

Modula-2: Abstractions for Data and Programming Structures (Using ISO-Standard Modula-2)

2002-2003 Edition

by

[Richard J. Sutcliffe](#)

Table of Contents

[Shareware Information](#)

[Acknowledgements](#)

[Preface for the Instructor](#)

[Intro for the Student](#)

[1. Planning for the Solution of Problems](#)

[2. From Plan to Program](#)

[3. Basic Program Structure Abstractions](#)

[4. Program Organization and Procedures](#)

[5. Iterations, Enumerations and Arrays](#)

[6. Program Organization and Modules](#)

[7. Solving Real World Problems in Modula-2](#)

[8. Data Storage Issues](#)

[9. Structured Data--Intermediate Techniques](#)

[10. Intermediate Program Structuring--Pitfalls and Techniques](#)

[11. Intermediate Programming--Data and Techniques](#)

[12. Pointers and Dynamic Data--
Introduction](#)

[13. Searching and Sorting](#)

[14. Intermediate Dynamic Data
Structures](#)

[15. Advanced Data Types and
Techniques](#)

[16. Generic Modula-2](#)

[17. Advanced Applications](#)

[18. Introduction To Graphics](#)

[19. Object Oriented Modula-2](#)

[Afterword](#)

[Appendices](#)

[Answers](#)

[About the Author](#)

Author

Rick Sutcliffe is Professor of Computer Science and Mathematics at Trinity Western University (7600 Glover Rd., Langley BC CANADA V2Y 1Y1). Before joining the University, he taught High School Math, Physics, and Computer Science for twelve years. He has written monthly columns for computer and high tech magazines, as well as numerous other articles and reviews for a variety of publications. He has represented Canada on the international committee charged with producing the standard for Modula-2 since its inception in 1987 and has taught and written in the language since 1983. His other publications include materials on ethical and social issues in technology, and some fiction.

[Contents](#)

Acknowledgements

Where used, "Apple" is a trademark of Apple Computer Corporation, "IBM" of International Business Machines, and "MS-DOS" of Microsoft Reference to these or any other trade names in no way implies that any of the parties mentioned endorse the use of this text in connection with their products.

References to Trinity Western University are not intended to imply that the University officially endorses all or any part of the contents of this text.

For the Second Edition:

This book could not have been written without the help of a number of people, and their assistance is gratefully acknowledged: My wife Joyce, without whose patience it could not have been finished; Jan Davis and Karen Jackson, my first secretaries, who typed most of the manuscript and fixed many spelling mistakes; and Tom Rowan, who tested many of the examples, and found more errors for me to correct. Richard Thomas, Glen Little, and Milton Schmidt tested and corrected all the programs in the final manuscript. Rich Gleaves, formerly of Volition Systems, Chris Cale of Logitech Corp., Lyle Bingham and Richard Ohran of MRI, and Mike Weisert of Borland International all spent much of their time on the telephone with me and provided valuable assistance. These, and Hochstrasser computing were all generous in keeping me up-to-date on their products and documentation. The final manuscript had all the programs tested and corrected, and the answer key was prepared by Richard Thomas, Glen Little, and Milton Schmidt, whose salaries, in part, were paid from a pre-publication grant from Merrill. The patience of two classes of Trinity Western students who suffered through the many corrections to the first edition are also appreciated. Surely two years of class testing have done some good!

For the Third Edition and the changes through 2003:

I am greatly indebted to my colleagues on the ISO committee (ISO/IEC/JTC1/SC22/WG13) that has met since 1987 to work on the standard for Modula-2. Roger Henry, Mark Woodman, and Martin Schoenhacker, the first three chairs of WG13 have been especially helpful, and their ideas and leadership are reflected extensively in the standard. Much of the fruit of the standards labour is reflected in the organization and examples of this book, which is a "Third Edition" in a formal sense only, as the material has been completely rewritten at least twice since the second (Merrill) edition. Gord Tisher assisted with testing and proofreading portions of the third edition and with the author's implementation of the standard libraries. He also implemented the HTML markup for the on-line edition. Dana Aldom and Nathan Sutcliffe incorporated numerous corrections and changes into the current version. Randy Yap produced the answer key and did corrections for the 1999-2000 edition. Many more of my students have now learned programming with Modula-2 in the intervening years, and the experience has, on the whole, enriched both me and them. I trust that this will continue to be the case in the future.

The original files for this book are maintained using an Apple Macintosh computer with the NisusWriter word processor. Sandra Silcot's Nisus macros (modified by Gord Tischer) were very useful in the process of converting to an HTML markup.

Preface For The Instructor

In its earliest years, Modula-2 was often regarded as an appropriate second computer language, one best studied after the student had mastered Pascal. However, Modula-2 did not simply add modern concepts to its rusty parent; it also inherited all of the clarity, simplicity and utility of Pascal, and is therefore a worthy successor, both in the classroom and for commercial production purposes.

Whether students are already experienced computer programmers or are relative beginners, many want to learn a language or other computer skills, on their own or with some assistance from a formal instructor. All of these need a text which is at once easy to read and yet complete enough to be used as a reference. It is therefore my hope that this book will be used and enjoyed as much by individual hobbyists and self-taught programmers with their own machines as by the mainframe-linked college and university students who constitute its normal market. After all, microcomputers--and the hobbyists and "hackers" they have brought to the scene--have caused a fresh breeze to blow through the whole realm of computer science. One purpose of this book is to open a window in the ivory tower and let in some fresh air by making Modula-2 accessible to beginning students in the formal classroom in a non-threatening manner. The student-oriented, casual and informal style of this book is therefore deliberately chosen, and in its first two editions, was very well received by students. This, coupled with a very broad coverage of the language, should place it squarely in the mainstream of modern computer education.

It must be kept in mind, however, that this is an introductory book, not an exhaustive treatise, and many things are not covered in depth. It is assumed that this course must be accompanied with or followed by more detailed treatments of algorithm construction and software engineering. Thus, such matters as the software engineering life cycle, object-oriented design, and program verification are discussed only lightly, are under other headings, or are not mentioned at all. On the other hand, it departs from the traditional text by attempting to cover typical data structures topics in an integrated fashion throughout. Instructors who wish to go into EBNF (Extended Backus-Naur Formalism, see [Appendix 3](#)) descriptions, or to get right into designing with modular design charts are certainly welcome to supplement this material--it provides an introduction to computer science through a language, but does not cover all possible topics.

Constructive criticism on all aspects of this book will be greatly appreciated. My students, adopters of the second (Merrill publishing) edition, and reviewers have already provided some. I have every intention of keeping this book alive through more editions and of including suggested improvements, especially those of students. After all, they are the "end-users" and their edification is the final judgement on this text. Finally, a word on the organization of the book itself. The interchapter notes constitute a broad introduction to the history of computing and to important ethical and other issues. They often have little to do (directly) with the programming content, and are intended to help set students in the appropriate cultural context in which this book exists. Old hands or hackers of long standing could skip some of these little words without damaging themselves seriously.

The first five to seven chapters may suffice as an appropriate one-semester introduction to programming for the beginner depending on what topics other than the language are covered. The pace at first is

designed to accommodate those who come to the course with only a modest background in computing. The next five to seven chapters could serve as the basis for a second programming course for these same students.

On the other hand, Chapters 7 and following could be selected from to provide the meat of a course for students already familiar with another programming language. These students could skim the first portion of the text in a few days, and begin serious work at Chapter 6. Thus the student with previous computing experience could cover through Chapter 9 (and selections beyond) in a four semester hour course. Depending on how many other topics are to be taken, the remaining material could be incorporated into a subsequent course, replacing the traditional data structures course, because of the integration of material. This book is intended to provide a relatively complete coverage of the main points of the Modula-2 language, but not for all possible versions of the language. It generally is complete for the ISO standard versions, though some references to others are included as well. With minor adaptations, particularly in those chapters dealing with input, output and utility modules, it should suffice for most installations. Teachers will almost certainly have to supplement those sections dealing with file handling--this is the area in which there is the greatest difference between versions of Modula-2. The author has spent several years on the ISO standards committee for Modula-2 as Canada's representative, and much material from the forthcoming standard has now been incorporated into the text. It is the hope of this committee that all vendors will eventually conform to the standard.

The assignments were kept somewhat general in the last edition, but specific problems from mathematics, business, chemistry, and physics have now been added. No student will want to do them all, nor any teacher to assign them all. Sufficient variety has been provided, especially in the earlier chapters, so that both those with a technical background, and also those wanting only a less technical introduction, can find suitable exercises. My approach has been to assign the general "Questions" to all students, and then to give them several choices from among the "Problems" so as to cater to their particular developing interests.

There are many other ways to select material. For instance, many of the examples and exercises are mathematically oriented, reflecting my own background and interests, but the general flow of the book does not depend heavily on this orientation. It is possible for an instructor to change this to a large extent for students on a less technical track. After all, a classroom teacher designs all her own examples for the lectures in any case, so these can (and probably should) represent a very different group of problems. In the latter chapters, some of the problems are quite challenging, and could be made the basis for major programming projects, even in a second or subsequent course.

Note that numerous changes were made to the HTML version for the 1997 and 1999-2000 edition. In addition, Chapters 12 and 13 were reversed; a new chapter of Generic Modula-2 was inserted before the original Chapter 16; some material that was out of place in chapter 7 was removed; and many small improvements were made. At this point, Chapters 1-7 are an introduction to programming; Chapters 8-12 make a good second course; Chapters 13-16 are intended for a course in data structures, and Chapter 17-19 are the first of several more in an advanced programming course. Note, however, that chapter 19 can also be used at an earlier and introductory level, perhaps in conjunction with a seminar introducing object oriented programming.

This book does not attempt to fully document all the features that may be available in every Modula-2 installation, nor does it attempt to teach how to operate any particular operating system. Such matters are the responsibility of the lab instructor and student. I would suggest that students should be very familiar

with the editing, filing, compiling, and linking functions of their particular system no later than by the time they finish Chapter 1, so as not to impede progress through the rest of the book.

[Contents](#)

Introduction For The Student

Why Compute?

If you are somewhat of a novice, and have flipped through the pages of this book, you may be wondering what you have gotten yourself into. Nothing of what you see makes much sense, and you may be having second thoughts about learning computing after all.

On the other hand, you may be fascinated by computers, already have a working knowledge of one or more computer languages, and want to upgrade your skills--you also have glanced through the book and are excited about what you see. Actually, you probably are so eager to get going that you will just skim this introduction and the first chapter for the essentials, and even that only if your instructor requires you to read it, so most of what is said here will be addressed to the other group.

So why compute? The high speed with which computers can perform tasks makes them suitable for solving problems that require a large number of repetitive steps, sorting through vast quantities of data, or performing complex calculations--provided in the latter case, that the calculation is first broken down into a series of simple steps by the programmer.

The ability to store data (such as the characters that form the words comprising this textbook) makes the computer useful indeed to writers. Few people who have learned to use a good word processor would ever go back to producing their manuscripts by pen and ink and then laboriously typing each of several drafts. After the first draft, corrections are made by instructing the computer through the word processing program to delete a paragraph here, insert a word there, correct a spelling somewhere else, move a chapter to a different place, and so on.

In the old system, a novelist who had just completed the final typed draft of a 500 page masterpiece, and was faced with the demand from an editor to change the name of the main character from John Doe to Sam Crud, might quail at the task of revising and re-typing all those many, many sheets. Now, a few keystrokes on the word processor, and the whole thing is done. Creating the new typed copy is simply a matter of waiting for the printer to crank it all out; one can read a book while it is happening.

Accountants and bookkeepers computerize their operations to gain speed, accuracy, and efficiency. No more the endless hours of searching for the missing \$0.21 by which the balance is off; now you know that the errors must be in typing the numbers in, because the totals derived by the computer are not subject to the kind of human error we all make when doing a massive addition (such as mentally transposing two digits.)

Want to look up some obscure piece of knowledge? If you know the name of the library, or database, where it is stored, you can usually dial it up on the telephone or connect to it on the network using your computer, search for and obtain what you want, and then store or print out a copy for your own use. If you are a researcher, this means that you can have accurate information about the problems you are working on at all times.

Computers provide a competitive edge in many areas of human endeavor; they can and do free people from drudgery and give them more power over their world. By giving people more time to think, more time to plan, a wider scope to create new ideas, more readily available information, more control over

their environment, and more and more time (potentially) to spend with people, computers and similar high technology devices are ushering in a new type of society--one characterized by a great flowering of creativity, a vast outpouring of new ideas (especially of new writing), and instant access to virtually any information.

This "Information Age", or "Communications Age", or "New Renaissance", is already well under way. While not everyone who lives in the new era does all the things described here, there are few occupations untouched by computers. So many people use them for so many things, that those who have no knowledge of them are functionally illiterate. Education for life must include some work with computers as a matter of necessity, not as an option.

Whether you are an old hand at programming and are just picking up another language, or whether you are a novice, these things should add some excitement and anticipation to your study--some degree of feeling that you are a part of history in the making.

The majority of people in the New Renaissance will be satisfied by the ability to make good use of applications other people have written. In fact many of you will do a large percentage of your computing with only one or two programs, and will never need anything else.

Some of you are more curious, and like to know how it's done, even if you may never be part of a large programming team rushing the latest "wunderprodukt" to market. You may never go beyond a course or two in Computer Science before your interest begins to wane, and you decide you have had enough theory to suit your needs.

Still others have or will develop a talent for the reduction of large problems to smaller more manageable pieces, and derive a great deal of satisfaction from mechanizing solutions and seeing the results as working programs. For you, programming is more than merely interesting; it has the potential to become a livelihood and a profession, and programming is just the starting place--you must go on to study many additional aspects of the craft before being ready to hang out a professional's shingle.

Returning to the beginner--you may have looked around and seen that the world in which you live is changing faster and faster as more and more new technology impacts on the way people live and work. It seems as though everyone and her aunt is involved with computers, and there are people around who talk an incomprehensible "high priestly" language full of words like ROM and Modula-2. You smile politely, but vacantly, hoping that no one notices your ignorance. You realize that even if you were to shout in frustration: "Stop the world, I want to get off!" no one will have the time to say much more than: "Go ahead, get off."

Whoever you are, you have bought this book on your own, or have taken a University course that requires it, and you may be expecting from it an easy initiation to certain of the mysteries of Computer Science. You may think that you can breeze through it, perhaps sit in on a few lectures, and then you will be able to sling the buzzwords with the best of them.

If you are part of the group using this book as a second or subsequent language, you already know that it's not that easy. This book is intended to introduce freshman University students to one means of giving instructions to computers (i.e., a programming language or notation.) Learning any worthwhile skill, especially a communications one, takes time, dedication, and a lot of hard work. However, it will not be assumed that you already are, or plan to become a professional in the field, though a book such as this one could be the first step along that path.

While you may not become a professional programmer, your life is going to be touched by computers, whether you like the idea or not. By learning the rudiments of a modern programming language, you will

gain an appreciation for how computers work, and for how applications are actually programmed. Not only will computers be demystified for you, but your own problem solving skills will be sharpened, and you will be able to apply the design principles to other areas of your work as well. In addition, if you are ever in a position of management, you will not only know what you can expect of your computer personnel, you will know what not to expect.

In the spirit of the information age, this book is available on the internet to all who wish to use it. All its contents are, however, copyright, and cannot be used as if they were your own work in your course; any such use must be acknowledged. Moreover, the book is not free, but shareware. See the copyright section for more information on payment.

The course that this text represents will cost you a large investment in time and energy. It will be a lot of work. It is for your future, and mine.

Enjoy, eh?

Richard J. Sutcliffe

Trinity Western University

Fort Langley, British Columbia--1986, 1992, 1995, 1997, 1998, 1999, 2002

[Contents](#)

Chapter 1

Planning for the Solution of Problems

- [1.1 What is Problem Solving?](#)
 - [1.2 Problem Solving and Computing](#)
 - [1.3 Top Down Analysis](#)
 - [1.4 Tools for Problem Solving](#)
 - [1.5 The Role of Abstraction](#)
 - [1.5.1 Abstractions and Computing](#)
 - [1.6 Data Representation Abstractions](#)
 - [1.6.1 Data and Information](#)
 - [1.6.2 Encoding \(Representing\) Data](#)
 - [1.6.3 Atomic and Compound Entities](#)
 - [1.6.4 The Concept of Type](#)
 - [1.6.5 Variables and Constants](#)
 - [1.7 Data Manipulation Abstractions \(Expressions\)](#)
 - [1.7.1 Precedence](#)
 - [1.7.2 Expression compatibility](#)
 - [1.8 Abstractions for Computing Machines](#)
 - [1.8.1 Computer Hardware Organization](#)
 - [1.8.2 Computer Software Organization](#)
 - [1.8.3 Computing Notations](#)
 - [1.9 Abstractions for Instructing Machines--Program Structures](#)
 - [1.9.1 Program Control Abstractions](#)
 - [1.9.2 Encoding \(Representing\) Programs--Pseudocode](#)
 - [1.9.3 Syntax, and Semantics](#)
 - [1.9.4 The Control of Errors](#)
 - [1.10 Chapter Summary](#)
 - [1.11 Assignments](#)
-

Chapter 2

From Plan to Program

[2.0 Chapter Goals](#)

[2.1 Giving Birth--A First Modula-2 Program](#)

[2.2 The Anatomy of an Infant Program](#)

[2.2.1 What is a Module?](#)

[2.2.2 Reserved Words](#)

[2.2.3 Standard Library Tools](#)

[2.2.4 A Name for the Baby--Identifiers](#)

[2.2.5 Strings](#)

[2.2.6 Summary](#)

[2.3 How to Solve a Problem](#)

[2.3.1 Analysis](#)

[2.3.2 Planning and Refining a Solution](#)

[2.3.3 Data Tables and Sample I/O](#)

[2.3.4 Refining the Solution](#)

[2.3.5 Execution and Satisfaction](#)

[2.4 Documenting the Solution](#)

[2.4.1 External Documentation](#)

[2.5 Variables in Modula-2](#)

[2.5.1 Simple Variable Types](#)

[2.5.2 Standard Identifiers](#)

[2.6 Literals and Constants](#)

[2.6.1 Literals](#)

[2.6.2 Constants](#)

[2.7 Expressions for Constants and Variables](#)

[2.7.1 Precedence in Modula-2](#)

[2.7.2 Mixed Expressions and Modula-2](#)

[2.8 Simple Output Methods](#)

[2.9 REAL Variables](#)

[2.9.1 Real Operations](#)

[2.9.2 The Format of Real Output](#)

[2.10 Type Compatibility and REAL](#)

[2.11 Interactive Programs](#)

[2.12 An Extended Example \(Bank Interest\)](#)

[2.12.1 Input in non ISO Versions](#)

[2.13 Chapter Summary](#)

[2.14 Assignments](#)

[Programming Note--I/O In Non-Standard Modula-2](#)

[**Contents**](#)

Chapter 3

Basic Program Structure Abstractions

[3.0 Chapter Goals](#)

[3.1 Statement Sequences](#)

[3.2 Simple Selection](#)

[3.3 Boolean Variables and Expressions](#)

[3.4 Repetition \(1\) -- the WHILE Statement](#)

[3.5 Repetition \(2\) -- the REPEAT Statement](#)

[3.6 Analysis of Loops](#)

[3.6.1 Loops and Boolean Flags](#)

[3.7 Counting Loops](#)

[3.8 Replacing Loops With Closed Forms](#)

[3.9 Qualified Import](#)

[3.10 Insect Control Again--Some Common Errors](#)

[3.11 Style and Prettyprinting](#)

[3.12 An Extended Example \(Day Number in a year\)](#)

[3.13 Chapter Summary](#)

[3.14 Assignments](#)

[First Interregnum--A Short History of Computing](#)

[Contents](#)

Chapter 4

Program Organization and Procedures

[4.0 Chapter Goals](#)

[4.1 What is a Procedure, and Why Use It?](#)

[4.2 Writing and Calling Procedures](#)

[4.3 Value and Variable Parameters \(Introduction\)](#)

[4.4 Predicates](#)

[4.5 Function Procedures](#)

[4.6 Summary of Built-in Procedures](#)

[4.7 Some Stylistic Considerations](#)

[4.8 Recursion](#)

[4.9 An Extended Example \(Compound Amounts\)](#)

[4.10 Chapter Summary](#)

[4.11 Assignments](#)

[Contents](#)

Chapter 5

Iterations, Enumerations, and Arrays

- [5.0 Chapter Goals](#)
 - [5.1 Abstract and Transparent Data Types in Modula-2](#)
 - [5.2 Making One's Own Data Types](#)
 - [5.2.1 Ordinal and Enumerated Types](#)
 - [5.2.2 Subranges Of Existing Types](#)
 - [5.2.3 Summary of some Modula-2 compatibility issues:](#)
 - [5.2.4 Summary of some Modula-2 types](#)
 - [5.2.5 Making comparisons](#)
 - [5.3 Indexed Data Types--Arrays](#)
 - [5.3.1 A First Look at String Variables](#)
 - [5.4 The FOR Statement](#)
 - [5.4.1 The FOR Loop and the WHILE Loop](#)
 - [5.4.2 The FOR Loop in Use](#)
 - [5.5 Manipulating Arrays](#)
 - [5.6 Arrays as Parameters](#)
 - [5.7 Multi-Dimensional Arrays](#)
 - [5.7.1 Arrays of More than Two Dimensions](#)
 - [5.7.2 Multidimensional Open Array Parameters](#)
 - [5.8 Manipulating Multi-Dimensional Arrays](#)
 - [5.9 An Extended Example \(Finding Prime Numbers\)](#)
 - [5.10 Chapter Summary](#)
 - [5.11 Assignments](#)
-

[Contents](#)

Chapter 6

Program Organization and Modules

- [6.0 Chapter Goals](#)
- [6.1 What Did You Say a Module is?](#)
- [6.2 Libraries--How to Borrow a Module and Sign itOut](#)
- [6.3 The Standard Library \(1\)--I/O](#)
 - [6.3.1 ISO Standard I/O Modules](#)
 - [6.3.2 Classical I/O--The InOut Family](#)
 - [6.3.3 Alternate Origins and Destinations](#)
 - [6.3.4 Redirection](#)
 - [6.3.5 Writing Special Characters To a Terminal](#)
- [6.4 The Standard Library \(2\)--Mathematical Functions](#)
 - [6.4.1 Square Root](#)
 - [6.4.2 Exponential and Logarithmic Functions](#)
 - [6.4.3 Trigonometric Functions](#)
 - [6.4.4 Conversions](#)
 - [6.4.5 Summary of RealMath](#)
 - [6.4.6 Other Mathematical functions](#)
- [6.5 Starting Your Own Libraries](#)
 - [6.5.1 Modules and User-Defined Data Types](#)
- [6.6 Handling Errors in Library Modules](#)
- [6.7 Modules and Design Considerations](#)
- [6.8 Libraries--an Overview](#)
- [6.9 An Extended Example \(Coordinate Geometry\)](#)
- [6.10 Chapter Summary](#)
- [6.11 Assignments](#)
 - [Second Interregnum--Nellie and the Pirates](#)

Chapter 7

Solving Real World Problems in Modula-2

[7.0 Chapter Goals](#)

[7.1 Introduction to Some Applications of Modula-2](#)

Part A--Strings

[7.2 Communicating in English](#)

[7.3 Library String Functions](#)

[7.4 Comparing and Manipulating Strings](#)

[7.5 An Application for Strings--Program Menus](#)

[7.6 Message Encoding and Cryptography](#)

Part B--Other Applications

[7.7 Some Statistical Tools](#)

[7.8 Random Numbers](#)

[7.9 Longer Cardinals](#)

[7.10 Matrices](#)

[7.10.1 Matrix Operations](#)

[7.10.2 Matrices and Determinants](#)

[7.11 Applications from Physics](#)

[7.12 Further Applications From Business](#)

[7.13 Chapter Summary](#)

[7.14 Assignments](#)

[Contents](#)

Chapter 8

Data Storage Issues

[8.0 Chapter Goals](#)

[8.1 Storage--An Introduction](#)

Part A--Machine and System Level Storage Issues

[8.2 An Introduction to the Lower Level](#)

[8.2.1 General Considerations](#)

[8.2.2 Low Level Numeric Notations](#)

[8.2.3 Machine Level Data Storage](#)

[8.2.4 Hexadecimal and Octal Notation](#)

[8.3 High Level Access to Low Level Facilities in Modula-2](#)

[8.3.1 The Module SYSTEM](#)

[8.3.2 Variables at Fixed Addresses](#)

[8.3.3 Hexadecimal and Octal Notation in Modula-2](#)

[8.4 Extended Low Level Examples](#)

[8.4.1 Keyboard Reading--Operating System Level](#)

[8.4.2 Generic Swap](#)

Part B--Input, Output, and Files

[8.5 Files--Introduction and Terminology](#)

[8.5.1 Sequences, Streams, and Channels](#)

[8.5.2 Sequential and Random Access Files](#)

[8.5.3 Planning to Use Streams with Files](#)

[8.6 Text I/O in ISO Standard Modula-2](#)

[8.6.1 The Restricted Stream Model](#)

[8.6.2 The Rewindable Sequential Stream Model](#)

[8.6.3 File Text I/O in non-Standard Modula-2](#)

[8.7 Binary I/O](#)

[8.8 Notes on File I/O](#)

[8.8.1 Special Notes on the Macintosh Operating System](#)

[8.9 Standard Channel I/O in ISO Modula-2](#)

[8.10 Lower Level I/O in ISO Modula-2](#)

[8.11 An Extended Low Level I/O Example--TermFile](#)

[8.12 Chapter Summary](#)

[8.13 Assignments](#)

Chapter 9

Structured Data--Intermediate Techniques

[9.0 Chapter Goals](#)

[9.1 Structured Data Revisited](#)

Part A--Sets

[9.2 Representation and Membership](#)

[9.2.1 Set Union](#)

[9.2.2 Set Intersection](#)

[9.2.3 Set Difference](#)

[9.2.4 Symmetric Set Difference](#)

[9.2.5 One Element at a Time](#)

[9.3 Set Comparisons](#)

[9.3.1 Set Equality and Inequality](#)

[9.3.2 Subset](#)

[9.3.3 Superset](#)

[9.4 Sets and the I/O Library](#)

[9.5 Sets at the Low Level](#)

[9.5.1 Bitsets](#)

[9.5.2 Packed Sets](#)

[9.6 An Extended Example](#)

Part B--Records

[9.7 Declaring and Assigning to Records](#)

[9.8 Using Records](#)

[9.9 Records and Arrays--Which, or Both?](#)

[9.10 Getting Physical With Records](#)

[9.10.1 RawIO in Non-Standard Modula-2](#)

[9.11 Records and Random Access Files](#)

[9.12 An Extended Example \(Inventory\)](#)

[9.12.1 Inventory with Raw Sequential I/O](#)

[9.12.2 Inventory with Raw Random I/O](#)

[9.13 Chapter Summary](#)

[9.14 Exercises](#)

Chapter 10

Intermediate Program Structuring

[10.0 Chapter Goals](#)

[10.1 Introduction](#)

Part A--Scope and Visibility Issues

[10.2 Blocks, Global and Local Variables, Side Effects](#)

[10.2.1 Procedure Blocks and Scope](#)

[10.2.2 Side effects and Counting Loops](#)

[10.2.3 Other Global side effects](#)

[10.2.4 Nested Procedure Scopes](#)

[10.3 Parameters Revisited](#)

[10.3.1 The Scope of Parameters](#)

[10.3.2 Parameters and side effects](#)

[10.4 Procedure Types and their Variables](#)

[10.5 Local Modules--Scope and Visibility Rules](#)

[10.5.1 Standard Identifiers and Scope](#)

[10.5.2 Dynamic Modules](#)

[10.6 An Extended Example--Fibonacci Sequences](#)

[10.7 Library Modules--Scope and Visibility Rules](#)

[10.7.1 Access to Imported Libraries at Inner Scopes](#)

[10.7.2 Visibility in Library Modules](#)

[10.7.3 Opaque Types--a Brief Introduction](#)

Part B--Program Control and Error Handling Issues

[10.8 Transfer of Control](#)

[10.8.1 GOTO and Other Noxious Weeds](#)

[10.8.2 RETURN in a Regular Procedure](#)

[10.8.3 Repetition Revisited--The Generalized LOOP Statement](#)

[10.9 Error Handling Revisited](#)

[10.9.1 Typical Library Errors](#)

[10.10 Controlling Program Termination](#)

[10.10.1 HALT and Abnormal Termination](#)

[10.11 FINALLY: Termination Detection and Cleanup](#)

[10.12 Exceptions](#)

[10.12.1 Language Exceptions](#)

[10.12.2 Library Exceptions](#)

[10.12.3 Handling Exceptions](#)

[10.12.4 Exceptions and Termination](#)

[10.12.5 User-Defined Exceptions](#)

[10.13 An Extended Example--Fractions and Exceptions](#)

[10.14 Chapter Summary](#)

[10.15 Assignments](#)

[Contents](#)

Chapter 11 Intermediate Programming-- Data and Techniques

[11.0 Chapter Goals](#)

[11.1 Introduction](#)

Part A--Programming Techniques

[11.2 Recursion Revisited](#)

[11.2.1 The Knight's Tour--An Extended Example](#)

[11.3 Selection Revisited-The CASE Statement](#)

[11.4 Pragmas](#)

[11.5 Efficiency in Large Programs](#)

[11.5.1 Controlling Run-Time Checking](#)

[11.5.2 Compiling For Specified Environments](#)

[11.5.3 Fine Tuning Loops](#)

[11.5.4 Linking, Program Libraries and Speed](#)

[11.5.5 Efficiency--A Summary](#)

Part B--Intermediate (Structured) Data Issues

[11.6 Structure Constructors](#)

[11.6.1 Array Constructors](#)

[11.6.2 Record Constructors](#)

[11.7 The Variant Record--a Chameleon](#)

[11.8 An Extended Example--Variant Records](#)

[11.9 Chapter Summary](#)

[11.10 Assignments](#)

[Contents](#)

Chapter 12

Pointers and Dynamic Data

[12.0 Chapter Goals](#)

[12.1.1 Pointer Variables](#)

[12.1.2 Pointer References](#)

[12.1.3 The value NIL](#)

[12.2 Applications of Pointers](#)

[12.2.1 Pointers and Parameters](#)

[12.2.2 Pointers and Sorting](#)

[12.3 Pointer Arithmetic](#)

[12.4 Dynamic and Static Memory](#)

[12.4.1 Static Memory Use](#)

[12.4.2 Procedures and the Stack--Automatic Dynamics](#)

[12.4.3 Dynamic Memory and the Heap--Program Controlled Dynamics](#)

[12.5 Managing Dynamic Memory in Modula-2](#)

[12.6 An Example--Dynamic Records and Files](#)

[12.7 Towards A Dynamic Array ADT](#)

[12.8 Pointers and Return Types](#)

[12.9 Opaque Types Revealed](#)

[12.10 Pointers and Lists](#)

[12.10.1 Declaring The Linked List Apparatus](#)

[12.10.2 Maintaining The Linked List](#)

[12.11 Variations on the List Theme](#)

[12.11.1 Circular Lists](#)

[12.11.2 Two-Way Lists](#)

[12.12 Variant Dynamic Records](#)

[12.13 Chapter Summary](#)

[12.14 Assignments](#)

Chapter 13

Searching and Sorting

[13.0 Chapter Goals](#)

[13.1 Searching](#)

[13.1.1 Linear \(Sequential\) Searches](#)

[13.1.2 Binary Searches](#)

[13.2 Introduction to Sorting](#)

[13.2.1 The Bubble Sort](#)

[13.2.2 The Selection Sort](#)

[13.3 Analyzing Sorting Routines](#)

[13.4 Inserting Methods](#)

[13.4.1 The Insert Sort](#)

[13.4.2 The Shell Sort](#)

[13.4.3 The Combsort](#)

[13.5 Advanced Sorting of Arrays](#)

[13.5.1 The QuickSort](#)

[13.6 Sorting With Auxiliary Storage](#)

[13.6.1 The Merge Sort](#)

[13.6.2 Pointers and Sorting](#)

[13.7 An Extended Example \(Searching in Text\)](#)

[13.8 Chapter Summary](#)

[13.9 Exercises](#)

[Contents](#)

Chapter 14

Intermediate Data Structures

[14.0 Chapter Goals](#)

[14.1 Introduction to Intermediate Data Structures](#)

[14.2 Lists Revisited](#)

[14.3 Queues](#)

[14.4 Stacks](#)

[14.5 Tables](#)

[14.6 An Introduction to Table Indexing Methods](#)

[14.7 Trees](#)

[14.8 An Extended Example--A Binary Search Tree](#)

[14.9 Chapter Summary](#)

[14.10 Assignments](#)

[Contents](#)

Chapter 15

Advanced Data Types and Techniques

[15.0 Chapter Goals](#)

[15.1.1 B-trees Defined](#)

[15.2 Implementing and Testing a Semi-Generic B-tree](#)

[15.3 Heaps](#)

[15.3.1 Heaps Defined](#)

[15.3.2 Heaps as Binary Trees](#)

[15.3.3 Defining the Heap](#)

[15.4 Implementing and Testing a Semi-Generic Heap](#)

[15.5 Array Implementation and Sorting With Heaps](#)

[15.5.1 Heapsort](#)

[15.6 Toward More Generic Structures](#)

[15.6.1 Low Level Assignment Routines](#)

[15.8 Pointers and Memory Management Revisited](#)

[15.8.1 Orphans](#)

[15.8.2 Garbage](#)

[15.8.3 Fragmentation](#)

[15.8.4 Defragmentation and Garbage Collection](#)

[15.8.5 Handles](#)

[15.9 Pointers and Generic Structures](#)

[15.10.1 Generic Sorts Defined](#)

[15.12 Assignments](#)

[Contents](#)

Chapter 16

Generic Modula-2

[16.0 Chapter Goals](#)

[16.1 Generics In the Base Language \(Revisited\)](#)

[16.1.1 Semi Generic Methods and Structures](#)

[16.1.2 Limitations of Fully Generic Techniques](#)

[16.1.3 Limitations of Fully Generic Structures](#)

[16.1.4 Summary](#)

[16.2 Generic Separate Library Modules](#)

[16.2.1 Generic Definition Modules](#)

[16.2.2 Generic Implementation Modules](#)

[16.2.3 Formal Module Parameters](#)

[16.3 Refining Separate Library Modules](#)

[16.3.1 Refining Definition Modules](#)

[16.3.2 Refining Implementation Modules](#)

[16.3.3 Multiple Refinements](#)

[16.3.4 Actual Parameters](#)

[16.4 Refining Within a Program Module](#)

[16.5 Refining Within an Implementation Module](#)

[16.6 Making New ADTs from Old With Generics](#)

[16.7 Extended ADT Examples](#)

[16.7.1 Generic Lists](#)

[16.7.2 Generic Queues](#)

[16.7.3 Generic Stacks](#)

[16.8 Dependency and Order in Generic Modula-2](#)

[16.8.1 Module Dependencies](#)

[16.8.2 Nested Module Refinement Order](#)

[16.8.3 Module Initialization and Termination Order](#)

[16.8.4 Import and Export Lists](#)

[16.9 Summary and Comparison With Other Notations](#)

[16.10 Chapter Summary](#)

[16.11 Assignments](#)

Chapter 17

Advanced Applications

[17.0 Chapter Goals](#)

[17.1 Standard and Non-Standard Numeric Types](#)

[17.2 Complex Numbers](#)

[17.2.1 Complex Numbers Defined](#)

[17.2.2 Implementing non-ISO Complex Numbers](#)

[17.2.3 Testing the non-ISO Complex Implementation](#)

[17.2.4 Opaque non-ISO Complex Numbers](#)

[17.2.5 Testing the Opaque non-ISO Complex Implementation](#)

[17.3 ISO Complex Types and Support](#)

[17.3.1 ISO COMPLEX Math Library Support](#)

[17.3.2 Implementing ComplexMath](#)

[17.3.3 Input and Output](#)

[17.4 Electrical Circuits and Complex Numbers](#)

[17.5 Very Long Cardinals--The Type Decimal](#)

[17.6 Binary Coded Decimal Fixed Point Types](#)

[17.6.1 BCD Support in p1 Modula-2 \(Optional\)](#)

[17.7 A Suggested Project--Polynomials](#)

[17.8 The Date and Time](#)

[17.8.1 The Module SysClock](#)

[17.8.2 Time and Date I/O](#)

[17.8.3 Time and Date Arithmetic](#)

[17.9 A Closer Look at Whole Number I/O](#)

[17.9.1 Common Conversion Information Modules](#)

[17.9.2 Scanning For Whole Number Input](#)

[17.9.3 High Level String Conversion Routines](#)

[17.9.4 High Level Whole Number I/O](#)

[17.10 Chapter Summary](#)

[17.11 Assignments](#)

Chapter 18

Introduction To Graphics

[18.0 Chapter Goals](#)

[18.1 Basic Graphics Concepts](#)

[18.1.1 Discrete Grids--Graphing Pixels](#)

[18.1.2 Where is the origin?](#)

[18.1.3 Measuring Angles](#)

[18.2 A Graphics Environment](#)

[18.3 Using The Module GraphPaper](#)

[18.4 Recursive Drawing--Fractals](#)

[18.4.1 Snowflake Fractals](#)

[18.4.2 A Tree Fractal](#)

[18.4.3 Singly-Recursive Snowflake-like Fractals](#)

[18.4.4 Sierpinski's Curve](#)

[18.5 String Art](#)

[18.6 An Extended Example--Implementing GraphPaper](#)

[18.6.1 Defining the Module GraphWindow](#)

[18.6.2 Implementing GraphWindow in MacOS](#)

[18.6.3 Implementing GraphWindow in Windows NT](#)

[18.6.4 Implementing GraphPaper in MacOS](#)

[18.6.5 Implementing GraphPaper in Windows NT](#)

[18.7 Chapter Summary](#)

[18.8 Assignments](#)

[Contents](#)

Chapter 19

Object Oriented Modula-2

[19.0 Chapter Goals](#)

[19.1 Introduction to Object Oriented Thinking](#)

[19.2 Object Oriented Terminology](#)

[19.2.1 Basic Definitions](#)

[19.2.2 Reference versus Value Objects](#)

[19.3 Getting Started with Object Oriented Modula-2](#)

[19.3.1 A Little History](#)

[19.3.2 Some Simple OOM-2 Programs](#)

[19.3.3 Summary of Basic OOM-2 Traced Class Semantics](#)

[19.4 Untraced Objects](#)

[19.5 Assignment and Comparison of Objects](#)

[19.6 Encapsulation of Classes in Separate Modules](#)

[19.7 Inheritance](#)

[19.7.1 Why Inherit](#)

[19.7.2 Inheritance in OOM-2](#)

[19.7.3 Assignment Compatibility between Classes and Subclasses](#)

[19.7.4 Overriding Methods in Subclasses](#)

[19.7.5 Class and Object References](#)

[19.7.6 Why Single Inheritance?](#)

[19.8 Abstract Classes](#)

[19.9 Guard Statement](#)

[19.10 Additions to the Libraries](#)

[19.10.1 Exceptions](#)

[19.10.2 Coroutines](#)

[19.10.3 The Module Garbage Collection](#)

[19.11 Extended Example--Points and Vectors](#)

[19.12 Example Outline--Personnel Records](#)

[19.13 On the Use of Programming Paradigms](#)

[19.14 Chapter Summary](#)

[19.15 Assignments](#)

[Programming Note--Comparisons With Other Object Notations](#)

Afterword: You and Modula-2--Why did you Really buy This Book?

You have done one or more whole courses in Modula-2 now, and what can you conclude? What have you gained, or what have you become? Are you now a professional computer scientist?

To the latter, no, not unless you were before starting this book, and you used its contents only to add another language to your growing repertoire. If this lofty status is indeed your goal, your next step is probably a course in the management of large programs and data structures. You will also need to learn about how that computer you have been using works (digital systems and hardware design), a few courses in such things as systems design, expert systems ("artificial intelligence"), computers and the law, computers and society, some applications courses (design and use) such as in data-bases, communications, networking, and so on. More work on advanced programming (graphics, GUIs and operating systems) would top things off nicely, and make you an apprentice. You have a long way to go. Are you a real programmer?

Same as above, to put it succinctly. This book has covered an introduction to the facilities of a single programming language, and while it is the author's belief that Modula-2 offers most of the modern facilities any programmer needs and a nearly ideal programming environment for development work, there are many other languages that will be more suitable (at least in the near future) for specific tasks. If you are planning to become a professional programmer, you should learn at least two other high-level languages that are substantially different from Modula-2, at least one of C++, Java, Pascal or Ada (each similar in their own way) and at least one machine language.

Moreover, you should be familiar with your main working language in several different operating systems (environments). If you have learned Modula-2 on a mainframe or large mini-computer, you should get a microcomputer version and learn that, or vice-versa. In any event, find one with full windowing and learn how to generate applications that take full advantage of such features. Above all, never stop learning, for computer science is changing and being changed so rapidly that you will quickly be left far behind if you stay in one place.

Has your life been enriched by this book?

Silly, presumptuous and inappropriate question. Still, one would hope so.

--Rick Sutcliffe, Ft. Langley 1988, 1990, 1995, 1999

Appendices

[Appendix 1--The Lexis of Modula-2](#)

[A1.1 Reserved Words \(Keywords\)](#)

[A1.2 Standard \(Pervasive\) Identifiers](#)

[A1.3 Standard Symbols](#)

[A1.4 Standard Operators](#)

[Appendix 2--Syntax Diagrams](#)

[A2.1 Lexis](#)

[A2.2 Syntax](#)

[Appendix 3--The Syntax of Modula-2](#)

[A3.1 A Notation to Describe Languages](#)

[A3.2 Some Examples of EBNF](#)

[A3.3 The Syntax of Modula-2 in EBNF](#)

[Appendix 4--Classical Library Modules](#)

[A4.1 High Level Input and Output](#)

[A4.2 Mathematical Functions](#)

[A4.3 SYSTEM and Other Low Level and System Access Modules](#)

[A4.4 Storage](#)

[A4.5 String Handling](#)

[A4.6 File I/O](#)

[A4.7 Character Information--ASCII](#)

[Appendix 5--ISO I/O Library](#)

[A5.1 An Overview of the ISO I/O Library](#)

[A5.2 I/O On Standard Channels](#)

[A5.3 Supplied Channels](#)

[A5.4 Specified Channels](#)

[A5.5 Channel Constants--IOConsts](#)

[A5.6 Device Independent Channel I/O--IOChan](#)

[A5.7 Device Drivers](#)

[A5.8 Device Module Constants--ChanConsts](#)

[A5.9 Linking Drivers to Channels--IOLink](#)

[Appendix 6--ISO Support Modules for This Text](#)

[A6.1 RedirStdIO](#)

[A6.2 Files](#)

[A6.3 Keyboard](#)

[A6.4 CharBuffer](#)

[A6.5 STerminal](#)

[A6.6 ACSCI](#)

[A6.7 SComplexIO](#)

[A6.8 SLongComplexIO](#)

[A6.9 ComplexIO](#)

[A6.10 LongComplexIO](#)

[A6.11 GraphPaper](#)

[A6.12 GraphWindow](#)

[Appendix 7--ISO Required System Modules](#)

[A7.1 SYSTEM](#)

[A7.2 COROUTINES](#)

[A7.3 EXCEPTIONS](#)

[A7.4 TERMINATION](#)

[A7.5 M2EXCEPTION](#)

[Appendix 8--ISO Utility and Information Modules](#)

[A8.1 Characters and Strings](#)

[A8.2 High Level String Conversion Modules](#)

[A8.3 Low Level String Conversion Modules](#)

[A8.4 SysClock--The Date and Time](#)

[Appendix 9--ISO Mathematics Library Module](#)

[A9.1 RealMath](#)

[A9.2 LongMath](#)

[A9.3 ComplexMath](#)

[A9.4 LongComplexMath](#)

[Appendix 10--ISO Process Support](#)

[A10.1 Processes](#)

[A10.2 Semaphores](#)

[Appendix 11--Modula-2 and Pascal](#)

[A11.1 Statement Syntax Differences](#)

[A11.2 Symbols](#)

[A11.3 Overall Structure](#)

[Appendix 12--Generic Modula-2 Syntax](#)

[A12.1 Keywords](#)

[A12.2 Diagrams of Changes to Base Language Syntax](#)

[A12.3 Generic Modula-2 Syntax Diagrams](#)

[A12.4 Changes To the Syntax of the Base Language in EBNF](#)

[A12.5 The Syntax of Generic Modula-2 in EBNF](#)

[Appendix 13--Object Oriented Modula-2 Syntax](#)

[A13.1 Keywords and Pervasive Identifiers](#)

[A13.2 Diagrams of Changes to Base Language Syntax](#)

[A13.3 Object Oriented Modula-2 Syntax Diagrams](#)

[A13.4 Changes To the Syntax of the Base Language in EBNF](#)

[A13.5 The Syntax of Object Oriented Modula-2 in EBNF](#)

[A13.6 Other Changes to the Base Language](#)

[A13.7 ISO Libraries Supporting Object Oriented Modula-2](#)

[Appendix 14--Bibliography](#)

[Answers to Questions and Selected Problems](#)

[Contents](#)

ANSWER KEY

WARNINGS:

1. As of 2002 06 18, this is a third, corrected answer set.
 2. No guarantees are expressed or implied.
 3. Answers to only a few of the problems are provided, and none after chapter 12.
 4. To keep this document short, problem answers are very brief, and should be taken as coding suggestions only. They are not suitable as guides for style or commenting.
-

[Chapter 1](#)

[Chapter 2](#)

[Chapter 3](#)

[Chapter 4](#)

[Chapter 5](#)

[Chapter 6](#)

[Chapter 7](#)

[Chapter 8](#)

[Chapter 9](#)

[Chapter 10](#)

[Chapter 11](#)

[Chapter 12](#)

[Chapter 13](#)

[Chapter 14](#)

[Chapter 15](#)

[Contents](#)

Appendix 3 The Syntax of Modula-2

A3.1 A Notation to Describe Languages

The purpose of this section is to describe a concise notation in which the syntax of computer notations (languages) can be unambiguously defined or described.

A computer language is a collection of sequences of symbols that are formed according to certain well-defined rules. Such rules constitute the grammar of the notation, or language. In a sense, a computer program can be thought of as corresponding to a sentence in some spoken language, and the component parts of a program are roughly analogous to such language constructs as subject, predicate, and object or complement.

Spoken languages, though they express ideas with a finite number of sounds (phonemes) are capable of expressing an infinite number of such ideas. Likewise, a programming notation may have a finite number of symbols and rules for manipulating these, but be capable of expressing an infinite number of programs.

There are four rules which allow one to create all the syntax definitions for any programming language. They are similar to the basic programming abstractions noted in chapter one of this text and elaborated on in Modula-2 for the writing of programs.

1. The Rule of Succession

If a language construction C uses two elements P and Q in succession, one expresses this as:

$$C = PQ$$

2. The rule of alternatives

If a language construction C uses one or the other of two elements P or Q , one writes this rule as:

$$C = P \mid Q$$

3. The rule of option

If a language construction C may be either P or nothing at all (empty), then one writes the syntax rule as:

$$C = [P]$$

4. The rule of Repetition

If a construction C consists of any number (including zero) of repetitions of P then one writes:

$$C = \{ P \}$$

The notation described by the four rules of succession, alternatives, option, and repetition with the notation above is called Extended Backus-Naur Formalism or EBNF for short.

A3.2 Some Examples of EBNF

Before looking at the complete definition of Modula-2 in this notation, consider a few particular cases and see how EBNF can save space and gain some precision.

One of the earliest definitions was that of an identifier. It went as follows:

An identifier is a sequence of letters and numbers that starts with a letter.

The same rule can be written in EBNF as:

```
identifier = letter { letter | digit }
```

As indicated in the rules above, the braces signify a repetition of any number of the elements inside them, and this can be either a letter or a digit.

A further example of the use of the vertical bar to indicate alternatives is given in the following series of definitions:

```
octalDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"  
digit = octalDigit | "8" | "9"  
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F"
```

Here, each definition is built up from the previous one in order to avoid needless repetition. Notice the use of quotes to surround elements which, if they are used, must appear exactly as given. That last line for instance tells us the rather familiar fact that a hexadecimal digit must be either a digit or a letter from "A" to "F".

The option notation is can be illustrated in the description of the import construction, where the *FROM* *moduleName* is an optional form. One could express this in EBNF as:

```
import = [FROM ident] IMPORT identList ";"
```

Notice how the EBNF shows clearly that the semicolon is required at the end of this construction.

A3.3 The Syntax of Modula-2 in EBNF

The concrete syntax in this section is taken from ISO/IEC IS 10514, the international standard for Modula-2.

```
compilation module =  
    program module | definition module | implementation module ;  
program module =  
    "MODULE", module identifier, [protection], semicolon,  
    import lists, module block, module identifier, period ;  
module identifier =  
    identifier ;  
protection =  
    left bracket, protection expression, right bracket ;  
protection expression =  
    constant expression ;  
definition module =  
    "DEFINITION", "MODULE", module identifier, semicolon,  
    import lists, definitions, "END", module identifier, period ;  
implementation module =  
    "IMPLEMENTATION", "MODULE", module identifier, [protection],  
    semicolon, import lists, module block, module identifier, period ;  
import lists =  
    {import list} ;  
import list =  
    simple import | unqualified import ;  
simple import =  
    "IMPORT", identifier list, semicolon ;  
unqualified import =  
    "FROM", module identifier, "IMPORT", identifier list, semicolon ;  
export list =  
    unqualified export | qualified export ;  
unqualified export =  
    "EXPORT", identifier list, semicolon ;  
qualified export =  
    "EXPORT", "QUALIFIED", identifier list, semicolon ;  
qualified identifier =  
    {module identifier, period}, identifier ;  
definitions =  
    {definition} ;  
definition =  
    "CONST", {constant declaration, semicolon} |
```

```

"TYPE", {type definition, semicolon} |
"VAR", {variable declaration, semicolon} |
procedure heading, semicolon ;
procedure heading =
    proper procedure heading | function procedure heading ;
type definition =
    type declaration | opaque type definition ;
opaque type definition =
    identifier ;
proper procedure heading =
    "PROCEDURE", procedure identifier, [formal parameters] ;
formal parameters =
    left parenthesis, [formal parameter list], right parenthesis ;
formal parameter list =
    formal parameter, {semicolon, formal parameter} ;
function procedure heading =
    "PROCEDURE", procedure identifier, formal parameters,
    colon, function result type ;
function result type =
    type identifier ;
formal parameter =
    value parameter specification | variable parameter specification ;
value parameter specification =
    identifier list, colon, formal type ;
variable parameter specification =
    "VAR", identifier list, colon, formal type ;
declarations =
    {declaration} ;
declaration =
    "CONST", {constant declaration, semicolon} |
    "TYPE", {type declaration, semicolon} |
    "VAR", {variable declaration, semicolon} |
    procedure declaration, semicolon |
    local module declaration, semicolon ;
constant declaration =
    identifier, equals, constant expression ;
type declaration =
    identifier, equals, type denoter ;
variable declaration =
    variable identifier list, colon, type denoter ;
variable identifier list =
    identifier, [ machine address], {comma, identifier,
    [machine address] } ;

```

machine address =
left bracket, value of address type, right bracket ;

value of address type =
constant expression ;

procedure declaration =
proper procedure declaration | function procedure declaration ;

proper procedure declaration =
proper procedure heading, semicolon, (proper procedure block,
procedure identifier | "FORWARD") ;

procedure identifier =
identifier ;

function procedure declaration =
function procedure heading, semicolon, (function procedure block,
procedure identifier | "FORWARD") ;

local module declaration =
"MODULE", module identifier, [protection], semicolon,
import lists, [export list], module block, module identifier ;

type denoter =
type identifier | new type ;

ordinal type denoter =
ordinal type identifier | new ordinal type ;

type identifier =
qualified identifier ;

ordinal type identifier =
type identifier ;

new type =
new ordinal type | set type | packedset type | pointer type |
procedure type | array type | record type ;

new ordinal type =
enumeration type | subrange type ;

enumeration type =
left parenthesis, identifier list, right parenthesis ;

identifier list =
identifier, {comma, identifier} ;

subrange type =
[range type], left bracket, constant expression, ellipsis,
constant expression, right bracket ;

range type =
ordinal type identifier ;

set type =
"SET", "OF", base type ;

base type =
ordinal type denoter ;


```

packedset type =
    "PACKEDSET", "OF", base type ;
pointer type =
    "POINTER", "TO", bound type ;
bound type =
    type denoter ;
procedure type =
    proper procedure type | function procedure type ;
proper procedure type =
    "PROCEDURE", [left parenthesis, [formal parameter type list],
        right parenthesis] ;
function procedure type =
    "PROCEDURE", left parenthesis, [formal parameter type list],
        right parenthesis, colon, function result type ;
formal parameter type list =
    formal parameter type, {comma, formal parameter type} ;
formal parameter type =
    variable formal type | value formal type ;
variable formal type =
    "VAR", formal type ;
value formal type =
    formal type ;
formal type =
    type identifier | open array formal type ;
open array formal type =
    "ARRAY", "OF", {"ARRAY", "OF"}, type identifier ;
array type =
    "ARRAY", index type, {comma, index type}, "OF", component type ;
index type =
    ordinal type denoter ;
component type =
    type denoter ;
record type =
    "RECORD", field list, "END" ;
field list =
    fields, {semicolon, fields} ;
fields =
    [fixed fields | variant fields] ;
fixed fields =
    identifier list, colon, field type ;
field type =
    type denoter ;
variant fields =

```

```

    "CASE", [tag identifier], colon, tag type, "OF",
    variant list, "END" ;
tag identifier =
    identifier ;
tag type =
    ordinal type identifier ;
variant list =
    variant, {case separator, variant}, [variant else part] ;
variant else part =
    "ELSE", field list ;
variant =
    [variant label list, colon, field list] ;
variant label list =
    variant label, {comma, variant label} ;
variant label =
    constant expression, [ellipsis, constant expression] ;
proper procedure block =
    declarations, [procedure body], "END" ;
procedure body =
    "BEGIN", block body ;
function procedure block =
    declarations, function body, "END" ;
function body =
    "BEGIN", block body ;
module block =
    declarations, [module body], "END" ;
module body =
    initialization body, [finalization body], ;
initialization body =
    "BEGIN", block body ;
finalization body =
    "FINALLY", block body ;
block body =
    normal part, ["EXCEPT", exceptional part] ;
normal part =
    statement sequence ;
exceptional part =
    statement sequence ;
statement =
    empty statement | assignment statement | procedure call |
    return statement | retry statement | with statement |
    if statement | case statement | while statement |
    repeat statement | loop statement | exit statement | for statement ;

```

```

statement sequence =
    statement, {semicolon, statement} ;
empty statement =
    ;
assignment statement =
    variable designator, assignment operator, expression ;
procedure call =
    procedure designator, [actual parameters] ;
procedure designator =
    value designator ;
return statement =
    simple return statement | function return statement ;
simple return statement =
    "RETURN" ;
function return statement =
    "RETURN", expression ;
retry statement =
    "RETRY" ;
with statement =
    "WITH", record designator, "DO", statement sequence, "END" ;
record designator =
    variable designator | value designator ;
if statement =
    guarded statements, [if else part], "END" ;
guarded statements =
    "IF", boolean expression, "THEN", statement sequence,
    {"ELSIF", boolean expression, "THEN", statement sequence} ;
if else part =
    "ELSE", statement sequence ;
boolean expression =
    expression ;
case statement =
    "CASE", case selector, "OF", case list, "END" ;
case selector =
    ordinal expression ;
case list =
    case alternative, {case separator, case alternative},
    [case else part] ;
case else part =
    "ELSE", statement sequence ;
case alternative =
    [case label list, colon, statement sequence] ;
case label list =

```

```

    case label, {comma, case label} ;
case label =
    constant expression, [ellipsis, constant expression] ;
while statement =
    "WHILE", boolean expression, "DO", statement sequence, "END" ;
repeat statement =
    "REPEAT", statement sequence, "UNTIL", boolean expression ;
loop statement =
    "LOOP", statement sequence, "END" ;
exit statement =
    "EXIT" ;
for statement =
    "FOR", control variable identifier, assignment operator,
    initial value, "TO", final value, ["BY", step size], "DO",
    statement sequence, "END" ;
control variable identifier =
    identifier ;
initial value =
    ordinal expression ;
final value =
    ordinal expression ;
step size =
    constant expression ;
variable designator =
    entire designator | indexed designator |
    selected designator | dereferenced designator ;
entire designator =
    qualified identifier ;
indexed designator =
    array variable designator, left bracket, index expression,
    {comma, index expression}, right bracket ;
array variable designator =
    variable designator ;
index expression =
    ordinal expression ;
selected designator =
    record variable designator, period, field identifier ;
record variable designator =
    variable designator ;
field identifier =
    identifier ;
dereferenced designator =
    pointer variable designator, dereferencing operator ;

```

```

pointer variable designator =
    variable designator ;

expression =
    simple expression, [relational operator, simple expression] ;

simple expression =
    [sign], term, {term operator, term} ;

term =
    factor, {factor operator, factor} ;

factor =
    left parenthesis, expression, right parenthesis |
    logical negation operator, factor |
    value designator | function call |
    value constructor | constant literal ;

ordinal expression =
    expression ;

relational operator =
    equals operator | inequality operator | less than operator |
    greater than operator | less than or equal operator |
    subset operator | greater than or equal operator |
    superset operator | set membership operator ;

term operator =
    plus operator | set union operator | minus operator |
    set difference operator | logical disjunction operator |
    string concatenate symbol ;

factor operator =
    multiplication operator | set intersection operator |
    division operator | symmetric set difference operator |
    rem operator | div operator | mod operator |
    logical conjunction operator ;

value designator =
    entire value | indexed value | selected value | dereferenced value ;

entire value =
    qualified identifier ;

indexed value =
    array value, left bracket, index expression,
    {comma, index expression}, right bracket ;

array value =
    value designator ;

selected value =
    record value, period, field identifier ;

record value =
    value designator ;

dereferenced value =

```

pointer value, dereferencing operator ;

pointer value =
value designator ;

function call =
function designator, actual parameters ;

function designator =
value designator ;

value constructor =
array constructor | record constructor | set constructor ;

array constructor =
array type identifier, array constructed value ;

array type identifier =
type identifier ;

array constructed value =
left brace, repeated structure component,
{comma, repeated structure component}, right brace ;

repeated structure component =
structure component, ["BY", repetition factor] ;

repetition factor =
constant expression ;

structure component =
expression | array constructed value |
record constructed value | set constructed value ;

record constructor =
record type identifier, record constructed value ;

record type identifier =
type identifier ;

record constructed value =
left brace, [structure component, {comma, structure component}],
right brace ;

set constructor =
set type identifier, set constructed value ;

set type identifier =
type identifier ;

set constructed value =
left brace, [member, {comma, member}], right brace ;

member =
interval | singleton ;

interval =
ordinal expression, ellipsis, ordinal expression ;

singleton =
ordinal expression ;

constant literal =

```
whole number literal | real literal |  
string literal | pointer literal ;  
constant expression =  
    expression ;  
actual parameters =  
    left parenthesis, [actual parameter list], right parenthesis ;  
actual parameter list =  
    actual parameter, {comma, actual parameter} ;  
actual parameter =  
    variable designator | expression | type parameter ;  
type parameter =  
    type identifier ;
```

[Contents](#)

1.1 What is Problem Solving?

Many books have been written in an effort to analyze and teach problem solving skills. Perhaps an even more fundamental question than the one posed in the title of this section is: *What is a problem?* Here are a few:

- one is at the bottom of Mt. Baker and wishes to be at the top
- one wishes to find the greatest common divisor of two numbers
- two children are listening to the Canucks hockey game but they are supposed to be sleeping
- a large mailing list needs to be sorted alphabetically so as to search for duplicates
- a wood-burning stove must be supplied with wood for the winter
- a teacher wishes to analyze a number of test scores and generate letter grades for report cards
- a grade six French student needs to memorize some verbs.

Just as the bare fact of a problem is a different thing to different people, so also is the process in view to solve them:

- To a high school algebra student, problem solving is the name for a formalized kind of game.
- To the owner of a business, it is finding the way to maximize customer satisfaction and profits.
- To a counsellor, it may involve restoring a client's self esteem and re-building broken relationships.
- In the information sciences domain, it is often thought of as the process whereby raw data is given meaning and thereby transformed into information or knowledge.

All these and many more situations that could be added to these lists have two things in common that make them "problems." These elements are:

- (a) a current state of affairs, and
- (b) a desired state or goal that is different from (a)

The problem solver seeks to use some process to transform the perceived current state (a) into the desired state (b.) In the narrowest sense, problem solving is the operation of some process by which the transformation of states is achieved. In a broader sense, problem solving encompasses both discovering the problem and the finding of an appropriate process, and not just the execution of the solving strategy after it is found. That is;

Problem solving is the description of current states and desired goals, and the finding and using of means to achieve those goals.

1.2 Problem Solving and Computing

Although many general things can be said about problem solving, the focus of this book is on the programming of computers so as to employ them as tools in the process. Of course, as they are manufactured, computing machines have only very limited capabilities. Computers do not pose problems, analyse them, or find solutions. They only run the programs that embody the solutions discovered by some human programmer, and for this, the hardware alone is insufficient. The addition of a programming notation (or language) such as Modula-2 produces a system capable of using the programmer's instructions to solve problems. All the intelligent planning to take use of this system once it has been assembled is entirely a human responsibility.

Computers can do only a few simple things, but they do them very quickly, so it is possible to employ them for rather sophisticated tasks. The programmer must set the task for the computer to perform with each step carefully detailed in some such notation--the complete sequence of which steps is termed a program. All the work this requires is worthwhile because the machine can execute the steps so quickly that one is still ahead of the game--despite the many hours it may take to plan and write down the necessary instructions.

It cannot be over-emphasized that the key to success in such an endeavour is careful planning. The problem to be solved or the task to be performed must be completely analyzed and fully understood by the programmer before any instructions to the machine can be written. What is the best method for doing this? There are two strategies for approaching the solution of a problem, however complex it may be. The first is to focus on the details, tackle each one separately, then gradually weld the partial answers into a coherent whole that provides a solution to the whole problem. This is known as a "bottom up" design, and there are times when variations of this method can be quite useful.

The second is to take an overview of problem as a whole, systematically break it into its parts, with a well-defined notion at each stage as to how the solution to the parts will fit into the whole. The actual solution is built by erecting the foundation and framework first and filling in the details later. From the general overview, one sketches what has to be done in broad terms, then refines the solution a little at a time until it is sufficiently detailed to write the steps in the chosen programming notation.

The process of solving a problem by breaking it down into successively smaller steps whose interrelationship is clearly defined and then solving each of the steps, is known as the top down method of design.

Modula-2 is an excellent language for this kind of structured approach to problem solving, for its facilities can be used to achieve this designed approach in a natural way.

For this technique to work, however, it is essential to clearly determine at the planning stage how the parts will fit together in the final program. The lines of communication between the tasks that each part of the solution is to accomplish must be clearly spelled out in advance. During the construction of the individual parts of a program, each one of them can be broken down in the same manner. Only when the

problem at hand (the whole project or a selected portion of it) has been reduced to a series of very detailed steps should the actual code-writing begin. At this point, there is (ideally) a one-to-one correspondence between the steps of the solution and the commands available in the programming notation being used. Rendering the solution in a correct program is then a simple matter. An even more sophisticated programmer will use the top-down method for general design purposes, but will occasionally set aside this work to tackle some small or more obvious part of the solution in bottom-up style. It is not easy to give definite rules for when this can be done. Professional programmers have considerably more independence to make such a decision for themselves than do apprentices, because there is more about the practice of problem solving that is obvious or intuitive to them. Most of the problems in this text will be solved in top-down style; the places where a combination top-down/bottom-up style is used will be noted.

[Contents](#)

1.3 Top Down Analysis

Problem solving itself presents us with a problem, for it is not immediately obvious what are the general techniques and strategies that can be applied to finding solutions. (This may be termed a *meta-problem*, that is, a problem to solve about problem solving.) It is clear that problem solving is a learned activity--one that human beings undertake first by imitating the methods of others, and only later by being innovative and creative. Furthermore, it is one thing to solve a problem (perhaps by lucky accident) and quite another to be able to generalize such solutions and apply variations of successful techniques to similar situations. That is, problem solving requires that its undertaker not just *do* something, but also think about the meaning of the problem, the desired goal, and the process of achieving that goal. There are a number of ways of elaborating this thought into a systematic set of steps, but all of them require the solver to do what is sometimes dull, plodding work that one **WADES** through systematically (**W**rite, **A**pprehend, **D**esign, **E**xecute, **S**crutinize). Consider each step in turn:

Write everything down. This applies to all stages of the process; it is not simply done at the end as an afterthought. Problems tend to come in related groups. The solution to one often applies to many others, or can be adapted to others. Thus, it is important that a record exist, not only of the result, but of the entire technique that brought about the solution. Otherwise, only the answer remains. Not even the original solver, much less any other person, can understand and apply a poorly documented solution some time later to another problem. The more complex and lengthy the process, the more important it is that a complete record is kept of the thought processes at each step.

Apprehend the problem

1. *Have a clear grasp of the existing state of affairs.* Enough must be known to give an adequate picture of the situation--not so much as to be overwhelming, and not so little as to be incomplete. The data must be sifted for sufficiency, correctness and relevance. If more data is needed, research must be done. If there is too much, only that which is germane is retained. You cannot calculate an area without measuring the dimensions or without knowing the formula. There is not much point in setting out to climb Mt. Baker from the foot of Mt. Robson, sorting the grocery list along with the mailing list, or studying French verbs from a Greek dictionary. A problem solver, like a physician, must understand the problem and must target causes rather than symptoms.

2. *Have a clear conception of the goal.* Definitions are important here. What is a "greatest common divisor?" What exactly is meant by an "alphabetic sort?" How many French verbs must be learned, and what must be known about them?

3. *Formulate the problem clearly.* Here the state-that-is, and the state-that-ought-to-be are compared in detail. Only the differences between the two states constitute the problem. If the question was something like "How much cheese did Italy export in 1906?" the answer may be obtained in the mere gathering of data, with no further work required. At this point, there is no problem, eh? On the other hand, if the question is to examine cheese exports for the last eight decades and formulate a five-year plan for the industry, most of the work still remains to be done after the data-gathering stage is complete.

Design a solution

1. *Consider related tasks.* Even a relative novice has some experience in problem solving, and has accumulated a repertoire or "library" of methods. It may be that the solution to a current problem can be built upon or expanded from that to a similar problem that has either already been solved, or is easier to do. Perhaps only a portion of the desired solution is available as a task already performed in solving another problem. It may be that the entire solution can be pieced together in parts from one's library of previous and related solutions.
2. *Break the main task into sub-tasks.* Complex problems are decomposable into a series of tasks, each simpler than the whole. Each of the sub-tasks span a portion of the gap between the problem and the desired solution. Climbing a mountain involves planning for each day's base camp, and establishing these one at a time before going on to the next one.
3. *Refine the sub-tasks into individual steps.* For each portion of the solution, employ the same technique as for the whole. There may be solutions available from the library of past experience for one of the steps, even if not for the whole solution. Individual steps may need to be broken down even finer before a technique becomes obvious. It is possible to tell that progress is being made toward a solution when the overall problem gives way to successive refinements into ever more detailed simpler steps. The solutions to the sub-tasks are developed into finished processes, tested individually wherever possible, and then combined as an organic whole. French is learned one verb at a time; fire wood is cut and bucked one tree at a time.
4. *Find a strategy to perform each (sub)task.* These may include guessing at a solution, drawing diagrams or pictures, looking for patterns, making tables or lists, employing direct, indirect, deductive or inductive logic, working backwards, breaking the problem into specific cases, or employing simulations or models. When suitable, it may also involve translating the problem into a mathematical or computing notation and using variables or formulas.

Execute the completed plan. Once the plan of action has been completely formulated, it has to be carried out. The mountain must actually be climbed, the French verbs memorized, the game played, the equation solved or the computer program run. This is the just do it stage.

Scrutinize the results.

1. *Is the answer correct and reasonable?* The answers obtained by the problem solving process now need to be considered carefully for reasonableness, and to see whether or not the desired goal has been reached. If the list is sorted backwards, or every student in the class is assigned a "Z-minus," or there is still one more slope to climb, then there are still aspects of the solution that need attention.
2. *Re-develop the solution until the desired goal is reached.* Since there is an element of trial-and-error to all problem solving, it is seldom the case that the first attempt at a solution is entirely satisfactory and free from error. This additional refinement by working through parts of the process above will therefore almost always be necessary.
3. *Look for a simpler way.* The first method found to a correct solution usually instructs the solver more in the process than in the problem. Insights gained by finding one way to obtain the desired result can often be applied to making the solution simpler and more straightforward. Perhaps there was an easier route up the mountain after all.
4. *Consider generalizations.* Where possible, the completed solution is best implemented as a general one to a whole class of similar problems. Even if this does not happen, the new solution always becomes

part of the solver's library, and its techniques are available for re-use on other problems at a later time. Once one mountain has been climbed, the whole process of climbing mountains becomes easier.

[Contents](#)right">ContentsContents

1.4 Tools for Problem Solving

Problem solving techniques are best illustrated with an example. In order not to confuse the general problem solving process with the specific methods of computer programming, this first illustration comes from the back yard.

Problem : Cut 10 cubic metres of fire wood.

Solution : (first pass)

Preliminaries

1. Compute the number and select the trees to fell.
2. Obtain tools and safety gear.

Initialization

3. Put on safety gear.
4. Prepare tools for use.

Execution

5. Cut down the selected trees.
6. Remove branches.
7. Buck the logs into short pieces.
8. Split the wood.
9. Pile the wood for drying.

Termination

10. Clean up the site.
11. Put away tools and safety gear.

This first cut at writing out the steps required to solve the problem is produced with the help of past experience, assistance from others, how-to manuals, and a little common sense. In this form, the solution is little more than an ordered list of steps toward the achievement of the goal. The would-be logger may find that the execution of some of these steps is now obvious and that they can be undertaken without any further thought. In such cases, one could say that parts of the solution are part of a personal library of previously used and familiar techniques. Others of the steps in the solution may be less obvious (to some people) and require further elaboration as follows:

2.1 From garage obtain tools:

gas, oil, chain saw, hex wrench, file, chain oil, maul

2.2 From cupboard obtain safety equipment:

boots, gloves, safety glasses

4.1 Sharpen chain saw:

select a tooth to sharpen

```

    repeat
        file the selected tooth
        select next tooth
    until all teeth are sharp
4.2 Tighten chain saw bar with hex wrench
4.3 If engine is two-cycle
    then mix oil and gas in correct proportions
    Fill fuel tank
    Fill oil chamber with chain oil
4.4 Inspect spark plug
    If spark plug is dirty
        then clean or replace spark plug
4.5 Inspect maul
    If maul is dull
        then file a new edge in maul
5. Start a count at zero trees cut
    While count is less than number of trees selected
        determine direction of fall
        cut main notch facing fall direction
        overcut higher back notch on opposite side
        yell "timber"
        duck
        add one to the count

```

Some, (or all) of the other steps could be refined in much the same manner. The degree of such refinement depends on

- the complexity of the problem
- the availability of tools
- the number of techniques that need no further elaboration but can be taken as "givens." Here, some of the givens are described by the words: compute, select, obtain, put on, file, tighten mix, fill, inspect, clean, replace, file (second type), determine, cut, overcut, yell, duck, remove, buck, split, pile, clean up, put away, start, add. Some of the latter are built-in to the logger (anyone can yell; most can add one to a number.) Others depend on experience--the words describe sub-tasks that are part of a person's (possible) library of skills.

As given here, each of these words is taken as an "atom" or indivisible action not requiring further elaboration. Each represents or abstracts a complex series of more elementary and detailed steps. If the instruction "fill" were not obvious enough for the person to know what to do next, it would have to be detailed in a further refinement. On the other hand, many people could not be bothered doing all this for themselves. They would lack both the tools and the expertise, so would simply call a professional to perform the whole job, obscuring all the detail written out above, and replacing it by the single abstraction: "pay logger to cut enough fire wood." This latter view of the task is very far removed from

the details of the task and its sub-tasks and could be termed a "high level abstraction." The one who arranges for a task to be done in this way has no interest in or knowledge of the details. Even the hired logger (an experienced professional) will pay little conscious attention to detailed planning, but will view the task as an organic whole (a single abstraction) and perform it as a comfortable routine that requires little thought.

At the other extreme of this abstraction vs. detail spectrum, one could try to program a robot to do the task. Some of the sub-tasks that seemed easy for a human being (inspect, yell, select) suddenly become very difficult. Not only would audio-visual sensors have to be built and programmed, but the robot would also have to be provided with a means to determine such things as:

- the natural and best falling directions
- when and which way to duck
- what to do when something goes horribly, terribly wrong

The task at hand requires judgement and the ability to abstract and perform instinctively (qualities of human intelligence.) For these reasons, the amount of detail involved in specifying the steps of the process for a robotic logging machine would increase by several orders of magnitude.

A good problem solution must balance off the elaboration of steps, the development of new techniques, and the employment of old familiar methods whose details are given no thought because they are part of a personal or hired experience library.

[Contents](#)

1.5 The Role of Abstraction

The concept of abstraction was introduced in the last section to aid in the discussion of planning the solution to problems. Abstractions are central to the discipline of computing, and indeed to scholarly studies as a whole. They also play an important role in daily life. The purpose of this section is to discuss the concept of abstraction in detail, define the term precisely, and outline its uses in a number of fields including computing science.

What is an abstraction?

The Western Judeo-Christian religious tradition holds that there is a God capable of holding in his thoughts all the details of the fine structure of the universe simultaneously. This knowledge, expressed as creative energy, brought the universe into being in the first place, and even now holds it together. Of course, not all moderns agree that God exists, much less is omniscient and all-powerful in this sense. At the same time, no one seriously believes it possible for any human being to achieve such universal awareness. Even mundane and ordinary objects (a chair, a tree, a cow, one human cell) are sufficiently complex to make such comprehension impossible. It has been centuries since a single human being could have even a passing acquaintance with all available scholarly knowledge. Today, it is not even possible for one person to learn the whole of any one discipline.

Fortunately, it is not necessary to have exhaustive knowledge about something in order to make appropriate use of it. Consider the automobile: One can enjoy riding in a car without knowing how to drive. It is not necessary to be able to build an automobile in order to drive one and not required that its builders be able to design one. The designers need not be able to produce the metals and plastics from which it is made. None of these must know how to refine the petroleum products required to run it. Road designers, builders, and mechanics are also specialists, and so are legislators, sales people, auto company executives, and parts manufacturers.

Each has different priorities for what must be known about an automobile. Each has an *essential* subset or extract of detail taken from all that is available to know about the subject. Each focuses only on the essentials to a particular role, and needs only a cursory acquaintance with details important to others. The same is true in science. There, it is clearly understood that no object can be comprehended in every detail down to the sub-atomic. The concentration on essentials and the exclusion of details makes scientific understanding manageable, and even possible. Such a technique gives a researcher an intellectual handle on the subject that would be impossible if knowing everything were deemed to be the only adequate kind of knowledge. It is therefore possible to conceive of things by knowing an appropriate and sufficient subset of their properties. In the light of these examples, it is possible to offer the following definition:

Abstraction is the process of organizing or digesting details in order to concentrate upon or grasp essential patterns or to see the big picture.

One aspect of abstraction is deciding which are the sufficient essential properties to the task at hand, and

which are details that can be ignored. This very much depends on the community within which the one doing the abstraction works, for to be useful, an abstraction must not only be meaningful, that meaning must also be communicated to others in the field. (If only one person understands something, and cannot transmit its essence to another's understanding, the abstraction is useless.) Therefore, the kinds of abstractions that come to be widely accepted depend on the level of knowledge and education of the community for which they are intended. For instance, a solar-system model for explaining atomic structure is sufficient for those who are not equipped to grasp the finer points of probability and quantum mechanics, but quite inadequate for researchers at the frontier of knowledge in the field. Likewise, there are a variety of models for explaining the workings of a modern economy, and these vary in complexity and usefulness depending on whose understanding is being addressed. The needs of most citizens are quite different from those of a politician trying to make a decision, or of a professional economist trying to provide the information for the decision.

Other Abstractions

This process of attempting to grasp a myriad of detail by the abstraction of broad outlines or essentials is not confined to the sciences or even to the academic disciplines that attempt to use the scientific method. Numerous examples are possible from all fields:

- A chart or graph is an abstraction of data or relationships into pictorial form, in order to allow them to be visualized, and therefore understood comprehensively.
- Words and numbers are symbolic abstractions of specific ideas.
- Whenever someone learns a skill or a trade, the necessary activities and actions are abstracted from all the possible knowledge. The essence of this kind of abstraction is that such skills become automatic, so that they can be exercised without thinking about the details at all.
- The manufacturing/wholesaling/retailing chain is an organizational abstraction that allows people to obtain goods without making them.
- Job specialization is an abstraction process that frees people from excess complication, allowing them to concentrate on a few useful skills. They can then deal with most of the necessities of life through other specialists in a similarly abstract manner.
- Money (whether expressed as precious metal, coin, paper, cheque, or electronically) is an abstraction for the wealth of nations, corporations, and individuals.
- A representative democratic state is an abstraction that allows individual input into the governing process without having to consider every detail of every person's stand on every issue.
- Any understanding of God is an abstraction for a being who is too complex ever to know entirely (except by himself.)

Thus, far from being the province of academics alone, abstraction is a process fundamental to all human activity. The totality of the abstractions people use is an important measure of the complexity of their society. The most sophisticated abstractions are those that allow people to perform complex tasks without much thought. Most industrial machines (and even bicycles) have to be operated abstractly--at a level of unconscious skill, for so long as the details must still be thought about, the task cannot be performed efficiently, if at all. (If you stop to think about what you are doing, you are likely to fall off your bicycle.)

While one could criticize the process of abstraction over many levels as removing people from "real" understanding, it is precisely such distancing that gives abstractions their power. It is not necessary to understand how cheese is made in order to enjoy it. Indeed, abstractions are the most useful when they are far removed from the thing being abstracted, when they have been refined to the point that the people who need them can usefully employ them in an automatic fashion.

Other Names for the Process

So important and pervasive is the process of abstraction that it has a variety of specialized names arising from different disciplines and from the terminology adopted by the various people who have considered various aspects of this activity.

A *digest* is a summary of that part of the material deemed by the one making the digest to be the most essential. It is an attempt to filter the data, removing the non-essential, redundant, or irrelevant. For instance, the data reported from experiments are nearly always digested from the entire set obtained; this is necessary for brevity and clarity. Digests are useful as quick reference cards for the operation of computer programs, or complex equipment.

A *model* is a representation of something in a more concrete or accessible form than the original. It may be also used of a scale model for some proposed project. The term conveys the idea of explaining or showing by means of an analogy to something else that is supposedly better understood. (I.e. for which there are believed to be adequate abstractions already) The term modelling may be used by scientists to describe parts of the process of theory formation. Data from the real world is modelled in some form that can be stored and manipulated in a computer.

Theory formation is an attempt to abstract into some simple statement the workings of the subject under study. This term tends to be less concrete than modelling, for a theory is an attempt to define rather than to model, though in practice the distinction is often a fine one.

A *paradigm* is also a way of looking at a subject by way of analogy or example. It too is a model, but this term tends to be used in a broader sense to describe abstractions of considerable importance or size (a collection of related abstractions.) One example is the evolutionary paradigm, within which are many models for origins. Another is that of the Marxist "class struggle," to which paradigm all Marxist theories of political science and economics must be bent.

A *meme* is a (perhaps indirectly perceived) transmittable idea that is the basis of a social movement or a political philosophy. Its spread through a population can be studied in a manner similar to that of an infection, because it is the nature of a meme to induce the desire to proselytize. A meme can be benevolent (e.g., the ideals of democracy), fatal to its holders (e.g. belief in ritual suicide) or fatal to others (e.g., Naziism and Stalinism).

A *world view* is a complete set of philosophic or religious presuppositions within which paradigms and individual abstractions are formed. It constitutes the total way in which a person does abstractions (thinks) about the real world, and finds its expression within the various communities of which the person is a member. It encompasses the complete set of memes that a person possesses and spreads. One may speak, for example, of a scientific world view, of a Christian one, of a liberal one, or of an American one. Within each of these there exist numerous specific views of parts of the world. World views act on the individual both consciously and unconsciously. For example, a computer scientist is influenced in the choice of problems to work on, the means of seeking solutions, and the ethics of presenting those solutions to employers and the marketplace by some world view. Indeed, all of his or her professional

actions are conducted within and must be evaluated in the context of a world view. The mention of an abstraction term, such as the title of a theory or the name of a world view system, evokes in the hearer a vision of a some set of beliefs, views, or typical activities. That evoked image will invariably be to some degree inadequate or incorrect, especially if the hearer is not a part of the community that devised or is described by the abstraction. When such a misconception takes place, it is often because the hearer holds to some popularly believed ideas about the group in question, in which case the hearer's own (mistaken) abstraction is called a stereotype. Plants, and animals, do not make abstractions; this is a uniquely human activity. Neither do computing machines originate abstractions, they are simply tools for the systematic examination or expression of human ones. Abstractions make thinking and communicating possible and allow people to understand and use the world and its processes, whether by science, computing machines, or otherwise. They make it possible to make, to build, to specialize and to cooperate. They are therefore the essential building blocks, not of science alone, but of human civilization itself. This section concludes with an attempt to abstract itself:

Abstractions are never the "real" thing, and therein lies both their power and their usefulness.

Abstractions are intellectual frameworks; they are not discoveries.

It is important to keep in mind, however, that useful abstractions must retain their essential connectedness to the real world. There may be many mental steps removed, but those steps should be clear and easily repeated or explained otherwise the abstraction loses touch with reality and becomes useless.

1.5.1 Abstractions and Computing

Computers are tools for problem solving that enable people to hide the detail of data manipulation and calculations with a variety of abstractions. For instance, a word processor embodies a way of thinking of and working with documents that is quite different that if they were expressed only on paper. The graphical interface found on modern computers allows the user to perform very complicated tasks with a minimum of effort (at a greater degree of abstraction) by comparison with the verbal interface found on more old-fashioned machines. One need not know how to make or even to program a computer in order to make productive use of it for such tasks as word processing in a graphical context. However, this text is about programming computers, and it is useful to observe that:

- A computer program is an abstraction of a problem into a specially devised symbolic language (notation) for the purpose of solving the problem.
- A language (computing notation) is an abstraction designed for the purpose of writing and communicating other abstractions (programs.) It hides the details of the low-level machine language from the programs. It could be termed a meta-abstraction.

It is also important to note that within the discipline of programming itself, the important abstractions

(about which there will be much more to say) can be grouped into two categories:

1. *Data abstractions*: These are the forms in which entities from the "real world" are represented in some computational notation. There are:

- a. data representation abstractions (data structures) and
- b. data manipulation abstractions (expression structures)

2. *Machine abstractions*: These are the ways in which the construction and operation of the computational process are represented. These pertain to:

- a. the computing apparatus itself (machine structures) and
- b. the instruction and manipulation of the machine (program structures.)

Of the four structure categories, this book is concerned with all but the computing apparatus itself, which category will receive only the brief treatment in this chapter. The other three (data, expression, and program structures) make up the main subject matter of any course in the art and science of writing programs. They will also be introduced in the remaining sections of this chapter, and will be specifically referred to in the programming goals for each subsequent chapter. Note that, depending on the depth of abstraction, these four may variously refer to single data items, simple expressions and instructions, entire data collections, complex formulas, whole programs, or something between.

[Contents](#)

1.6 Data Representation Abstractions

A great deal of human activity centres on the collection and processing of data. Sight, smell, touch, taste, and hearing provide personal data input for most of us. We also collect data about the stock market, the economy, budgets, financial institutions, population statistics, chromatography experiments, seismological surveys, the weather, baseball and hockey teams, library loans, and a host of other events and activities. Any such data collection is likely to contain too many individual facts to be comprehended, as a whole--it must be processed in some fashion to become useful.

1.6.1 Data and Information

Raw data, as it is initially collected, is of little value or use. The human brain organizes audio/visual (and other) inputs and the mind interprets these inputs and assigns meaning to them. Thus a pattern of vibrations in the air is interpreted as conversation or as music, and a pattern of retinal impulses is a rose or perhaps a lover. One sees and hears, but there is much more to this than just a few organized sensory inputs, for intelligent organization is required to give meaning to the stimuli. Likewise, economic and scientific data consists only of raw numbers (symbols) until it has been organized and interpreted. Once such higher levels of meaning have been attached to data, it is termed information. This intelligent act of attaching meaning to raw data is clearly an abstraction process. Indeed, assigning meaning may to some extent be thought of as a synonym for the whole abstraction forming activity.

At a somewhat more concrete level, it is often possible to automate certain repetitious calculating tasks that are part of the process of placing meaning on data collections. These tasks are ideally suited to modern computing machinery and are driven by sets of instructions that achieve the mechanical aspects of the data organization. One can go farther than this and say that certain standard meanings are collected and tabulated, then automatically assigned to the items in the data collection by the computer program. The latter is then just a fast and reliable extension of human intelligence bent to the abstraction task

1.6.2 Encoding (Representing) Data

However, there are practical issues to solve at a lower level than describing what data processing is. These centre on how data is communicated. Whenever one writes a symbol like "4" or 'four," a potential for communication exists, based on the fact that these symbols encode a certain idea. By convention, everyone encodes the same idea with these symbols, so communication is possible.

Computation devices must also encode data in some consistent way. There are two categories of such codes:

1. *External codes.* These are usually human-readable characters that are input into a computer or output from it using a keyboard, screen, printer, or other device. The most common form for this data is a character such as "4," "a," "%," and so on.

2. *Internal codes.* Because computer storage is based on electronic circuits that can be thought of as ON/OFF switches, internal storage is in a different format than that used for human interface with the machine. In this form, it is not directly accessible by a human user. The person using a computing machine does not usually need to know what kind of internal representation is employed for data, because there are input/output programs that translate the data between internal and external formats. There are some issues relating to data representation that do make a difference to programmers, however.

1.6.3 Atomic and Compound Entities

One of these issues is the level at which a data abstraction operates. Consider the symbol 4.25×10^{15} for example. On one level, this symbol can be thought of as having nine parts or components, one for each character used. On a second (higher) level, it consists of a mantissa (4.25) and an exponent (15) with the rest just punctuation. On a third level, one could take the symbol as an organic whole--a representation of a real number. This is what is usually done. Symbols like 27, -4.92, 16.3×10^{12} , and so on are normally thought of as atomic (indivisible) entities, and not in terms of the individual digits, signs, and decimal points.

There is data that is not like this. Consider

a set $A = \{1, 6, 9, 15\}$

an ordered pair $P = (2, 3)$

or a complex number $z = 4 - 6i$.

Each of these is not only an entity in itself, but has also component parts that are sufficiently important to be referred to in their own right. One may use the individual elements of a set, the coordinates in an n-tuple, the parts of a complex number, and so on, in particular computations. In such cases, the structural parts of the data item become important in themselves. On the other hand, one may also refer to the entity as a whole, without making use of the knowledge of its constituent parts.

Similar situations arise when a data item has an assortment of related components of various kinds.

For instance, we might think of a student record as an aggregate (or a collective) consisting of:

- name
- address
- phone number
- marks
- fees owed

Each constituent part of such a student record is a data item in itself, and so is the aggregate as a whole. Depending on what one is doing at the time (revising the marks, or storing the whole record) one might handle the data as an aggregate (a high level of abstraction) or by constituent parts (a low level of abstraction.) Indeed, in this case, a still higher abstraction may be employed if one wishes to deal with several such students as a group. Such a group could in turn be given a structure in a variety of ways:

- a number could be painted on the back of each
- they could be organized alphabetically by their names

- they could be seated in rows and columns in a classroom

Data items that are normally handled as indivisible whole items are called atomic or unstructured and those that have component parts are called structured or compound.

1.6.4 The Concept of Type

It should be clear that data items like 4, 6, or 12 are of a different kind than those like "A," "Henry", or {5,9,16}. It is also the case, though perhaps not quite so obvious that 4,6,12 are also of a different kind than, say, -3 or 7.6.

The classification of data items into different kinds is done for one or more of three reasons:

1. the underlying concepts being abstracted by the symbols are different.
2. different kinds of data are stored by computing devices in different ways.
3. different operations may be defined on different kinds of data

These considerations are more evident in the case of atomic data on the one hand, and highly structured data on the other. However, mathematicians (and others) find it useful to distinguish among:

a. *cardinals* (also called unsigned whole numbers) 0,1,2,3,.....

(The strict mathematical convention is to use the term whole number only for what are called here unsigned whole numbers and not at all for the ones that might have negative values; the terminology here is looser, and conforms to the ISO standard vocabulary.)

b. *integers* (also called signed whole numbers).....-3,-2,-1, 0, 1,2,3,.....

c. *reals* numbers like 4.7, -3.98, 2.5×10^7 , -6.3×10^{-5}

d. *characters* "A", "z", "2", " (whatever can be typed)

e. *strings* "Hi There" "Fred"

f. *boolean* false, true

g. various structures or collections of these basic kinds.

Note that one distinguishes a "4" (character) from a 4 (unsigned or signed whole number). One may often wish to distinguish a 4 from a 4.0 (real), or even between a 4 taken from the unsigned whole numbers and a 4 taken from the signed whole numbers. It may also be necessary to handle the character "4" differently from the one-character string "4".

Data that is of the numeric kinds (unsigned or signed whole number, and real) has certain operations defined for it. Indeed, without the operations of addition, subtraction, multiplication, division, and comparison, the numeric kinds would be incomplete. Anyone who uses such data assumes that the abstraction of numbers includes the ability to do such things.

On the other hand, these four operations mean little on, say, the character and boolean kinds. We could imagine other operations such as capitalize ("a") which produces "A" or possibly NOT (True) which produces False.

If the data is structured instead of atomic, the level at which an operation is applied on an item is also an important consideration. A group of students might have:

- operations on the whole group
 - assign the group to a different room
- operations on individuals as part of the group
 - add student to the group delete student from group re-position student within the group
- operations on the particular data associated with an individual
 - change the last name of a student enter a mark for a student update the fees owing for a student

Thus, on any given level of the abstraction scale, the concept of the kind or type of data is incomplete without either implicitly or explicitly including the operations associated with the data at that level. Such considerations lead to the following definition:

A specified set of items with certain properties and operations in common is called an abstract data type, or ADT for short.

Some ADT's are easy to represent in a computer. BOOLEAN has only two possible values (True, False). These could be internally represented by, say, a zero and a one, respectively.

However, the numeric ADTs present one with various representation difficulties. As one conceives of them in mathematical terms, both unsigned whole numbers and signed whole numbers have an infinite number of possible values. To allow for that, the computer's storage would have to be unlimited. Since this is not possible, every actual system restricts these two types to some specific range of values. On smaller machines, this is often -32768.. 32767 for signed whole number and 0.. 65535 for unsigned whole number. On larger ones, the usable range could reach the millions or billions--but there is always a specific limit beyond which one cannot represent for the machine what are perfectly ordinary numbers. Users of a given system must simply know what this limit is, and be prepared to live within it.

The Real ADT presents another problem. Not only must there be maximum and minimum limits as for whole number ADTs, but there also restrictions on the available precision. No computer can make provision for storing an indefinite number of significant figures (decimal places); there is always some limit. If this limit is, say, eight figures, then 0.0000000001 cannot be distinguished from 0.0, nor can the completeness property of reals always be followed. (That is, that between any two reals there is another real.)

In the case of a data structure, different kinds of limitations may arise, for large numbers of similarly structured data items may not all fit in the memory of a computer at once, so a means must be found to store these on some external medium. It is then also necessary to have a method of reading from this external storage, altering some or all of the data, and writing it back again. Of course, if the collection becomes sufficiently large, even the external storage may be too small; it is then time to buy more disk or tape drives.

For such reasons, actual instances of ADTs on specific computing systems are always approximations to the intellectual concept one is trying to represent. This situation does not differ appreciably from the "real world" wherein measuring instruments also have limited capacity and/or precision. The users of any machine (whether a computer or a tape measure) must live with its limitations.

An instance of an ADT, complete with specified limits, is called an implementation of the

ADT. One recognizes that such limitations vary from one computing platform to another by terming them implementation dependent.

It is also possible to distinguish three levels at which one could say that data exists:

- as a 'pure' abstraction or idea
- as a virtual implementation in some symbolic form (English description, mathematical or computing notation)
- as an actual or physical implementation within a computing machine. (expressed, say, as voltages applied to certain electrical circuits)

1.6.5 Variables and Constants

In performing computations, it is often convenient to refer to data or to a portion of its structure by a named place holder. In algebraic terms, such names abstractly represent some value in a calculation. One might write:

$x = 5$

$y = 6$

$z = x + y$

The first lines say to assign the name x to the value 5 and the name y to the value 6. The third line says to give the name z to the value obtained by adding the values represented by x and y .

Such named quantities fall into two broad categories--those understood to have a fixed value throughout the computation, and those whose value is subject to change during the course of the calculation. In writing: a formula like $area = \pi * radius^2$ for instance, it is understood that the value of π is fixed (approximately 3.14) whereas the values of the $radius$ and $area$ are subject to change depending on the particular instance of the problem. (See section 1.6)

Moreover, the type associated with the numbers when written literally (5, 6, 3.14) also attaches to the names used for them, so we say that these names too have a type. These considerations produce the following definitions:

A constant is a name for a value that is understood to be fixed.

A variable is a name whose value is subject to change during the course of a computation.

The type of a constant or a variable is the same as that of the abstract data type of which it names a particular instance.

This last definition may sound a little technical, but all it means in practice is that if the name π is given the value 3.14159, then π is of type real because its value is of type real.

Because such names are intended to abstract values, it is useful to employ descriptive names, so as to mirror function as part of doing the abstraction. Thus, one commonly employs names such as: *interest*, *rate*, *time*, *speed*, *distance*, *area*, *perimeter*, and so on, rather than using a , x , i , P , s , a , b , and the like.

Contents

1.7 Data Manipulation Abstractions (Expressions)

Along with the concept of an ADT (data and operations) comes the idea of writing down combinations of data items using some notation for the operation. The notational abstractions used for such operations as, say, addition, depend somewhat on those used to represent the data. For instance the operation *eleven plus twenty* could be represented as:

(|||||) (|||||)

This is what children do when they are learning how to make number abstractions and they count with sticks, marbles, pictures of teddy bears or jars of jelly beans.

At a higher level of abstraction, one combines several symbols into one, and could write:

XI + XX,

a notation that, despite its inconveniences, served Europe for many centuries.

Later, the concept of place value, and the idea of using zero as a place holder became accepted. With the adoption of this Arabic system, the modern numeric representation came into being, and along with it came streamlined rules for performing operations--all of which transfers more or less directly into most computing notations. Thus, we write

3 + 5 + 16 (addition)

7 - 2 (subtraction)

-4 * 6 (multiplication)

3.0 / 7.0 (division)

for various numeric data types. Note the use of * and / for multiplication and division, respectively. This practice is all but universal, for most keyboards lack the usual mathematical symbols for these operations.

A combination of data items with various operators that are available for that data type is called an expression.

Performing the operations and extracting a single numeric result is called evaluating the expression.

Thus the evaluation of 3 + 5 + 16 produces 24, of -4 * 6 produces -24, and so on.

Of course, the concept of type attaches to the result of evaluating the expression, as well as to the individual items that make it up. Thus, all the expressions above are evidently of numeric types.

Now consider expressions such as:

15 < 2.0, or

-3 = 7

These contain numeric data connected by comparisons rather than arithmetic operations. They produce the values *True*, *False*, and *False*, respectively, and are classified as Boolean expressions. We say:

The type of an expression is the same as the type of the data produced when the expression is evaluated.

Naturally, this applies to expressions with named symbols for constants or variables as well. One often writes formulas, with a variable on the left hand side, and an expression on the right hand side to mean that the expression is to be evaluated and the result (with its type) will have the name given on the left. Examples include:

interest = principal * rate * time

distance = speed * time

The intent in using such formulas is that they stand abstractly for a whole class of possible computations, the actual numeric details for which can be filled in later. Naturally, such facilities are available in most computing notations as well. In such cases, the name also represents a memory location, and one can think of the value as being deposited in that named location for later reference.

Like algebraic expressions, Boolean ones may also be represented by names, and combined to form more complex expressions. For instance, if p and q are Boolean expressions, then:

$\text{not } p$ is false whenever p is true, and vice versa,

$p \text{ and } q$ is true whenever both p and q are true, and

$p \text{ or } q$ is true whenever either of p or q (or perhaps both) is true.

In forming boolean expressions, "and," "or," and "not" are called connectives.

1.7.1 Precedence

Some expressions are ambiguous unless rules are adopted to make their meaning clear. Thus

$3 + 4 * 5$

could produce 35 or 23, depending on whether the addition or the multiplication is performed first. To ensure that such problems do not arise, mathematicians adopt a "convention" or set of rules to evaluate otherwise ambiguous expressions.

By this convention, multiplication and division are performed before addition and subtraction, but parentheses can modify this order. Otherwise, evaluation is done left to right. That is, the correct evaluation of the expression above produces 23. Such rules are not followed by many calculators, which evaluate expressions as one enters them. However, computers are more expensive than calculators, and one can reasonably expect that their programming notations can handle the mathematically correct order of operations.

Here are a number of evaluations, with the results shown at right:

$x = 2 + 6 / 3$	$4 ==> x$
$x = 3 - 6 * (7 + 3)$	$-57 ==> x$
$x = 3 - 4 + 6 * 7$	$41 ==> x$

When expressions contain booleans, numeric comparisons have the lowest priority, the *or* connective has the precedence of addition, the *and* connective has the precedence of multiplication, and the *not* has a

higher precedence than either. Again, parentheses can modify this order.

```
2 + 3 <= 5           true
(1 < 2) and (-4 < 7)   true
(2 >= 5) or (8 < 6)    false
not (1 = 1)           false
(4 < 1 and 3 < 4       cannot evaluate. 1 and 3 makes no sense.
```

1.7.2 Expression compatibility

As the last example illustrates, it makes little sense to mix data items of different types in the same expression. (What operator could be used? What type would result?) Thus $4.0 < \text{True}$ or $3 - \text{False}$ are rather obvious errors.

On the other hand, the numeric operations are defined (with essentially the same meaning) for several types. For example, one can write whole number or real addition expressions. This leads to some interesting difficulties when writing mixed expressions (containing more than one type.)

An expression like $-2 + 5$ can be evaluated mathematically to 3 without giving any thought to such issues, but in a computing machine things are not so simple. Some notations take a very strict view of mixed expressions. Because there is one signed whole number in the expression, they would assume that the 5 be taken from the signed whole number type rather than the unsigned whole number type. The result would be of the signed whole number type. On the other hand, if there is no context it is impossible to tell whether the underlying nature of symbols like 5 arise indeed signed or unsigned--they can be written in either type of expression.

When two ADTs share a common range and operation and instances of their symbols can be used together in a single expression, they are called expression compatible (over the common range). Otherwise, they are expression incompatible.

Expressions like $4.5 + 3$ are also easy to handle abstractly. This evaluates to 7.5 and so is a real expression. The fact that the 3 is converted into 3.0 (from an unsigned whole number or signed whole number to a real) is often ignored--outside the computer. Within the computing environment however, this conversion cannot be ignored, for the two data types may well be stored in very different ways and therefore be expression incompatible. In some notations, this conversion is performed automatically, making these types (at least appear to be) expression compatible. In others, the user is responsible to do conversions when data is not expression compatible. This particular conversion is called "floating" (for converting to floating point) and the expression may be written as $4.5 + \text{float}(3)$.

Similar explicit conversions may be required if it is known that data is of one type and the result is of another. For instance:

card (-3 - -7)

might be used to produce an unsigned whole number result, and

int (10 - 5)

might indicate this result is of signed whole number type.

It is also worth observing that although certain operators (+, -, *, /) work on several types, they mean slightly different things for each. (Internally to the machine, they could mean very different things.)

An operator that is defined for expressions of more than one type is said to be overloaded.

[Contents](#)

1.8 Abstractions for Computing Machines

1.8.1 Computer Hardware Organization

Let's take a little closer look at the way computing machines operate, so that some of the terms taken for granted throughout this book will not be confusing.

First, a definition of the "nuts and bolts".

The term hardware refers to the physical components, including the electronics, which make up the computer itself. That is, this word distinguishes the machine from the instructions it executes.

Every computer must have hardware that allows it to achieve the following tasks:

1. *Input.* A computer must be able to accept information from the outside world.

Early computers utilized punched paper cards or tape as their primary means of obtaining data. Later, magnetic media employed reels of tape, soft plastic "diskettes" of various sizes, or precision engineered metallic "hard" disks spinning at high speed. Other input devices include light pens, television cameras, "joysticks" for games, a mouse, page document reader, and even the human voice. However, while many people are predicting that one or more of these will replace the alphanumeric keyboard entirely over the next few years, the latter still remains the input device of choice for most people.

2. *Memory.* A computer must be able to store data.

One useful way of thinking about a computer is to view it as a collection of thousands of pigeon holes, much like those used in a post office to sort mail. Information can be stored in a specific location, and then it can later be retrieved for manipulation by the processor.

For practical purposes, there are two types of computer memory, each having its distinct purpose. One type contains the programs used to start up the computer, perhaps to test the rest of memory for flaws, and to begin the operation of the disk drive so as to obtain some larger program from there. Since this program must be available every time the machine is turned on, it is permanently coded at the time of manufacture. For this reason, such memory is called Read Only Memory, or ROM for short. A computer may have anywhere from a few hundred to tens of thousands of units of its available memory locations dedicated to the built-in ROM programs.

The programs coded into ROM at the time of manufacture are the firmware of the computer. This term is used to distinguish the programs from the ROM chips which contain them (hardware).

The majority of memory falls into the second category. This is read/write or "Random Access Memory" (RAM for short.) This is where user programs and data are stored. Turning off the power causes all the

information in RAM to vanish, so it is important to ensure that programs and data are also stored on an external device before this is done. In fact, careful management of resources dictates that a programmer working on a large project should store the work periodically as a precaution against a power failure that would wipe everything out.

3. *Processing.* A computer has a built-in ability to manipulate the stored data.

The chip (or collection of chips and circuits) that actually moves the data around in memory and manipulates it in other ways, is called the Central Processing Unit (CPU). It contains the circuitry to allow some simple arithmetic operations to be performed, to test the memory for the presence of certain results, and to perform a variety of other operations. Instructions to do these things are given as special numeric codes.

The limited capabilities represented by this "machine language" are not usually used directly by the person sitting in front of the computer console. Rather, other languages and programs are written in terms of the machine's language, and it is through these that the operator interacts with the machine.

If the memory are the post office pigeonholes, the processor can be thought of as the postmaster, moving items from box to box, sorting them, and taking collections of items from the boxes for further processing elsewhere.

4. *Control.* The various functions of the computer must be coordinated.

Some of the functions that control the input, processing, and output of the data also may be located in the CPU section (or chip). However, there are usually other specialized circuits for this purpose, and in some computers, these actually make up most of the electronic hardware.

Besides the routing of data to and from the correct memory locations and input or output devices, some of this circuitry is connected to certain hardware switch locations of the main memory, and interpret references to these by programs as signals to take action. (e.g. turn on the disk drive, invert the screen to white on black, etc.) This frees the CPU from much time-consuming and unnecessary activity, and allows it to be used more for processing than for control, speeding up the operation of the whole computer.

5. *Output.* A computer has ways to send information back to the outside world.

Many of the devices mentioned under the input section may simultaneously handle output. For this reason, the two functions are often grouped and referred to collectively as I/O. At one time, printers were the main devices dedicated strictly to output, but today a cathode-ray tube (such as is found in a television) is commonly used as a video display terminal for most purposes, with the printer being employed only at the last stages of a project when a final printed or "hard" copy of the results is desired. There have been many kinds of printers. Some printed hundreds of lines per minute, cost a small fortune, and were used only with mainframe installations. Others generated letter quality (typewriter style) output, but at only thirty characters per second (or less). Between these in speed, but at a much lower price were the dot matrix printers, which formed letters by making an array of dots on the page with pins, ink jets, or an electrical spark. Laser printers and other whole-sheet devices represent a newer technology that has moved rapidly from large networks to the individual small user. These are now available in colours and with large sheet sizes. A variety of other special purpose output devices also are available, particularly for graphics-oriented displays.

The specific technology employed to implement the physical machine changes rapidly, and need not be detailed further here. Hardware design, and even the assembly of pieces out of a box are not the concern of most people who use or program computers. Rather, they are interested in the way that the total

working environment *appears* to them as they use it.

The total environment presented to the user by the combination of hardware and software that is being employed at the moment is called the virtual machine.

Note that the virtual machine to someone employing a word processor or spreadsheet is quite different from that presented to someone who is programming the same computer. Each has a different abstraction for the computer, a different virtual machine. The same user has a different virtual machine at different times of use, depending on the software currently available for the task at hand.

1.8.2 Computer Software Organization

Thus, it is time to pay attention to the programs that actually run on a computer.

The programs resident in the RAM memory of a computer are collectively referred to as its software.

Software may refer to a purchased "canned" package used to operate an accounting or word processing system. It may also refer to a computer language together with some program written by the user in that language.

The software that handles the disk drives and other I/O, and generally provides the environment in which the programmer works, is referred to as the operating system of the machine.

If the task is actually writing programs, it is important to realize that a computer can take action upon only a limited vocabulary of instructions--usually fewer than one hundred words. However, once given the instructions to follow, the machine will do the set task so rapidly that the programmer saves time in the end, despite the work put into turning those instructions into code that the machine could follow. Here are a few basic definitions.

A set of instructions to a computer that is intended to make it perform a task is called a program. The person devising the program is called a programmer and the collection of all the instructions available to a programmer at a given time is called a programming notation or language.

Because the vocabulary of a computer is limited, a programmer must give the program instructions in a manner carefully chosen for clarity, accuracy, and efficiency. Much of the purpose of this book is to teach the rudiments of strategies for creating such programs. Its students will spend a great deal of time sitting in front of a computer typing in and running sample programs, for computer science, like mathematics (and other worthwhile things), can be learned only through the fingertips. That is, the "hands-on" aspect of programming is not an optional part of the course, it is the course. The book, the

professor and the lectures are teaching aides, but the theory they present is worthless unless it is used.

1.8.3 Computing Notations

Since programming *is* the central issue here, it is worthwhile to consider programming languages in general terms.

The central processing unit (CPU) of a computer can execute programs only through a limited number of instructions placed directly into the memory as numerical codes. These codes and their meanings are collectively referred to as the *machine language* for that particular processing system.

It is possible to write programs in machine language employing a text editor (like a word processor, but somewhat specialized) for the writing, and then use a program called an *assembler* to generate the actual code by translating the text file. However, most languages are not machine languages, but are written in terms of these low level codes. The commands in the higher level languages (such as Modula-2) are more like English words than the cryptic abbreviations used in assemblers, or the meaningless (to us) numbers that are the actual machine codes. Once a higher level program has been written out, the machine can translate this notation into the appropriate machine codes so that it can be executed on the processor. Of course, higher level languages have developed gradually, as has the hardware on which they run. At one time, only the low level machine codes were used, and entering these was a laborious process indeed. As they developed, these languages or notations became somewhat specialized, reflecting the biases of their creators and principal users.

Two Early High-Level Notations

The first high level language to gain common acceptance was FORTRAN (FORmula TRANslation) which was developed in the 1950s for numerical computations and scientific research. This language exists today in many versions, the most common of which is FORTRAN 77. The newest standard version FORTRAN 90 has just become available.

A second language from this early era in computing, which is still extensively used, is COBOL (Common Business Oriented Language). Again, there are many versions of this language, but they are all designed to make it easy to program the solution to business problems.

Though the language definitions themselves do not demand it, actual implementations of both FORTRAN and COBOL require that programs be written out in one set of codes as a text file (using the language vocabulary) and then must be translated by another program into the code that the machine itself can run.

An implementation of a language that is translated once from the programmed form to the machine version, and thereafter run from the machine version, is said to be a compiled implementation. The program which performs the translation task is called a compiler.

BASIC

BASIC (Beginner's All Purpose Symbolic Instruction Code) was developed as a teaching tool rather than as a major problem-solving notation. It also has many versions, some of them with almost as much power as the FORTRAN of which is it sometimes considered an abbreviation. BASIC is often called a

"quick-and-dirty" language, because it allows the programmer to write code and get fast results, using the computer as a kind of giant calculator. Unfortunately, it is very "loose" (the dirty part) and its users easily develop rather sloppy ways of thinking and working that are detrimental to the planning of large programs. The following definition happens to apply to most (but not all) versions of this language.

An implementation of a language which is translated from the written code into the machine code as the program is run, and which must be translated this way every time it is run, is said to be an interpreted implementation.

BASIC, in most of its incarnations, is not suited for large programming projects because its design does not enforce good programming habits. Also, because it is frequently implemented as an interpreted language rather than as a compiled one, its programs are rather slow.

Pascal and Modula-2

In the early 1970s, the Swiss computer scientist Niklaus Wirth devised a new teaching language which he called Pascal. Eventually, Pascal became the mainstream language among university computer science faculties around the world. Because it was designed for teaching, Pascal had many shortcomings for programming commercial applications, and there have come to be several enhanced versions. Perhaps the best known was the P-system version developed at the University of California at San Diego. Here, the product of the compiler was not machine code, but an intermediate called P-code, which itself had to be interpreted when the program was executed.

The advantage was that all that was needed to take the compiled program to another computer was the appropriate final stage interpreter for the target machine, because the P-Codes themselves were the same for all machines. Since much of the operating system (filer, editor, etc.) was also written in P-Code, the same virtual machine was presented to the operator or programmer, regardless of the type of hardware employed. Wirth, by the way, was also the one who devised the P-code; the UCSD version was merely the major commercial implementation of this, and provided what became the standard operating system to contain it.

There were other contenders for the title of "standard" Pascal, however. The International Organisation for Standards (ISO) produced a version in 1978, and later the American National Standards Institute (ANSI) also published similar but not quite identical standards for the language. These standard versions of Pascal were widely implemented on minicomputers and mainframes and were commonly used in educational institutions, though these often developed local dialects of their own.

Because of the wide experience with Pascal, and the many extensions of it which others created, Wirth decided to derive a new language of his own from this base. Apparently he believed that the many attempts to enhance Pascal confirmed the belief that it was fundamentally flawed, and that he should start again, rather than do a patch job of his own. He had already produced a language called Modula, whose principal feature was the "module." This allowed programs to be compartmentalized for easier design and error detection.

Borrowing much of the style from Pascal, and the module concept from Modula (which was never very widely used) Wirth developed in 1978 and published in 1980 his description of a new language which he called Modula-2. Besides the language, he devised a new intermediate code which he called M-code, and designed and built a computer (the Lilith, which was an optimized workstation for the language, and

whose native machine language was M-Code.) He also designed and distributed both M-Code and machine language compilers to implement the language.

It is important to note that Modula-2 is specifically designed for programming large complex systems, and many comments later in this text will serve to point students toward such tasks, though the examples will, of necessity, be at an elementary level.

Modula-2 gained wide acceptance in a short period of time, and wherever it was introduced it quickly replaced Pascal as a teaching language. It also came to be used for large production purposes in a way that Pascal never was. As a result, it too became the subject of a standardization process begun on the international level by ISO in 1987, and completed with the release of the final standard document in 1996. The results of that process are reflected in this revision of the text.

More recently, generic programming extensions and object oriented extensions have also been proposed by the ISO Modula-2 committee and these are expected to be adapted in late 1997 or early 1998.

Some Other Modern Languages

Two other languages which are generally compiled and have been used in universities for teaching purposes are PL/1 (Programming Language one) and "C". The former was a creation of IBM, and the latter was contributed by Bell labs. (Actually no predecessors "A" or "B" ever existed outside the lab, but there is reputed to have been a Canadian language whose name was pronounced like the former, but was spelled "EH?".) In recent years, C has been extended to allow it to be "object oriented," and the new variations are called C++ and Objective C.

Still another, whose use has been mandated by the United States government as a standard for defence critical applications, is called Ada, in honour of the Countess of Lovelace (1815-1852), the first computer programmer. Ada is also a descendent of Pascal, but is an enormous language/programming environment. While it must be used for certain military contracts, it has not so far been found to be suitable for beginning instruction, nor does it have many implementations on microcomputers.

A new language from Sun Microsystems called Java borrows notational style from C++ and programming style from Smalltalk. It is an interpreted language and its code can, in theory, be run on any platform. The idea and functionality of Java were also borrowed by Microsoft to create C#.

Wirth has also not been content to rest on his laurels, and has produced new notations in the Modula-2 style, called Oberon and Oberon-2. These are experiments in minimalism in computing notations, and may give rise to a new mainstream language at a later time. Others have also experimented with the Modula family of languages, producing Modula-3, object oriented Modula, and several other variations.

Other Notations for Problem Solving

In addition to those mentioned above, there are many other specialty languages with small but dedicated followings. There are also a number of very high level (so-called "fourth generation or 4GL") computing environments available that are not so much languages as they are a means to solve problems without writing programs. These are often designed more for business purposes than for scientific ones, however, because in the latter case, it is virtually impossible to anticipate ahead of time even the general structure of problems, and scientists and mathematicians often need the flexibility that a high level notation such as Modula-2 supplies. Moreover, 4GL's are usually interpreted and can be exceedingly slow.

In addition, such programming environments as databases and spreadsheet programs often serve more as languages than they do as applications, and there are even programmable word processing programs that

1.9 Abstractions for Instructing Machines--Program Structures

The fundamental concepts of the modern computing machine were enunciated by John Von Neumann in the late 1940s. These included both the idea of the encoded stored program, and of the implementation of programs as the step-by-step execution of a series of instructions. Devices built on such principles are often termed von Neumann machines. The exact nature of both the machine and of the instructions varies widely, but there are certain themes that are common to almost all of them.

To illustrate the fact that these methods are not confined to computing devices, it is worthwhile to observe that the tree cutting example given in [section 1.4](#) incorporated all the major ways of arranging instructions that are employed in modern computing notations.

1.9.1 Program Control Abstractions

Sequence

The simplest of these abstractions is in fact the *sequence*--one instruction following another in order.

Example:

1. Compute the number and select the trees to fell.
2. Obtain tools and safety gear.
3. Put on safety gear.
4. Prepare tools for use.
5. Cut down the selected trees.
6. Remove branches.
7. Buck the logs into short pieces.
8. Split the wood.
9. Pile the wood for drying.
10. Clean up the site.
11. Put away tools and safety gear.

Selection

The second is *selection* which refers to the choice among two or more alternative tasks depending on certain circumstances encountered when the solution is actually executed.

Example:

```
If spark plug is dirty
    then clean or replace spark plug
```

Here, the choice is between cleaning a dirty spark plug, or doing nothing. There may be a variety of outcomes from examining a fouled plug, so it too may result in some selection:

Example:

```
If spark plug is dirty
    then if spark plug is also damaged
        then replace spark plug
```

```
        examine points
        if points are damaged
            then replace them too
        else just clean spark plug
else reinstall spark plug
```

Repetition

The third organizational tool for problem solving is the *repetition* or *iteration* of a series of steps under the control of some condition. Because one or more instructions are being repeated again and again, the programming structure that expresses this is called a *loop*.

Example 1:

```
While count is less than number of trees selected
    determine direction of fall
    cut main notch facing fall direction
    overcut higher back notch on opposite side
    yell "timber"
    duck
    add one to the count
```

Example 2:

```
repeat
    file the selected tooth
    select next tooth
until all teeth are sharp
```

These two methods of repeatedly executing a sequence of statements differ only in that the first one involves checking the condition for continuing at the top of the loop (before the instructions are executed,) and the second one postpones the checking of the exit condition until the end of the instruction sequence.

Depending on the position of the test for exiting a statement repetition (a loop) the construction is known as top-of-loop tested or as bottom-of-loop tested.

A variation of the while loop (and therefore top-of-loop tested) uses an explicit counter to step through the repetitions. It might be expressed as follows:

```
for count = 1 to the number of trees selected
    determine direction of fall
    cut main notch facing fall direction
    overcut higher back notch on opposite side
    yell "timber"
    duck
```

Every programming notation uses one or more of these repetition methods (and perhaps others) for expressing repetition; the precise details for Modula-2 can await the need.

Composition

The fourth method of combining instructions involves naming a section of code, and then using only its name in the main

program sequence. This abstraction technique--of letting the name of some code stand for the whole--is termed *composition*. There was a great deal of this in the tree cutting example, in the use of such words as: compute, select, obtain, put on, file, tighten mix, fill, inspect, clean, replace, file (second type), determine, cut, overcut, yell, duck, remove, buck, split, pile, clean up, put away, start, and add. Each was employed to stand for a series of complex actions that it was thought unnecessary to detail in the instruction sequence itself.

The actual details of the composed or abstracted items are to be found either in a separate library from whence they are called upon only by name, or they may be elaborated somewhere else in the program, in an effort to avoid clutter in the main sequence of instructions. Sometimes such compositions are termed *subprograms*, although this term may have a more specialized meaning in certain computing contexts. A more Modula-2 like term for a composition is *procedure*.

Parallelism

The final method of conceiving of the execution of instructions departs entirely from the Von Neumann model. It is not implemented on a single processing device, but on many simultaneously. (One could imagine an army of chainsaw-wielding loggers cutting down an entire forest at the same time--an impossible task if one of them had to do it alone.)

The execution of simultaneous von Neumann machines all cooperating toward the achievement of a single controlling master task is called parallelism or parallel processing.

Hardware and software that allow for parallelism are relatively new, and this topic will not be studied extensively here, though Modula-2 does have some facilities for taking advantage of such ideas, and these will be explored in a later chapter.

1.9.2 Encoding (Representing) Programs--Pseudocode

The solution to the wood cutting problem does not just illustrate the planning process used in creating programs and the control methods used in expressing instructions. The point-form English in which the solution was expressed is known as *pseudocode*. This same style is used to express solutions destined for eventual coding into some specific computing notation. The advantages of writing the solution out in pseudocode first are that:

1. one need not pay particular attention at this stage to the specific grammatical details of the actual code in a particular notation
2. the pseudocode is general enough so that the solution can later be expressed in any one of several different actual coding notations
3. writing in pseudocode forces the programmer to pay sufficient attention to detail to ensure that the solution is completely thought out
4. the pseudo code is easy to examine for possible efficiency improvements and for the elimination of logical errors.

There is no hard and fast rule for deciding when to stop refining into steps at the pseudocode stage, and when to begin writing actual code in the selected notation. The transition ought to take place when it is obvious to the writer how all remaining abstractions can be expressed in the actual programming notation--a point that is reached by different people at different times, depending on the ability to conceptualize large tasks as an organic whole. For most people, it suffices to represent the "tricky bits" where unusual or hard-to-follow computations are being performed. There is a name for these:

A technique to perform a calculation, expressed as a series of steps or instructions, is called an algorithm.

Example 1:

Express in pseudocode a method for swapping the values of two variables x and y.

Solution:

One might be tempted to write, simply,

```
x <- y  (put the value of y into location x)
y <- x  (put the value of x into location y)
```


However, this will not do, for if these two commands are executed in sequence, both the variables x and y will end up naming the value originally named by x . What is needed to conduct a working swap algorithm is a place holder to name one of the values temporarily, thus:

```
temp <- x
x <- y
y <- temp
```

NOTE: One may also write:

```
temp = x
x = y
y = temp
```

as long as it is clear that the value named on the right hand side is given to the name on the left hand side. That is, the equal sign is being used as an operation, and not as a statement that two things are identical.

Example 2:

Express in pseudocode a technique of adding up a sequence of numbers indexed from $item_1$ through to $item_{20}$.

The use of a variable to hold the sum, as well as an indexing scheme is necessary. Initially, the variable should be set to zero, as it will be employed to hold a running sum. The code here is expressed in a repeating fashion:

```
sum = 0 -- this variable holds the running total
count = 1 -- this is the index counter
repeat
    sum = sum + itemcount
    increase the count by 1
until count <= n
    read currentReal
    if currentReal "increment" the counter)
until counter = 21
```

Notice that the erroneous version would never complete the task because the value for a_1 would be written continuously until the user interrupted the program.

Example 5:

Write an algorithm to add the first n positive integers, where the number n is determined by asking the user of the program.

```
write the message "Type the number of integers to be added"
read the value for max number
set running sum to zero
set current integer to one
repeat
    add current integer to running sum
    increase current integer by one
until current integer equals (max number + 1)
```

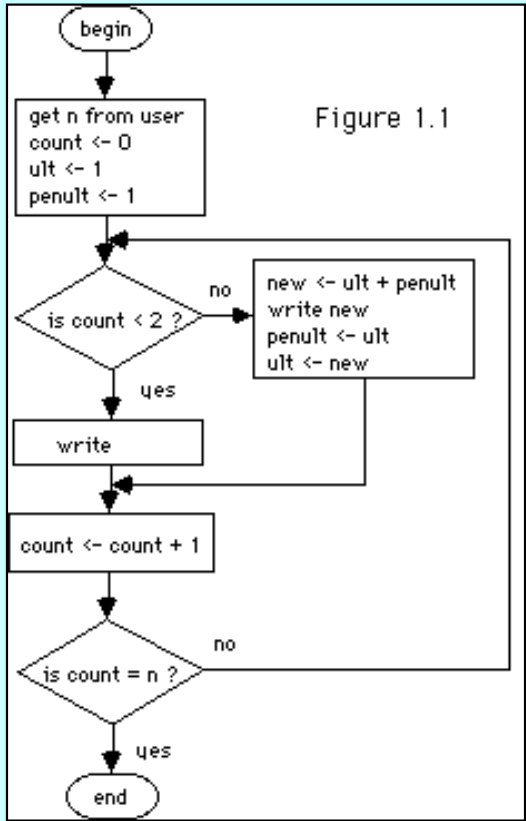
The interesting point to notice here is the use of the running sum. Observe that it must be set to zero initially. This is because in

a "real" computer, *running sum* names a memory location in the machine which may or may not have had a value previously stored in it. For the purposes of this algorithm, the assumption must be made that *running sum* has some undetermined value before being cleared to zero, making this step essential.

Example 6:

(by Gordon Tisher)

Write an algorithm to print out the first n numbers of the Fibonacci sequence. The Fibonacci sequence is the sequence of numbers 1, 1, 2, 3, 5, 8, 13, 21, 34 . . . where each number (from the third on) is the sum of the previous two. A flowchart of the algorithm is as follows:



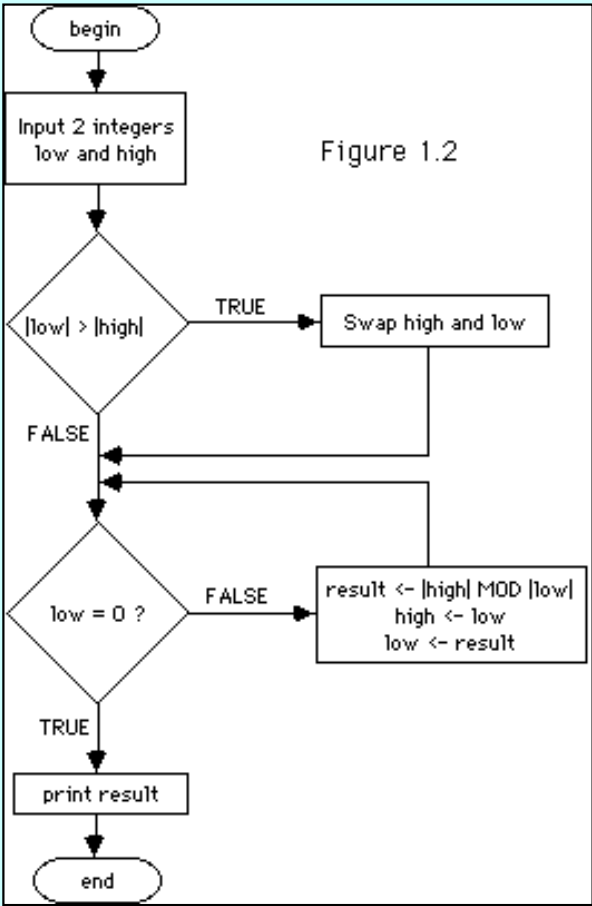
The pseudocode for this would be:

```
Fibonacci
get n from user
count ← 0
ult ← 1
penult ← 1
repeat
    if count < 2 then
        write "1"
    else
        new ← ult + penult
        write new
        penult ← ult
        ult ← new
    end if
    count ← count + 1
until count = n
end Fibonacci
```

Example 7:

(by Dana Aldom)

Write an algorithm to find the greatest common denominator (GCD) of two integers. The GCD is the highest number that divides evenly into both integers. Using Euclid's algorithm, the flowchart would be as follows:



The pseudocode for such an algorithm could be written as follows:

```
Euclid
  Input two integers: high and low
  If  $|low| < |high| \text{ MOD } |low|$ 
    high  $\leftarrow$  low
    low  $\leftarrow$  result
  end while
  print result
end Euclid
```

Most of the following chapters will have at least one sample problem worked out in full, from initial problem analysis through to one or more attempts to frame a suitable algorithm in pseudocode, and on to actual code. Since personal and institutional standards for this aspect of the work vary widely, these examples can only serve as a rough guide for the appropriate amount of detail that may be required from a student. For more precise guidelines, prevailing local standards must be consulted.

1.9.3 Syntax, and Semantics

Unlike the loose and rough style employed for pseudocode, actual programming notations have very exacting rules that determine how code that is acceptable to the translator program must be submitted. For instance, in Pascal one writes a while

loop (top-of-loop tested) as:

while <boolean expression><boolean expression>"brain," discovered that an errant moth had landed between the two contacts of a relay. Although beaten to death, the creature still served to prevent the proper electrical contact from being made. Having located the cause of the mental dysfunction, the technician removed the unfortunate creature from its impalement and brought it to the white-coated senior operators. Recognizing this for the historic moment it was, they had the brave moth incarcerated in one of their notebooks, immortalized for all time as the first "bug in the system."

Unless a program doesn't do anything, or only comprises a line or two, it will almost certainly contain errors or *bugs*. These may be of two types:

Syntax errors result from incorrect spelling, misplaced or missing punctuation (such as semicolons) or the otherwise incorrect use of some part of the notation. The compiler should discover these errors and notify when attempting to compile text into code. Some complete Modula-2 systems include a context-oriented editor, which tries to prevent the writer from entering a syntactically incorrect statement in the first place.

Logical errors, on the other hand, are the result of insufficient planning, fuzzy thinking, or poor program organization. They are caused by a failure to express the meaning of the problem in a fashion that can be translated into a solution. To put it another way, the steps in abstracting the real world problem to a computing notation are in such cases either ill-understood, mistakenly applied, or both. These errors often do not surface until the program is run and starts to produce unexpected results.

It is a fundamental principle of good program planning that one must plan before writing, write before typing, check before running, and test the program carefully before delivering the finished product to customers (or to the teacher). The cardinal sin in programming is sitting down at the terminal or microcomputer with nothing but the statement of the problem, and beginning to write code. As far as the inevitable syntax errors are concerned, it would be good to remember that the computer can do only what you tell it; its "intelligence" is very limited (unlike that of the programmer) and the slightest mistake will make communication attempts unintelligible to it. Careful planning can eliminate most bugs before they hatch. Good proofreading will squash most of the rest, and the remainder will be served up on the plate either at compile time or at run time. These, one must patiently fly swat, one by one. Much more will be said about such matters in subsequent chapters; in the meantime, fore-warned is fore-armed.

[Contents](#)

1.10 Chapter Summary

In this chapter you learned:

- about the nature of the problem solving enterprise
- how to refine the solution to problems
- what abstractions are, and about
 - data representation abstractions (data structures)
 - (atomic and compound data, type, variables, constants)
 - data manipulation abstractions (expression structures)
 - (expressions, precedence, compatibility)
 - the computing apparatus itself (machine structures)
 - hardware (input, control, memory, output)
 - software (built-in, and user programmed)
 - programming notations
 - the instruction and manipulation of the machine (program structures.)
 - sequence, selection, repetition, composition, and parallelism
 - how to represent solutions in pseudocode
 - syntax, semantics, and errors

1.11 Assignments

Questions

1. What is the difference between top-down and bottom-up design?
2. What are the major steps that must be undertaken to solve problems using top-down design?
3. "Driving a car is an abstraction." Explain.
4. What is an abstraction?
5. "A human being is more than just a machine made of meat." Comment on this statement in the light of this chapter, and your own knowledge of the state-of-the-art of computational machine capabilities.
6. Elaborate on the contention that money--however expressed--is an abstraction of wealth.
7. What is a paradigm? a model? a world view? a meme?
8. "Abstractions are intellectual creations; they are not discoveries." Comment on this statement made in the text. Perhaps you have a different view of the matter.
9. Into what two major categories do the abstractions of interest to the study of computing fall? Break each of the categories into two major sub-categories as well.
10. What is the difference between data and information?
11. Classify as compound or atomic entities: (a) a class of students, (b) the number 4.5, (c) a book, (d) a computer program, (e) the slope of a line, (f) the slope of a line when expressed as rise/run.
12. Classify according to unsigned whole number, signed whole number, real, character, string or boolean types. (Some might be more than one; some might be compound types.)
 - (a) 23.589
 - (b) 4
 - (c) (3, 4)
 - (d) "Yesterday, today and forever"
 - (e) -12
 - (f) $5 < 2$
 - (g) $15 - 9$
 - (h) $4.3 - 1.3$
 - (i) $(6 < 2)$ and (5 "virtual machine?"
21. What are the major tasks of an operating system?
22. Who was the inventor of two major programming languages, and which ones are they?
23. What language is principally used for business applications?
24. What is the difference between a compiler and an interpreter?
25. Explain the five principal abstractions for structuring programs.
26. Explain, with examples, the difference between top-of-loop and bottom-of-loop testing.
27. What are the advantages of writing out programs in pseudocode before coding them in an actual programming notation?
28. What is an algorithm?

29. What is the difference between the syntax and the semantics of a notation?
30. What are the two major sources of errors in computer programs, and how are they best prevented?

Problems

31. Evaluate the following expressions, giving the value produced for the variable. You may assume that the variable is of the correct type to name the simplified entity from the expression. If any cannot be evaluated for some reason, indicate why not.
- (a) $x = 12 + 34 * 15$
 - (b) $y = 1.2 - 3.4 / 1.7$
 - (c) $t = 3 + 5 * 8 - 6 / 2 + 7$
 - (d) $m = (1 < 5) \text{ and } (-2 < 0)$
 - (e) $b = (3 \leq 3) \text{ or false}$
 - (f) $r = (\text{not } (5 + 4 < 2)) \text{ and } (5 / 1 < 1)$
 - (g) $c = 15 < \text{true}$
 - (h) $q = (1.5 * 106) * 1000$
 - (i) $n = ((15 - 2 * 3) < 4) \text{ or not } (62 - 10)$
 - (j) $Z = 4.5 / 1.5 + \text{float } (4)$
32. Write an algorithm in pseudocode to write out the first twenty squares. Use a loop.
33. Write an algorithm in pseudocode to test whether a number is even or odd.
34. What is the output of the pseudocode:

```
for i = 1 to 10
  for j = 1 to i
    write the value of i
    write a comma and a space
    write the value of j
    go to the next line
  end "for j" loop
end "for i" loop
```

35. Write an algorithm in pseudocode to add the squares of the first n positive integers, where the number n is determined by asking the user of the program.
36. Write an algorithm in pseudocode to add ten numbers, each of which is typed by the user.
37. Write an algorithm in pseudocode to add n numbers, each of which is typed by the user. The user must first be asked for n .
38. Modify the algorithm in #37 to add the squares of the numbers as well as the numbers themselves in separate totals.
39. Write an algorithm in pseudocode to sort three numbers in order from smallest to largest.
40. The following piece of pseudo-code is meant to be an algorithm to compute the average of a sequence of real numbers. However, it will not work as shown. Make one correction so that it will.

```

Compute Average
  read n (* number of reals to do *)
  set count to 1
  set partial sum to 0
  while count <= n
    read currentReal
    add currentReal to partial sum
    add 1 to count
  end while
  set average to partial sum/ count
  write out average
end Compute average

```

41. Write an algorithm to find the largest and the smallest number of a sequence of numbers.
42. Combine the corrected algorithm for #40 and #41 to produce a single piece of pseudo code to examine a sequence of numbers and determine the largest, smallest, and average of the numbers.
43. A matrix is a rectangular array of numbers, such as:

1	4	9	-7	6
3	2	89	5	34
1	-56	7	8	13

This particular one is a three row by five column matrix. Write an algorithm for adding together all the cells of the matrix.

44. Generalize the pseudo code for the last problem by having the number of rows and the number of columns as named constants *numrows* & *numcols* rather than as specific numbers (three and five.)
45. You have a column of numbers a_1, a_2, a_3 through a_{12} that represent the monthly profits of your organization. You want to project these to the following year by adding 5% to each one and putting the result into another column numbered b_1, b_2, \dots through b_{12} . Write the pseudocode for this. (Note that the two columns constitute a matrix.)
46. You have three columns of numbers labeled revenue, expense and net profit. Each is numbered 1,2,3,... lastItemNumber. The first two are filled in. Write the pseudocode to construct the third.
47. As in #46 above, but the columns are labeled price, GST (tax) and total. Only the first column is filled in. The second is constructed by finding 7% of the number in the first, and the third by taking the total of the first two. Write pseudocode to do this.

2.0 Chapter Goals

The purpose of this chapter is to turn the theoretical considerations of chapter one into some practical realizations to put some flesh on the ideas of problem solving by using Modula-2 as the program writing notation. A simple program that writes a message to the screen is given first, and its parts are analyzed carefully. Following this, the technique of problem solving is elaborated with additional examples in Modula-2. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

No new data types are taken up in chapter 2.

Realized in the Modula-2 notation:

cardinal, integer, character, real, longreal, and literal strings

Data Manipulation Abstractions

General:

variable initialization

Realized in the Modula-2 notation:

variables, constants, literals, declaration, naming, assignment, expressions, type, type compatibility, type conversion, input and output, formatting of output.

Programming Abstractions

General:

No new programing abstractions are taken up in chapter 2. Several of the ones introduced in Chapter one are discussed further. Planning, refining, coding, and documentation of programs are elaborated.

Realized in the Modula-2 notation:

statement, repetition (while loop,) computer program, importing from libraries of pre-written routines (input and output,) commenting.

2.1 Giving Birth--A First Modula-2 Program

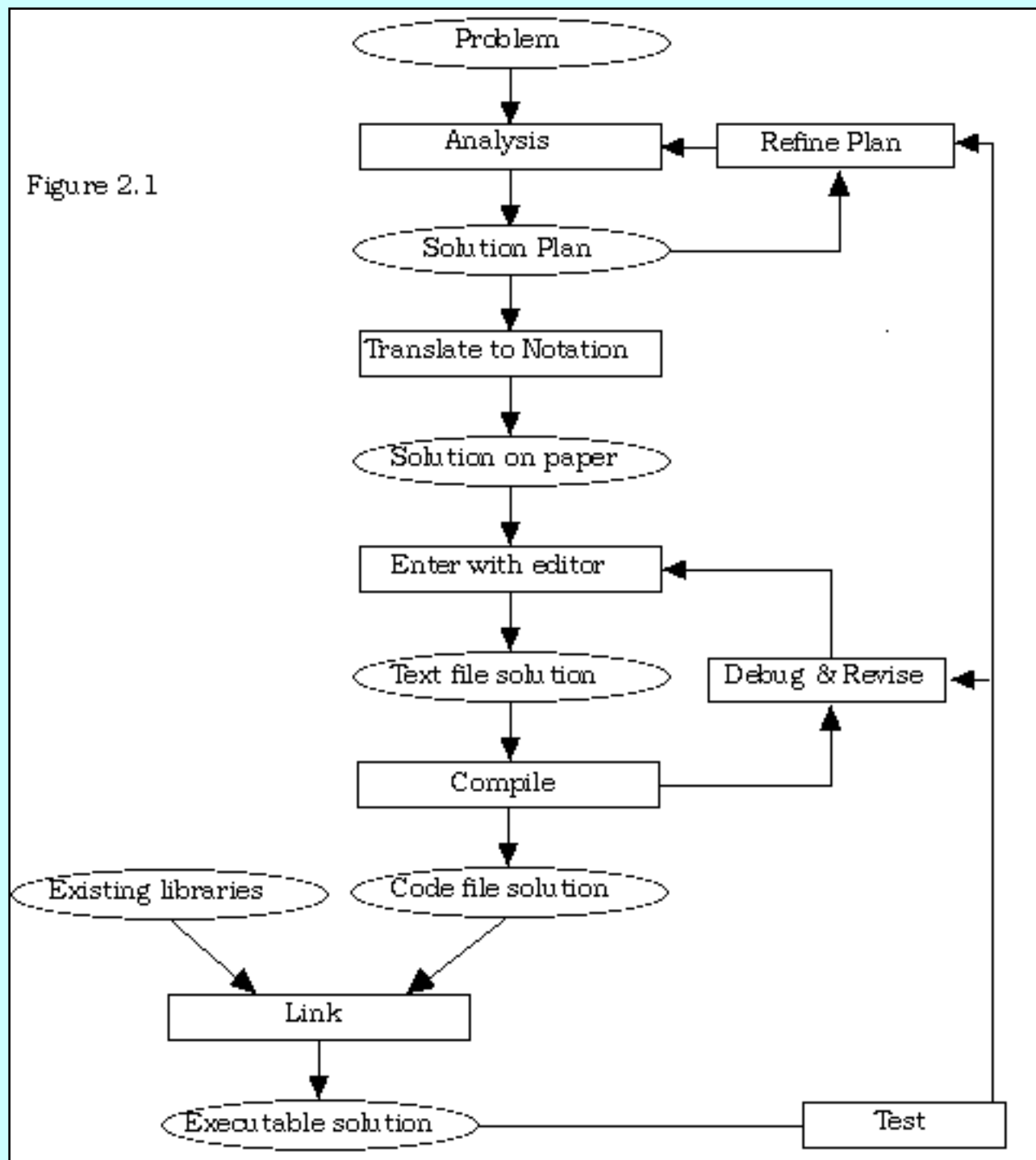
Programs written in the Modula-2 notation are generally compiled. (See [section 1.8.3](#).) That is, the programmer cannot simply write out the instructions, type *run* and get immediate results on the console. Rather, one uses an editing program to create a text version of the program and then compiles this into a code version that the machine can execute.

In a typical implementation of Modula-2 (and of many other notations) the writing and execution of a program involves the following steps:

1. Carefully define the problem to be solved. (The discussion of Chapter 1 will be followed up on in more detail throughout the text.)
2. Refine the plan into tasks and sub-tasks as necessary
3. Write out the solution to the problem using a high level notation such as Modula-2.
4. Enter a text version of the program into the computer using an editing program and save it for later reference.
5. Run the compiler program to translate the text version to a machine-readable form.
6. Link the file(s) created at step two with others previously compiled (or supplied with the system) to generate a program that will actually run by itself.
7. Run the program.
8. Test the program and redevelop as necessary.

Figure 2.1 shows a developer's-eye view of a computing system being used for problem solving:

Figure 2.1



In some cases, direct action by the programmer is not required to achieve linking. Linking may be performed automatically at run time (and therefore be invisible to the programmer.) On the other hand, it may require that a separate program be run using the compiled file(s) as input and an executable file as output.

In this book, little emphasis will be placed on the compiling, linking, and interpreting processes. Instead, the focus will be on writing efficient, error-free programs that will compile and run smoothly. The details of how to use particular computers, their operating systems, editors, compilers, and linkers, are beyond the scope of this book. Such instruction must be obtained from separate manuals supplied with the specific system, or provided in an instructional lab.

Of the major functions of a computer (input, storage, processing, control, and output), only the processing step lies within the definition of a program given in the introduction. Storage usually is

handled more or less automatically by the system, though we shall later see that the programmer also can control this to a limited extent. Even input and output are, in some senses, peripheral to the task of processing, despite the fact that without them the computer would have no means of obtaining its data or of delivering any results. Consider yourself: If you cannot articulate your opinions (output) you may as well not have any. So, what good is a box that calculates but does not give the answers to the outside world?

Perhaps more important is the fact that input and output are handled differently on various computers. That is, these are *hardware-dependent* functions. When the routines that handle these operations are entirely contained within the computing notation itself, programs written for one computer are unlikely to work on another one. For these reasons, the input and output functions have been separated from the Modula-2 language proper. They must be imported into the programs from another part of the system called the library. The contents of this library always look the same to the programs that employ it (This is what standards are for), but the code that implements the library will be quite different on different machines. This importation is illustrated in the first sample program, shown below.

```
MODULE HiThere;  
  
FROM STextIO IMPORT  
    WriteString;  
  
BEGIN  
    WriteString ("Hello Master, how may I serve you?");  
END HiThere.
```

The output from this program is the line:

```
Hello Master, how may I serve you?
```

NOTE: The place from which *WriteString* is usually obtained is called *STextIO* or *InOut*

[Contents](#)

2.2 The Anatomy of an Infant Program

All Modula-2 programs must have the general form shown in the sample above. The word `MODULE` comes first, followed by some descriptive name. If there are any `IMPORT` commands, these come next. The words `BEGIN` and `END` (which must be all in upper case) mark off the *block* of actual program instructions or *statements*. They serve more or less as parentheses, even though they are spelled out as words. There may be several such blocks opened and closed in this manner throughout a program (more on this later), but it is important to note that this main one must end with a period, and that any block that has a name such as *HiThere* as this one does must have that name mentioned again just after the corresponding `END`. Note the following definition:

A statement is an instruction directing a computer to take some action. It usually is written out in a high level notation for subsequent translation by the computer into one or more low level instructions and their corresponding actions.

The semicolons between statements are not optional, except before an `END` marker. They separate the statements from each other, in effect telling the computer that one instruction is finished and that it is time to start another one. As indicated, an `END` marker is not a statement, but punctuation, so the semicolon can be left out before it. There are far more complex programs, but this one, perhaps the simplest that actually does something, is a start. In the short sections that follow, some aspects of this program are analyzed in detail.

2.2.1 What is a Module?

This section considers the first word of the sample program, a word that every program will have at or near the beginning.

A Module is a container to hold the items and information that constitute all or part of an executable program.

Obviously, this thing called a module is important. It doesn't take much deep thinking to realize that the whole Modula-2 programming notation is named after it. Unfortunately, this initial attempt at a description does not tell the whole story about modules, and a complete description must wait until later in this book. It will have to suffice for now to say that a program is a module, and that there are other modules available in the system from which we can import things (like *WriteString*) that may be useful to us. For instance, if all the entities mentioned below actually existed, this too would be a legitimate program:

```
MODULE Lunch;  
  
FROM LunchBag IMPORT  
    sandwich, apple, Eat;  
  
BEGIN  
    Eat (sandwich);  
    Eat (apple);  
END Lunch.
```

Here, the modules are *Lunch* and *LunchBag*, while *sandwich* and *apple* are evidently some sort of entities that can be acted upon, and *Eat* is some kind of procedure (like *WriteString*) that causes an action to take place. The term *procedure* will be defined more carefully later.

2.2.2 Reserved Words

Certain special punctuation markers in the sample program are capitalized (and printed in bold for emphasis.) All such special symbols were defined and reserved for a particular purpose by Niklaus Wirth when he first designed Modula-2. Every programming notation has such symbols and uses them in much the same way as does Modula-2.

A Modula-2 reserved word is a special word or marker used to outline the structure of a program. It must be entirely written in upper case letters, and it cannot be used for any other purpose.

The reserved words encountered in the sample program were: MODULE, BEGIN, FROM, IMPORT, and END. Others will be encountered later, and each one will be pointed out as a reserved word when it first occurs in an example. A complete list can be found in [Appendix 1](#).

2.2.3 Standard Library Tools

Other items (such as *WriteString*) are imported from a pre-existing library, and can also be thought of as being, in some sense, a standard part of every Modula-2 system. Many modern programming notations share this concept; it is not unique to Modula-2.

A Modula-2 Standard Library Item is an entity that can be found by a particular name and in a particular library in every implementation of standard Modula-2.

When Niklaus Wirth defined the Modula-2 notation, he included a few sample or suggested library modules along with his definition of the language proper. Many of these, and their entire contents, have been adopted and used for a wide variety of Modula-2 versions and constitute a *de facto* standard for Modula-2 libraries. There is also a *de jure* international standard specified by the International Organization for Standards (ISO) for Modula-2, and it also specifies certain (different) libraries that are required to accompany any implementation designated as *standard* under its terms.

Most implementors follow (or at least include) material from the *de facto* standard suggested by Wirth. Others have adopted the ISO modules for standard Modula-2 (though that is new), and still others ignore both and design libraries as they see fit. For this reason, the name of typical imported procedures (such as *WriteString*) may vary, and so may the name of the library module from which they are imported. In this text, the ISO standard for Modula-2 will be used for most programs, and any that use other libraries will be noted.

NOTE: There are also *language* differences between Wirth's definition of Modula-2 and the ISO definition. There are even some between different editions of *Programming in Modula-2*. These too will be observed in the context where they are relevant.

2.2.4 A Name for the Baby--Identifiers

In the examples above, each module had a name. This is required both in the MODULE heading and in its corresponding END. However, not only modules have names. In fact, there are many other entities in a typical program that also have to be identified in this manner. (There were some in the last example.)

For instance, if one wanted to write code using a formula like:

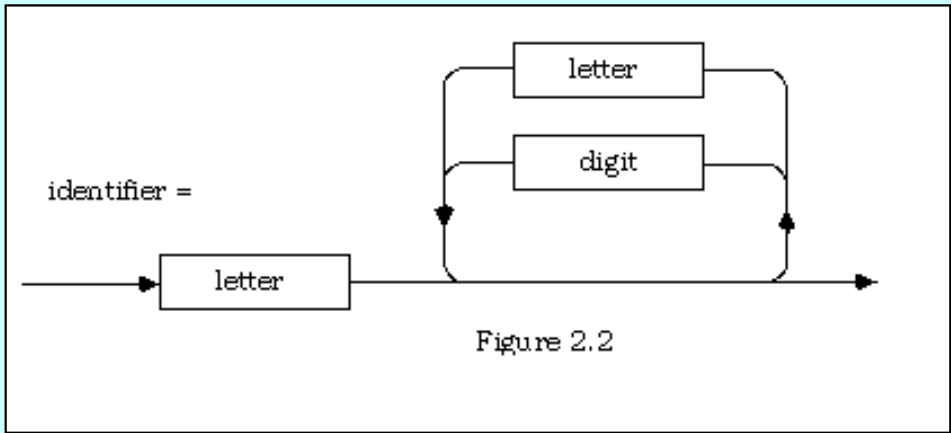
$a = b + c$ (in Modula-2 it is $a := b + c$)

then *a*, *b*, and *c* must all be names for program entities (variable names.) One could call them *number1*, *number2*, and *number3*, or any other name--provided that the name is legal in Modula-2.

The technical term for such names is *identifier*. It is important to know the rules that must be followed for creating identifiers.

A Modula-2 identifier is a sequence of non-blank letters or digits beginning with a letter.

This definition is illustrated in Figure 2.2 by what is sometimes termed a *railroad diagram*. The arrows indicate the sequence of symbols or characters that define an identifier.



The alternate path, or *loop* shows clearly that, after an initial letter, any sequence of letters and numbers is permitted. Notice that the definition not only excludes blanks, but also any special symbols, such as +, -, /, \$, %. To conform with past practice in other languages, or with the requirements of other software on the system, some versions of Modula-2 may allow the dollar symbol, spaces and/or some of these other symbols that are supposed to be forbidden. In this book, "plain vanilla" rules will be followed in such matters, and such compromises will be ignored.

NOTE: In ISO standard Modula-2 the low-line (also sometimes called an underscore) is counted as a letter. This was done to conform to the practice in Pascal. However, it is easier to type and to read, say, *SideOfSquare* than *side_of_square*, so the former style will be used exclusively in this texts. Programmers are urged to avoid the use of low-lines in identifiers unless these are absolutely necessary to conform to some local practices over which they have no control.

Here are a few examples of identifiers:

Correct:

FixItUp	FRED
CompoundInterest	Showboat1
Rate	Blank25
Time	hello

Incorrect:

(contains dash)

Compound Interest	(has a blank)
What%	(illegal symbol)
23Skidoo	(does not start with a letter)
Next-Time	

Permitted in ISO standard, but very poor style:

this_identifier
this__longer__identifier (how many low-lines are there?)

Although the definition itself does not make this clear, there is also a distinction made between upper case and lower case letters. Consider the following examples:

These are Different:

Help, help, HELP, HeLP, etc.
BEGIN, Begin (the first is a reserved system word)
Something, SomethingIAt

NOTE: As the second-to-last example shows, it is essential to type every identifier or command with the correct (consistent) upper case or lower case letters. Do not type *Writestring* or *writestring* instead of *WriteString*, for example.

In some implementations of Modula-2, the last two in this list of examples might not actually be interpreted as different by other parts of the system, even if they are by the language. For instance, in dealing with file names on the disk or other external device, only the first eight or ten (or some other number) of characters may be checked. Operating systems, and

notations other than Modula-2 may recognize only upper case letters, or treat upper case and lower case versions of a letter as the same. These peculiarities are holdovers from days when computer terminals had only upper case letters on their keyboards. They apply only when sending such information to the operating system, not within the program.

NOTE: 1. As previously indicated, words like MODULE, BEGIN, END, FROM, and IMPORT are reserved words, and cannot be used as identifiers.

2. Other words are also already taken. They too are entirely capitalized and are called standard identifiers. (See [section 2.5.2](#) for a full discussion.) Still others may be commonly imported identifiers (such as *WriteString*.) It would be unwise to use these as names for something else, even though you may get away with it.

3. There is a certain style or taste to the matter of appropriate capitalization of identifiers. Specific rules shall be outlined later; for now, imitate.

2.2.5 Strings

The sample program contained the line:

```
WriteString ("Hello Master, how may I serve you?")
```

The portion between the double quotes is called a *literal string* and the effect of this particular statement will be to write out the characters between the quotes onto the computer's main output device (usually a screen, but sometimes a printer).

A literal string is a sequence of characters that is enclosed between either single quotes or double quotes.

Here are some examples of literal strings:

```
"How are you doing today?"
'Do you think "hello" is the correct word?'
"Don't you like tea?"
"Your mark is 100%."
'What a nice day!'
"He said 'Hi there!' to me yesterday."
'The distance between studs is 16".'
```

Notice that if the string contains a single quote (apostrophe) it must be in double quotes and if it contains double quotes it must be enclosed in single quotes. No string can contain both single and double quotes, but there are no other restrictions on the characters it contains, except that it cannot run contain a carriage return.

```
WriteString ("This is an example of an illegal string, because it is too long to fit
on one line of the original text form of the program.")
```

Here is how to do this legally:

```
WriteString ("This one is better because it ")
WriteString ("has been broken up into pieces, ")
WriteString ("each of a more manageable size.")
```

Notice the spaces at the end of each of the first two literals in this last example. These are needed to ensure that the words in the successive strings will in fact be separated from one another in the output as it is printed on the screen (or elsewhere). This does not, however, solve another problem--if one writes several strings like these one after another, the computer will perform a *carriage return* of its own when it gets to the end of a line on the screen.

A carriage return is an action taken by an output device such as a screen or printer that causes the printing position for the next character to be placed at the beginning of a new line.

This could be right in the middle of a word. In fact, if one had a 40 column screen, the output from this last example would be:

```
This one is better because it has been broken up into pieces, each of a more manageable size.
```

To prevent this, the programmer inserts carriage returns in the correct places (between words) by using the *WriteLn* statement. The output in the next two examples has been formatted in this way.

Here's the same one again:

```
WriteString ("This one is better because it ");
WriteLn;
WriteString ("has been broken up into pieces, ");
WriteLn;
WriteString ("each of a more manageable size.");
WriteLn;
```

The purpose of *WriteLn* is to reposition the cursor (the next character location) to the beginning of the next line. Here's another example--this time a complete Module.

```
MODULE Notice;

FROM STextIO IMPORT
    WriteString, WriteLn;

BEGIN
    WriteString ("* * * * *");
    WriteLn;
    WriteString ("*   This   Program   *");
    WriteLn;
    WriteString ("* *   Copyright   * *");
    WriteLn;
    WriteString ("* * *   2005   * * *");
    WriteLn;
    WriteString ("* * * * * by * * * *");
    WriteLn;
    WriteString ("* Nellie   Hacker *");
    WriteLn;
    WriteString ("* * * * *");
    WriteLn;
END Notice.
```

The output from this program is:

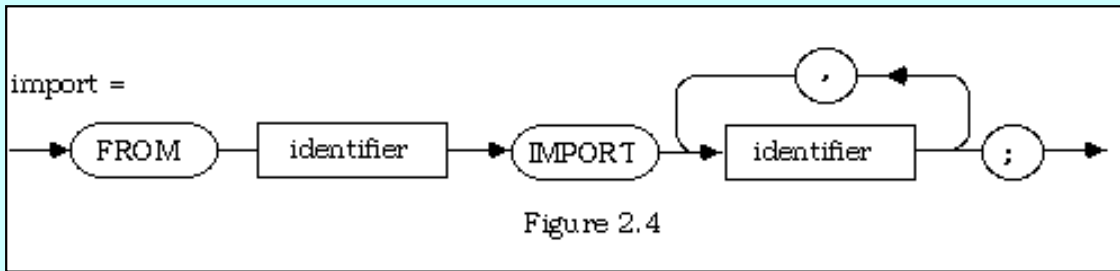
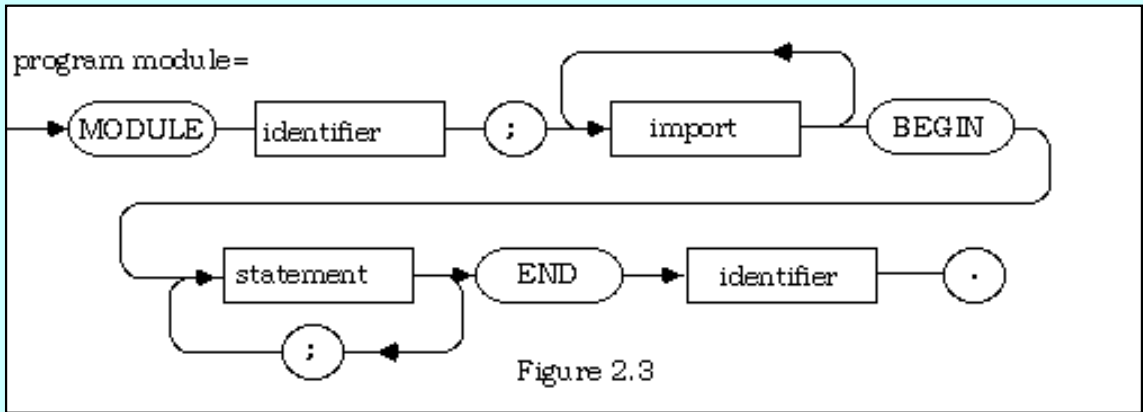
```
* * * * *
*   This   Program   *
* *   Copyright   * *
```

* * * 2005 * * *
* * * * by * * * *
* Nellie Hacker *
* * * * * * * * * *

- NOTES:** 1. It is a good idea to use a *WriteLn* after the last actual output of text to ensure that the cursor is correctly positioned for the next output.
2. If *WriteString* is imported from another place than *STextIO*, say from *InOut*, then is also *WriteLn*.

2.2.6 Summary

The railroad diagram in figures 2.3 and 2.4 summarize the discussions of syntax (correct structure) of a module as discussed thus far.



These diagrams are more complex than Figure 2.2, with *necessary* reserved words or symbols in circles and ovals and the parts supplied by the programmer in boxes. Notice that the diagram places no limit on the number of import lists or statements.

NOTE: Neither of these diagrams is complete. More will be added to both later in the text. Complete versions are in [Appendix 2](#).

2.3 How to Solve a Problem

With these remarks on the structure of simple Modula-2 programs, it is time to return to the theme of problem solving. This was discussed at the start of chapter one; here that general discussion will be applied specifically to the kinds of problem for which a computer-assisted solution is anticipated. During the course of the following discussion, a simple problem will be posed and a complete solution provided. This will culminate in a program that embodies the solution in a coded form.

Most problems appear insurmountable at first--that seems to be their very nature. Many students (and some teachers), on being presented with one, will scan it over lightly in an attempt to get the whole picture in their mind at once and then, after a few minutes, throw up their hands in despair and shout "I don't know where to start!" Worse still, they may make dozens of trips to the professor or teaching assistants, each time with the same poignant question: "What's the next step?"

Note, however, that the way problem solving was defined in [section 1.1](#) presupposes that a problem is something that has a solution. That which has no solution is not a problem in the sense used in this text. Thus, the person faced with a problem is here assumed to be one who *can* solve it, given a suitable strategy or technique. The task is to find such a technique, then to employ the computer to implement it. That is, a computer programmer is a problem solver who uses a particular tool (the computer) as part of the solution process.

In [section 1.3](#), eight goals for a problem solver were detailed. It is time to turn those goals into a specific strategy. Thus, though the step-by-step problem solving system that follows still has enough generality to be modified for other types of problem, it has been formulated here with the assumption that computer assistance will be employed in the final process.

2.3.1 Analysis

Goals:

Write Everything Down.

- 1. Have a clear grasp of the existing state of affairs.*
- 2. Have a clear conception of the goal.*
- 3. Formulate the problem clearly.*

Strategic Step 1 Write the problem out.

This forces the would-be problem solver to slow down an overheated mind long enough to read the problem carefully instead of trying to absorb it as a whole by staring at entire paragraphs at once. If necessary, copy the whole thing word for word or highlight it in the book, or tap a pencil on each word while reading it. Before beginning to consider what the solution might be, one must know what the problem means--and that presupposes knowing what the words say.

Example problem:

Write a program that can raise a whole number (say 4) to a positive whole number power (say 6.)

Strategic Step 2 Ask whether a computer is appropriate.

There is no sense using a sledgehammer to crack a peanut or a chain saw to cut a toothpick. Likewise, there is no point in using a computer for simple additions or multiplications. They can be done mentally, or with a calculator, but it is a waste of both human and machine resources to write a computer program for such tasks. True, some of the examples in this book are trivial, but this is necessary to make teaching points simply. Students ought to remember that they have to walk before they can run. It will take a while to begin to realize the true power of the machine, and to write substantial programs to take advantage of that power.

Example problem suitability:

The sample problem posed above is well within the reach of a simple calculator, and is therefore of marginal suitability for computer solution. However, the solution does illustrate how a number of the programming ideas discussed in chapter one are actually implemented in Modula-2.

Still, some problems may be too general, or too easy to be appropriate for computer solution. On the other hand, they may be out of a machine's reach for other reasons, such as the requirement for some thinking process that is outside the domain of machines. Here are examples of the kind of problem that might not at this time be appropriate for computer solution.

1. How do we solve Canada's economic problems?
2. What is the aesthetic value of that painting?
3. Who will win the Stampeders vs. Lions game?
4. Who is right--the Evolutionists or the Creationists?

Partial answers might be obtainable for the first problem by computer analysis of available data (provided that the meaning of the words "economic" and "problem" can be agreed upon). Reference to some base of information also may give an answer to the second question, but only as a dollar value, and even then only if the painting had a previous sales history. The third question can be responded to in terms of probabilities based on an analysis of the past (perhaps computer-assisted) but cannot be answered as stated here. The fourth question is not answerable in any final sense that would be satisfying to all concerned, either via data analysis or the scientific method, since there is no evidence compelling enough to convince either side that its religious or philosophical beliefs are wrong. Each will continue to "keep the faith" regardless of the other's interpretation of the data, and neither is likely to experience a conversion to the other view as a result of any computerized analysis.

There is a third category of problem that may be inappropriate to tackle with the particular resources at hand. These are the questions that could be analyzed by a machine, but not, say by a personal computer, for it may be too slow, or lack a suitable notation, peripherals, or storage space. For instance, one cannot run Canada's income tax collection system or design aircraft frames on a personal computer. One may also find that certain computing notations lack some facilities needed to solve a particular problem efficiently. In this case, the programmer must learn how to work within a different programming environment in order to get the desired answers.

Lack of knowledge about what the computer can do to solve a problem constitutes a fourth difficulty in this category. Even this many years into the computer revolution, many people have a knowledge of computing limited to what they read in the gushy reviews typical of many trade magazines. They often believe in such ephemeral enthusiasms and rush out to buy the latest brand-name computer solution for their accounting, inventory, or information flow troubles. However, computerizing a badly organized business will not solve any of its existing problems; it will only ensure that they occur more rapidly, in greater numbers, and with more potential for harm. An already efficient operation can be improved with carefully chosen machines and software, but an inefficient one will only become worse with the addition of a computer.

Caveat computorae! To err is human. To really foul things up requires a computer.

Murphy's Law: It only takes a millisecond for everything to go wrong using a computer.

Strategic Step 3 Re-write the problem in your own words.

At this third stage, one is aiming for an understanding of what is being asked. This is the stage where one begins to go from the general to specifics. If one's conception of what is known and what is required for the solution are vague, there is no point in carrying on. Use headings and point form, and concentrate on the given information and the desired result. Try to be precise. Write out any formulas that are stated or implied--in the latter case, it may be necessary to look something up.

Example problem restatement:

Given: Two whole numbers, one the base and the other the exponent

To Do: Compute $\text{base}^{\text{exponent}}$

Desired Result: print the result

Formula: none. Use a repeated multiplication

2.3.2 Planning and Refining a Solution

Goals: 1. Consider related tasks.

2. Break the main task into sub-tasks.

Strategic Step 4 Re-Use Previous Work Where Possible

If you have written any previous programs in Modula-2, there will likely be parts that one can copy into the new solution in order to avoid typing them again. These should be noted now. If this is a first program in Modula-2, one might think that there is nothing to be re-used. However, that is far from the case. Every implementation of Modula-2 comes with a substantial library of routines designed to solve certain problems commonly encountered in writing programs. As noted in the discussion of the simple example earlier in this chapter, those library routines include code for input and output. There are also pre-

programmed mathematical functions, utilities for using other parts of the system (such as the computer's clock), and many others. Almost all Modula-2 programs import from these libraries, so a note should be made at any stage of the planning process when it is observed that the library can be used.

Example problem library use:

This problem has output requirements. It will, therefore make use of the standard *STextIO* and *SWholeIO* Modula-2 libraries. No other libraries are required.

Strategic Step 5 Break the problem into steps.

This is done first into larger tasks, then into smaller ones. Simple programs may not have this distinction, but most do. Ultimately, individual detailed program statements must be fed to the computer's editor one statement at a time for later compilation. The place to decide how to do this is not at the terminal, as the programmer sits down to start typing. Rather, the steps to solve the problem must be written out ahead of time in enough detail to allow a more or less direct translation into computer code. The amount of detail necessary varies with experience and familiarity with the question at hand, but most problems worth solving by computer require several refinements first into broad detail, then into finer steps. If several restatements and refinements are necessary, they must be documented. The time spent at this stage saves many hours later. Sloppily designed and poorly thought out programs simply do not work.

Example problem refinement:

1. Input Section obtain base
 obtain exponent
 (these will be stored in the program)
2. Computation calculate $\text{base}^{\text{exponent}}$
3. Output print out final result

Second refinement of problem: (same numbers as above)

1. Set up values of base and exponent
 Assign the value of the base to a cardinal variable
 Assign the value of the exponent to another cardinal variable
2. Computation
 set the result initially to the base
 set a counter to one
 while the counter is less than the exponent
 multiply the result by the base and increase the counter
3. Print out the final answer

2.3.3 Data Tables and Sample I/O

Goals: Write everything down.

Strategic Step 6 List all variables and imports.

Before actually beginning to code, write down the names of all variables that will be used and the names of all library items required. This section serves as a quick reference when writing the code, so that a name is not inadvertently used for the wrong thing.

Example problem data table:

```
Variables: base, exponent, counter, result--all cardinals
Imports from STextIO: WriteString, WriteLn
           from SWholeIO: WriteCard
```

Strategic Step 7 Show what input is required and what output is expected

Here, the exact form of everything that will appear on the computer screen is specified. The final program is correct if it matches the specifications that were written for it in advance. Note that this section does not contain a sample of the actual output produced after the program is run; it specifies what the output for a given input will be *before* the program is ever written.

Example problem sample I/O:

Input:

Set the base to 5 and the exponent to 6 in the code.

Output:

5 raised to the power 6 equals 15625

2.3.4 Refining the Solution

Goals: Refine the sub-tasks into individual steps.

Strategic Step 8 Sketch the solution using pseudocode

Writing the solution in pseudocode is just another refinement, this time into a shorter and less wordy form. Pseudocode resembles the final program, but there is no need to pay attention to the exact syntax of the

computing notation that will be used.

Example problem pseudocode:

```
Write "This program will raise a given base to a given exponent"
Assign the base
Assign the exponent
result <-- base
counter <-- 1
while counter <-- exponent
    result <-- result * base
    counter <-- counter + 1
write base
write "raised to the power"
write exponent
write "equals"
write result
```

Strategic Step 9 Refine the code completely before entering it

Do not waste scarce computer resources by doing initial rough copies of code at the keyboard. Ideally, what the programmer does type is so well thought out and so carefully written out (by hand) that it compiles and runs error free the first time. Actually, the world may not always turn in this fashion, but it is nice to try. In the case at hand, the final code looks like this:

```
MODULE Powers;

FROM STextIO IMPORT
    WriteString, WriteLn;

FROM SWholeIO IMPORT
    WriteCard;

VAR
    base, exponent, counter, result: CARDINAL;

BEGIN
    WriteString ("This program raises a base to a power");
    WriteLn;
    base := 4;
    exponent := 6;
    result := base;
    counter := 1;
    WHILE counter < exponent
```



```

DO
    result  := result *  base;
    counter := counter + 1
END;

```

```

WriteCard (base, 0);
WriteString ( " raised to the power ");
WriteCard (exponent, 0);
WriteString ( " equals ");
WriteCard (result, 0);
WriteLn;

```

END Powers.

NOTES: 1. Observe the specific syntax for the listing of variables and giving their type at the start of the program, for assignment and for the Modula-2 version of the WHILE loop. The assignment operator := is a reserved symbol; WHILE and DO are reserved words, and the name CARDINAL is a built-in identifier (see section 2.5.2.)

2. The purpose of the number 0 in the *WriteCard* statements will be given later.

3. Observe the use of spaces in the *WriteString* statements in order to separate the words from the numbers.

4. If using non standard-conforming Modula-2, *WriteCard* may be imported from *InOut*. Only one FROM..IMPORT line is needed instead of two.

There you have it--the top-down *design* of a small program in all its gory detail. As mentioned earlier, the amount of detail may vary, as may the number of times one must refine the problem into steps, but the *kind* of planning outlined here is not optional.

2.3.5 Execution and Satisfaction

Goals: 1. *Execute the completed process.*

2. *Re-develop the solution until the desired goal is reached.*

Strategic Step 10 Compile, link and run the finished program

The task is not complete until the program has been run and the output checked against the specifications. Output from a few sample runs should be recorded and made a part of the documentation for the project.

Example problem sample output:

First run: (using the code above)

```

This program raises a base to a power
4 raised to the power 6 equals 4096

```

Second run: (with the base set to 3 and the exponent to 4)

```
This program raises a base to a power
3 raised to the power 4 equals 81
```

Strategic Step 11 Check actual output with specifications and correct errors

Only the final code was given above. The first time it was compiled there were several punctuation errors, including some missing semicolons and a missing colon in an assignment operator. The sample program could also use a little more work (not done here) in order to catch erroneous inputs such as incorrectly typed numbers. It also produces an incorrect result if the exponent is zero (try it.) That possibility was not allowed for in the original problem statement, so the solution is correct. However, one could add additional code, either to give a correct result for this case, or to exclude it by printing an error message whenever zero is input for the exponent.

[Contents](#)

2.4 Documenting the Solution

Goals: Write everything down. (Reprise)

At Every Strategic Step Document what you have written.

Good programs are not only well planned and systematically written, but also are thoroughly documented. There are two major categories of documentation that must be produced if a program is to be used properly or even understood by its own writer at a later date:

2.4.1 External Documentation

A manual must be produced for the ultimate user of the program. This external documentation must explain in a clear, non-technical fashion what is expected of the user at each step, and what the program will do with the information provided. In commercial practice, a person other than the programmer usually writes the manuals in order to avoid the use of technical jargon, and orient the documentation to the user. It is best to write the user manual before coding the program, specifying completely what every screen will look like, what all the commands are and exactly what they will do. The user manual then becomes part of the specifications for writing the program. One or two sample user manuals will be given later as part of extended examples.

2.4.2 Internal Documentation

The program itself ought to contain carefully written documentation. This material is of three types:

1. Names

The rules for constructing names were discussed in section 2.2.4. These rules apply to all names, not just to modules. It is not enough, however, to be able to construct names that follow the syntactical rules for Modula-2. Names ought also to be given creatively, so that they provide part of the program documentation, assisting the programmer to read and understand the work at a later time. Modules, variables, and other items that require names should therefore be identified descriptively. That is, a module name should reflect its function, and that of an item in the module should reflect its use or role in the program.

Poor Module Names: Snafu, Tarfu, Fubar, MyProgram, ThisCode, Thingy.

Some of these are easy or traditional, but they have no meaning in the context of the work being undertaken.

Good Module Names: Hello, ComputeInterest, FindArea, ComputePowers.

These state, imply, or at least hint at the purpose of the module they name.

Poor Variable Names: i, j, k, n, p, r, t, x, var, thing, a1, a2, a3.

Such names may be easy to type without much thought, but they produce cryptic and nearly unreadable code.

Good variable names: interest, number, principal, rate, time, side, base, exponent.

By describing their role in the program, such names immediately inform the reader of the code what is supposed to be happening when the program is run. This makes both understanding and modification feasible. There is nothing more embarrassing than being unable to understand your own code just weeks after writing it. (Except, perhaps being unable to bring out a new version of a commercial package because no one in the company understands the code.) It also illustrates that:

Programs are meant to be read by humans, not by computers.

2. Comments

Writing a program can be a long and tedious process. The more useful the program is expected to be, the longer and more complex the source code (text file) is likely to become. If anyone wants to change a section later, will it be possible to find out why they were written the way they were? Does the programmer even understand the beginning of the program by the time the end is reached? Can someone else read and modify the work several years later?

To ensure that there will be positive answers to these questions, all programming notations provide a mechanism for inserting explanatory notes within the original text file. These comments are ignored when the compiler reads the file, but are necessary for human beings to be able to read it intelligently. In Modula-2, comments are enclosed in comment parentheses, which consist of the reserved symbol `(*` to begin a comment and the reserved symbol `*)` to end one. Here is the current example, heavily commented:

```
MODULE PowersCommented;  (* This is the title line. *)

(* Written by R.J. Sutcliffe *)
(* as an introductory example *)
(* and to demonstrate comments *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 01 25 *)

(* Note that comments such as the one above ought always to be included in the source
file in order to identify its writer and purpose. (* Observe that comments can extend
over several lines and can include other comments. *) *)

FROM STextIO IMPORT    (* comments can go here, too *)
    WriteString, WriteLn;
FROM SWholeIO IMPORT
    WriteCard;

(* Alternately, have in non-ISO Modula-2:
FROM InOut IMPORT
    WriteString, WriteLn, WriteCard;
*)

VAR
    base, exponent, counter, result: CARDINAL;

BEGIN    (* The program starts here *)
    (*
    Introductory Section: note that a program ought
    also to identify itself when run. Note too this
    alternate style for multi-line comments.
    *)
    WriteString ("The program Powers");
    WriteLn;
    WriteString ("was written by R.J. Sutcliffe");
    WriteLn;
    WriteString ("as an introductory example");
    WriteLn;
    WriteString ("This program raises a base to an exponent");
    WriteLn;

    (* Set up the key variables *)
    base := 4;
```

```

exponent := 6;

(* calculation section *)
result := base;  (* initially, set the result to the base *)
counter := 1;
WHILE counter < exponent
  DO
    result := result * base; (* multiply base enough times *)
    counter := counter + 1
  END;

(* display the result *)
WriteCard (base, 0);
WriteString ( " raised to the power ");
WriteCard (exponent, 0);
WriteString ( " equals ");
WriteCard (result, 0);
WriteLn;

END PowersCommented.

```

A comment is any material enclosed within the comment parentheses which are "(" and ")."

The student should observe carefully how effective comments are used in the longer and more detailed examples in this book.

3. On-line Help

The meaning of most programs of any substance will be ambiguous at times even to experienced users. At such times, it may be useful to be able to select a "help" option within the program itself, rather than have to look the information up in a printed manual. Because of space limitations, this type of documentation will not be fully illustrated in the programs in this text. However, there will be instances of on-line help in such situations as error handling. Commercial programs, however, require such documentation just to be regarded as acceptable in the modern marketplace. In such projects, on-line assistance must be planned for at each step of the program design. It should be complete, (though not so complete as the printed documents) and should be relevant to the context of the program at the time the help is requested. Apart from this remark, further comment on this point is beyond the scope of this text.

[Contents](#)

2.5 Variables in Modula-2

As in the sections following the "hello world" example, a careful analysis now needs to be given of the major new idea introduced in the example of the [section 2.3](#)--that of the Modula-2 variable. Variables as names for data representation abstractions were discussed in general terms in [section 1.6.5](#), and the reader is asked to review that material before beginning with this portion of the text.

Good programs are versatile, and give different results depending on the data being fed into them (even if the result is electronic indigestion). For that reason, programs must make extensive use of the computer's memory to store data and partial results, and to retain final answers pending the time when they will be printed out or otherwise communicated back to the real world. If, for each machine, (and each piece of code) the programmer had to write code to find available memory locations for data and then store things in those locations, the task would be unmanageable.

Fortunately, any good programming language has a facility to control all this work--all the programmer has to do is give a name to the pigeonhole where the data is to be stored, and the compiler automatically generates code to allocate the right amount of memory, give it the desired name, find it again whenever the name is subsequently used, and store values there whenever the program demands. In Modula-2 programs, it is necessary to give variables their name and type before the program code itself begins. For example, in the last program, the line

```
VAR
  base, exponent, counter, result: CARDINAL;
```

causes code to be generated to set aside enough memory to store the values of the four named variables. It also ensures that their type will be **CARDINAL**. In all subsequent references to these, code is generated to find the same location by its name. For instance, when the code for the line

```
result := result * base;
```

is executed, the locations named by *result* and *base* are looked up and the values retrieved. The two values are multiplied, and the answer is stored back in the location *result*. Likewise, the code for the line

```
counter := counter + 1;
```

causes the number stored at the location named by *counter* to be increased by one. The following definitions are useful at this point:

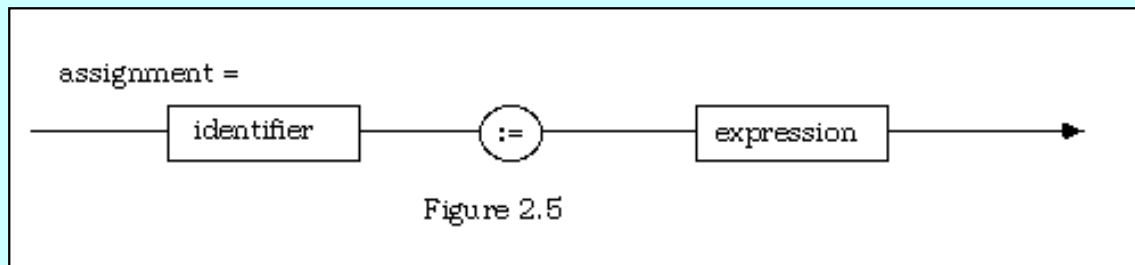
A Modula-2 variable is a name for a memory location, the contents of which can be changed by a program.

The symbol `:=` is called the assignment operator, and is the means by which the name on its left is given to the value on its right.

The difference may seem unimportant, but this last definition is technically more correct than saying that the value is assigned to the name, though the distinction is not always carefully maintained, even in this text.

NOTE: There should be a space before and a space after an assignment operator.

The assignment also illustrates the simplest kind of statement--the *assignment* statement. Its railroad diagram is shown in figure 2.5.

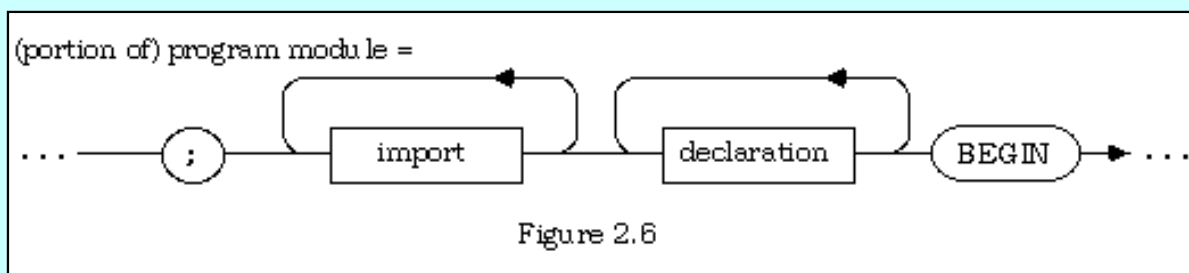


2.5.1 Simple Variable Types

In some computer notations, variables come into existence at the time they are first *used* in the program. Moreover, the type of contents the variable may have will be determined by the context in which it is used (these may, for instance, be alphabetic characters instead of numbers). Modula-2, on the other hand, is what is known as a *strongly typed language*. This means that every variable is of a particular kind or type and only values of the same type can be assigned to it.

A Modula-2 variable is given its name and type, and memory is set aside for the value under a VAR heading. This is called declaring the variable. The declaration of a variable must include its type. The entire section of the program Module preceding the word BEGIN, excluding any import statements, is called the declaration part.

This leads to a revised diagram of a program module. (Figure 2.6)



Once the type of a variable has been appropriately declared, the variable may be assigned only values of that type. For instance, only numbers such as -30, 0, 45, 62, etc. can be assigned to a variable of type INTEGER and only single characters such as 'c', '*', 'H', or '5' can be assigned to a variable of type CHAR (short for Character). Some rules for type declarations must also be observed. For example, in a declaration such as:

VAR

```
firstNumber : INTEGER;  
ch : CHAR;
```

NOTES: 1. VAR is a reserved word which must be capitalized.

2. Each declaration of another type must be separated from the ones before with a semicolon.

3. There should be spaces before and after each colon and a new line should be started after the VAR and before any actual declarations. (This is not absolutely required, but it makes the program look neater.)

This declaration creates the identifiers *firstNumber* and *ch*, with their types fixed for the balance of the program. In subsequent statements, an assignment like *firstNumber* := 'M' or *ch* := 5 would be rejected by the compiler as being in "type conflict." That is, one is not allowed to give a name of one type to a value of another. If there are several variables of the same type to declare all at once, they may be separated by commas. Thus,

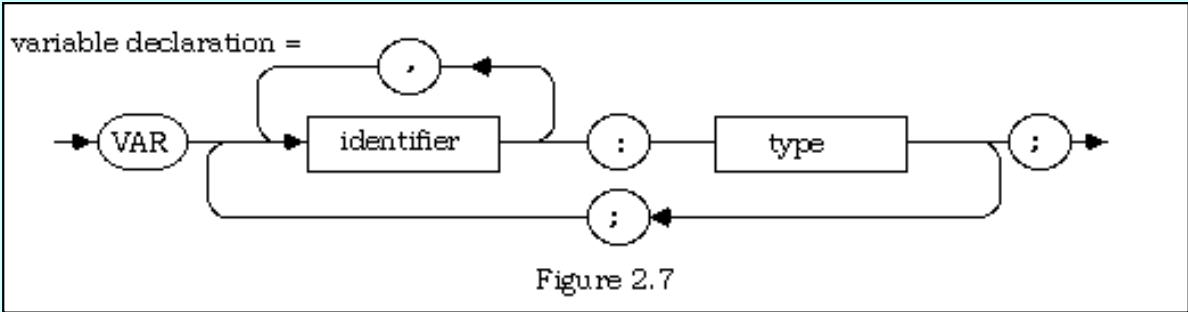
VAR

```
counter, size, length, int : CARDINAL;
```

has the same effect as the declaration below and conserves space in the source file.

```
VAR  
  counter : CARDINAL;  
  size : CARDINAL;  
  length : CARDINAL;  
  card : CARDINAL;
```

The syntax of the declaration is diagrammed in Figure 2.7.



The values a variable can take on are not only limited by its type--CHAR for instance, is restricted to single characters--but also by a predefined range. This is *not* a restriction imposed by Modula-2 as a programming notation, but by each individual computer and/or its operating system. In some older systems, the type CARDINAL, for instance can range only from 0 through 65535. The similar type INTEGER on such systems ranges from -32768 through 32767. In newer systems, these are 0 through 4294967295 for CARDINAL's and the INTEGER range is from -2147483647 through 2147483647.

- NOTES:** 1. CARDINAL, INTEGER and CHAR are built-in identifiers. As detailed later (section 2.5.2), it is unwise to attempt to reuse these names for program identifiers.
2. These limits are typical of certain small computers and can in some circumstances be very restrictive. Some systems offer a much larger range for CARDINAL and INTEGER. (See chapter 15 for a fuller discussion of the reasons these particular numbers arise as limits.) The reader must obtain information for the actual system being used from the implementation manuals, or by experimentation.
3. Alternatively, many versions of Modula-2 have one or both of the nonstandard types named by the built-in identifiers LONGCARD and LONGINT. The range of these two data types is much greater than that of the regular types, as each has more memory set aside for its data.
- Although one cannot assign INTEGER or CARDINAL variable names to CHAR values, or vice versa, it is possible to freely assign CARDINAL and INTEGER names to each others' values, but only in the overlapping part of their domain.

Example:

Assuming the ranges given above, suppose one had:

```
VAR  
  firstInt, secondInt : INTEGER;  
  firstCard, secondCard : CARDINAL;  
  ck, ch : CHAR;
```

Then the following are correct:

```
firstInt := -50;  
secondInt := 4500;  
firstCard := 62500;  
ch := 'X';
```



```
ck := "m"; (* single or double quotes allowed *)
secondCard := secondInt; (* This is okay. *)
firstCard := firstCard + 3000; (* This is almost too big for some older systems.
*)
```

and, the following are incorrect:

```
firstInt := firstCard; (* error when code is executed. *)
secondCard := firstInt; (* assigning negative to a cardinal. *)
ch := 'ab'; (* This won't work; 2 characters. *)
```

NOTE: Creating a variable name and giving it a type via the VAR statement does not also give it a value. Until the first assignment statement for that variable is encountered, it is undefined.

Assigning the first known value to a variable is called initializing it.

To illustrate the last point, consider the calculation section in the last major example:

```
(* calculation section *)
result := base; (* initially, set the result to the base *)
counter := 1; (* What if this initialization is omitted? *)
WHILE counter < exponent
DO
    result := result * base; (* multiply base enough times *)
    counter := counter + 1
END;
```

Suppose that the line initializing *counter* were left out. When the code entered the WHILE loop, the value of *counter* would be whatever happened to be in that piece of the machine's memory before the program began to run. It may already be greater than *exponent*, in which case the loop will not be executed even once. It may be some value between one and *exponent* in which case a further computation will be made, but the wrong number of multiplications will be performed. Assuming, say, that there are 65536 possible values for a CARDINAL, and *counter* has one of these chosen at random, the probability that the program would produce the correct result is 1/65536 or 0.000153. Since the range for CARDINAL is much larger on many systems, the actual situation is probably much worse than this. Notice that similar incorrect results can also be obtained by leaving out the line initializing *result*.

Moral: always initialize variables before expecting them to have any particular value.

2.5.2 Standard Identifiers

It is important to note that since the capitalized words CARDINAL, INTEGER, and CHAR name types, they too are identifiers. Like reserved words (VAR, IMPORT, MODULE, etc.) they are part of (are built into) the Modula-2 notation rather than imported from a library.

A Modula-2 Standard Identifier is a name that is built-in to the notation. It must be written entirely in capital letters.

Though like reserved words in that they are built-in to the notation and must be capitalized, standard identifiers differ from reserved words in two important respects:

1. They are indeed names, rather than structural punctuation markers.
2. They may be reused for other purposes, as unwise as this may be.

Thus,

```
VAR  
  IMPORT : CARDINAL ;
```

is an error the compiler will report, but

```
VAR  
  CARDINAL : INTEGER ;
```

may be foolish, for it redefines the identifier `CARDINAL` as a variable, cutting off access to the standard Modula-2 type `CARDINAL`, but it is permitted.



2.6 Literals and Constants

2.6.1 Literals

Sample programs thus far have contained a number of instances where some value was typed literally. This has been true of all the items written out by `WriteString` statements thus far. In

```
WriteString ("Hi there.")
```

the quoted string *Hi there* is called a string literal. Likewise in the assignment

```
counter := 1;
```

the value *1* on the right hand side of the assignment statement is a cardinal literal.

An entity specified by its value, rather than by a pre-declared name, is called a literal.

More accurately, the symbols in the string *Hi there* and the digit in the numeral *1* are not values themselves, but symbolic encodings of those values. (The values themselves are mental abstractions, and *1* is a symbol or numeral for the idea, not the idea itself.) Technically that is, literals are also names. This observation prompts the following more accurate (if perhaps harder to follow) definition:

A literal is any entity whose name is an encoding of its value.

Examples:

```
firstInt := 5;
secondInt := firstInt + 7;
character := 'C';
```

In these examples, the symbols *5*, *7*, and *'C'* are all literals. The compiler convert such literals into the correct form necessary to do the assignment in each case. Note that there is a type implied in a literal, even if the type is ambiguous. Thus, some literal assignments can be made more than one way; others cannot be done at all. If *card* is of type cardinal, and *int* is of type integer, then:

```
card := 7;
int := 7;
```

are both correct, as the literal 7 can be interpreted as either type. However,

```
card := -7;
```

is clearly incorrect, and the compiler can catch this error without having to postpone checking until the program is run.

2.6.2 Constants

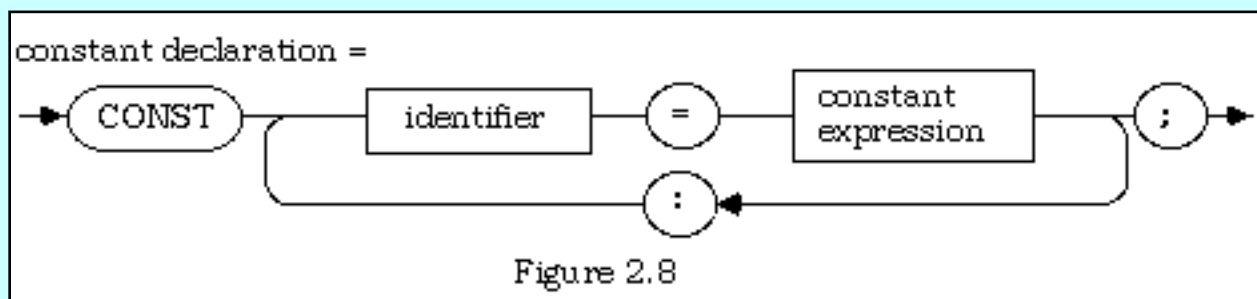
These were discussed in general terms in [section 1.6.5](#) along with variables. Suppose that in the course of the sample program, *Powers* the exponent was always equal to five. Rather than read this number in from the keyboard every time, or even write the variable assignment

```
exponent := 5;
```

in the program, it may be desirable to name it as a constant.

A Modula-2 constant is an identified value that remains the same throughout a program; that is, it is not subject to later re-assignment.

Constants are declared in a manner similar to variables. Unlike variables, they simultaneously receive a (fixed) value. For instance, to use the above number as a constant, the identifier exponent would be removed from under the VAR declaration and placed under a CONST declaration instead. The syntax for CONST declarations is shown in Figure 2.8.



```
CONST (* Another reserved word *)  
  exponent = 5;
```

```
VAR  
  base, counter, result: CARDINAL;
```

This has two potential advantages: the constant is changed into the correct internal form for the computer's use only once rather than several times. This may gain some speed but, more important, if it

is later decided that the number in question should be *six* rather than *five*, it need be changed only once up at the beginning of the program, rather than several times throughout, with the possibility of missing one or two instances.

Once constants have been declared, they can be used wherever the original literal could be used because the type of the constant is checked before making an assignment. If it is not compatible, an error is generated. Moreover, the type does not need to be explicitly declared in the CONST statement because it will be assigned automatically. One cannot therefore get away with writing *character := 'C' + amount* where *character* is of type CHAR and *amount* equals 5 (a constant), because the compiler records that *amount* can only be of type INTEGER or CARDINAL. The compiler also will automatically treat all negative whole number constants as being of type INTEGER. Thus,

CONST

```
number = -4;  
ten = 10;  
space = " ";
```

creates the INTEGER constant *number*, the CARDINAL/INTEGER (whole number type) constant *ten*, and the CHAR constant *space*.

NOTE: The identifiers of constants, unlike those of variables, are *equated* to their values rather than being assigned to them. This might not seem like much of a difference, especially when the same kinds of names can be given to both, but Modula-2 forces the distinction to be made by requiring the "=" symbol instead of the ":=" operator. Confusing these two is a common beginners' error.

[Contents](#)

2.7 Expressions for Constants and Variables

This material follows up on the general comments about expressions made in [section 1.7](#). Here, the specific concern is the kinds of things one is allowed to write on the right-hand-side of a Modula-2 assignment statement. This section will be concerned only about type INTEGER and CARDINAL. Additional expressions that are appropriate for other data types will be discussed later.

A Modula-2 expression is a combination of literals, constants, and variables using addition, subtraction, multiplication, division, and/or such other operations as may be defined for the type of entity being combined.

*In an expression, symbols such as +, -, and * are called operators and the entities being combined are called operands.*

Here are some examples of valid expressions:

```
4 + 5
7 - firstNum
secondNum + thirdNum - fourthNum
```

If *firstNum* and *fourthNum* are large enough cardinals the last two may cause errors at runtime. As in other notations, multiplication is indicated by an asterisk.

```
3 * 5          (* three times five *)
4 * num        (* one cannot write 4num, 4(num) or 4  num *)
```

Division of INTEGER or CARDINAL values uses the / or DIV operators. The answer will have any fractional part removed, so that it will still be an entire number of the same type. For both / and DIV, the dividend (left operand) may be negative, but DIV requires that the divisor (right operand) be positive. The / operator simply does a division and throws away the fractional part. However, the DIV operator has the effect of reducing the dividend to the *next lower* multiple of the divisor first, then dividing. The effect is the same for positive dividend, but not for negative ones.

```
7 / 2          (* result:  3 *)
9 DIV 4        (* result:  2 *)
-35 / 2        (* result: -17 *)
-35 DIV 12     (* result: -3  first reduce to -36, then divide *)
-42 / 10       (* result: -4 *)
-42 DIV 10     (* result: -5  first reduce to -50, then divide *)
7 / -2        (* result: -3 *)
9 DIV -2       (* result:  the compiler reports an error *)
-15 / -3       (* result:  5 *)
-12 DIV -4     (* result:  an error *)
card / 0       (* result:  a divide-by-zero error *)
int DIV 0      (* result:  a divide-by-zero error *)
```

NOTE: Care must be taken to ensure that the result of an arithmetic expression is neither too large nor too small to fit in the allowable range for the type of the variable one is attempting to assign it to.

An attempt to assign to a variable name a value which is larger than the maximum allowable, or smaller than the minimum allowable is said to overflow or underflow, the variable respectively.

That is, the expressions (a - b) and (a + b) may not be valid CARDINALs, even though both a and b have been correctly assigned CARDINAL values.

Example:

Suppose one had:

```
VAR
firstCard, secondCard, thirdCard : CARDINAL;
firstInt, secondInt : INTEGER;
```

Then these assignments are correct:

```
firstCard := 6000;
secondCard := 5000;
firstInt := -4000;
thirdCard := firstCard + secondCard; (* no problem here *)
```

and these may cause overflows, depending on the respective ranges:

```
thirdCard := firstCard * secondCard;
(* answer may be too big for CARDINAL *)
secondInt := firstCard * 30;
(* answer may be too big for INTEGER *)
```

WARNING: Compilers are supposed to generate code to check at run time for overflows and underflows of CARDINAL/INTEGER (and similar) assignments. However, there are Modula-2 implementations in which this is not done, and this fact could make errors difficult to track down.

Another pair of operators worth mentioning here are REM and MOD. On the one hand, $c := a \text{ DIV } b$ and $c := a / b$ assign only the whole number part of the appropriate division to c, with the remainder being discarded. REM and MOD do the opposite, discarding the quotient to the division and retaining only the remainder. REM works for both positive or negative divisors and dividends, but MOD requires that the divisor be positive. REM produces the remainder of a / division, and MOD the remainder of a DIV division. Therefore, the results are the same for positive dividends, but different for negative ones. A REM result always has the same sign as the dividend, but a MOD result is always positive (the distance from the next lower multiple of the divisor to the dividend.)

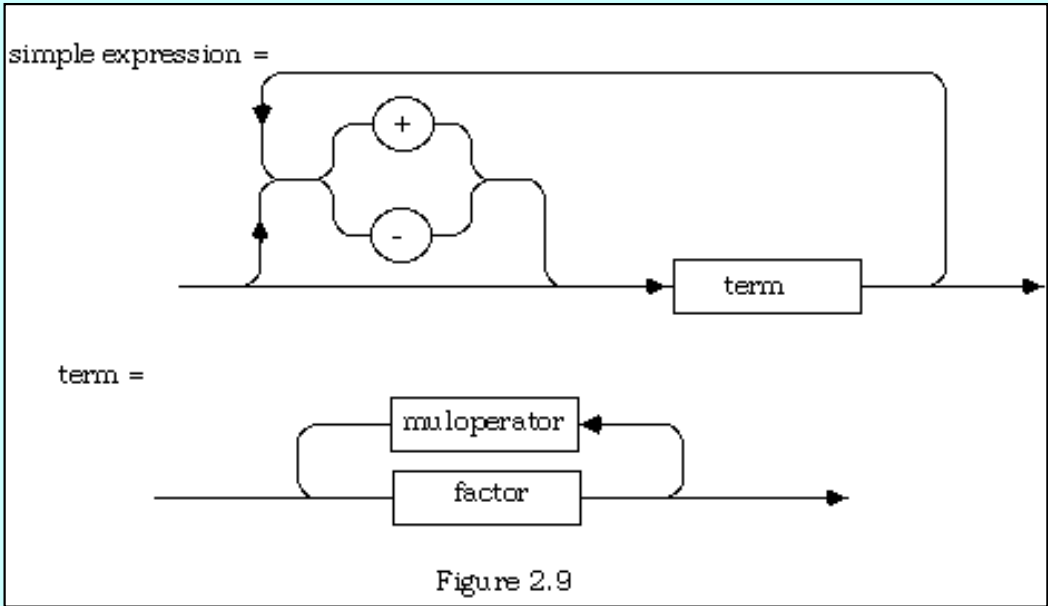
Examples:

```
4 REM 2 produces 0
4 MOD 2 produces 0
12 REM 5 produces 2
12 MOD 5 produces 2
-22 REM 6 produces -4
-22 MOD 6 produces 2 (* distance from -24 to -22 *)
32 REM -3 produces 2
32 MOD -3 produces an error
49 REM 0 produces an error
49 MOD 0 produces an error
```

NOTES: 1. Another way to obtain a MOD b is to compute a REM b first. Then, a MOD b = a REM b if a REM b = 0, and, a MOD b = b - a REM b if a REM b < 0

2. REM, MOD and DIV are all reserved words, as are all words indicating an operation.

Parts of an arithmetic expression that are joined by a multiplication or division are called *factors* and those joined by an addition or subtraction are called *terms*. More complex expressions can be formed by combining factors and terms. Figure 2.9 is a diagram of a simple expression. Note that it may start with a "+" or a "-" and that a "muloperator" can be *, /, REM, DIV or MOD (thus far.)



Expressions can be used anywhere on the right-hand side of a variable assignment (`a := a + 2`) or a constant equate (`octave = 2 * unison`). To further illustrate the latter, suppose three constants were used in a program, namely 20, 40, and 70, and the programmer knew that even if these changed, the second would always be twice the first, and third would be thirty more than the second. The constants could be declared as:

```

CONST
firstConst = 20;
secondConst = 2 * firstConst;
thirdConst = secondConst + 30;

```

2.7.1 Precedence in Modula-2

Refer to [section 1.7.1](#) for a general discussion of this topic, where it is explained that Mathematicians adopt a convention or set of rules to evaluate otherwise ambiguous expressions such as:

`4 + 5 * 8` or `6 - 12 / 2` or `(12 - 5) (3 + 9) - 4 (6 - 3 / (-1))`

Modula-2 is capable of following these standard rules for evaluating arithmetic expressions. That is, multiplication and division are performed before addition and subtraction, and parentheses can modify this order. This is not the case with many calculators, which evaluate expressions as they are entered. Here are a number of assignments, with the results shown at right.

<code>firstNum := 4 + 5 * 6</code>	Answer: 34
<code>secondNum := 4 DIV 3 + 5 MOD 4</code>	Answer: 2
<code>thirdNum := (3 - 4) * 6</code>	Answer: -6
<code>fourthNum := 9 * (5 - 20 DIV 3)</code>	

Of course, both *thirdNum* and *fourthNum* must be of type `INTEGER`, or the last two assignments will fail to work. Notice that the "*" operator is necessary in the last case; one cannot simply write `fourthNum := 9(5 - 20 DIV 3)`, using the parenthesis to

imply multiplication.

2.7.2 Mixed Expressions and Modula-2

In Modula-2, mixed expressions, say, those involving both INTEGER and CARDINAL types, are forbidden. That is, in the terminology of [section 1.7.2](#), INTEGER and CARDINAL are not expression compatible, even though (over their common range) we have seen that they are assignment compatible. For instance, if *firstCard* is of type CARDINAL and *firstInt* and *secondInt* are both of type INTEGER, then it is illegal to write either:

```
firstInt := firstCard + secondInt;  
firstCard := firstCard + firstInt;
```

because the right-hand side of the expression contains two different data types. The way to get around this, if necessary, is to convert the value of one of the members on the right to the data type of the other as follows:

```
firstInt := VAL (INTEGER, firstCard) + secondInt;  
firstCard := firstCard + VAL (CARDINAL, firstInt);
```

Of course, the numbers being converted must be in the correct range for the other variable, or there will be an overflow error. Conversions using VAL will be covered in more detail in [section 2.10.1](#).

[Contents](#)

2.8 Simple Output Methods

Once the programmer has the means to deal with numbers within programs, it also becomes necessary to print the answers to the screen where they can be seen by the program operator. The sample program module *powers* in section 2.3 contained the statement

```
WriteCard (exponent, 0);
```

for writing out the value of *CARDINAL* to the screen. There is also a *WriteInt* statement and both are, like *ReadCard* and *ReadInt*, imported from *SWholeIO* (Or, in non ISO Modula-2, from *InOut*). The second number in the *WriteInt* and *WriteCard* statements gives the size of the space to write in. Any extra spaces not needed for the number itself are placed to the left of the number. If not enough spaces are provided for, exactly the correct number will be taken. We say that the numbers are *right justified* in the specified field. That is,

```
WriteString ("the number is");  
WriteCard (6, 0);
```

produces the output

```
the number is 6
```

with one space before the 6. (Zero room provided is handled as a special case in ISO standard Modula-2 and a single space is prefixed. In non ISO standard versions, there may be no space at all written in this situation.) The statements

```
WriteString ("the number is");  
WriteCard (6, 3);
```

produce the output

```
the number is  6
```

with two spaces before the six, and the statements

```
int := -13;  
WriteString ("the number is");  
WriteInt (int, 4);
```

produce the output

```
the number is -13
```

with a single space, because one of the four spaces allowed for is needed for the negative sign. Thus, the second number in *WriteCard* and *WriteInt* allow for some formatting of the output number. On the other hand, if one does not care how many spaces are used, the second number in *WriteInt* or *WriteCard* should be zero. This was done in the *powers* example, where the space between the strings and the numerals was created by putting spaces into the strings, rather than having them written by *WriteCard*. This method is preferred when the number of digits in the answer may vary and works in both ISO and non-ISO libraries.

```
(* display the result *)
WriteCard (base, 0);
WriteString ( " raised to the power ");
WriteCard (exponent, 0);
WriteString ( " equals ");
WriteCard (result, 0);
WriteLn;
```

To illustrate, consider a program designed to print out a small table of cardinals (1 through 10) with their squares and cubes. The planning will not be given in detail, but the algorithm used can be expressed in pseudocode as:

```
print table headings
set base to 1
while base <= 10
  print base
  print square of base
  print cube of base
  increase base by one
end while
```

If this algorithm is implemented as:

```
MODULE SquareCube;
(* by R. Sutcliffe *)

FROM STextIO IMPORT
  WriteString, WriteLn;
```

```

FROM SWholeIO IMPORT
    WriteCard;

VAR
    base: CARDINAL;

BEGIN
    (* Write column headings *)
    WriteString ("Number ");
    WriteString ("Square ");
    WriteString ("Cube");
    WriteLn;
    base := 1;
    (*on each line print first, second, and third powers of the base*)
    WHILE base <= 10
        DO
            WriteCard (base, 0);
            WriteCard (base * base, 0);
            WriteCard (base * base * base, 0);
            base := base + 1;
            WriteLn;
        END;
    END SquareCube.

```

the output will look like this:

```

Number Square Cube
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

```

because no spacing has been included. (In many non ISO versions, the numbers will be jammed together with no spaces at all, but the ISO standard requires one space to be output before the number in the special case that the room allowed is zero.) Putting one extra space into the WriteCard statements is not sufficient, for only the one digit numbers will then be properly separated. Instead, the program should lay

out a well-spaced table. Moreover, this is a good place to use a constant, for the upper limit of the table may well be subject to change. If it is re-written as:

```
MODULE SquareCubeFormatted;
(* by R. Sutcliffe *)

FROM STextIO IMPORT
    WriteString, WriteLn;

FROM SWholeIO IMPORT
    WriteCard;

CONST
    maxToDo = 10;

VAR
    base: CARDINAL;

BEGIN
    (* write column headings *)
    WriteString ("          Number");  (* allow 15 spaces for each *)
    WriteString ("          Square");
    WriteString ("          Cube");
    WriteLn;
    base := 1;
    (* on each line print first, second, and third power of base *)
    WHILE base <= maxToDo
        DO
            WriteCard (base, 15);
            WriteCard (base * base, 15);
            WriteCard (base * base * base, 15);
            base := base + 1;
            WriteLn;
        END;

END SquareCubeFormatted.
```

the output is

Number	Square	Cube
1	1	1
2	4	8
3	9	27
4	16	64

5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

There is also a procedure in *STextIO* (or in *InOut*) to output CHAR variables. This is the *WriteChar* statement (called *Write* when found in *InOut*), and it too can output either literal characters, or those designated by constant or variable names. Here is a brief example:

```
VAR
    ch : CHAR;

BEGIN
    ch := 'R';
    WriteChar (ch);
    WriteChar ('. ');
    WriteChar ('S');
    WriteChar ('. '); (* outputs my initials *)
```

and, another:

```
MODULE GetIt;
(* by R. Sutcliffe *)
(* prints out a little joke *)

FROM STextIO IMPORT
    WriteString, WriteLn, WriteChar;

FROM SWholeIO IMPORT
    WriteCard;

CONST
    wye = "Y"; (* Character constants are allowed. *)
    four = 4;

BEGIN
    WriteChar (wye);
    WriteChar (wye);
    WriteString ("UR ");
    WriteChar (wye);
    WriteChar (wye);
```

```
WriteString ("UB ");
WriteChar ("I");
WriteChar ("C");
WriteString (" UR ");
WriteChar (wye);
WriteChar (wye);
WriteCard (four, 2);
WriteString (" ME!");
WriteLn;
END GetIt.
```

the output of which is

```
YYUR YYUB IC UR YY 4 ME!
```

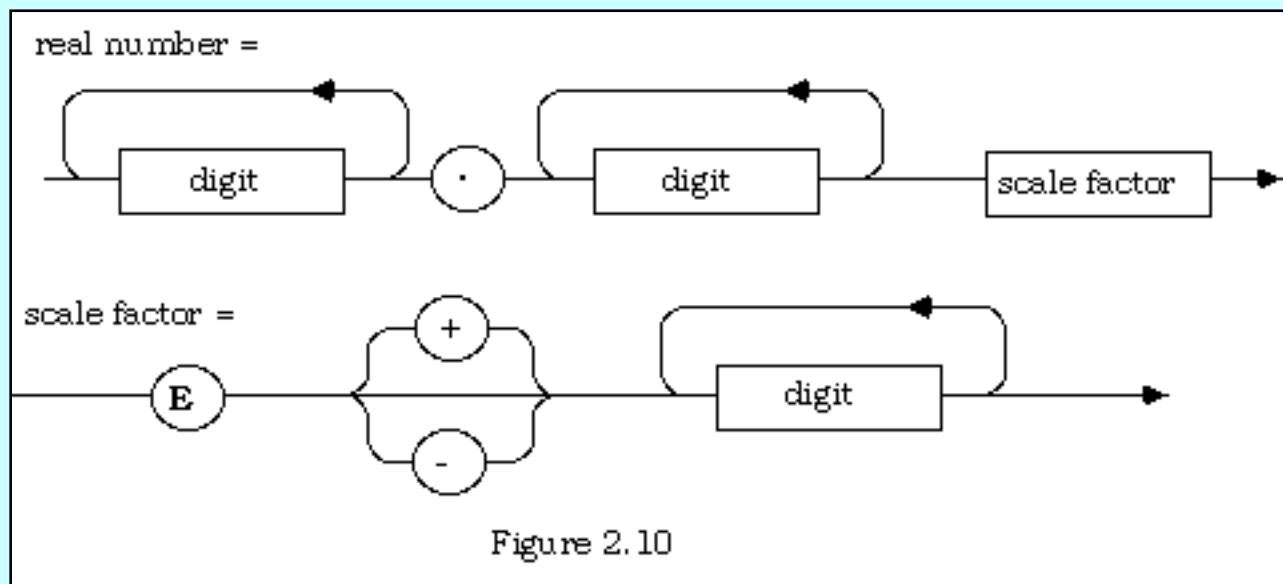
[Contents](#)

2.9 REAL Variables

It must be made clear at the outset that a REAL variable is no more "real" than an INTEGER, at least not in the sense that the latter is, say, a figment of the imagination. Mathematically, the integers can be regarded as a subset of the reals, though this is not true in Modula-2, because the two are different types.

*A Modula-2 **REAL** number is one that is expressed as a series of significant figures including a decimal point, and optionally followed by a scale factor (power of ten).*

Figure 2.10 illustrates this.



The notation used for REALs is the standard or scientific one, with a slight change to make it more computer readable. Consider the following representations:

<u>Decimal</u>	<u>Scientific</u>	<u>Modula-2</u>
2000	2.000×10^3	2.000E+03
0.058	5.8×10^{-2}	5.8E-02
	6.023×10^{-23}	6.023E-23

NOTE: For such simple exponents as in the first two examples, the numbers may also be entered for a Modula-2 program as 2000. and 0.058 respectively. That is, the scale factor indicator and digits after the decimal may not be always necessary, but the decimal point is. Note that one may not substitute a lower case *e* for *E* when writing literals in a program.

2.9.1 Real Operations

The operations and their precedence are the same for REALs as for INTEGERS and CARDINALs, except that REM, MOD and DIV are not defined and REAL division is indicated only with the / (slash).

```
firstReal := 5.0 / 3.0;           Answer:  1.66666
secondReal := 7.5 + 1.1 * 2.0;    Answer:  9.7
```

To output these answers to the screen, the appropriate *WriteReal* statement is imported as usual. One must import it from *SRealIO* (In non ISO standard versions, the library may be called *RealIO* or *RealInOut* or these items may simply be included in *InOut*.) As in the previous statements of this type, a number following the REAL to be printed specifies the size of the field of blanks in which it is to be right justified.

The following module is typical of many that can be written to do conversions or other computations using a well-defined formula and having a simple output. As its only purpose is to demonstrate output of reals, and few other new concepts are present, no detailed plan is provided.

Example:

Write a module to convert a 14 degree Fahrenheit temperature to Celsius units.

```
MODULE ConvertToCelsius;
(* by R. Sutcliffe *)

FROM STextIO IMPORT
  WriteLn, WriteString, WriteChar;

FROM SRealIO IMPORT  (* note separate import statements *)
  WriteReal;

CONST
  Fahrenheit = 14.0;  (* this can be changed *)
  conversion = 32.0;  (* difference between zero points *)
  ratio = 5.0 / 9.0;  (* ratio between Celsius and Fahrenheit *)

VAR
  Celsius : REAL;

BEGIN
  Celsius := ratio * (Fahrenheit - conversion);
  WriteReal (Fahrenheit, 12);
  WriteChar ('F');
  WriteString (" degrees equals ");
```

```
WriteReal (Celsius, 12);  
WriteChar ('C');  
WriteString (" degrees.");
```

END ConvertToCelsius.

The output from this program module is as follows:

```
14.000000000000F degrees equals -10.0000000000C degrees.
```

- NOTES:** 1. The number of significant figures and the maximum size of the exponent allowed depends on the computer system being used. Here, twelve figures have been printed. A typical size range for many small computers is about $10^{-308} < \text{real} < 10^{308}$, and similarly for negative reals.
2. There are other ways to format real number output that will be considered later.
3. The type **LONGREAL** is also available, and this extends the above range considerably.

Example:

Write a module to compute the area and circumference of a circle from a given radius.

```
MODULE CircleAreaCirc;  
  
FROM STextIO IMPORT  
    WriteLn, WriteString;  
  
FROM SRealIO IMPORT  
    WriteReal;  
  
FROM RealMath IMPORT  
    pi;  
  
CONST  
    twopi = 2.0 * pi; (* note the real constant expression *)  
  
VAR  
    area, circumference, radius : REAL;  
  
BEGIN  
    radius := 4.0;  
    (* calculate area and circumference *)  
    area := pi * radius * radius;  
    circumference := twopi * radius;  
    (* inform the user *)
```

```

WriteString ("A circle of radius ");
WriteReal (radius, 0); (* spaces provided in the string parts *)
WriteString (" units has a circumference of ");
WriteLn;
WriteReal (circumference, 0);
WriteString (" units");
WriteString (" and an area of ");
WriteReal (area, 0);
WriteString (" square units.");
END CircleAreaCirc.

```

The output from this program module is as follows:

```

A circle of radius  4.0000000 units has a circumference of
 2.5132740E+1 units and an area of  5.0265480E+1 square units.

```

2.9.2 The Format of Real Output

Sometimes a programmer wishes to print out real numbers, say to represent things like wages and bank balances in decimal or *fixed* form. At other times, the preference is for scientific or *floating point* notation. At still other times, engineering notation (in which the powers of ten are always multiples of three) is desired.

Fixed	Floating	Engineering
21421.5	2.14215E+4	21.4215E+3

Sometimes the number of significant figure needs to be specified. At other times it is the number of figures after the decimal that is of importance. For instance, when printing money values such as \$15.34, fixed notation with two decimal places is necessary.

In ISO standard Modula-2 the library *SRealIO* has three additional procedures besides *WriteReal* for writing out real numbers.

```

WriteFloat (real, sigFigs, width);

```

writes out the specified *real* in floating point form with leading spaces if required to make the total space taken equal to *width* characters. One leading space is used if the width is zero. (This is the same behaviour as the *width* parameter for *WriteCard* and *WriteInt*.) If the value of *sigFigs* is greater than zero, it specifies the number of significant figures in the result. If the user of this procedure specifies zero figures, the particular implementation is free to define the outcome, and the manuals will have to be consulted. A sign is written only for negative values, and the exponent part is only written if it is not zero. No decimal point is included if there are no figures after it to print. The code

```
WriteFloat (pi, 4, 0);  
WriteLn;  
WriteFloat (1000.0 * pi, 4, 10);  
WriteLn;  
WriteFloat (10000.0 * pi, 6, 10);  
WriteLn;  
WriteFloat (pi/10000.0, 2, 10);
```

produces the output

```
3.141  
3.141E+03  
3.14159E+04  
3.1E-04
```

```
WriteEng (real, sigFigs, width);
```

writes out the specified *real* in engineering form (exponents are multiples of three) with leading spaces if required to make the total space taken equal to *width* characters. The rest of the behaviour of *WriteEng* is the same as that of *WriteFloat*. The code

```
WriteEng (pi, 4, 0);  
WriteLn;  
WriteEng (1000.0 * pi, 4, 10);  
WriteLn;  
WriteEng (10000.0 * pi, 6, 10);  
WriteLn;  
WriteEng (pi/10000.0, 2, 10);
```

produces the output

```
3.141  
3.141E+03  
31.4159E+03  
310E-06
```

```
WriteFixed (real, place, width);
```

writes out the specified *real* in fixed point form (ordinary decimal form) and rounded off to *place* figures after the decimal, with leading spaces if required to make the total space taken equal to *width* characters. The rest of the behaviour of *WriteFixed* is the same as that of *WriteFloat*. The code

```
WriteFixed (pi, 4, 0);  
WriteLn;  
WriteFixed (1000.0 * pi, 4, 10);  
WriteLn;  
WriteFixed (10000.0 * pi, 6, 10);  
WriteLn;  
WriteFixed (pi/10000.0, 2, 10);
```

produces the output

```
3.1416  
3141.5930  
31415.930000  
0.00
```

Notice that the last number is zero to three figures, so that is all that gets written.

Returning to the idea of printing currency values, the code

```
WriteChar("$");  
WriteFixed(pi,2,5)
```

produces the output

```
$ 3.14
```

with one space before the leading digit because four of the five spaces are occupied.

```
WriteReal (real, width);
```

behaves like a call to *WriteFixed (real, place, width)*; if the number can fit in the width provided. The value of *place* is chosen to exactly fill the remaining field. If the width is insufficient, the call to *WriteReal* behaves like one to *WriteFloat (real, sigFigs, width)*; with a value of *sigFigs* at least one, limited to those that can be included in the given width along with the exponent and sign. The code

```
WriteReal (pi, 0);  
WriteLn;  
WriteReal (1000.0 * pi, 10);  
WriteLn;  
WriteReal (10000.0 * pi, 10);  
WriteLn;  
WriteReal (pi/10000.0, 10);
```

produces the output

```
3.1415927
3141.59270
31415.9270
3.1415E-04
```

This matter presented somewhat of a difficulty for early implementors of Modula-2, because no such flexibility was defined or even suggested by Wirth as part of the standard operating environment. Indeed, in some Modula-2 implementations, it may not be possible to achieve the goal of formatting reals in these ways without doing some very hard programming work.

In fact, *non-ISO* standard compliant versions of Modula-2 may provide something like:

```
WriteReal (real, width, n);
```

where *real* represents the value to be written, and the second number is the width of the space to write in as before. The meaning of the third number, in such versions may be either the number of significant figures of the real to print, or the number of decimal places of the real to print. The library location of the alternate *WriteReal* may vary in such versions. Moreover, some non-standard versions of *WriteReal* switch to scientific notation whenever a negative sign is placed in front of the third supplied number. Observe that the usual ordering of the second and third numbers in non-ISO versions is often the opposite to that in ISO standard packages.

NOTES: 1. The user documentation for the compiler package of a non-ISO standard implementation must be checked carefully to determine which version of the *WriteReal* procedure has been provided, and in what library it is located.

2. Where more than one *WriteReal* is available, a specific implementation may even give them different names and/or place them in different libraries.

[Contents](#)

2.10 Type Compatibility and REAL

In [section 2.7](#), it was observed that CARDINAL and INTEGER are assignment compatible, even if they are not expression compatible. Assignment compatibility is in this case a small concession to the normally strict rule against mixing types--these two are used in similar ways. It can be achieved easily because most computers also store values of these two types in similar ways. However, both the use and the storage of REAL values is quite different than for either INTEGER or CARDINAL values. Therefore one cannot in Modula-2 write either of

```
firstReal := -7;  
secondReal := firstReal + card;
```

where *firstReal* and *secondReal* are REAL and *card* is CARDINAL. That is:

In Modula-2 the type REAL is neither assignment compatible nor expression compatible with CARDINAL or INTEGER.

However, there is a way to convert numbers to and from REAL representation. If *re* is of type REAL and *ordnum* is of type INTEGER or CARDINAL, then

```
re := FLOAT (ordnum)
```

will safely convert *ordnum* to a REAL, and assign the resulting value to *re*. Likewise,

```
card := TRUNC (re)
```

will safely convert the whole number part of *re* to a CARDINAL and assign the resulting value to the variable *card*. Furthermore,

```
ordnum := INT (re)
```

will safely convert the whole number part of *re* to an INTEGER and assign the resulting value to the variable *ordnum*.

NOTES: 1. These two are examples of what are called functions, or, more properly in Modula-2 function procedures.

2. The above definitions hold for ISO standard Modula-2. In some older implementations of Modula-2 the FLOAT may only work on one of the types INTEGER or CARDINAL and TRUNC conversions may produce an INTEGER rather than a CARDINAL. There may not even be an INT function procedure.

3. These two operations allow the creation of mixed expressions. Naturally, FLOAT and TRUNC have the precedence of parentheses.

4. TRUNC does not round off to the nearest cardinal, it "hacks off" the decimal part instead.

Examples:

Assume *re* is REAL, *int* is INTEGER, and *card* is CARDINAL

<code>re := 2.6 + FLOAT (5);</code>	Answer: 7.6
<code>int := 26 - 2 * INT (re);</code>	Answer: 12
<code>card := 26 - TRUNC (re * 2);</code>	Answer: 11
<code>int := INT (-4.75) - 6;</code>	Answer: -10

2.10.1 Additional Conversion Functionality

Many computers have more than one format for storing real numbers. Typically, there is one for reals with a somewhat limited range of significant figures and exponent, and one or more additional types with a greater range for both. Because of this, ISO standard Modula-2 provides for a second real type called LONGREAL. The precision and the exponent range for this type must be at least as great as for the type REAL. Thus, if only one real type is available in the underlying machine, the two real types would essentially be the same. However, since this is implementation defined (and somewhat exceptional), the compatibility rule is:

The Modula-2 type REAL and the Modula-2 type LONGREAL are not compatible.

Thus, besides the conversion functions above, ISO standard Modula-2 adds LFLOAT to convert to the type LONGREAL. Since REAL values need conversion to LONGREAL as well, LFLOAT takes any of CARDINAL, INTEGER, or REAL. In addition, INT, TRUNC, and FLOAT can take any numeric type that need to be converted to INTEGER, CARDINAL, or REAL, respectively. Note that this behaviour cannot be counted on in non-ISO standard versions.

Examples:

Assume *int* is an INTEGER, *card* is a CARDINAL, *re* is a REAL and *lre* is a LONGREAL.

<code>int := INT (-3.5);</code>	Answer: -3 (works like TRUNC)
<code>lre := LFLOAT (2.7);</code>	Answer: 2.7 (* but stored as a LONGREAL *)
<code>lre := LFLOAT (-3);</code>	Answer: -3.0 (* stored as a LONGREAL *)
<code>re := FLOAT (-3);</code>	Answer: -3.0 (* stored as a REAL *)
<code>re := FLOAT (INT (5.1));</code>	Answer: 5.0
<code>card := TRUNC (-3.5);</code>	Answer: error; presumably need to use INT
<code>int := INT (lre);</code>	Answer: be careful; could overflow!

- NOTES:** 1. Nonstandard versions that add such types as LONGINT and LONGCARD may have additional conversion functions.
2. Many nonstandard versions add the conversion function CARD, to convert to CARDINAL, instead of or in addition to TRUNC.
3. In the ISO standard, TRUNC, INT, FLOAT, and LFLOAT are simply short notation for the corresponding use of VAL (mentioned briefly in [section 2.7](#)). That is,

INT (r) means VAL (INTEGER, r),

TRUNC (r) means VAL (CARDINAL, r),

FLOAT (i) means VAL (REAL, i), and

LFLOAT (i) means VAL (LONGREAL, i).

The Modula-2 function procedure VAL(newType,item) safely converts the value stored in item into a corresponding value of the type newType.

[Contents](#)

2.11 Interactive Programs

The sample programs given thus far all produced a single output. Their data was right in the code, either as literals that were written out, or as constants that were operated on to produce an output. It is all very well to have a program produce some answers, but communication ought to be a two-way street. Programs are much more versatile if they can print out messages to the user asking for input data, read the answers typed into variables, and then act on this data. The procedures to achieve the reading of data are also located in the modules *STextIO*, *SWholeIO*, and *SRealIO*. (Or from *InOut* and *RealInOut*) The following are among the entities available for `IMPORT`.

From *STextIO* or *InOut*

ReadChar--takes in one character and puts it into CHAR variable memory (If from *InOut*, called just *Read*)

From *SWholeIO* or *InOut*

ReadInt--accepts and assigns an INTEGER

ReadCard--does the same for a CARDINAL

From *SRealIO* or *RealInOut* (or possibly from *InOut*)

ReadReal--ditto for a REAL

Before giving examples, two definitions are in order:

A program that operates on a set of fixed data obtained either from within the program itself or from a file is said to be a batch style program.

A program that interacts with its user via prompts to obtain the data on which its execution depends is called interactive.

Here, for instance are two earlier batch style examples re-written in interactive style:

Example:

Write a module to convert user-supplied Fahrenheit temperatures to Celsius units.

```
MODULE ConvertToCelsiusI;
(* by R. Sutcliffe *)
(* to illustrate interactive style *)

FROM STextIO IMPORT
  WriteLn, WriteString, WriteChar;
(* The latter is called simply "Write" when in InOut. *)

FROM SRealIO IMPORT
  ReadReal, WriteReal;

CONST
  conversion = 32.0;  (* difference between zero points *)
  ratio = 5.0/9.0;    (* ratio between Celsius and Fahrenheit *)

VAR
```

```
Fahrenheit, Celsius : REAL;
```

```
BEGIN
```

```
  (* ask user for data *)
  WriteString ("What is the Fahrenheit value to convert? ");
  ReadReal (Fahrenheit);
  WriteLn;
  (* do the calculation *)
  Celsius := ratio * (Fahrenheit - conversion);
  WriteReal (Fahrenheit, 12);
  WriteChar ('F');
  WriteString (" degrees equals ");
  WriteReal (Celsius, 12);
  WriteChar ('C');
  WriteString (" degrees.");
```

```
END ConvertToCelsiusI.
```

The output from this program module is as follows: (user input in boldface)

```
What is the Fahrenheit value to convert? 14.0
14.0000000000F degrees equals -10.0000000000C degrees.
```

Example:

Write a module to compute the area and circumference of a circle from a user-supplied radius.

```
MODULE CircleAreaCircI;
```

```
  (* by R. Sutcliffe *)
```

```
  (* to illustrate interactive style *)
```

```
FROM STextIO IMPORT
```

```
  WriteLn, WriteString;
```

```
FROM SRealIO IMPORT
```

```
  ReadReal, WriteReal;
```

```
VAR
```

```
  radius, area, circumference : REAL;
```

```
CONST
```

```
  pi = 3.1415926;
```

```
  twopi = 2.0 * pi;
```

```
BEGIN
```

```
  (* ask user for data *)
```

```
  WriteString ("What is the radius of the circle? ==");
```

```
    (* fancier prompt *)
```

```
  ReadReal (radius);
```

```
  WriteLn;
```

```

(* calculation *)
  area := pi * radius * radius;
  circumference := twopi * radius;
(* output *)
  WriteString ("A circle of radius ");
  WriteReal (radius, 0);
  WriteString (" units has a circumference of ");
  WriteReal (circumference, 0);
  WriteString (" units");
  WriteLn;
  WriteString (" and an area of ");
  WriteReal (area, 0);
  WriteString (" square units.");
END CircleAreaCircI.

```

The output from this program module is as follows: (user input in boldface)

What is the radius of the circle? ==> **13.0**

A circle of radius 1.3000000E+1 units has a circumference of 8.1681404E+1 units
and an area of 5.3092914E+2 square units.

[Contents](#)

2.12 An Extended Example (Bank Interest)

Example problem:

Compute the amount there will be in my bank account after one year if the interest is compounded quarterly and any withdrawals I make are entered immediately after the interest is paid.

Example problem suitability:

The bank account example given above is well within the reach of a simple calculator, and is therefore of somewhat marginal suitability for computer solution. However, the solution does illustrate how a number of the programming ideas discussed in chapter one are actually implemented in Modula-2.

Example problem restatement:

Given: A bank account, possibly containing money.

To Do: Compute interest, make withdrawals every 3 months

Desired Result: final balance.

Formula: $\text{interest} = \text{principal} * \text{rate} * \text{time}$.

Example problem library use:

This problem has both input and output requirements. It will, therefore make use of the ISO standard Modula-2 library. Because of the withdrawals, four separate simple interest calculations will be made, rather than one compound interest calculation. No mathematical or financial libraries are required.

Example problem refinement:

1. Input Section obtain bank balance
 obtain interest rate
 obtain quarterly withdrawal amounts
2. Computation calculate interest each quarter
 subtract withdrawal each quarter
3. Output print out final balance

Second refinement of problem: (same numbers as above)

1. Ask the user for the initial balance
 Read the balance and assign the value to a real variable
 Ask the user for the interest rate
 Read the rate and assign the value to another real variable
2. set quarter to one
 while quarter is less than or equal to four
 For the current quarter

```

    Compute the interest using
        interest = principal * rate * time
    Add this to original balance
        balance = balance + interest
    Ask user for withdrawal amount
    Read the amount and assign the value to a real variable
    Subtract the withdrawal from last balance
        balance = balance - withdrawal
    add one to the quarter
end while

3. After four quarters
    Print out the final balance

```

Pseudocode:

In this refinement, only portions of the code that involve several steps, formulas, or computations (i.e. an algorithm) are detailed. In this example, the last natural language refinement is followed very closely, so this section might have been left out in this case, but it is included to make the model for students to follow as complete as possible.

```

quarter = 1
while quarter <= 4
    interest = principal * rate * time
    balance = balance + interest
    write message "what is withdrawal amount?"
    read (amount)
    balance = balance - withdrawal
    quarter = quarter + 1
end while

```

Example problem data table:

```

Variables:    principal, rate, interest, withdrawal--all reals
              quarter--a cardinal

Constants:    time = 0.25

Imports      from STextIO (InOut): WriteString, WriteLn, SkipLine
              from SWholeIO (InOut): WriteCard
              from SRealIO (RealInOut) ReadReal, WriteReal

```

NOTE: The purpose of *SkipLine* is to clear the end of line state that exists following one input line before attempting to read the next input line. This is necessary whenever inputs are separated by typing a carriage return. This was not previously needed, because only one input per program was being employed.

Example problem sample I/O:

```

Input:
What is the opening balance? 1000.00
What is the annual interest rate?
Enter as a decimal; Example: 6.5% is 0.065. 0.08
How much do you wish to withdraw in quarter number 1? 10.00
How much do you wish to withdraw in quarter number 2? 9.00

```

How much do you wish to withdraw in quarter number 3? **11.00**

How much do you wish to withdraw in quarter number 4? **6.00**

Output:

The final balance in your account is 1044.57

Example problem sample User Manual:

Description: Bank Interest is a small application designed to maintain a simple bank account record. It allows the user to specify the opening balance, and annual interest rate at the beginning of the program, and then to make quarterly withdrawals. Interest is calculated on the balance before the withdrawal.

Operation: If in an MS-DOS or UNIX environment, type the name of the program from the operating system prompt. If in a Graphics User Interface (GUI) environment, click twice in close succession on the icon representing the program. A banner will appear giving the name of the program and some information about the author. Following this the prompt

What is the opening balance?

will appear on the screen. Type in the amount that the bank account had at the start of the year. Following this, the prompt

What is the annual interest rate?

Enter as a decimal; Example: 6.5% is 0.065.

will appear on the screen. Type in the interest rate in the form shown. Now, for each quarter, a message like

How much do you wish to withdraw in quarter number 1?

will appear on the screen. Respond with the amount being withdrawn for that quarter in the form 10.23. That is, give the number in decimal form and without a dollar sign. The program will finally respond with a message like:

The final balance in your account is xxx.xx

Example problem code:

```
MODULE BankInterest;
```

```
( *  
    Name: Nellie Hacker  
    Student Number: 922001  
    CMPT 141 Fall 1991  
    Assignment #1  
    Bank Interest Computation  
*)
```

```
FROM STextIO IMPORT  
    WriteString, WriteLn, SkipLine;
```

```
FROM SWholeIO IMPORT  
    WriteCard;
```

```
FROM SRealIO IMPORT  
    ReadReal, WriteFixed;
```

```

CONST
    time = 0.25; (* that is, one quarter *)

VAR
    principal, rate, interest, withdrawal : REAL;
    quarter : CARDINAL;

BEGIN
    (* print informative material *)
    WriteString ("This program computes interest on an account");
    WriteLn;
    WriteString ("quarterly, and then allows for withdrawals.");
    WriteLn;
    WriteLn;
    WriteString ("It was written by");
    WriteLn;
    WriteString ("Nellie Hacker");
    WriteLn;
    WriteString ("for CMPT 141");
    WriteLn;
    WriteString ("Assignment #1 Due 1992 09 15");
    WriteLn;
    WriteLn;
    (* get user input *)
    WriteString ("What is the opening balance? ");
    ReadReal (principal);
    SkipLine;
    WriteLn;
    WriteString ("What is the annual interest rate? ");
    WriteLn;
    WriteString ("Enter as a decimal; E.g.: 6.5% is 0.065. ==");
    ReadReal (rate);
    SkipLine;
    WriteLn;

    quarter := 1;
    WHILE quarter <= 4
        DO
            interest := principal * rate * time;
            (* compute new principal *)
            principal := principal + interest;
            WriteString ("What is the withdrawal in quarter number ");
            WriteCard (quarter, 0);
            WriteString ("? =="); (* complete a readable prompt *)
            ReadReal (withdrawal);
            SkipLine;
            WriteLn;
            (* recompute principal *)
            principal := principal - withdrawal;
            quarter := quarter + 1;
        END;
    (* print output *)
    WriteString ("The final balance in your account is $ ");

```



```
WriteFixed (principal, 2, 0);
WriteLn;
END BankInterest.
```

Example problem actual output (user input in bold):

This program computes interest on an account quarterly, and then allows for withdrawals.

It was written by
Nellie Hacker
for CMPT 141
Assignment #1 Due 1992 09 15

What is the opening balance? **1000.00**
What is the annual interest rate?
Enter as a decimal; E.g.: 6.5% is 0.065. ==> **11.00**
What is the withdrawal in quarter number 2? ==> **5.00**
What is the withdrawal in quarter number 4? == "state" the input achieves. *SkipLine* removes any more input on the current line until this state is reached; then it passes the state by and readies the next "line" of data for reading.

2.12.1 Input in non ISO Versions

Non ISO standard libraries may employ a procedure *ReadLine* for essentially the same purpose as *SkipLine*. On the other hand, they may simply regard the end-of-line as a character to be read, rather than as a state to be in, and therefore must handle this differently. There are two possibilities:

1. *Readxx* statements may be set up to skip any "white space" (including carriage returns) before doing their read. If this is the case, the carriage returns do not need to be read separately, for they will be taken out of the input and thrown away by the next read statement.

2. They may require the user program to handle the carriage return as a character, but not supply any special procedure for doing so. In this event, the user program should import the procedure *ReadChar* or its equivalent *Read* from *InOut*, declare a character variable, say

```
VAR
  cr : CHAR;
```

and replace all occurrences of *SkipLine* in the above program with

```
Read (cr);
```

to read off and discard the carriage return.

In any event, the matter cannot simply be ignored, or the program will not operate correctly.

[Contents](#)

2.13 Chapter Summary

This chapter covered these topics:

- what a Modula-2 program is
- what the basic structure of a Modula-2 program is
- what a module is
- how to give correct names (identifiers) to Modula-2 entities
- how to write comments
- how to distinguish among literals, variables and constants
- how to assign values to variables and equate values to constants
- about the types INTEGER, CARDINAL, CHAR, and REAL
- about expressions, precedence and some type conversions
- how to talk to a program and make it talk to a user

It included discussion of the following Modula-2 built-ins:

Reserved Words	Standard Identifiers
BEGIN ... END	CARDINAL
CONST	CHAR
DIV	FLOAT
DO	INT
FROM ... IMPORT	INTEGER
MOD	LONGREAL
MODULE	LFLOAT
REM	REAL
WHILE	TRUNC
VAR	VAL
	CARD*
	LONGCARD*
	LONGINT*

* indicates a non-standard word that *may* be present in some versions.

Symbols Imports

From STextIO (InOut):	ReadChar (Read)
+	WriteChar (Write)

-		WriteLn
*		WriteString
=		SkipLine
/	From SWholeIO (InOut):	ReadCard
E		WriteCard
:=		ReadInt
		WriteInt
	From SRealIO:	ReadReal
	(RealInOut)	WriteReal
		WriteFloat (ISO only)
		WriteFixed (ISO only)

indicates the usual name in non ISO compliant versions using InOut.

[Contents](#)

2.14 Assignments

Questions

1. What are the functions of: (a) an editor; (b) a compiler; (c) a linker; (d) an interpreter? Determine whether linking is a separate step on your system.
2. Why is it that input and output is not in the Modula-2 language proper, but has to be imported from a separate library?
3. What is the general format of a Modula-2 program?
4. Determine whether any special characters not normally legal in Modula-2 identifiers are allowed on your system.
5. Determine from the manuals and write down in a handy place the limits your system imposes on the numeric types `CARDINAL`, `INTEGER`, `REAL`, and `LONGREAL`. Make a note of any types such as `LONGINT` and `LONGCARD` that may be provided, and note their limits as well.
6. Determine from the manuals and write down in a handy place the exact syntax and semantics (meaning) of the I/O procedures discussed in this chapter, organized by the names of the modules in which they are found. (If your version is ISO standard, you have nothing to do.)
7. Which of the following identifiers are illegal in Modula-2 and why? `FirstFruit`, `Result#1`, `Goody2Shoes`, `Execute User`, `12Dozen`, `MODULE`, `Canada'sBest`, `AVeryLongIdentifier`, `THISYEARSIMPORTS`, `Begin`, `This_low_life_identifier`
8. Define the term *identifier*.
9. Define the term *statement*.
10. Define the term *standard* in the context of Modula-2.
11. Distinguish among the terms *reserved word*, *standard identifier*, and *standard library identifier*.
12. Distinguish among the terms *literal*, *constant*, and *variable*.
13. Make a list of three problems a small computer would be suitable for, three more that only a large machine could handle, and three that would be unsuitable for computer solution at all. (Do not use any of the illustrations in this chapter; make them as unlike these as possible.) Give your reasons in each case. Discuss your list with several other people.
14. What is the difference between the way a constant is given a value and a variable is given a value in Modula-2?
15. What is a Modula-2 expression?
16. What does it mean to say that `CARDINAL` and `INTEGER` are assignment compatible but not expression compatible?
17. Correct the right hand side of each assignment statement so that it will work properly. Assume that *int* is of type `INTEGER`, *card* is of type `CARDINAL`, *lre* is of type `LONGREAL`, and *re* is of type `REAL`.
 - a) `card := card + int;`
 - b) `re := int;`

- c) `lre := re;`
- d) `lre := card;`
- e) `re := card;`
- f) `re := FLOAT(VAL(INTEGER, 6));`
- g) `card := TRUNC (-5);`

18. Without writing any programs, determine the answers to the following expressions. A few cause errors. Note these and state why there is an error.

- a) $4 + 5 * 7 - 3$
- b) $8 \text{ **DIV** } 2 * 10 \text{ **MOD** } 3$
- c) $4 / (2 - 6 \text{ **DIV** } 3)$
- d) $-15 \text{ **REM** } 6 - 3 * 14 / 4$
- e) $2 - 5 * (4 + 7)$
- f) $3.2 - 16.7 * (4.1 + 5.9)$
- g) $15.25 - 12 \text{ **DIV** } 4$
- h) $2.4 + 1.5 * \text{**FLOAT**}(4)$
- i) $2.5 * \text{**FLOAT**}(15 \text{ **MOD** } -4)$
- j) $\text{**FLOAT**}(11.7)$
- k) $(1.5 \text{ E}+09) * (3.0 \text{ E}+04)$
- l) $(2.8 \text{ E}+12) / (7.0 \text{ E}-04)$
- m) $(7.5 \text{ E}-18) / (1.5 \text{ E}-03)$
- n) $\text{**FLOAT**}(8 \text{ **DIV** } 2 + 6 \text{ **DIV** } 5) + 5.9$
- o) $(7.5 \text{ E}-18) / 10000$
- p) $-12 + \text{**TRUNC**}(7.8)$
- q) $\text{**LFLOAT**}(8) + \text{**FLOAT**}(7)$

19. What is the exact result output from each of the following? Be sure to indicate spaces where present, and to describe the nature of any apparent logical errors.

- a)

```
WriteCard (4, 0);
WriteInt (-2, 4);
```
- b)

```
WriteString ("answer =");
WriteInt (-6, 0);
```
- c)

```
counter := 1;
WHILE counter < 5
  DO
    WriteCard (counter, 3);
    WriteLn;
  END;
```
- d)

```
counter := 1;
WHILE counter < 10
  DO
    counter := counter - 1;
    WriteCard (counter, 3);
  END;
```

```
e) counter := 3;
  WHILE counter < 20
  DO
    WriteCard (counter, 3);
    counter := counter * 2;
  END;
```

Problems

20. Write a program that will print out to the screen a message of the type

```
*****
** This program was written by **
*****      <your name<date<your course<professor's name>      ***
*****
```

That is, the program should print out an ownership message for your subsequent work within a fancy box made up of stars or some other symbols. Try to get the whole thing so that it is centered in the screen. (Many screens are eighty columns wide, and are twenty four rows high.)

21. Write a batch style program to convert 45 pounds to kilograms. (One pound equals 0.453592 kilograms.)

22. Write a batch style program to convert 45 miles per hour to metres per second. (One mile equals 1606.344 metres.)

23. Write a batch style program to convert 17 hours 15 minutes 43 seconds to seconds.

24. Write a batch style program to compute how many times your heart would beat if you lived for seventy-five years. (Assume it beats 72 times per minute, and a year has 365.25 days.)

25. Rewrite the module [*Powers*](#) in section 2.3 as an interactive program.

26. Rewrite the module [*SquareCubeFormatted*](#) in section 2.8 as an interactive program. The user should be prompted for the starting and ending number of the table to print.

27. Rewrite the program in question 21 above in interactive form.

28. Rewrite the program in question 22 above in interactive form.

29. Rewrite the program in question 23 above in interactive form. The user should be prompted for all three pieces of input data.

30. Rewrite the program in question 24 above in interactive form. The user should be prompted for a pulse rate and a number of years.

31. Rewrite the module [*BankInterest*](#) in section 2.12 so that the withdrawals come out before interest is calculated on the remaining balance, then there is an opportunity to make a deposit. Have the program print a chart with the headings *opening balance*, *withdrawal*, *interest*, *deposit*, and *new balance*. Each line after that should be one time period. Have the program do monthly figures rather than quarterly ones. Write new refinements, pseudocode, and users' manual.

NOTE: The following programs should have complete plans, including applicable refinements, pseudocode, documentation, sample data, and actual runs. All should be interactive.

32. Write a program to compute and display answers to more general simple interest problems. (interest = principal * rate * time) The user should be prompted for three pieces of input data.
33. Write a program that will compute and print on the screen the area and perimeter of a rectangle, given that the length and width are real and are to be typed in from the keyboard.
34. Write a payroll module to compute the wages of secretary Nellie Hacker. The user should be prompted for the hourly pay, the hours worked on each of the five working days of the week, and pay Nellie one-and-a-half times her regular salary for working more than 8 hours in a given day. Use only the syntax discussed in this chapter.
35. Write a module that will compute the cost of operating your car given the number of kilometres travelled, the number of litres of fuel consumed per hundred kilometres, and the cost of gasoline in cents per litre.
36. (Alternate to #35 in countries using the old Imperial units) Write a module that will compute the cost of operating your car given the number of miles travelled, the number miles travelled per gallon of fuel consumed, and the cost of gasoline in cents per gallon. (You may use either of the gallon sizes you are familiar with.)
37. Write a module that will convert miles per gallon to litres per hundred kilometres. Be sure to note which gallon size you are using for the conversion (Imperial or U.S.)
38. Write a program that will compute and print to the screen the answers to cardinal multiplications in the following form:

```

      225
      112
-----
      450
      225
      225
-----
     25200

```

39. Write a program that will compute and print to the screen the answers to cardinal divisions in the following form:

```

      34   R 13
-----
15 ) 523

```

40. Write a program that will sum a set of purchases, print a subtotal, apply a sales tax, print and add this, then print a grand total. The user should be asked repeatedly for a new purchase amount until a figure of 0.0 is entered. The sales tax may be a constant.

Programming Note--I/O In Non-Standard Modula-2

Several references were made in Chapter two to the classical or pre-standard versions of Modula-2. In general, but with many variations, the entire contents of the standard modules *SWholeIO* and *STextIO* were found in the classical module *InOut*, and contents of *SRealIO* were found in the classical module *RealInOut*. [Section 6.3.1](#) and [Appendix 7](#) can be consulted for further details. The program here is a little illustration in classical Modula-2. As can readily be seen, there is not much difference between it and an ISO program.

```
MODULE ALittleMath;

  (* This program will ask for two numbers and multiply them.  *)

FROM InOut IMPORT
  WriteString, WriteLn, Read;
FROM RealInOut IMPORT
  ReadReal, WriteReal;

VAR
  userInput1, userInput2, ans : REAL;

BEGIN
  (* fetch input *)
  WriteString ("Give me a number, please. ");
  ReadReal (userInput1);
  WriteLn;
  WriteString ("And a second one, if you would. ");
  ReadReal (userInput2);
  WriteLn;
  (* do calculation *)
  ans := userInput1 * userInput2;
  (* output *)
  WriteString ("The answer to your ");
  WriteString ("multiplication problem is ====");
  WriteReal (ans, 15, 3);

END ALittleMath.
```

Contents

Chapter 2

From Plan to Program

[2.0 Chapter Goals](#)

[2.1 Giving Birth--A First Modula-2 Program](#)

[2.2 The Anatomy of an Infant Program](#)

[2.2.1 What is a Module?](#)

[2.2.2 Reserved Words](#)

[2.2.3 Standard Library Tools](#)

[2.2.4 A Name for the Baby--Identifiers](#)

[2.2.5 Strings](#)

[2.2.6 Summary](#)

[2.3 How to Solve a Problem](#)

[2.3.1 Analysis](#)

[2.3.2 Planning and Refining a Solution](#)

[2.3.3 Data Tables and Sample I/O](#)

[2.3.4 Refining the Solution](#)

[2.3.5 Execution and Satisfaction](#)

[2.4 Documenting the Solution](#)

[2.4.1 External Documentation](#)

[2.5 Variables in Modula-2](#)

[2.5.1 Simple Variable Types](#)

[2.5.2 Standard Identifiers](#)

[2.6 Literals and Constants](#)

[2.6.1 Literals](#)

[2.6.2 Constants](#)

[2.7 Expressions for Constants and Variables](#)

[2.7.1 Precedence in Modula-2](#)

[2.7.2 Mixed Expressions and Modula-2](#)

[2.8 Simple Output Methods](#)

[2.9 REAL Variables](#)

[2.9.1 Real Operations](#)

[2.9.2 The Format of Real Output](#)

[2.10 Type Compatibility and REAL](#)
[2.11 Interactive Programs](#)
[2.12 An Extended Example \(Bank Interest\)](#)
 [2.12.1 Input in non ISO Versions](#)
[2.13 Chapter Summary](#)
[2.14 Assignments](#)
[Programming Note--I/O In Non-Standard Modula-2](#)

Contents

3.0 Chapter Goals

As the discussions of chapter one indicated, this book is concerned in equal measure with the representation and manipulation of data on the one hand, and the technique of structuring programs on the other. Chapter two began the process of fleshing out the former. This chapter focuses on realizing three of the four program structure abstractions mentioned in chapter one.

Data Representation Abstractions

General:

The concept of a flow of data or data stream is introduced.

Realized in the Modula-2 notation:

boolean variables

Data Manipulation Abstractions

General:

redirection of standard data streams from within a program

Realized in the Modula-2 notation:

boolean expressions

Programming Abstractions

General:

fine tuning loops, writing efficient code, replacing loops with closed forms where possible; style and prettyprinting; further discussion of preventing errors

Realized in the Modula-2 notation:

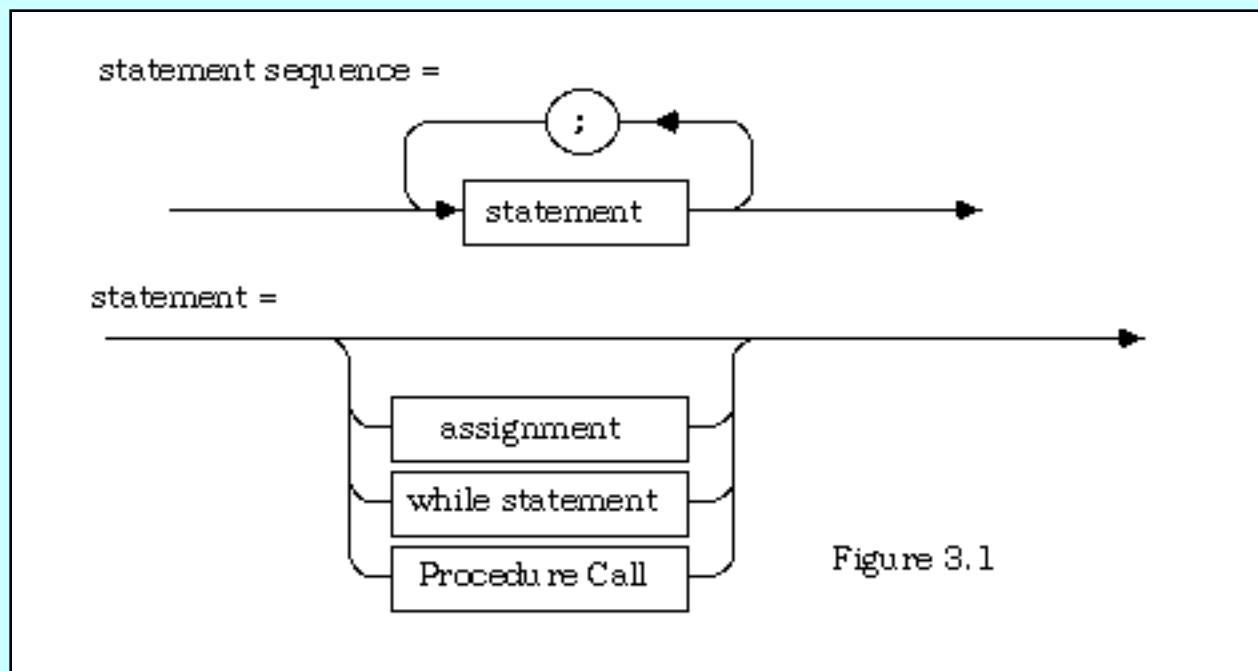
the IF .. THEN .. ELSIF .. ELSE construction for selection; the WHILE, and REPEAT forms of repetition; qualified import; prettyprinting, the use of FROM alone for qualified import

3.1 Statement Sequences

Although a few machines can run two or more programs simultaneously, most modern computers execute programs as sequential steps (one after the other). That is, programs consist of commands written in a particular order, and the compiler for the programming notation preserves this order when translating into machine-readable code. Chapter two introduced several examples of Modula-2 commands or statements. Examples in that chapter included numerous examples of sequences of such statements. Here is another:

```
WriteString ("This program computes the area of a square");  
WriteLn;  
WriteString ("It was written by");  
WriteLn;  
WriteString ("Nellie Hacker");  
WriteLn;  
WriteString ("for CMPT 141");  
WriteLn;  
WriteString ("Assignment #2 Due 1992 09 22");  
WriteLn;  
WriteLn;
```

Figure 3.1 summarizes the syntax of the statement sequence, and details the statement types encountered thus far. The second diagram is incomplete, of course. It will be added to in this chapter and for several chapters to come. A complete version is in [Appendix 2](#).



Notice that a statement need contain nothing at all! Several consecutive semicolons therefore constitutes a valid statement sequence, though it is a rather uninteresting one. Thus, (redundant) semicolons are allowed before an END (or in similar situations,) where they are not required because they are not separating statements. This can be helpful to the programmer who frequently goes back to add new statements to the end of a sequence, and just as frequently forgets to insert a new semicolon before the first of the new statements. If the sequence ends in a semicolon, one need not remember to put one in when extending it.

[Contents](#)

3.2 Simple Selection

A statement sequence requires the computer to step through a program precisely in the order in which the instructions in the original text file are written.

But, what if the solution demands one of the following:

1. That one or more steps be skipped if a variable has a certain value?
2. That one of two or more possible actions be taken depending on, say, what the user responds at the keyboard?
3. There be several alternative sections of code that could be executed depending, say, on the value of some variable?

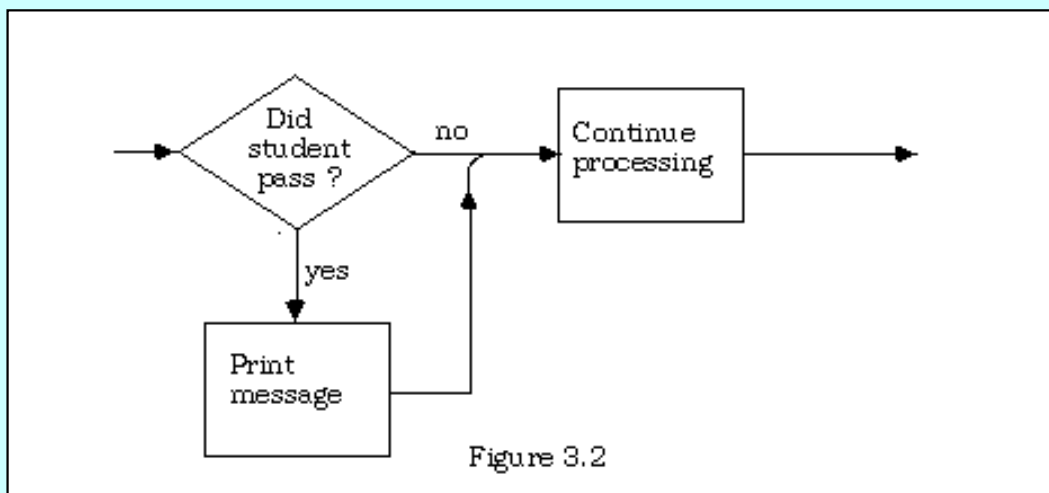
The abstract solution mechanism presented in chapter one to handle such situations was called *selection*. Almost all computer languages implement actual selection statements that allow for changing the "flow of control" through the program in any of these situations.

Here are the general forms which your code can follow to handle each of the above three:

Situation 1

One or more steps should be skipped whenever a variable equals a particular value. Suppose a segment of code is calculating average marks, and is to print a congratulatory message for everyone who passes the course, but to print nothing otherwise.

This could be diagrammed as in figure 3.2:



In Modula-2, the code might be written something like this:

```
(*common code executed prior to entering selection section*)
WriteString ("Your mark is");
WriteReal (mark, 10);
WriteLn;
(*selection section*)
IF mark >= 50 THEN "congratulations, you passed the course";
    WriteLn; (* semicolon optional before an end *)
END;
WriteString (nextMessage);
WriteLn;
(* now, carry on with rest of program *)
```

NOTES: 1. The reserved word **IF** must be accompanied by both the **THEN** and the **END**. These are also reserved words. The **END** of an **IF** is not the **END** of a named block, so it is not followed by a period.
2. Observe the positioning of semicolons and the way in which the statements are blocked out for easy readability. As

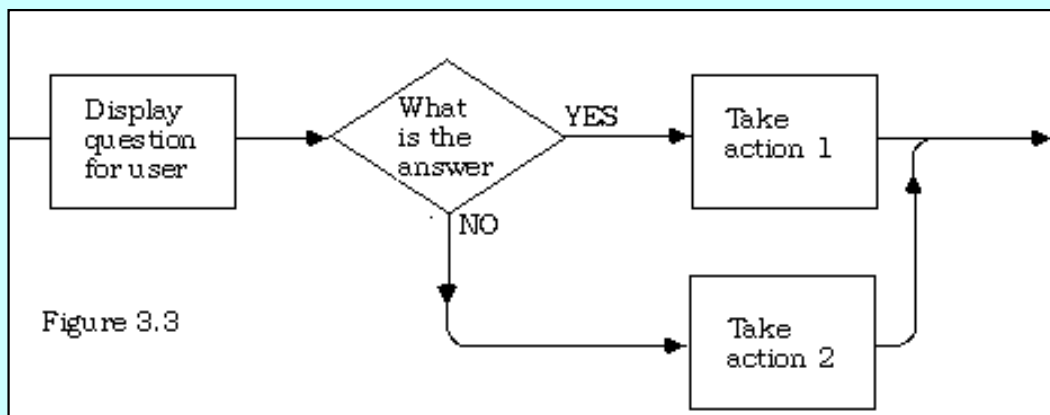
mentioned above, the semicolon immediately before the END is optional.

Here is a section of code that with some modifications might appear in many types of programs. It arises whenever the user has been given a series of numbered choices (a menu) and then must respond. The code will repeat until a correct choice has been made.

```
choice := 8; (* a cardinal type, by the way *)
WHILE choice " enter your choice (0 - 7) here ==");
    ReadCard (choice);
    IF choice "Your choice was invalid.  Try again.");
        WriteLn;
    END; (* of the if *)
END; (* of the while *)
```

Situation 2

One of two or more possible actions is to be taken depending on, say, what the user responds at the keyboard. Suppose the two choices depend on whether the user responds with a Y for yes or an N for no. Figure 3.3 illustrates the desired flow of control.



Going beyond the specific illustration in figure 3.3, the idea behind this version of selection can be applied in a wide variety of circumstances.

Example 1:

This first example is a complete program designed to sort two input numbers into increasing order and then print them. It differs from the previous two in that two alternatives are present within the IF statement.

```
MODULE SortTwo;

FROM STextIO IMPORT
    WriteString, WriteLn, SkipLine;

FROM SWholeIO IMPORT
    ReadCard, WriteCard;

CONST
    fieldLength = 6;

VAR
    num1, num2 : CARDINAL;

BEGIN
```

```

(* information *)
WriteString ("This program sorts two cardinal numbers from smallest to largest.");
WriteLn;
(* collect the numbers from the user *)
WriteString ("Enter the first number please. ==");
ReadCard (num1);
SkipLine;
WriteLn;
WriteString ("Enter the second number please. ==");
ReadCard (num2);
SkipLine;
WriteLn;
WriteString ("From least to greatest, the numbers are: ");
(* decide which way to write them and do it *)
IF num1 < num2
  THEN
    WriteCard (num1, fieldLength);
    WriteString ("", "");      (* comma between *)
    WriteCard (num2, fieldLength);
  ELSE
    WriteCard (num2, fieldLength);
    WriteString ("", "");
    WriteCard (num1, fieldLength)
  END;  (* if *)

WriteLn;

END SortTwo.

```

Here is a run from this program with the user input marked in bold:

```

This program sorts two cardinal numbers from smallest to largest.
Enter the first number please. ==> 21
From least to greatest, the numbers are:      21,      43

```

NOTE: A new reserved word **ELSE** has been introduced to indicate an alternative action within the **IF** statement. One of the two alternatives will always be followed.

Example 2:

A tin can is shaped like a cylinder, and may or may not have a top lid. Write a program to compute the area and volume of the cylinder.

Discussion:

In order to save space, a complete plan for this program will not be provided. However, the following (very rough) pseudocode serves to indicate how to implement such alternate paths through the program.

```

Print an introductory heading
Print a prompt asking for the radius of the can
Read the answer into the real variable radius

```

```

Print a prompt asking for the height of the can
Read the answer into the real variable height
Compute the area of one lid using  $A = (\pi)r^2$ 
Compute the area around the side using  $B = 2(\pi)rh$ 
Compute the volume using  $V = Ah$ 
Set the initial total area to the side area
Print a prompt asking if the cylinder has a top lid
If the answer is Yes then
    total area = side area plus twice lid area
Else if the answer is No then
    total area = side area plus lid area
Print the results in an informative way

```

Here is the code:

```

MODULE CylVolArea;

(* Written by R.J. Sutcliffe *)
(* to illustrate IF..THEN..ELSIF *)
(* using P1 for the Macintosh computer *)
(* last revision 1993 02 11 *)

FROM STextIO IMPORT
    WriteString, WriteLn, SkipLine, ReadChar;

FROM SRealIO IMPORT
    ReadReal, WriteReal;

CONST
    pi = 3.14159265;  (* or, import it from RealMath *)

VAR
    radius, height, lidArea, sideArea, totalArea, volume: REAL;
    answer : CHAR;

BEGIN
    (* information *)
    WriteString ("CylVolArea was written by R.J. Sutcliffe");
    WriteLn;
    WriteString ("as an example of IF .. THEN .. ELSE");
    WriteLn;
    WriteLn;
    WriteString ("This code computes cylinder volumes and areas.");
    WriteLn;
    WriteLn;

    (* Gather the information from the user *)
    WriteString ("Please enter the radius of the cylinder ==");
    ReadReal (radius);
    SkipLine;  (* consume the carriage return after the real *)
    WriteLn;
    WriteString ("Please enter the height of the cylinder ==");
    ReadReal (height);

```

```

SkipLine;  (* consume the carriage return after the real *)
WriteLn;

(* compute some results *)
lidArea := pi * radius * radius;
sideArea := 2.0 * pi * radius * height;
totalArea := sideArea; (* initialize; more to be added later *)
volume := lidArea * height;

(* Find out if cylinder has a top lid, and act accordingly *)
WriteString ("Does this cylinder have a top lid? Y or N ==");
ReadChar (answer);
SkipLine;  (* skip to next line of input *)
WriteLn;

(* start to print out answer *)
WriteString ("For a cylinder with these dimensions ");

(* add appropriate lid area for one or two lids *)
IF (answer = "Y") OR (answer = "y")
THEN
    totalArea := totalArea + 2.0 * lidArea;
    WriteString ("and two lids ");
ELSIF (answer = "N") OR (answer = "n") THEN
    totalArea := totalArea + lidArea;
    WriteString ("and one lid ");
END;
WriteString ("the volume is ");
WriteReal (volume, 0);
WriteLn;
WriteString (" cubic units, and the area is ");
WriteReal (totalArea, 0);
WriteString (" square units.");
WriteLn;
WriteLn;

(* pause and wait for user to get a good look *)
WriteString ( "Press any key to continue");
ReadChar (answer);

END CylVolArea.

```

When this program was run twice with similar input (shown in bold) the following results were obtained:

Run#1

CylVolArea was written by R.J. Sutcliffe
as an example of IF .. THEN .. ELSE

This code computes cylinder volumes and areas.

Please enter the radius of the cylinder ==> **20**
Does this cylinder have a top lid? Y or N ==> **10**

Please enter the height of the cylinder ==> **N**

For a cylinder with these dimensions and one lid the volume is 6.2831855E+3 cubic units, and the area is 1.5707964E+3 square units.

Press any key to continue

NOTES: 1. This code introduces two more reserved words, namely OR and ELSIF. Some other computing languages use ELSE IF (two words) and may do so in a slightly different way.

2. The logic here is as follows. If the user responds with a "Y" (uppercase or lowercase--they are different characters) then the area is calculated with two lids. If the response is "N" then the area is calculated with one lid, and if it is neither then the control of the program passes beyond the END of the IF with none of the four statements under the control of the IF being executed.

3. The last section (waiting for a keypress) may be necessary in some systems if their standard input/output erases the screen as soon as the program is finished.

The effect of this code is to interpret any answer other than "Y," "y," "N," or "n" as meaning that there are no lids at all. As this may be a rather undesirable result, the code could be rewritten as:

```
IF (answer = "Y") OR (answer = "y")
  THEN
    totalArea := totalArea + 2.0 * lidArea;
    WriteString ("and two lids ");
  ELSE (* assume a no answer if not yes *)
    totalArea := totalArea + lidArea;
    WriteString ("and one lid ");
  END;
```

If all values of an answer other than one particular one cause the same action, then that action is called the default.

In this case, the code has been modified to make "No" the default answer. If it is desired to check the answer and allow only one of the four specified ones to have any effect, the IF statement can be surrounded by a WHILE loop in the following manner:

```
answer := "X"; (* initialize it to none of the four *)
WHILE (answer # "Y") AND (answer # "y")
      AND (answer # "N") AND (answer # "n")
DO
  ReadChar (answer);
  IF (answer = "Y") OR (answer = "y")
  THEN
    totalArea := totalArea + 2.0 * lidArea;
    WriteString ("and two lids ");
  ELSIF (answer = "N") OR (answer = "n") THEN
    totalArea := totalArea + lidArea;
    WriteString ("and one lid ");
  ELSE
    WriteString ("Invalid answer; please answer Y or N ");
    WriteLn;
  END; (* if *)
END; (* while *)
```

If this is done, the code will cycle endlessly through the WHILE loop until the user does indeed respond with one of the required answers.

NOTE: The new reserved word AND has been introduced here.

Situation 3

There are several alternative sections of code that could be executed depending on the value of some variable. In this case, the code may have several ELSIF sections, and may or may not have an ELSE. What follows is a general framework for the IF..THEN..ELSIF..ELSE..END construction that can be used as a model for a variety of code.

```
IF variable = value1
  THEN
    Statement Sequence1;
  ELSIF variable = value2 THEN
    Statement Sequence2;
  ELSIF variable = value3 THEN
    Statement Sequence3;
  ...

  ELSIF variable = valueN THEN
    Statement SequenceN;
  ELSE
    Default Statement Sequence;

END;
```

Example:

The Aldergrove Credit Union super saver plan pays interest to its depositors at the prime rate provided they have over \$20 000 on deposit, half a percentage point less if over \$15 000, another half a percent less if over \$12 000, with similar steps down at \$10 000, \$8 000, \$6 000 and each \$1000 thereafter. Write a program to compute the interest payable on any amount input by the user.

Discussion:

This program is a straightforward application of a multiple part IF statement, and is presented without any planning documents. Note that the minimum rate is achieved below \$1000, that is, after eleven reductions amounting to 5.5%. Between this point and the \$5000 threshold, the rate goes up in regular steps; after that in irregular ones.

```
MODULE SuperSaverInterest;

(* Written by R.J. Sutcliffe *)
(* to illustrate IF..THEN..ELSIF..ELSE *)
(* using Metropolis Modula-2 for the Macintosh computer *)
(* last revision 1993 02 11 *)

FROM STextIO IMPORT
  WriteString, WriteLn, ReadChar, SkipLine;

FROM SRealIO IMPORT
  ReadReal, WriteFixed;

VAR
  deposit, basicRate, actualRate, interest: REAL;
```

```
answer : CHAR;
```

```
BEGIN
```

```
WriteString ("Program written by R.J. Sutcliffe");  
WriteLn;  
WriteString ("as an example of IF .. THEN .. ELSIF .. ELSE");  
WriteLn;  
WriteLn;  
WriteString ("This program computes interest on deposits.");  
WriteLn;
```

```
(* Gather the information from the user *)  
WriteString ("What is the principal amount on deposit? ==");  
ReadReal (deposit);  
SkipLine; (* reset to start of next line *)  
WriteLn;  
WriteString ("What is the prime interest rate? ");  
WriteString ("Please enter in decimal form. (6% = 0.06) ==");  
ReadReal (basicRate);  
SkipLine; (* reset to start of next line *)  
WriteLn;
```

```
(* Now compute the actual rate from the basic rate and deposit. *)  
IF deposit > 15000.00 THEN  
    actualRate := basicRate - 0.005  
ELSIF deposit > 10000.00 THEN  
    actualRate := basicRate - 0.015  
ELSIF deposit > 6000.00 THEN  
    actualRate := basicRate - 0.025  
ELSE  
    actualRate := basicRate - 0.025 - 0.005 * FLOAT (TRUNC (deposit) DIV 1000);  
END;
```

```
(* Take into consideration a really low prime that might otherwise produce a  
negative interest *)  
IF actualRate < 0.0  
    THEN  
        actualRate := 0.0  
    END;
```

```
(* make the computation of interest and print the result *)  
interest := deposit * actualRate;  
WriteString ("The annual interest on $");  
WriteFixed (deposit, 2, 0);  
WriteString (" at ");  
WriteFixed (100.0 * actualRate, 2, 0);  
WriteString ("% is $");  
WriteFixed (interest, 2, 0);  
WriteLn;  
WriteLn;
```

```
WriteString ( "Press any key to continue");  
ReadChar (answer);
```

END SuperSaverInterest.

Once again, notice the use of semicolons. As usual, placing them before such reserved words as **END**, **ELSE**, and **ELSIF** is optional.

When this program was run twice with similar input (shown in bold) the following results were obtained:

Run#1

Program written by R.J. Sutcliffe
as an example of IF .. THEN .. ELSIF .. ELSE

This program computes interest on deposits.
What is the principal amount on deposit? ==> **0.03**
The annual interest on \$ 1000.00 at 0.00% is \$ 0.00

Press any key to continue

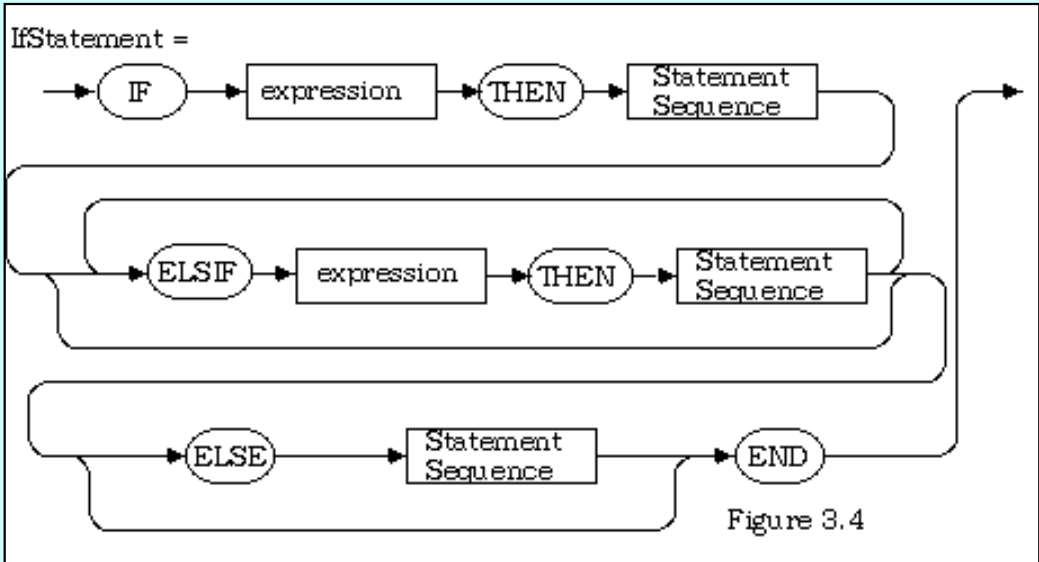
Run#2

Program written by R.J. Sutcliffe
as an example of IF .. THEN .. ELSIF .. ELSE

This program computes interest on deposits.
What is the principal amount on deposit? ==> **0.08**
The annual interest on \$ 10000.00 at 6.00% is \$ 600.00

Press any key to continue

All the forms of the IF statement have now been used in the examples of this section. The complete syntax diagram is given in figure 3.4:



3.3 Boolean Variables and Expressions

In summary, the most general form of the IF ... THEN ... ELSIF ... ELSE construction as used in the previous section is as follows:

```
IF (Boolean Expression 1)
  THEN
    Statement Sequence 1;
  ELSIF (Boolean Expression 2) THEN
    Statement Sequence 2;
  ...

  ELSIF (Boolean Expression n) THEN
    Statement Sequence n;
  ELSE
    Statement Sequence n + 1;
END;
```

All the ELSIFs and the ELSE are optional, depending on the logic required by the program. Note that the form (*Boolean Expression i*) is used here instead of a specific comparison as in all the examples thus far. There was a brief discussion of Boolean expressions in [section 1.7](#). The purpose of this section is to specify more exactly what constitute valid boolean expressions in Modula-2, for much more elaborate ones than these can be used in a variety of program structures.

This brings us to the question you may have been ready to ask when you read the heading for this section and the first IF. "What exactly is this thing called a "Boolean expression"? Two related definitions are useful:

An Modula-2 expression is of type BOOLEAN if it can be evaluated unambiguously as "true" or "false".

A Modula-2 variable of type BOOLEAN can have one of the two values TRUE, or FALSE (all three words are standard identifiers).

Previous examples have already illustrated the use of the reserved symbols = and << less than

<= less than or equal to

<& both of the conditions must be true

OR at least one of the conditions must be true

NOT or ~ reverses the value of the following logical expression

As in other Modula-2 expressions, any combination of these is allowed, and parentheses can be used to change the natural order of evaluation. Here are a few examples to review the use of these:

<u>Expression</u>	<u>BOOLEAN Value</u>
5 <= 5	TRUE
(10 < 5) AND (3 < 4)	FALSE
('Y' = 'y') OR FALSE	FALSE
NOT (5 = 6)	TRUE
(4 = 5) AND ((3 = 7) OR TRUE)	FALSE

NOTES: 1. The order of evaluation is:

First NOT (~)

Second *, /, DIV, MOD, AND (&)

Third +, -, OR

Last the relational operators =, <>, <, <=

2. When evaluation reaches an AND, if the left expression is FALSE, the right expression is not evaluated. Likewise, if evaluation reaches an OR, if the left expression is TRUE the right expression is not evaluated. Most computer languages cannot be relied upon to take these shortcuts and may evaluate all parts of every boolean expression. Another more formal way of putting this is to say that if p and q are Modula-2 boolean expressions, then:

p AND q means: IF p THEN q ELSE FALSE

p OR q means: IF p THEN TRUE ELSE q

3. Expressions separated by AND or OR should be enclosed in parentheses. If this is not done, the correct order of evaluation may not be followed, for the statement may not be syntactically correct. If x, y, and z are numbers, then

x < y **AND** y < y) **AND** (y < (y AND y) <= 7) & (3 < 10)

would be evaluated as:

(6 **DIV** 3 + 2 = 5) **OR** **FALSE** & (3 < 10)

(2 + 2 = 5) **OR** **FALSE** (& not evaluated)

(4 = 5) **OR** **FALSE**

FALSE

Example 2.

IF (a < b) **OR** ((c < d) **AND** ((y < 0.05))

Should a in fact be less than b the entire portion on the right of the OR will not be evaluated as the left side alone would be sufficient to give a value of TRUE for the whole expression.

Example 3.

IF (false expression) **AND** (enormous expression several lines long);

In this case, *enormous expression* will not be evaluated, again saving considerable time when the code runs.

As indicated above, there are boolean variables in Modula-2 as well as boolean expressions. These are declared like those of any other type, under a VAR heading. They may then be used throughout the program to represent various boolean values. As usual, declaring one of these names does not give it to any particular value--the variable must be initialized before it is known to be either TRUE or FALSE; until then it is undefined. Here is a little program which offers the user a sample choice between two actions and stores the information from the choice in as the value of a boolean variable:

Example 4.

Two groups of students are taking a course in Modula-2 at the same time. The Cmpt141 students have High School computing with high marks and so do a four semester hour course. The Cmpt 143 students attend the same lectures, but have a previous university level course in Pascal. Because it is much easier to learn a second computing notation at this level than a first, these students receive two semester hours credit. They are also exempt from a major assignment. Their marks are calculated as follows:

CMPT 141 CMPT 143

Labs 25% (All must be done to gain credit) 25%

Midterms 30% 35%

Major Paper 10% (essay) Not required

Final Exam 35% 40%

The task is to write a program that will take the percentage marks from the three or four categories and produce a final percentage grade for the course. In what follows, much of the planning has been left out for the sake of brevity.

Pseudocode:

```
Print an informative heading
Initialize the student's totalMark to zero
Print a prompt asking which course the student is taking
Set the value of the boolean variable in141 using the answer
Print a prompt asking if all assignments were handed in
Set the boolean variable assignOK using the answer
IF assignOK then
    Print a prompt asking for the lab percentage
    Read the result to the real variable labs
    Set totalMark to 0.25*(labs)
    Print a prompt asking for the midterm percentage
    Read the result to the real variable midterms
    Print a prompt asking for the final exam percentage
    Read the result to the real variable fExam
    If in141 then
```

```

    Print a prompt asking for the essay percentage
    Read the result to the real variable essay
    totalMark = totalMark+.3*(midterms)+.35*(fExam)+.1*(essay)
Else
    totalMark = totalMark+.35*(midterms)+.4*(fExam)
Print the final mark embedded in a course-specific message
If Not (assignOK) Then
    Print an explanation

```

In addition, the code presented here implements a common technique to allow for a repetition of the entire program at the request of the user. This also employs a boolean variable.

MODULE Modula2CourseMarks;

```

(* Written by R.J. Sutcliffe *)
(* to illustrate the use of Boolean variables *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 02 12 *)

```

FROM STextIO **IMPORT**

```

    WriteString, WriteLn, ReadChar, SkipLine;

```

FROM SRealIO **IMPORT**

```

    ReadReal, WriteFixed;

```

CONST

```

    labWt = 0.25;
    midtermWt141 = 0.3;
    midtermWt143 = 0.35;
    finalWt141 = 0.35;
    finalWt143 = 0.4;
    essayWt = 0.1;

```

VAR

```

    labs, midterms, essay, fExam, totalMark : REAL;
    answer, cr : CHAR;
    in141, assignOk, again : BOOLEAN;

```

BEGIN

```

    WriteString ("Modula2CourseMarks written by R.J. Sutcliffe");
    WriteLn;
    WriteString ("as an example in the use of Boolean variables");
    WriteLn;
    WriteLn;

```

```

WriteString ("It computes the final percentage mark for ");
WriteLn;
WriteString ("students enrolled in Cmpt 141 and Cmpt 143. ");
WriteLn;
WriteLn;
again := TRUE;
WHILE again
    DO
        totalMark := 0.0;

        (* Gather the information from the user *)
        WriteString ("Is the student in Cmpt141? Y/N ==");
        ReadChar (answer);
        SkipLine; (* consume the carriage return after the char *)
        WriteLn;
        in141 := (answer = "Y") OR (answer = "y");
        WriteString ("Were all assignments complete? (Y/N) ==");
        ReadChar (answer); (* Note that variables can be re-used *)
        SkipLine; (* consume the carriage return after the char *)
        WriteLn;
        assignOk := (answer = "Y") OR (answer = "y");
        IF assignOk
            THEN
                WriteString ("What % mark was earned on labs? ==");
                ReadReal (labs);
                SkipLine; (* consume the carriage return after real *)
                WriteLn;
                totalMark := labWt * labs;
                WriteString ("What was the midterm mark? ==");
                ReadReal (midterms);
                SkipLine;
                WriteLn;
                WriteString ("What was the final exam mark? ==");
                ReadReal (fExam);
                SkipLine;
                WriteLn;
                IF in141
                    THEN
                        WriteString ("What mark was the essay mark? ==");
                        ReadReal (essay);
                        SkipLine;
                        WriteLn;
                        totalMark := totalMark + midtermWt141 * midterms

```

```

        + finalWt141 * fExam + essayWt * essay;
    ELSE
        totalMark := totalMark + midtermWt143 * midterms
        + finalWt143 * fExam;
    END; (* if in141 *)
END; (* if assignOk *)

(* print the result *)
WriteString ("The final mark for Cmpt");
IF in141
    THEN
        WriteString (" 141 ");
    ELSE
        WriteString (" 143 ");
    END;
WriteString (" was ");
WriteFixed (totalMark, 2, 0);
WriteString ("%.");
WriteLn;
IF NOT assignOk
    THEN
        WriteString ("No credit when assignments not done.");
        WriteLn;
    END;

WriteLn;
WriteString ( "Do another calculation? Y/N ==");
ReadChar (answer);
again := (answer = "Y") OR (answer = "y");
SkipLine;
WriteLn;
END; (* of the while loop *)

```

END Modula2CourseMarks.

Here is a run from this module:

Modula2CourseMarks written by R.J. Sutcliffe
as an example in the use of Boolean variables

It computes the final percentage mark for
students enrolled in Cmpt 141 and Cmpt 143.

Is the student in Cmpt141? Y/N ==> **Y**

What % mark was earned on labs? ==> **90**
What was the final exam mark? ==> **72.5**
The final mark for Cmpt 141 was 89.00%.

Do another calculation? Y/N ==> **N**
Were all assignments complete? (Y/N) ==> **N**

In programs like this, one sets a boolean from keyboard input and then uses the value one or more times to determine which course to take through the code.

When the value of a boolean variable is used one or more times to determine the appropriate path through a program, the variable is called a flag.

NOTE: It is redundant, superfluous, and too much excess of unnecessary extra code to write `IF assignOk = TRUE` when it suffices to use `IF assignOk`.

[Contents](#)

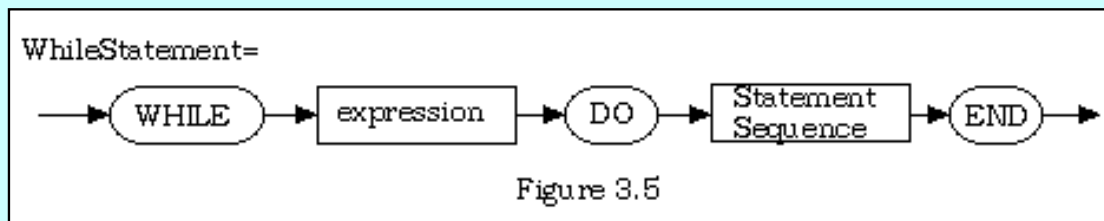
3.4 Repetition (1) -- the WHILE Statement

A number of program examples thus far have employed repetition in the form of the WHILE loop, and there is very little more to say about this Modula-2 construction that the astute reader has not already gleaned from these examples. The general form is:

```
WHILE Boolean Expression (* i.e. while "Expression" is TRUE *)  
  DO  
    Statement Sequence;  
  END;
```

When the END of the loop is reached, program control cycles back to the WHILE and the Boolean expression is checked again. If it is still TRUE, the loop executes once more. If it is FALSE, control passes to the next statement after the END of the WHILE loop.

If the expression happens to evaluate to FALSE when the WHILE is first encountered, the whole statement will be skipped. In the terms used in section 1.9.1, the WHILE loop employs *top-of-loop testing*. The syntax diagram is in figure 3.5:



Example:

Write a program to read in a sentence and find the average number of letters in each word. (This could be used as one test for the reading level of textual material.)

Discussion:

A somewhat simplified plan will be given for this program, as the WHILE loop has already been well illustrated.

Problem restatement:

Given Input

a sequence of letters ending with a carriage return

Processing to do

determine the number of letters

determine the number of words

compute the average word length

Desired Output

inform user of computed average

Problem refinement:

1. Ask the user for a sentence.
2. Read the response in, one character at a time.
 Count the non-blank characters as letters.
 Count blanks as words.
 Add one word (the last word.)
3. Divide letters/words for the average.
4. Inform user of total.
5. Give option of doing it again.
 Recycle if yes.
 End if no.

Entities required:

- o Variables: currentLetter, answer : CHAR
 letters, words : CARDINAL
 again : BOOLEAN
- o Imports: WriteString, WriteLn, WriteReal, ReadChar, SkipLine,
 ReadResults, ReadResult (latter two to determine whether at end of line)

Code:

```
MODULE AverageWordLength;
```

```
FROM STextIO IMPORT  
    WriteString, WriteLn, ReadChar, SkipLine;
```

```
FROM SIOResult IMPORT  
    ReadResult, ReadResults;
```

```
FROM SRealIO IMPORT  
    WriteFixed;
```

```
VAR  
    currentLetter, answer : CHAR;  
    letters, words   : CARDINAL;  
    again : BOOLEAN;
```

```
CONST  
    space = ' ';
```

```
BEGIN  
    WriteString ("This program computes the average ");  
    WriteString ("length of the words in a sentence.");  
    WriteLn;  
    WriteLn;  
    again := TRUE;       (* initialize boolean *)
```

```

WHILE again
  DO
    letters := 0;      (* initialize counters *)
    words := 0;

    (* get data *)
    WriteString ("Type a sentence ending in a return. ");
    WriteLn;
    ReadChar (currentLetter);      (* process first char. *)

    WHILE ReadResult() # endOfLine
      (* continue until end of line *)
      DO (* process the line *)
        IF currentLetter = space
          THEN
            words := words + 1;
          ELSE
            letters := letters + 1;
          END;      (* if *)

        ReadChar (currentLetter);      (* obtain next char *)
      END; (* while *)      (* and recycle loop *)

    SkipLine;
    WriteLn;
    words := words + 1;      (* add one for last word at end *)
    WriteString ("The average word length was ");
    WriteFixed (FLOAT (letters) / FLOAT (words), 2, 0);
    WriteLn;
    WriteString ("Do you want to do another one? ");
    ReadChar (answer);
    SkipLine;
    WriteLn;
    again := (answer = 'Y') OR (answer = 'y');

  END;      (* first WHILE *)

  (* here we either recycle or fall out to the next statement *)

  WriteString ("AverageWordLength processing concluded.");
  WriteLn;
  WriteString ("Press a key to continue.");
  ReadChar (answer);

END AverageWordLength.

```

NOTE: The entities *ReadResults* and *ReadResult* are the only two items in the library module *SIOResults*. The first is a variable type that may take on values such as *endOfLine* and *allRight* (among others). The second makes an enquiry about the last performed read operation on the standard input/output channel used by such modules as *STextIO*, *SRealIO*, and *SWholeIO*. It then returns a value of the type *ReadResults* that the main program may compare with the

desired outcome.

This program works, as far as it goes, but no output is provided here, for the reader will no doubt have noticed that there are a few flaws in the thinking. One of these is that if two blanks are left after a word, the program will count two words. This could be avoided by having another character variable called, say, *lastOne* and assigning this the value of *currentLetter* at the end of each pass through the inner WHILE loop. Then the check for whether one has a word or not at the beginning of the loop could include a comparison of *lastOne* to *space*. The code could look like this:

```
Read (currentLetter);      (* process first char. *)
lastOne := currentLetter;
WHILE ReadResult() # endOfLine
  DO  (* process the line *)
    IF (currentLetter = space) AND (lastOne # space)
      THEN
        words := words + 1;
      ELSE
        letters := letters + 1;
      END;      (* if *)
    lastOne := currentLetter;
    ReadChar (currentLetter);      (* obtain next char *)
  END;  (* while *)      (* and recycle loop *)
```

Another flaw is that a real sentence ends with a mark like a period and may have quotes, apostrophes, commas, and other punctuation symbols that are not letters. The calculation here counts these as letters. Moreover, it would be nice not to be limited to a single sentence, but to have some means other than the end of line state to determine when processing is complete. In one of the exercises, the reader is asked to rewrite this Module to correct some of these deficiencies.

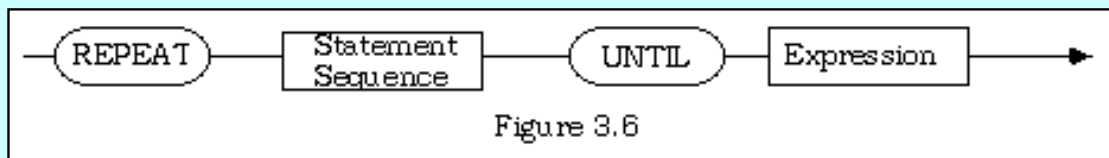
[Contents](#)

3.5 Repetition (2) -- the REPEAT Statement

The while loop tests the indicated boolean expression at the top of the loop and does not execute the loop if the condition is false. There are times, however, when one wishes to have the loop execute at least once in any case, and then to test the condition before continuing for a second pass. As indicated in [section 1.9.1](#), this is achieved with the repeat loop construction. In Modula-2, as in many computing notations, the general form of this type of loop is:

```
REPEAT
    Statement Sequence;
UNTIL Boolean Expression;
```

or diagrammatically as in figure 3.6:



Notice that the UNTIL concludes the statement sequence so that there is no END. There is also no DO. This type of loop repeats the enclosed statement sequence as long as the boolean expression is FALSE and then, when it becomes TRUE, control "falls out" to the next statement beneath. Some of the code in the examples above could have been written more efficiently. Instead of

```
again := TRUE;
WHILE again
    DO
        statement sequence;
        WriteLn;
        WriteString ("Do you want to do another one? ");
        ReadChar (answer);
        WriteLn;
        again := (answer = 'Y') OR (answer = 'y');
    END;
```

it is more efficient to write:

```
REPEAT
    statement sequence
    WriteLn;
    WriteString ("Do you want to do another one? ");
    ReadChar (answer);
    WriteLn;
    again := (answer = 'Y') OR (answer = 'y');
UNTIL NOT again;
```

Here is a simple problem from mathematics whose solution illustrates the use of the REPEAT statement and at the same time shows the necessity of careful planning and an intimate knowledge of the problem itself.

Example:

Write a program which will find the Greatest Common Divisor (GCD) of two given numbers.

Discussion:

When solving problems like this one, it is essential that the terms used in the question are clearly understood.

The Greatest Common Divisor (GCD) of two numbers is the largest positive whole number which divides evenly (without remainder) into both. If their Greatest Common Divisor is one, two numbers are said to be relatively prime.

For instance,

$\text{GCD}(6, 8) = 2$

$\text{GCD}(24, 160) = 8$

$\text{GCD}(25, 16) = 1$, so 25 and 16 are relatively prime.

$\text{GCD}(2, 0) = 2$ (every number divides zero)

A number of methods are commonly taught in Junior High School mathematics to solve such problems. To illustrate, consider the following method for determining the Greatest Common Divisor of 432 and 1800.

Solution A:

1. List all the divisors of 432 and 1800.
2. Examine the two lists and select the largest number appearing in both.

Algorithm for listing divisors:

```
empty the list
set trial divisor to 1
repeat
  divide number by trial divisor
  if remainder is zero then
    put trial divisor in list
    put quotient in list
  add one to trial divisor
until trial divisor is greater than the square root of the number
```

The latter provision is feasible since the algorithm adds both to the bottom and the top of the list simultaneously. Thus, once the square root of the number is passed on the way up, the top end contains all the divisors down to that point as well, and the process may be concluded.

Algorithm for selecting the largest number in two lists:

```
repeat
```

```

find the largest number in each of the two lists
if these are not equal then
    discard the greater of these two numbers from its list
until the largest numbers in each list are equal; this is the GCD

```

For the example cited, this method produces the two lists:

432:1,2,3,4,6,8,9,12,16,18,24,27,36,48,54,72,108,144,231,432

1800:1,2,3,4,5,6,8,9,10,12,15,18,20,24,25,30,36,40,45,50,60,72,75,90,100,120,150,200,225,300,360,450,600,900,1800

Finally, examination of the lists for the highest number in both quickly produces 72 as the GCD.

There are a number of difficulties with this method. It uses the concept of square root, which is not very satisfactory when working with whole numbers. How to translate the "list" abstraction employed here into a computing notation is not at all clear. Neither is how to implement the idea of discarding an item from a list. Such difficulties cause one to consider another method, and this one too is taught in high school mathematics.

Solution B:

1. Factor the two numbers into products of primes.
2. Construct the GCD as the product of the largest common number of factors of each prime taken individually.

Thus, $432 = 2^4 * 3^3$ and $1800 = 2^3 * 3^2 * 5^2$

Examining the common factors gives the GCD as $2^3 * 3^2 = 72$.

However, this method is even more difficult to implement as an algorithm and translate into a computing notation. It requires that the problem be refined into methods for determining primes, keeping a list of them, and maintaining lists of the prime factors of numbers, together with the appropriate exponents. Such a refinement will not be given here, as there is a better way. To see what that better way is, it is necessary to examine carefully the definition of a divisor.

*We say that d is a divisor of x if there is a number m with $x = d * m$.*

Thus, for two positive numbers x and y with a common divisor d , there would be numbers m and n with $x = m * d$ and $y = n * d$. (There is always at least one such number, because 1 is a common divisor of all other natural numbers.)

To progress from this point requires a flash of insight. Suppose that x is the larger of the two numbers. Observe that $x - y = m * d - n * d = (m - n) * d$

The latter expression gives the difference of the two numbers as a product of a natural number with their common divisor d . Put another way, this means that every divisor of the two given numbers is also a divisor of their difference.

In particular, this is true of the greatest common divisor of the two:

either $\text{GCD}(x, y) = \text{GCD}(x - y, y)$ OR $\text{GCD}(x, y) = \text{GCD}(x, y - x)$ (1)

(depending on whether $x > y$)

It ought also to be evident that

If $x = y$ then $\text{GCD}(x, y) = x$ (2)

These two formulas help by reducing the problem to that of finding the GCD of successively smaller pairs of numbers than the originals, until the pair is equal, at which point either is the GCD. A trace of this logic by hand for the earlier example produces:

$\text{GCD}(1800, 432) = \text{GCD}(1368, 432)$

$= \text{GCD}(936, 432)$

$= \text{GCD}(504, 432)$

$= \text{GCD}(72, 432)$

$= \text{GCD}(72, 360)$

$= \text{GCD}(72, 288)$

$= \text{GCD}(72, 216)$

= GCD (72, 144)

= GCD (72, 72)

= 72

This hand trace reveals that yet another simplification to the algorithm is possible. At one stage, the number 72 was subtracted several times in succession so what is left is really just the remainder after the larger number was divided by the smaller. For convenience, assume that the first number is the larger, and keep them in order throughout. That is,

$\text{GCD}(x, y) = \text{GCD}(y, x \bmod y)$ (1a)

(assuming always $x \geq y$) "This program computes the GCD "; WriteString ("of two whole numbers."); WriteLn; **REPEAT** (* get the numbers *) WriteString ("Enter the first number =="); ReadCard (x); (* read first number *) SkipLine; (* and end of line *) WriteLn; WriteString ("And now, the second =="); ReadCard (y); (* read second number *) SkipLine; (* and end of line *) WriteLn; (* arrange the numbers in order *) **IF** $y > 0$ **DO** gcd := y; (* let the gcd be the smaller of the two *) $y := x \bmod y$; (* replace smallest with new remainder *) $x := \text{gcd}$; (* and larger with smaller from last step *) **END**; WriteString ("The GCD is "); WriteCard (gcd, 0); WriteLn; (* check for another *) WriteString ("Do you want to do another one? "); ReadChar (answer); SkipLine; (* and end of line *) WriteLn; again := (answer = 'Y') **OR** (answer = 'y'); **UNTIL NOT** again; **END** GCD.

Here is a run from this module with some of the on-screen carriage returns deleted to save space.

```
This program computes the GCD of two whole numbers.
```

```
Enter the first number ==> 4780
```

```
The GCD is 20
```

```
Do you want to do another one? y
```

```
Enter the first number ==> 1001
```

```
The GCD is 1
```

```
Do you want to do another one? y
```

```
Enter the first number ==> 5
```

```
The GCD is 5
```

```
Do you want to do another one? n
```

If this example illustrates nothing else, it is that while simple algorithms take considerable intelligent human effort to discover, the repetitive calculations can then be left to the computing device. In this case, the algorithm that was finally coded here was first reported on by the Greek mathematician Euclid about 300 B.C.

[Contents](#)

3.6 Analysis of Loops

In some of the examples of repetition, REPEAT .. UNTIL was more appropriate than the WHILE loop, because it saved a few statements, and because the loop was to execute at least once. Which of the two types of loop one uses depends on whether the test for repetition is to be at the top of the loop or at the bottom, that is, whether one needs to have the loop execute at least once in all cases. This is, however, a minor design consideration, for as the examples showed, it is possible to use the WHILE construction all the time by initializing the value of the controlling variable before entering the loop. Because this is so, some teachers discourage or even forbid their students from using the REPEAT..UNTIL construction at all. That is, there may be considerations other than program design that determine the details of the code. Local customs should be followed in such matters. This text will use the WHILE construction most of the time, but REPEAT will be employed when it is more natural. Many of the comments made in this section about the WHILE loops apply to the REPEAT loop as well.

In any case, the principle of top-down-design as applied to this situation means that one should choose the type of loop before starting to code. There must be a reason for putting each statement on paper, and the programmer must know what that reason is and be able to explain it.

3.6.1 Loops and Boolean Flags

In some programs, it may be necessary to execute the code in the loop indefinitely until some special condition is reached. This was done in several of the earlier examples in this chapter when the bulk of the program code was surrounded by a *REPEAT..UNTIL NOT again* construction.

If a loop can be exited only on some special value of the boolean expression controlling it, that value is called a sentinel value. If a variable is used to hold the value, it may be called a sentinel variable.

Example:

Write a module to simulate a simple four function calculator.

Discussion:

Such calculators maintain two values called the *x-register* and the *accumulator*. The x-register holds the most recently entered value, and the accumulator holds the most recently obtained result. Here, the code will expect an alternating sequence of one-character symbols (the operations) and numbers (the new x-register each time). It will perform the binary operations as (x-register *operation* accumulator)


```

=="FourBanger was written by R.J. Sutcliffe"); WriteLn; WriteString ("to illustrate Boolean flags in
loops"); WriteLn; WriteLn; WriteString ("This program simulates a four function calculator."); WriteLn;
WriteString ("You enter an operation and then a number"); WriteLn; WriteString ("The running result
will be displayed."); WriteLn; WriteString ("If you enter 'E' for the operation the program exits.");
WriteLn; WriteString ("The default operation is = which means 'display number.'"); WriteLn; WriteLn;
(* Initialize *) accum := 0.0; xReg := 0.0; REPEAT (* write out the accumulated value *) WriteFixed
(accum, 6, 25); WriteLn; ReadChar (op); (* no end of line, expect number now *) (* check for operation
*) opOK := (ReadResult() = allRight) AND ((op = "+") OR (op = "-") OR (op = "*") OR (op = "/")); IF
CAP (op) # "E" (* not exit *) THEN (* obtain a number *) ReadReal (xReg); numOK := (ReadResult()
= allRight); SkipLine; IF NOT numOK (* num is bad for some reason *) THEN xReg := 0.0; END; IF
opOK THEN (* go back and see what the operation was and do it *) IF op = "+" THEN accum :=
accum + xReg; ELSIF op = "-" THEN accum := accum - xReg; ELSIF op = "*" THEN accum :=
accum * xReg; ELSIF (op = "/") THEN IF (xReg # 0.0) THEN accum := accum / xReg; ELSE
WriteString ("*** Divide by zero error ***"); WriteLn; END; END; ELSE accum := xReg; END; END;
UNTIL CAP (op) = "E"; END FourBanger.

```

NOTES: A new function procedure CAP has been introduced to replace such boolean expressions as (*op* = "e") *OR* (*op* = "E") by *CAP* (*op*) = "E". CAP is a standard identifier.

A run of FourBanger is recorded below, with some of the on-screen carriage returns deleted to save space.

```

FourBanger was written by R.J. Sutcliffe
to illustrate Boolean flags in loops

```

```

This program simulates a four function calculator.
You enter an operation and then a number
The running result will be displayed.
If you enter 'E' for the operation the program exits.
The default operation is = meaning 'display number'.

```

```

                                0.000000
+9.0                            9.000000
-8.8                            0.200000
*789.0                         157.799850
/8.0                           19.724981
/0.0
*** Divide by zero error ***
                                19.724981
*777.0                         15326.310547
e

```

Observe the necessity of capturing the value returned by *ReadResult* before calling *SkipLine*. If this were not done, and *SkipLine* were called first, the latter would reset the state obtained by *ReadResult* and the actual result of the operation *ReadReal* would be lost.

Instead of having *ReadResult* in a separate module (as in the ISO standard) to indicate success in reading from the standard I/O channel, many older versions of Modula-2 have a boolean flag in the library module that is imported and checked after every read operation. This is true of both the classical *InOut* and *RealInOut* modules. In such versions, one might write:

```
FROM InOut IMPORT
  ReadInt, Done;
```

and then in the program, one could use the value of *Done*, set by *InOut*, to determine whether a correct integer had been typed. Indeed, one could so arrange things that *Done* served as a flag to the program to proceed. If it were not TRUE, the user could be forced to try again until it were. Code could look like this:

```
REPEAT
  WriteString ("Enter the number here ==");
  ReadInt (theNumber);
  numOK := Done;  (* non-ISO version *)
  IF NOT Done
    THEN
      WriteString ("error in number typed ; please try again");
      WriteLn;
    END;
  Read (cr);
  (* these versions read carriage return as a character *)
UNTIL numOK;
```

The classical module *RealInOut*, where available separately from *InOut*, also has a variable *Done* that can be imported into a program module and used by it. The code

```
IF CAP (op) # "E"
  THEN
    ReadReal (xReg);
    numOK := (ReadResult() = allRight);
    SkipLine;
  END;
IF NOT numOK
```

in the example *FourBanger* would be written:

```
IF CAP (op) # "E"
  THEN
    ReadReal (xReg);
```

```
    numOK := Done;  
    Read (cr); (* get carriage return following *)  
END;  
IF NOT Done
```

See [section 3.9](#) for what to do if it is necessary to import both the *Done* from *InOut* and the *Done* from *RealInOut* into the same program.

[Contents](#)

3.7 Counting Loops

Loops are often used to repeat a sequence of steps according to some numbered pattern, instead of doing so until a special condition is realized as in the last section. There may be a predetermined number of times to repeat the loop, or the number of times may depend on the values of one or more program variables.

A repetition of a sequence of program steps according to some numbered pattern is called a counting loop or an iteration.

A couple of typical situations follow:

```
count := 1;  
WHILE count <= 10  
  DO  
    Statement Sequence;  
    count := count + 1;  
  END;
```

or

```
count := 1;  
WHILE count <= 10  
  DO  
    Statement Sequence;  
    count := count * 2;  
  END;
```

The first loop executes for *count* equal to 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10. At the end of the tenth time through, *count* is 11 and the boolean expression controlling re-entry to the loop is now false, so control passes to the next statement following the END of the WHILE loop. The second loop executes with *count* set to 1, 2, 4, and 8, and then terminates with *count* equalling 16. The following simple program, presented without any commentary, illustrates a counting loop. It prints a table of ten consecutive integers together with their squares and cubes.

```
MODULE SquareCube;
```

```
(* Written by R.J. Sutcliffe *)
```

```

(* to illustrate the use of counting loops *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 02 16 *)

FROM STextIO IMPORT
  WriteString, WriteLn, ReadChar, SkipLine;
FROM SWholeIO IMPORT
  ReadInt, WriteInt;

CONST
  numToDo = 10;

VAR
  curNumber, finish : INTEGER;
  answer, cr : CHAR;
  again : BOOLEAN;

BEGIN
  (* write information *)
  WriteString ("SquareCube was written by R.J. Sutcliffe");
  WriteLn;
  WriteString ("as an example in the use of counting loops.");
  WriteLn;
  WriteLn;
  WriteString ("This program computes the squares and cubes ");
  WriteLn;
  WriteString (" of ten integers starting with ");
  WriteString ("the one you provide. ");
  WriteLn;
  WriteLn;

  REPEAT
    (* Gather the information from the user *)
    WriteString ("Please enter the first integer ==");
    ReadInt (curNumber);
    SkipLine;
    finish := curNumber + numToDo;
    (* print headings for the table *)
    WriteString ("Number      Square          Cube          ");
    WriteLn;
    WHILE curNumber <= finish
      DO
        WriteInt (curNumber, 6);
        WriteInt (curNumber * curNumber, 10);

```

```

    WriteInt (curNumber * curNumber * curNumber, 16);
    WriteLn;
    curNumber := curNumber + 1;
END;
(* see if it should be done again *)
WriteLn;
WriteString ( "Do another sequence? Y or N ==");
ReadChar (answer);
again := (answer = "Y") OR (answer = "y");
SkipLine;
WriteLn;
UNTIL NOT again;

```

END SquareCube.

NOTE: In some non-standard versions of Modula-2, the LONGINT type may be needed to hold larger numbers than can be coped with by the INTEGER type. Where both exist, the exact limitations of the two must be determined from the individual system manuals.

Here is a successful run:

SquareCube was written by R.J. Sutcliffe
as an example in the use of counting loops.

This program computes the squares and cubes
of ten integers starting with the one you provide.

Please enter the first integer ==> **y**
Please enter the first integer ==> **n**

On occasion, one might want to terminate a counting loop prematurely, because some condition has occurred that requires it. Such code could look something like:

```

count := start;
WHILE count <= finish
  DO
    Statement Sequence;
    IF interesting condition
      THEN
        count := finish + 1; (* value to guarantee exit from loop *)
      ELSE
        count := count + increment; (* continue processing *)
      END; (* if *)
    END; (* while *)

```

Setting the loop counter to an exit value forces the loop to terminate when the second END is reached. This sort of construction can be used to avoid errors in a program or as a possible evasive action following a check on the validity of input data. This example also illustrates that the number of times a loop will execute may not be known when the program is written. Suppose, for instance, that *start* were greater than *finish* for some reason. The loop would not execute at all, and control would pass to the next statement.

[Contents](#)

3.8 Replacing Loops With Closed Forms

Sometimes, the solution to a problem appears to call for a loop, but closer analysis reveals that the loop can be replaced by a single formula. When this can be done, the code is much more efficient, for just the one formula is evaluated with just a few calculations, rather than many passes through a loop, with one or more computations each time. This situation is common when the repeated computations follow a pattern of some kind. An initial example (and others will be pointed out later in the text) comes from the study of sequences and series, which arise naturally in a variety of applications of mathematics, especially in business

Problem:

Find the sum of the first 1000 positive integers.

Discussion:

The simplest approach, and one that might be hoped for by a primary teacher trying to keep a bright student out of his way for a while, would be to perform 999 additions, using the pseudocode:

```
set the sum to zero
set the current number to one
while the current number is less than 1000
    add the current number to the sum
    increase the current number
print the result
```

A more enterprising student might notice that if the numbers are written down twice, the second underneath the first and backward, the result can be obtained much more quickly:

```
Sum    =    1 +    2 +    3 +    4 + ...
Sum    = 1000 + 999 + 998 + 997 + ...
2*Sum  = 1001 + 1001 + 1001 + 1001 + ... (1000 times)
```

Thus twice the desired sum is $1000 * 1001$, so the actual sum is half this amount. In general therefore, the sum of the first n positive integers can be found by evaluating the formula

$$\text{Sum}_n = \frac{n(n+1)}{2}$$

More generally still, one can observe that the sequence 1, 2, 3, 4, 5, ... is just a particular instance of a class of such sequences with a defined starting point and specific amount by which each term is incremented to get the next. Others include:

2, 4, 6, 8, 10, ...
 23, 31, 39, 47, ...
 100, 201, 302, 403, ...

A sequence $a_1, a_2, a_3, a_4, \dots$ of terms in which there is a fixed common difference d between successive terms is called an arithmetic sequence.

Such a sequence can be expressed as:

$a_1, a_1 + d, a_1 + 2d, a_1 + 3d, a_1 + 4d, \dots$

and the pattern makes it clear that a formula for the n th term is

$$a_n = a_1 + (n - 1)d$$

By following the example of the consecutive integers, the sum of the first n terms is:

$$\text{Sum}_n = a_1 + a_1 + d + a_1 + 2d + a_1 + 3d + \dots + a_1 + (n - 1)d$$

$$\text{Sum}_n = a_1 + (n - 1)d + a_1 + (n - 2)d + a_1 + (n - 3)d + a_1 + (n - 4)d + \dots + a_1$$

$$2\text{Sum}_n = 2a_1 + (n - 1)d + 2a_1 + (n - 1)d + 2a_1 + (n - 1)d + \dots (n \text{ times})$$

Thus the sum is given by:

$$\text{Sum}_n = \frac{n(2a_1 + (n - 1)d)}{2}$$

Or, since $a_n = a_1 + (n - 1)d$, this can be expressed as:

$$\text{Sum}_n = \frac{n(a_1 + a_n)}{2}$$

It is now a simple matter to write a module to find the n th term, and the sum of the first n terms of an arithmetic sequence, without using any loops at all.

MODULE ArithSeq;

```
(* Written by R.J. Sutcliffe *)
(* to illustrate replacing loops with formulas *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 02 16 *)
```

FROM STextIO **IMPORT**

WriteString, WriteLn, ReadChar, SkipLine;

FROM SWholeIO **IMPORT**

ReadInt, WriteInt;

VAR

termnum, first, last, difference, sum : **INTEGER**;

```
answer : CHAR;  
again : BOOLEAN;
```

BEGIN

```
(* information on program *)  
WriteString ("ArithSeq was written by R.J. Sutcliffe");  
WriteLn;  
WriteString ("to illustrate replacing loops with formulas");  
WriteLn;  
WriteLn;  
WriteString ("This program computes the specified term ");  
WriteString ("and partial sum");  
WriteLn;  
WriteString (" of the terms of an arithmetic sequence ");  
WriteString (" of integers that you specify. ");  
WriteLn;  
WriteLn;
```

REPEAT

```
(* Gather the information from the user *)  
WriteString ("Please enter the first term ==");  
ReadInt (first);  
SkipLine;  
WriteString ("What is the difference between terms? ==");  
ReadInt (difference);  
SkipLine;  
WriteString ("What is the number of the term you want ==");  
ReadInt (termnum);  
SkipLine;  
(* computation -- no loops *)  
last := first + (termnum - 1) * difference;  
sum := termnum * (first + last) / 2;  
(* output *)  
WriteString ("Term number ");  
WriteInt (termnum, 0);  
WriteString (" of the sequence is ");  
WriteInt (last, 0);  
WriteString (" , and the partial sum to that point is ");  
WriteInt (sum, 0);  
(* recycle? *)  
WriteLn;  
WriteLn;  
WriteString ( "Do another sequence? Y or N ==");
```

```
ReadChar (answer);  
again := CAP (answer) = "Y";  
SkipLine;  
WriteLn;  
UNTIL NOT again;  
  
END ArithSeq.
```

One run produced the results:

ArithSeq was written by R.J. Sutcliffe
to illustrate replacing loops with formulas

This program computes the specified term and partial sum
of the terms of an arithmetic sequence of integers that you specify.

Please enter the first term ==> **1**

What is the number of the term you want ==> **y**

Please enter the first term ==> **5**

What is the number of the term you want ==> **y**

Please enter the first term ==> **2**

What is the number of the term you want ==> **n**

[Contents](#)

3.9 Qualified Import

To this point, examples have imported from no more than two or three modules. Some programs import from twenty or thirty, and some programmers get to know the contents of those modules very well as their programs grow longer. In addition, there will occasionally be an identifier of the same name in two different library modules. One cannot import both of these, as they have the same name. That would amount to declaring two different variables with the same name--something the compiler forbids, of course. In order to use both these identifiers in a program, one must first include in the import section of the module the line:

```
IMPORT aModuleName;
```

This is a different kind of import and requires one to adopt a slightly different style of documenting and using imported items. Instead of writing

```
FROM STextIO IMPORT  
    WriteString, WriteLn;
```

or

```
FROM SWholeIO IMPORT  
    WriteInt, WriteCard;
```

and referring to the imported identifiers as WriteString, WriteLn, WriteInt, and WriteCard, one may instead write the imports as:

```
IMPORT STextIO;  
IMPORT SWholeIO;
```

If this is done, the program has access to the entire contents of the imported module, but must refer to the items in them by also using the module name, in the manner of the following code:

```
STextIO.WriteString ("The results are:");  
STextIO.WriteLn;  
SWholeIO.WriteInt (theInt,5)  
SWholeIO.WriteCard (theCard, 6)
```

and likewise for any other items in these or other library modules.

An identifier that is constructed from the name of the entity that owns it, followed by a period, followed by its own name, is called a qualified identifier.

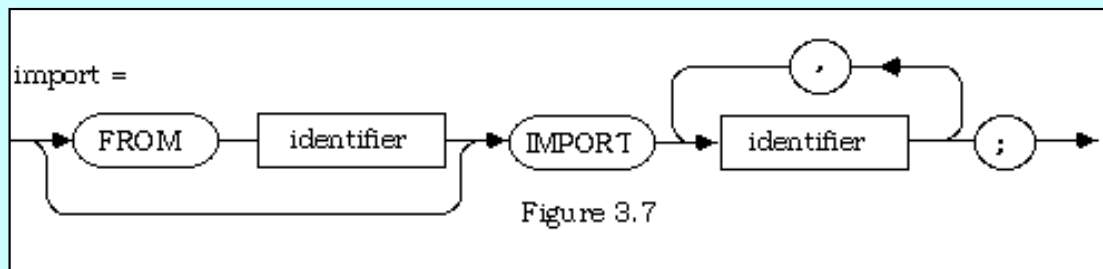
An identifier that is referred to by its name alone is said to be unqualified.

Whenever this is done, any items belonging to the imported module become available for use in the importing module, provided their own names are qualified by their module name, as: *ModuleName.itemFromModule*. Since it may not be necessary to refer to all the identifiers of such a library in a qualified manner, it is permitted to do both styles of importing

from the same module in one program.

NOTE: Programming notations other than Modula-2 may not allow both qualified and unqualified imports from the same library. They may not even have both in the language.

This new syntax forces a revision to the diagram of the import as shown in figure 3.7:



For example, the simple high level I/O libraries sketched by Wirth in *Programming in Modula-2* and variously implemented by a number of manufacturers since, had a global variable *Done* in both *InOut* and *RealInOut* for checking the validity of *Readxx* operations. As these traditional modules (in some variation) are usually provided, the following non-standard example may provide a useful illustration:

Example:

Write a program to raise a real number to a cardinal power. Ensure that the data read from the user is correct before starting the calculations.

Discussion:

The *Done* variables will be used to check input validity, and the user will not be permitted to go on if the data is incorrect. To achieve this, an error message will be given and a retry allowed. However, only two tries will be allowed in each category.

```
MODULE CardPowerOfReal;  (* non-ISO standard *)

(*  Written by R.J. Sutcliffe *)
(*  to illustrate qualified import *)
(*  using Metrowerks Modula-2 for the Macintosh computer *)
(*  last revision 1993 02 16 *)

(* This non-standard program Module will request from its users a real and a cardinal
and will raise the first number to the power of the second. Note that this is not a
very efficient method of doing this; this topic will be returned to. *)

FROM InOut IMPORT
  ReadCard, WriteString, WriteLn, Done, Read;
IMPORT RealInOut;
FROM RealInOut IMPORT
  ReadReal, WriteReal;

CONST
  maxTries = 2; (* typing errors not allowed to go on forever *)

VAR
  baseNumber, result : REAL;
  exponent, count, retries : CARDINAL;
  again, cardOK : BOOLEAN;
  answer, cr : CHAR;
```

BEGIN

```
WriteString ("This program raises real numbers to cardinal powers.");
WriteLn;
REPEAT (* main program repeat loop *)
  retries := 0;

  REPEAT (* trying to get a valid real for the base *)
    WriteString ("Type in a Real ===");
    ReadReal (baseNumber);
    Read (cr); (* get carriage return too *)
    IF NOT RealInOut.Done (* note how this is referred to*)
      THEN
        WriteString ("Incorrectly typed real");
        WriteLn;
        retries := retries + 1;
      END;
  UNTIL RealInOut.Done OR (retries = maxTries);

IF RealInOut.Done (* else, loop finishes; too much bad data *)
  THEN
    retries := 0;

    REPEAT (* trying to get valid cardinal for the exponent *)
      WriteString ("Type in a Cardinal ===");
      ReadCard (exponent);
      cardOK := Done;
      Read (cr); (* get carriage return too *)
      IF NOT cardOK (* unqualified IMPORT reference *)
        THEN
          WriteString ("Incorrectly typed cardinal");
          WriteLn;
          retries := retries + 1;
        END; (* if not cardOK *)
    UNTIL cardOK OR (retries = maxTries);

    IF cardOK (* else, loop finishes; too much bad data *)
    THEN
      result := 1.0; (*initialize answer *)
      count := 1;
      WHILE count <= exponent
        DO (* multiply base by itself enough times *)
          result := result * baseNumber;
          count := count + 1;
        END; (* while *)
      WriteString ("The answer is ");
      WriteReal (result, 10);
      WriteLn;
    END; (* if Done *)

  END; (* if RealInOut.Done *)

WriteString ("Do you want to do another one? ");
Read (answer);
```

```
Read (cr); (* knock the carriage return after the char off *)
WriteLn;
again := CAP (answer) = 'Y';
UNTIL NOT again;

END CardPowerOfReal.
```

NOTE: The variable *cardOK* was employed because the *Read* following the *ReadCard* also resets *Done*. The behaviour of *Done* in such respects varies considerable from one version of non-ISO Modula-2 to another--a good reason to read the documentation carefully, or use a standard (predictable) version.
Here is a run from this program with user input indicated in bold.

This program raises real numbers to cardinal powers.

Type in a Real ==> **4**

The answer is 39.06250

Do you want to do another one? **y**

Type in a Real ==> **a**

Incorrectly typed real

Do you want to do another one? **y**

Type in a Real ==> **t**

Incorrectly typed cardinal

Type in a Cardinal ==> **4.56**

Type in a Cardinal ==> **7**

The answer is 40997.17

Do you want to do another one? **n**

Notice how the various features of the program were all tested to ensure that the error trapping was operating as required.

[Contents](#)

3.10 Insect Control Again--Some Common Errors

The most common syntax errors in writing repetition control structures (WHILE, REPEAT) in Modula-2 are caused by leaving out critical parts of the statements. The compiler can notify one about some of these, or the programmer may see error messages like the following. (Numbers may or may not be attached.)

```
31  Error in REPEAT statement
32  'UNTIL' expected
33  Error in WHILE statement
34  DO expected
```

In such cases, as with incorrect use of "!=" or "=", or missing semicolons, it is usually easy to find the error once the compiler points it out. A missing END can be much trickier, however, because the compiler may not flag any error at all until it reaches the END of the Module and then discovers that it is short by one END. Careful prettyprinting, particularly in blocking out DO .. END sections, should help to prevent this, but sometimes the programmer will need to ask someone else to read the code to discover the "obvious" error that has been stared at for an hour without being seen.

Another potential problem is the never-ending or ad nauseam loop. This usually comes about by specifying a Boolean condition that may never actually happen. These may be simple counting errors such as:

```
cardNum = 5;
WHILE cardNum # 0
  DO
    Statement Sequence;
    card := card - 2;
  END;
```

for which loop the flag value is passed by without being achieved. On the other hand, they may be more subtle, as in the loop:

```
realNumber := 10.7;
WHILE realNumber # 0.0
  DO
    Statement Sequence;
    realNumber := realNumber - 0.1;
  END;
```

The problem here is that REAL values, are not necessarily represented exactly, and because of rounding-off errors, *realNumber* may never equal zero. Its value may get very close to zero, but miss by a small fraction. If possible, it would be better to use, say:

```
realNumber := 10.7;
WHILE realNumber < .00001  (* Is it close to zero? *)
  DO
    Statement Sequence;
    realNumber := realNumber - 0.1;
  END;
```

The case sensitivity of Modula-2 may also be a rich source of problems because of identifiers that the programmer believes

match one another but that do not (*myVariable* is not the same as *MyVariable*, for instance). Beginners get messages like "Identifier not declared or incorrect class" many times until they learn to take sufficient care ahead of time to avoid this problem.

It is also important to ensure that the identifiers themselves convey information about their role in the program. The statement $a := p * x * x$ is much clearer when written as $area := \pi * radius * radius$, for instance. This applies to constants as well as variables, and to any other entities requiring identifiers. The rare quality of common sense is recommended in such matters.

The names of identifiers are a part of the correct internal documentation of a program.

While on the subject of constants, it is worth remembering that they are in fact constant. Many programs are rejected by the compiler because the programmer attempts to reassign a constant. Some people avoid this problem by using variables instead of constants, but it is far better to declare several constants and use them consistently than to attempt to make a variable serve for more than one of them.

However, even if the program compiles and appears to run correctly the first time, it may still contain errors. These are referred to as "run-time" or "execution" errors, and are caused by insufficient planning. Here are a number of methods for detecting and eliminating--or preventing--run-time errors.

1. Include temporary print statements in the code.

If a program has been having trouble with a statement like *WriteReal* (*a/b*, *n*) for instance, the programmer can place the temporary print statements *WriteReal* (*a*, *n*) and *WriteReal* (*b*, *n*) just before the line causing the error message. Perhaps because of an earlier logical error the value of *b* is zero, and of course it cannot then be used as a divisor. Likewise, it may be necessary to print out the value of some counter or boolean value before, during, or after a loop. The temporary print statements can be removed when the program has been completely debugged.

Once the error has been located the incorrect code is re-designed, but with the "test prints" retained. It is then tested again.

Rewriting a program to eliminate one error could have the side effect of introducing another one, so this may have to be done several times.

2. Test a program with alternate data.

If the first keyboard input tried is accepted by a program that obtains its raw data in that way, there is no guarantee that it will always work as expected. If a positive number less than 10 is required, what happens if zero or a negative, or a larger number than 10 is supplied? Will it still work? Code may have to include a trap for invalid data, such as the following:

REPEAT

```
WriteString ("What is the number? ");
WriteLn;
ReadCard (number);
allOK := (ReadResult() = allRight) AND (number < 10);
SkipLine;
IF NOT allOK
    THEN
        WriteString ("Number must be between 0 and 10. ")
        WriteString ("Try again.");
        WriteLn;
    END;          (* if *)
```

UNTIL allOK;

As in previous examples, *ReadResult* is imported from *SIOResults* and can enquire about the result of the last invocation of the procedure *ReadCard*. The value of *allOK* (declared in the program) is in this case set by checking to see if *n* is greater than zero as well; if *allOK* is now *FALSE*, the input must be tried again. A *ReadInt* could have been used instead of a *ReadCard*. In

that case, the value of *allOK* would not be *FALSE* due to a negative number being read, but would be caught afterward in the portion *allOK := (ReadResult() = allRight) AND (number < 10)*.

NOTE: In programs like this one, as well as in non-standard versions employing the global variable(s) *Done*, care must be taken to examine or save the result of the read action before performing another *Readxx* for such action would leave the program with a mistaken view of the situation. This could happen if another *Readxx* were inserted before the *IF NOT allOK* on some later revision of the program. The value returned by *ReadResult* was therefore saved in another variable. This is particularly important when using *InOut.Done*, for some versions of *InOut* will set *Done* on both input and output operations. Others do not require a character read to remove the carriage return from the input stream after a *Readxx*. However, some do, and it is best to plan for such behaviour.

Some readers may be using other non-standard procedures such as *SimpleIO.ReadCard* whose syntax is indicated in the following revision of the above fragment:

```
REPEAT (* not even classical standard *)
  WriteString ("What is the number? ");
  WriteLn;
  ReadCard (number, done); (* resets program variable done *)
  allOK := done AND (number < 10);
  Read (cr, done); (* read character after the number *)
  IF NOT allOK
    THEN
      WriteString ("Number must be between 0 and 10. Try again.");
      WriteLn;
    END;          (* if *)
UNTIL allOK;
```

In such non-standard versions, all *Read* procedures take a second parameter that is a program-defined BOOLEAN.

The point of all this is: Will a program work if supplied with inappropriate data or not? If not, it must have traps for bad data so that there is some action it can take as an alternative to "crashing."

3. Hand check results for reasonability.

It is amazing how many silly answers are accepted just because they were given by a calculator or a computer (or an economist, a politician, or a teacher). Even programmers press the wrong keys all too often. What seems like a perfectly good program producing valid results may in fact be outputting utter nonsense.

A programmer should watch for:

- 1) Answers that are impossibly large or ridiculously small. (Such as ending up with a million dollars in your bank account.... That's funny, I thought it was more like 74 cents.)
- 2) Situations that might cause large roundoff errors. The number of significant figures for a REAL is limited, so numbers close to zero may have a large percentage error. A lengthy chain of computations involving such quantities could yield very inaccurate results.
- 3) The computer prints the same result every time, regardless of the input data. This probably means there is the wrong identifier in a *WriteCard* or *WriteReal* statement or the programmer forgot to write the code that computes the number it is supposed to output.
- 4) Too many things on one line all run together. Some *WriteLn* statements need to be inserted in appropriate places in order to make the output more readable. One could object that poorly formatted output does not constitute a logical error, but such an objection is unlikely to impress a customer (or a teacher).

There shall be more to say about insect control in later sections of the text. As each new statement is considered, new possibilities for errors arise. Meticulous attention to detail in design, coding, and testing is the only way that these can be avoided or detected.

4. Check loops for efficiency and correctness.

A great deal of a computer's time is generally spent on the execution of repetitions or loops. Such sections can often be improved (or corrected so that they work) if special care is given to the following points.

First, progress must be made toward some goal as the loop executes. There is no point in writing such things as:

```
WHILE count < 2
  DO
    number := number + 1
  END
```

or

```
WHILE number1 # number2
  DO
    number1 := number2 + 1
  END
```

because the required condition can never be reached (another ad nauseam loop like those earlier in the chapter). The same kind of thing might happen through carelessness, as in the following example:

```
WHILE count # 0
  DO
    count := count - 2
  END;
```

Here, unless count is both positive and even when the loop is entered, the condition *count* = 0 can never be achieved.

A reader might complain that these examples are trivial and their difficulties obvious. This is correct on both counts, but spectacular and complex code may hide from sight just such follies as these. Buried in a long section inside a loop may lurk the *ad infinitum* dragon which forces the code into an eternal purgatory of repetition until the user presses a reset button, or the central computer decides that the program has used too much CPU time and cuts the user off the line.

Second, to improve efficiency, a loop should not perform an identical calculation on each pass. Such code should be executed before the loop begins. Thus

```
answer := 1;
number := 1;
WHILE number < 10
  DO
    realNumber := x + y + z;
    answer := answer * number;
    WriteCard (answer, 6);
    number := number + 1
  END;
```

should be written instead as:

```
answer := 1;
number := 1;
realNumber := x + y + z
WHILE number
```

Here, the first comment is not closed off because the pair `"*)"` has a space. If comments could not be nested (as in most Pascals), the `"*)"` on the fourth line would

close the first one, thus neatly removing the *INC (count);* from the program. The Modula-2 compiler scans to the end of the program, if necessary, looking for the close of the comment and flags the error when the comment close is not found. In such cases, the programmer is not informed until the error is verified to exist and it may therefore take some time to determine the correct location of the offending code. If such a problem is reported, the editor's *search-and-replace* function ought to be employed to locate the "**)*" typographical error. Some good programming editors also have a "balance parentheses" function that can be employed to find and correct such errors.

6. Watch for spelling and punctuation errors.

When a program is coded and entered with the editor, careful attention must be paid to such things as the spelling, type, and appropriateness of identifiers, the positioning of commas and semicolons, ensuring that quotes on strings are all present and are the same kind of quote (both single, or both double), closing off parentheses correctly, and properly declaring all objects (and with the same spelling) before using them.

Some notations allow *Var* instead of *VAR*, but Modula-2 does not. On the other hand, a Modula-2 compiler does not care if one puts in extra semicolons where none are actually required (before *ELSE*, *ELSIF* and *END*). It does complain if some are left out that are needed to separate statements, or if one is dropped into the middle of a statement, say, before the reserved word *THEN*. Others to watch for include a missing period at the end of a program, failure to match the module name at the *END* with the one in the header, and an incorrect spelling of an imported identifier or a library module. These are all errors that can be detected with a little proofreading before the first attempt at compiling.

[Contents](#)

3.11 Style and Prettyprinting

All the program examples thus far have followed a certain written style. The beginner should study carefully the textual layout (also known as *prettyprinting* conventions) employed for these examples. Observe particularly the use of indentation and blank lines. These conventions are not necessary to produce *working* programs (the compiler couldn't care less about prettyprinting). Rather, they are required to produce *readable* programs.

Style is a matter of good taste, to be sure, but it is more than that: it is a matter of professional ethics as well. Programs written for a client must be readable and modifiable. That is why it is important to incorporate informative and effective comments in the body of programs. Non-trivial commenting for understandability and prettyprinting for readability go hand-in-hand with good overall design. Without these, one cannot produce programs that can be maintained, and cannot therefore produce good programs. Comments included in a program to explain its steps are part of its internal documentation. Because of the difference it makes in readability, prettyprinting is also part of internal documentation. Consider an extreme example of unreadability. Since a Modula-2 compiler ignores carriage returns, a program could be written as:

```
MODULE Test; BEGIN Statement Sequence; END Test.
```

that is, all on one line. In practice, this would make things very nearly impossible to read, and even though the compiler would accept it, no computing science teacher or programming supervisor ever would.

Here are some general rules for laying out programs in a clear and readable fashion:

A. Prettyprinting Rules

1. Blank lines should be placed before such headings as **CONST**, **VAR**, **TYPE**, and **PROCEDURE**. (The latter two will be taken up later.) They should also be placed before and after any comment extending over more than one line. They may also be placed after each **END** but the last if desired. Creative and liberal use of blank lines to set off sections of code is encouraged.
2. Spaces should be placed before and after all arithmetic operations and the symbols ":", "=", and ":", before the opening parenthesis which follows the name of a procedure (such as *WriteString*), and after any commas (say, in import lists.)
3. All statements and declarations begin on a new line.
4. The body of every **CONST**, **VAR**, and **TYPE** declaration as well as of every **BEGIN .. END**, **IF .. THEN .. END** or similar construction, is indented one tab or two spaces from the heading. The **END** of such a structure always appears directly below the keyword that starts it (such as **THEN**, or **DO**).
5. Specifically, the correct indentation for an **IF .. THEN .. ELSIF .. ELSE** is as follows:

```
IF Boolean Expression 1
  THEN
    Statement Sequence 1;
  ELSIF Boolean Expression 2 THEN
    Statement Sequence 2;
    ...

  ELSE
    Statement Sequence n + 1;
END;
```

In this construction, only the first THEN appears on a line by itself. Strictly speaking, it is an integral part of the first IF, and that is why there must be no semicolon before it, but lining THEN up with the ELSIF and ELSE keywords makes the whole block easier to read. The idea is to improve the layout, not to emphasize the THEN as if it were a more important part of the statement.

6. If a FROM is used, the list of imports should start on the line following the library module name.

7. When DO is used, it is indented on a line by itself. This applies to WHILE loops, and to any similar construction.

8. No source line should contain more than about 72 characters.

Besides these layout rules, there are a number of other style rules that should be followed in the choice of identifiers:

B. Identifier Style Rules

1. Use an identifier name which is self-documenting, such as *Principal*, rather than a cryptic character such as *P*.

2. The identifiers of constants and variables start with a lower case letter and are nouns; those of Procedures start with upper case letters. If they perform an action (eg *WriteString*) they are verbs and if they return a value (eg *ReadResult*) they are nouns. BOOLEAN identifiers should normally be adverbs.

3. Multi-word identifiers have each new word after the first starting with a capital (eg *WriteString*). The low-line often used in Pascal is allowed in ISO standard Modula-2, but its use ought to be discouraged by anyone interested in readable programs. Write *myIdentifier*, rather than *my_identifier*.

4. Only reserved words and standard identifiers are entirely capitalized, except in the case of certain imports, where the programmer must simply match the library style. (Authors of library modules should themselves follow this rule.)

5. Module names may have any case, as there may be constraints imposed by the underlying operating system.

Here is a bad example:

```
MODULE BadGradePoint;
FROM STextIO IMPORT WriteString,WriteLn,ReadChar;
FROM SWholeIO IMPORT WriteCard;
VAR x:CHAR;y:CARDINAL;
BEGIN WriteString("What is your letter grade?  A .. F ==");
ReadChar(x);x:=CAP(x);IF x="A"THEN y:=4 ELSIF x="B"THEN y:=3
ELSIF x="C"THEN y:=2 ELSIF x="D"THEN y:=1 ELSE y:=0 END;
WriteString("The grade point for that letter grade is ");
WriteCard (y,2); WriteLn;IF y<2 THEN WriteLn("You failed.");
END;END BadGradePoint.
```

Could you tell what that program was for? (It did compile!) Here it is again in a better form:

```
MODULE BetterGradePoint;

(* This program takes a letter grade input from the keyboard and converts it into a
grade point numeral. *)

FROM STextIO IMPORT
    WriteString, WriteLn, ReadChar;
FROM SWholeIO IMPORT
    WriteCard;

VAR
    userInput : CHAR;
    gradePoint : CARDINAL;

BEGIN
```

```

(* First obtain the letter grade *)
WriteString ("What is your letter grade?  A .. F ==");
ReadChar (userInput);
userInput := CAP (userInput);

(* and now, convert it to the correct number of points *)
IF userInput = "A"
  THEN
    gradePoint := 4
  ELSIF userInput = "B" THEN
    gradePoint := 3
  ELSIF userInput = "C" THEN
    gradePoint := 2
  ELSIF userInput = "D" THEN
    gradePoint := 1
  ELSE
    gradePoint := 0
  END;

(* output results *)
WriteString ("The grade point for that letter grade is ");
WriteCard (gradePoint, 2);
WriteLn;

IF gradePoint >= 2 THEN
  WriteString ("You passed.");
END;

END BetterGradePoint.

```

NOTE: The degree to which prettyprinting and style rules are enforced (as well as the details of these rules) depends not on the implementation of the language, but on the installation (site) at which it is used. Teachers or supervisors will surely vary these rules, but students and other low apprentices do so at their own peril.

[Contents](#)

3.12 An Extended Example (Day Number in a year)

Problem:

Write a program that will calculate the consecutive date number of any day in the year.

Suitability:

This is a tedious calculation by hand. It could be a component of a program that, say, computes interest on periods of time less than a year.

Restatement:

The program is to count January 1 as day number 1, and December 31 as either 365 or 366, depending on whether it is a leap year. The input will be given as three cardinals representing the year, month, and date within the month.

Sample data:

Input:

1990

4

15

Output:

04 15 is day number 105 in 1990.

Library use:

The SWholeIO library will be employed to read and write the cardinal data.

Refinement:

The program DateCalc will:

1. Read data in the form of three cardinals, representing the number of the year, of the month, and of the date in the month, respectively.
2. The consecutive date in the month will be calculated.
3. The result will be printed in the form
mm dd is day number nnn in yyyy.
4. The program will exit.

Second Refinement:

- 1.1 print a prompt to the screen for the year data
read the year
- 1.2 print a prompt to the screen for the month data


```

        read the month
1.3 print a prompt to the screen for the date data
        read the date
2.0 Set the result to zero
2.1 Add to the result the number of days to the end of the previous month
    If month is 12 add 334
        Else if month is 11 add 304
        Else if month is 10 add 273
        Else if month is 9 add 243
        Else if month is 8 add 212
        Else if month is 7 add 181
        Else if month is 6 add 151
        Else if month is 5 add 120
        Else if month is 4 add 90
        Else if month is 3 add 59
        Else if month is 2 add 31
2.2 Add the date in the month to the result
2.3 Adjust for leap years
    If the month is greater than 2 then
        If the year is divisible by 4 but not by 100 (unless also by 400)
            add one to the result

```

Data Table:

```

1. Imports required
    From STextIO: WriteString, WriteLn, ReadChar, SkipLine
    From SWholeIO :ReadCard, WriteCard
2. Variables Required:
    year, month, day, result : Cardinals
    response : Char

```

User Manual:

DateCalc is a simple utility designed to compute the consecutive number in the year of any date entered. DateCalc runs on any Macintosh computer.

To use DateCalc, either type its name, followed by the <enter>

After noting the information, press a key and the program will exit.

Errors:

DateCalc will not check to see if you have entered a valid date; the responsibility is the user's to avoid such things as 1990 02 34.

Possible future enhancements:

Add the option of reading the data from a disk file

Add checking for the validity of the date.

Improve the efficiency of the computation by devising a formula.

Code:

```

(* Name: Daniella Christian
   Student Number: 052001
   CMPT 141 Fall 2008
   Assignment #2

```

```

    Calculating date numbers
*)

MODULE DateCalc;

FROM STextIO IMPORT
    WriteString, WriteLn, ReadChar, SkipLine;
FROM SWholeIO IMPORT
    ReadCard, WriteCard;

VAR
    day, month, year, result : CARDINAL;
    usingFile, again : BOOLEAN;
    response : CHAR;

BEGIN
    (* write out header information *)
    WriteString ("Name: Daniella Christian");
    WriteLn;
    WriteString ("Student Number: 052001");
    WriteLn;
    WriteString ("CMPT 141 Fall 2008");
    WriteLn;
    WriteString ("Assignment #2");
    WriteLn;
    WriteString ("Calculating consecutive date numbers");
    WriteLn;
    WriteString ("This program calculates the consecutive ");
    WriteString ("date in the year ");
    WriteLn;
    WriteString ("from user supplied information");
    WriteLn;
    WriteLn;

    REPEAT (* main repeat loop *)
        WriteString ("Enter the year number here ====");
        ReadCard (year);
        SkipLine;
        WriteLn;
        WriteString ("Enter the month number (1 - 12) here ====");
        ReadCard (month);
        SkipLine;
        WriteLn;
        WriteString ("Enter the day number here ====");
        ReadCard (day);
        SkipLine;
        WriteLn;

        (* do the calculation *)
        result := 0;  (* initialize the result *)

        (* add on right number of days to end of last month *)
        IF month = 12
            THEN

```

```

    result := result + 334
ELSIF month = 11 THEN
    result := result + 304
ELSIF month = 10 THEN
    result := result + 273
ELSIF month = 9 THEN
    result := result + 243
ELSIF month = 8 THEN
    result := result + 212
ELSIF month = 7 THEN
    result := result + 181
ELSIF month = 6 THEN
    result := result + 151
ELSIF month = 5 THEN
    result := result + 120
ELSIF month = 4 THEN
    result := result + 90
ELSIF month = 3 THEN
    result := result + 59
ELSIF month = 2 THEN
    result := result + 31
END;

```

```

(* now add the day in the month *)
result := result + day;

```

```

(* finally, adjust for leap years *)
IF (month " is day number ");
WriteCard (result, 4);
WriteString (" in that year.");
WriteLn;
WriteLn;

```

```

WriteString ( "Do you wish to do another? Y or N ==");
ReadChar (response);
again := CAP (response) = "Y";
SkipLine;
WriteLn;

```

```

UNTIL NOT again;

```

```

END DateCalc.

```

Results:

This module was run with the following result:

```

Name: Daniella Christian
Student Number: 052001
CMPT 141 Fall 2008
Assignment #2

```

Calculating consecutive date numbers

This program calculates the consecutive date in the year
from user supplied information

Enter the year number here ====>3
Enter the day number here ====> y
Enter the year number here ====>12
Enter the day number here ====> n

Further refinement:

There is an algorithm that allows the calculation to be done in a formula rather than by using the IF ... THEN construction. To see how this works, consider first the "rough estimate" that each month has 30 days, and consider how the 31 day months affect the total number of days to be added for the following month. We ignore February for the moment.

Mon:	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct
Nov	Dec									
No.:	1	2	3	4	5	6	7	8	9	10
11	12									
Adj:	0	1	1	2	2	3	3	4	5	5
6	6									

Each time there is a 31 day month, one must adjust the total by 1. Now, up to August, the adjustment formula is: month DIV 2, but after that it is: (month +1) DIV 2. This can be expressed in a single adjustment as: (month + month DIV 9) DIV 2. From this one must subtract 2 if the month is after February, and this quantity can be expressed in a formula as: 2*((month + 9) DIV 12). Try it!
The leap year adjustment changes the "2" in front of this expression to: 2 - (4 - year MOD 4) DIV 4 + (100 - year MOD 100) DIV 100 - (400 - year MOD 400) DIV 400) [The first term of this produces a 1 when the year is divisible by 4, the second removes it when the year is divisible by 100; and the third adds it back in when it is divisible by 400. Thus the two day adjustment for an ordinary February is altered as appropriate to a one day adjustment.]
Combining all these gives the result in a single Modula-2 expression:

```
result := 30 * (month -1) + day +
          (month + (month DIV 9)) DIV 2 -
          (2 - (4 - (year MOD 4)) DIV 4 +
            (100 - (year MOD 100)) DIV 100 -
            (400 - (year MOD 400)) DIV 400) *
          ((month + 9) DIV 12);
```

and this does the entire calculation, eliminating the two IF..THEN constructions in the original code.

Yet another refinement:

The formula used in the second version is rather cumbersome, and may be difficult to understand, especially if the code is examined without the analysis leading up to it (and perhaps even then). Perhaps a compromise ought to be struck between the easy low-technology method with its long IF .. THEN statement, and the high-technology method with its complex formula. Consider a further analysis of the problem. Starting in March, and going through to the following February, the total number of days and the average number of days from March first to the end of the previous month is given by:

Mon:	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Jan	Feb									
No.:	0	31	61	92	122	153	184	214	245	275
306	337									
Av:	0	31	30.5	30.7	30.5	30.6	30.7	30.6	30.6	30.6

30.6 30.6

This chart suggests numbering the months starting in March and multiplying this number by 30.6 to get the number of days elapsed to the first of the current month. Using a method similar to that above, the month number is adjusted by adding 12 if it starts out at 1 or 2. This initial adjustment changes the range of months from 1 .. 12 to 3 .. 14.

```
month := month + 12 * ( 12 - month) DIV 10
```

The expression

```
(month - 3)
```

is needed in the final formula to change this range to 0 .. 12, counting from March to February.
The chart below of the revised month numbers gives the result of doing this, first as a raw answer, (*Av) then rounded off to the nearest day (#). The latter are exactly what are desired.

Mon:	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Jan	Feb									
MNo.:	0	1	2	3	4	5	6	7	8	9
10	11									
*Av:	0	30.6	61.2	91.8	122.4	153	183.6	214.2	244.8	275.4
306	336.6									
#	0	31	61	92	122	153	184	214	245	275
306	337									

Rounding a number off to the nearest whole number can be expressed as:

```
cardNum := TRUNC (realNum + 0.5)
```

This gives, for the number of days from last March 1:

```
daysSinceMarch := TRUNC (30.6 * (month -3) + 0.5);  
(* with revised month *)
```

Adding 59 to the result for the first two months, and then removing the excess days (January and February have, in effect been moved to the following year), produces:

```
result = (59 + daysSinceMarch + day)
```

followed by a reduction if greater than 365 (because January and February are not really in the next year, it just made it an easier formula to consider them that way for the moment) and an increase if a leap year.
The code can be revised to have three more variables, *adjMonth*, *daysSinceMarch*, and *leap* and then modularized into a series of steps, instead of rolling everything into a single formula. That is, the algorithm can be expressed on several lines instead of one:

```
leap:= (year MOD 400 = 0)  
      OR ((year MOD 4 = 0) AND (year MOD 100 # 0));  
adjMonth := month + 12 * (( 12 - month) DIV 10);  
daysSinceMarch := TRUNC (30.6 * FLOAT (adjMonth - 3) + 0.5);  
result := (59 + daysSinceMarch + day);  
IF result > 2)
```

```
THEN  
    INC (result);  
END;
```

NOTE: DEC is a standard identifier similar to INC.

There are other ways of doing this as well. It would be nice, for instance, to have a procedure for rounding real numbers off to the nearest integer or cardinal. No such procedure is built in to Modula-2, though there are some in the *RealMath* and *LongMath* libraries, and in the next chapter, directions will be given for writing one.

[Contents](#)

3.13 Chapter Summary

This chapter covered these topics:

- how to employ selection in a Modula-2 program IF .. THEN
- about BOOLEAN variables and expressions.
- more about repetitive statements
 - the WHILE .. DO .. END
 - the REPEAT .. UNTIL
- how to do qualified import
- more about design and bug-proofing
- about organizational style and prettyprinting.

It included discussion of the following Modula-2 built-ins:

Reserved Words:	Standard Identifiers:
AND	ABS
ELSE	BOOLEAN
ELSIF	CAP
IF	DEC
NOT	FALSE
OR	HALT
THEN	INC
TO	TRUE
REPEAT	
UNTIL	
IMPORT (new sense)	

Symbols:	Imports:
&	From SIOResult:
<=	ReadResult
<	ReadResults
>=	
>	
# or <>	
~	

Contents

3.14 Assignments

Questions

1. What is a boolean variable?
2. What is the correct order of evaluation for boolean expressions?
3. Given the boolean variables p, q and r with p = TRUE, q = FALSE and r = TRUE, what are the values of the following BOOLEAN expressions?
 - a. p AND q OR NOT r
 - b. p OR q AND NOT r
 - c. p OR q AND q OR r
 - d. p AND r AND NOT q
4. Evaluate the following BOOLEAN expressions.
 - a. $(1 \leq 5) \text{ OR } (6 \text{ DIV } 2 = 3)$
 - b. $(3 + 5 < 9) \text{ AND } (14 \text{ MOD } 3 = 2)$
 - c. $(5 \# 7) \& (2 \text{ DIV } 5 < 1)$
 - d. $(7 \text{ DIV } 2 - 13 \text{ MOD } 5 < 8) \text{ AND NOT } ((2 = 9) \text{ OR } (7 > 39) \text{ OR } (-3 < 20 + 4 * -3))$
 - f. $\text{NOT}(5 + 4 < 1)$
5. What could be improved in the statement sequence:

```
IF finished = TRUE
    THEN
        a := a + 1
    END;
```

6. What could be improved in the statement sequence:

```
flag := TRUE;
WHILE flag
    DO
        WriteString ("enter a cardinal here ==");
        ReadCard (myCard);
        flag := (ReadResult () = allRight);
    END;
```

7. What is wrong with the loop:

```
count := 11;
```

```

WHILE count # 0
  DO
    statement sequence;
    DEC (count, 2);
  END;

```

8. What is wrong with the loop:

```

REPEAT
  count := 7;
  statement sequence;
  DEC (count, 2);
UNTIL count < 0;

```

9. What is wrong with the loop:

```

realCount := 10.0;
REPEAT
  statement sequence;
  DEC (realCount, 2);
UNTIL realCount < 0;

```

10. What is wrong with the loop:

```

realCount := 10.0;
REPEAT
  statement sequence;
  realCount := realCount - 1.0;
UNTIL realCount = 0;

```

11. Show by writing a few lines of code in a skeletal framework, how to check the outcome of read operations in *STextIO*, *SWholeIO* and *SRealIO*

12. Show by writing a few lines of code in a skeletal framework, how one could import a variable called *Done* from each of two modules called *RealInOut* and *InOut* into the same program. Include code showing how the two are referred to in the program.

13. Check the manuals for your system, and write down how to do the following (where applicable):

- obtain a printout of a program
- obtain a "compile time listing" with statistical details on the lines, an index into the code compiled, and other details as may be available.
- obtain a "hard copy" of a run of a program
- list data files to the printer

14. Research and describe the "year 2000 problem."

15. List and discuss several common strategies for avoiding errors; for finding errors after they have occurred.

Problems:

16. Write a program that will either convert inches to centimetres or vice versa, depending on user input at run time.

17. Write a program that will compute the perimeter of either a square given one side, a triangle given all three sides, or a circle given the radius. The task is chosen by the user and the program must ask the appropriate questions in each case.

18. Expand the small section of code related to [figure 3.2](#) into a full blown program that will ask for four marks, find their average, and then print appropriate pass/fail messages.

19. Modify the program [SortTwo](#) to sort three real numbers into order.

20. Modify the program [AverageWordLength](#) to improve the algorithm for determining the number of words. Add statements to keep track of and print out the number of words of 1, 2, 3...10, and more than 10, letters. Ensure that the statistical summary you print at the end is clear and easy to read.

21. The least common multiple (LCM) of two numbers is the smallest number that they both divide into. Write a program LCM to calculate this from two inputs. (Hint: research this first; there is a relationship between GCD and LCM.)

22. Extend the mini-calculator to include exponentiation (x^y) as well as the option to add, subtract, multiply and divide.

23. Research, plan, write and test programs to compute

a. the sum of the first n odd numbers $1 + 3 + 5 + 7 + \dots + 2n-1$

b. the sum of the first n even numbers $2 + 4 + 6 + 8 + \dots + 2n$.

c. the sum of the first n squares $1 + 4 + 9 + 16 + \dots + n^2$.

In each case, you must not use a loop to do the calculation.

24. Extend the GCD program further to allow for three input numbers; for n input numbers.

25. Modify #16 above to ask how many conversions are to be done and carry on from there.

26. Write a program which will read in a sequence of cardinals from the keyboard and then inform the user what is:

a. the largest of the numbers.

b. the smallest of the numbers.

c. the number of cardinals entered.

d. the total of the numbers.

e. the average of the numbers.

The user should be prompted to enter some specific value as a flag to terminate the sequence. (Hint: You could use zero for this, but a letter would work too, as it would set *SIOResult.ReadResult()* to a value other than *allRight*). The five pieces of information all need to be computed, but a menu should be given to the user and only the statistic that the user *wants* should be provided.

27. The Board of Governors at Trinity Western University has decided to grant a 3% pay raise, and they wish to know

a. what the new salaries will be

- b. what the total new salary bill will be
- c. how much extra money this will cost the University.

Write a program that will answer all their questions, given the following table of present salaries.

Amount	Number of faculty receiving this
\$41 000	2
\$41 300	4
\$42 000	8
\$42 100	6
\$43 600	5
\$43 000	2
\$44 000	8
\$44 500	2
\$45 000	2
\$45 300	1

28. Modify the solution to #27 so that those receiving less than \$42 500 receive 4%, those between \$42 501 and \$43 999 receive 3% and those receiving \$44 000 or more receive 2%. Compare the total cost of the two methods.
29. Write a program which prints out the decimal equivalents of the series of fractions $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, etc. in a column. (You can't continue this forever, but do it for fourteen steps.) Now print a column beside this one with the sum of all the fractions printed up to that point. The numbers should start .5, .75, .875, etc. Because the intermediate results are wanted, one cannot dispense with a loop. Can the sum ever reach the total one? Why or why not?
30. Check your answer to #29 by writing a loop to add the first twenty terms in this sequence and print just the result. The first fifty. The first hundred. Hmmm.
31. Modify the Module [BetterGradePoint](#) to use reals for the points and to compute the grade point average of up to five subjects. You could use a nonexistent grade such as "Z" to indicate when the averaging is finished.
32. Add to #31 the ability to take into account the number of semester hours of credit that a course is worth, so that courses have their individual grade points weighted by the number of credits (0 - 5) that the course is worth. Add also the ability to give an extra third of a point for courses with marks like "A+" or "C+" and to subtract a third of a point for those with the corresponding "minus" letter grades.
33. Go back to Chapter 2, problem #38-39. Expand this program to handle addition, subtraction, and division as well as multiplication. The choice of which operation to do will be up to the user. In the case of division, the full long division algorithm is to be written out with all the steps shown, along with the quotient and remainder.
34. Using the GCD algorithm developed in this chapter, develop a program which will "reduce" common fractions and print out the result in the following form:

$$\begin{array}{rcl} 60 & & 3 \\ --- & = & --- \\ 100 & & 5 \end{array}$$

35. Add to #34 the ability to read in the fraction to be reduced as a decimal numeral.
36. Write a program that accepts a positive cardinal and either produces its prime factors or states that it is a prime.
37. Write a program to process payroll data. Numeric input should consist of the hourly pay, number of hours at regular time, and number of hours overtime (paid at time-and-a-half.) Deduct 6% for pension, 2.5% for unemployment insurance, 20% for taxes, and \$20 if the employee is in the stock sharing plan. Output should consist of a neat table showing hours, gross pay, deductions, and net pay.
38. Write a program to convert the units miles per gallon to the units litres per hundred kilometres and vice-versa. (Look up conversion factors.) Test the program by having a user input distances and fuel consumption in either metric, imperial (or U.S.) units, and producing both results as output.
39. Write a program that will find the sum of the digits of a number and then say if it is divisible by 9, by 3 but not 9, by neither.
40. The motion of an object under a constant acceleration a is given by the formula $d = ut + .5at^2$, where d is the distance travelled, t is the time elapsed in seconds, and u is the initial velocity. Write a program that will take these inputs and print a table of values of distance vs time for a period of 20 seconds at intervals of one second; for a period of n seconds at intervals of Δn seconds.
41. Write a complement to the program [DateCalc](#) that accepts as input the year, and the day number within that year and produces the date.
42. Add to the consecutive date calculation program the ability to check for invalid dates. (Hint: convert a date, and convert it back. Does it equal what it started as or not?)
43. Look up Julian dates and write a program to accept as input a year, month, and day and compute the Julian date.
44. Add to #43 the ability to convert Julian dates to conventional dates.
-

[Contents](#)

First Interregnum--A Short History of Computing

People have long realized the competitive advantages that could be realized by having available more efficient data storage and computational ability. From counting on the fingers, to making marks on the walls of caves, to the invention of picture numbers, to the modern check or banknote, there has been a steady progression away from directly manipulating the objects that computations describe and toward the use of abstractions for the originals. Mechanical devices have played an important part in this sequence. More than one culture came up with the idea of placing beads on a string (the abacus.) In some places, these are still the preferred calculating device after several thousand years. A skilled operator can calculate the cost of a large number of purchases on an abacus much faster than most people could enter them into a calculator.

Some who have studied the ancient British monument known as Stonehenge have come to the conclusion that it was an enormous calculating device for making astronomical predictions. Other monuments left by the Babylonians, South and Central American Indians, and South Sea Islanders may have had similar purposes. The Scottish mathematician John Napier (1550-1617) devised Napier's bones and published tables of logarithms intended to simplify tedious arithmetic computations. These led directly to the wooden or bamboo slide rule, known and loved by many student generations prior to the development of inexpensive electronic calculators.

To the French mathematician and theologian Blaise Pascal (1623-1662) goes the honor of inventing the first mechanical adding machine (1642). It was based on a system of gears similar to those in a modern automobile odometer and was used for computing taxes. However, parts for the device could not be manufactured with sufficient precision to make it practical, and it never became widely used. About thirty years later, the famous German mathematician and co-inventor (with Newton) of calculus, Gottfried Wilhelm von Leibniz (1646-1716), made a similar but more reliable machine that could not only add and subtract but also multiply, divide, and calculate square roots. There were many people who improved calculating machines over the next century, and by 1900 they had an important place in government and commerce. As late as the mid 1960s electromechanical versions of these calculators could do only basic four function arithmetic, weighed thirty pounds, and took up half a desktop.

Meanwhile, another idea important to the modern computer was emerging--that of the stored program or instruction sequence. This idea arose in connection with the development of automatic looms by the French inventor Joseph Marie Jacquard (1752-1854). First shown at the 1801 Paris Exhibition, these looms used a collection of punched metal cards to control the weaving process. The machine, with some variations, is still used today, though it is now controlled by punched paper cards or tapes, or by direct connection to a microcomputer.

The first computer--a machine combining computational ability with stored programs--was designed by the British mathematician Charles Babbage (1792 - 1871). He worked on his "Difference Engine" for about eleven years before abandoning the project. Later, he designed a much more ambitious "Analytical Engine," that was intended to be an algebraic analogue of Jacquard's loom. Although Babbage even had a programmer for the engine (Lord Byron's daughter, Ada Augusta, the Countess of Lovelace,) this

machine was never constructed in his lifetime. Its concepts were not realized until 1944 when the Mark I computer was developed in the United States.

By this time, the punched paper medium had become standardized through the work of Herman Hollerith. He devised a card data storage and sorting system for the U.S. Census Bureau, which was first employed in the 1890 census. Hollerith left the bureau six years later to form his own company, the name of which was changed to International Business Machines in 1924.

Meanwhile, vacuum-tube technology had developed to the point where an electronic computer could be manufactured. The first of these were the British code-breaking devices Colossus Mark I and Colossus Mark II built in 1943 and 1944 for the British intelligence service at Bletchley Park. The latter attained speeds not matched by other computers for a decade. When the war was over, these machines were dismantled and their parts sold for surplus.

At about the same time, the groundwork of a number of researchers in the United States came to fruition in the construction of the Electronic Numerical Integrator and Calculator (ENIAC) by J. P. Eckert and J. W. Mauchly at the University of Pennsylvania. This machine, which contained over 18,000 vacuum tubes, filled a room six meters by twelve meters and was used principally by military ordinance engineers to compute shell trajectories. In subsequent years, many similar computers were developed in various research facilities in the United States and Britain. Such devices, which generally were limited to basic arithmetic, required a large staff to operate, occupied vast areas of floor space, and consumed enormous quantities of electricity.

Eckert and Mauchly were also responsible for the first commercial computer, the Universal Automatic Computer (UNIVAC,) which they manufactured after leaving the university. Their company was eventually incorporated into Sperry (now merged with Burroughs to become UNISYS), which still manufactures large industrial computers. Today, those early vacuum-tube monsters are referred to as "first-generation computers," and the machines that are their successors are called "mainframes."

The transistor, developed at Bell labs in late 1947, and its improvement during the early 1950s, was designed to replace the vacuum tube, reducing both electrical consumption and heat production. This led to the miniaturization of many electronic devices, and the size of typical computers shrank considerably, even as their power increased. The transistorized machines built between 1959 and 1965 formed the second generation of computers.

The price was still in the hundreds of thousands to millions of dollars, however, and such machines were generally seen at first only in the headquarters of large research and government organizations. Even by the mid-1960s, not all Universities had even one computer, and those that did often regarded them as exclusive toys for the Mathematicians and research scientists. There were occasional courses at the fourth-year level, but Freshman introductions to Computer Science had not yet become popular.

The invention of the integrated circuit dramatically changed things in the computing world. The first result was another, even more significant size reduction, for what once took up several floors of a large building now occupied a small box. The first of these third-generation computers was the IBM System 360, which was introduced in 1964 and quickly became popular among large businesses and universities. This size reduction also resulted in the first "pocket" calculators, which appeared on the market in the early 1970s. Even at the initial price of several hundred dollars, these put into the hands of the average person more computing power than the first UNIVAC had possessed. New models proliferated so rapidly and so many new features were incorporated into the pocket calculator that one company decided to have a chip designed that would allow it to program new functions so as to cut down the time necessary to

bring a new model to market.

The chip, called the 4004, gave way to the 8008, and then to the 8080 and 8080A. The latter became the backbone of the new small-computer industry, as numerous companies developed kits and fully assembled computers. In its later reincarnations by Zilog as the Z-80 and other descendants, such as the 8085, 8088, 8086, and now the 80186, 80286, 80386, 80486, Pentium, and P6, this invention lives on in millions of microcomputers. Not long after the 8080 became a commercial reality, Motorola developed the 6800 chip, which had the advantage to programmers of being cheaper and somewhat easier to work with than the 8080. It, too, became popular for a time, but soon gave way to other designs.

At about the same time the Z-80 was developed, the 6501 and 6502 chips were derived from the 6800 as low-cost industrial process controllers. In 1976, the 6502 was also used to build a small computer, this one entirely contained on a single board. It was called the Apple I, and Apple Computer Corporation went on to sell millions of the Apple][and its descendents, the][+, //e, //c and //GS, surpassing all other manufacturers of small computers in the process.

In 1977, Radio Shack joined the competition with its Z-80 based machines. In Europe, the equivalent popularizing role was played by Commodore (a Canadian company) and by Sinclair (a British firm). A few years later, IBM came into this market with the 8088-based PC. The mere presence of the giant changed the whole market for a time, with most other manufacturers seeking to make machines compatible with those of IBM. Eventually some of these "clone" makers, such as Compaq, became a larger presence in the market than IBM itself. By the mid 1990s, the machines generating the most attention were capable of storing more and manipulating larger numbers than anything previously seen in the microcomputer market. They were also capable of handling the processing requirements of the graphical user interface (GUI) first realized in the Xerox Star, the Apple Lisa and Macintosh, then in Commodore's Amiga and Atari's machines, and now being demanded by most computer users. The integration of circuits had now reached the point where millions of components were being crammed into a single chip. Between 1987 and 1991, major new commitments were made by Apple with the Motorola 68030-based Macintosh IIfx, IIfc and IIfi models and by IBM with their OS/2 machines. With the latter, IBM also followed Apple's lead into graphics-oriented software, helping to ensure this style of interface a continuing acceptance in the marketplace. Graphical user interfaces were also adopted by the makers of scientific workstations such as those made by Sun Microsystems, and were being attached to other machines running the UNIX operating system.

In the early 1990s, Microsoft, already the dominant manufacturer of operating systems for the Intel 80x86 chips and of applications for both these and the Macintosh platforms, had begun to market a GUI called Windows that was a rough imitation of the Macintosh Operating System. The courts ruled, however, that it was not a close enough imitation to fall under copyright law, and Windows gradually became dominant on the Intel based machines.

By 1995, Apple had formed partnerships with Motorola and IBM to develop new microprocessor technology and was already marketing machines based on the new PowerPC RISC chip, while IBM was porting its operating systems to the new chip as well. The two were readying new operating systems and preparing specifications for a common hardware platform on which to run them. Apple had licensed its operating system and the first Macintosh clones were appearing on the market. Microcomputers had become powerful enough that the minicomputer category had been all but crowded out of the market on price/performance considerations.

The late 1990s and early 2000s saw more changes. By this time, Apple had switched to the PowerPC

chips from Motorola and had rewritten its operating system as OS X to sit on top of free BSD UNIX for added stability, more security, and better multitasking. IBM's OS-2 had all but vanished from the marketplace, and, while newer variants of Microsoft Windows were dominant in consumer machines, many hobbyists were experimenting with LINUX, an open source version of UNIX, also out of concerns for stability and security.

While much of the marketing activity and most of the headlines were focusing on the microcomputer segment of the industry, the larger machines had undergone some startling changes as well. The fourth generation of supercomputers can be used in situations where the complexity of the calculations or the quantity of data is so great as to be beyond the ability of even an ordinary mainframe device. These machines are used by governments, the military, and in academic research institutions. Still newer generations of computers are on the drawing boards in the United States and Japan, and many of the new developments will undoubtedly filter down to become more consumer-oriented devices in the future. At the opposite end of the scale, pocket sized computing devices had also become important to some people. These ranged from the DOS based miniaturized version of the desktop sibling to the specialized personal time and communications organizer (Personal Digital Assistant or PDA.) Also called the Personal Intelligence Enhancement Appliance (PIEA) these devices boast handwriting recognition, wireless communications abilities, and sophisticated time management functions.

For most applications in the near future, however, microprocessor-based computing devices will have sufficient power to suit the majority of individual, academic, and business uses. They are inexpensive, easy to link (network) together for sharing other resources (such as storage devices and printers), and they run languages and other programs similar to those found on mainframe computers. Much of the new development work (particularly in programming and publishing) is being done with microcomputers in view, and it is safe to predict that the descendants of these machines are the ones most people will be referring to when they speak about computers in the future.

The larger machines, however, will also continue to grow and change, as will the organizations depending on them. Moreover, the computers of the future will be as different from those of today as these are from the ones of the late 1940s. They will be smaller (down to pocket size), faster, and with greater storage capacity. They will be integrated with video and communications technology to give immediate access to worldwide data bases. They will undoubtedly become easy to use, and at some point the need to offer university level courses in their operation will cease, for they will have become common technical appliances.

The Internet, especially the portions known as the World Wide Web (WWW) has become a kind of prototype for the universal distributed library (the Metalibrary) of the future, and most organizations have connections for e-mail, if for nothing else.

Computers have already profoundly changed many of society's institutions (business, banking, education, libraries). They will have an even greater effect on institutions in the future. They have also raised or caused new ethical issues, and these will need to be addressed in the interests of social stability. In addition, developments in computing have affected or given rise to other new products and methods in a variety of fields, further demonstrating the interdependence of ideas, society, and technology.

There are microprocessors in stereos, televisions, automobiles, toys and games. The entertainment and telecommunications industries are heavily dependent on new electronic technologies. Computers themselves are directly attached to research instruments that gather and interpret data in basic physics, chemistry, and biology experiments. The resulting changes and advances in scientific research have also

caused profound effects on society and its institutions. They have resulted in new social and ethical questions being raised, whose very asking could not have been anticipated in the industrial age. These include issues relating to software copyright, data integrity, genetic engineering, artificial intelligence, displacement of human workers by robots, and how to live in and manage an information-based society. Some of the technical trends and the possible social and ethical consequences will be examined and extrapolated in more detail in later sections of the book. It is at least possible to conclude at this point that the advent of the fourth civilization (aka "the information age") is owed more to the modern computer than to any other single invention of the late industrial period.

-- from the first edition; revised 1995 06 13 and 2002 06 10

Assignments

1. Write a full biography (minimum 1000 words) of one of the individuals whose name is important in the history of computing.
2. Select a specific time or technology important to the history of computing and write a fuller (minimum 1000 words) exposition than that contained here.
3. Write your own "future history" of computing.

[Contents](#)

Chapter 3

Basic Program Structure Abstractions

- [3.0 Chapter Goals](#)
- [3.1 Statement Sequences](#)
- [3.2 Simple Selection](#)
- [3.3 Boolean Variables and Expressions](#)
- [3.4 Repetition \(1\) -- the WHILE Statement](#)
- [3.5 Repetition \(2\) -- the REPEAT Statement](#)
- [3.6 Analysis of Loops](#)
 - [3.6.1 Loops and Boolean Flags](#)
- [3.7 Counting Loops](#)
- [3.8 Replacing Loops With Closed Forms](#)
- [3.9 Qualified Import](#)
- [3.10 Insect Control Again--Some Common Errors](#)
- [3.11 Style and Prettyprinting](#)
- [3.12 An Extended Example \(Day Number in a year\)](#)
- [3.13 Chapter Summary](#)
- [3.14 Assignments](#)
 - [First Interregnum--A Short History of Computing](#)

4.0 Chapter Goals

The purpose of this chapter is to elaborate on the *composition* abstraction for collecting sections of code under a single name. When the code in question is needed in the main program, only its name need be written. In Modula-2, composition is achieved by using the procedure mechanism, the code for which is placed in the declaration part of a module, and that can be used freely throughout the main program. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

Procedures as data types

Realized in the Modula-2 notation:

procedure types

Data Manipulation Abstractions

General:

code encapsulation, input and output parameters

Realized in the Modula-2 notation:

procedure value and variable parameters; formal and actual parameters

Programming Abstractions

General:

Top-down design and planning is extended to breaking a problem into sub-tasks that are then encapsulated as separate sections of code and called upon as desired.

Realized in the Modula-2 notation:

procedures, function procedures, recursion

4.1 What is a Procedure, and Why Use It?

This text has employed procedures many times already, even though no formal definition of the concept has been supplied. Any time a program uses something like:

```
WriteString ("answer here");  
ReadReal (theNumber);
```

or

```
INC (aNumber);
```

it relies on code found elsewhere to operate upon some item(s) and/or to obtain a result. Many of the built-in and imported identifiers name procedures.

A procedure is a unit of code designed to perform a particular sub-task of some main task. It is elaborated (written out) only once in some module, but can be used many times.

One way to think of a procedure is as a *black box* that performs some action whenever it is invoked by the flow of the program. The action taken may:

(i) be the same every time, as in:

```
WriteLn;
```

(ii) depend on data supplied to it, as in:

```
WriteString (string to write);
```

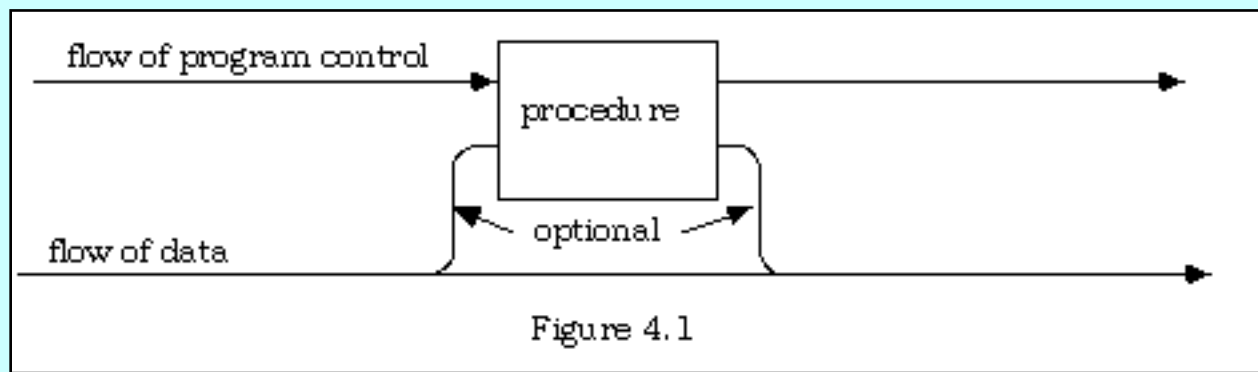
(iii) produce or alter data:

```
ReadCard (theNumber);
```

(iv) involve both (ii) and (iii):

```
ch := CAP (ch);  
INC (theNumber, theIncrement);
```

The possibilities are summarized in figure 4.1.



A procedure may best be regarded as code that is subordinate to a program in its operation--in the sense that the main program suspends operation in order to call the procedure to execute its code. When the latter is finished, the main program resumes control.

Besides the built-in procedures named by standard identifiers, (CAP, INC, etc.) and the ones available from various libraries, (WriteString, ReadCard, etc.) a program may define and use its own procedures.

Example: (first type; no data used or produced)

Write a program that can compute the voltage in a circuit given the voltage, current and resistance.

Discussion:

The relationship among these three is based on Ohm's law, which can be expressed as $E = IR$ where E is the voltage, I the current in Amps, and R the resistance in Ohms. The initial formulation of the solution is scarcely changed from programs in previous chapters, except that the routines that print out the program header have been removed to a procedure of the first type above--one that neither produces nor consumes data, but merely takes an action.

```
MODULE OhmsLaw1;
```

```
(* Written by R.J. Sutcliffe *)
(* to illustrate a simple procedure *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 02 25 *)
```

```
FROM STextIO IMPORT
  WriteString, WriteLn, ReadChar, SkipLine;
```

```
FROM SRealIO IMPORT
  ReadReal, WriteFixed;
```

```
( ***** )
```

```
PROCEDURE DisplayInfo;
```

```
BEGIN
```

```
    WriteString ("OhmsLaw1 was written by R.J. Sutcliffe");  
    WriteLn;  
    WriteString ("to illustrate a simple procedure");  
    WriteLn;  
    WriteLn;  
    WriteString ("This program computes the voltage in a circuit");  
    WriteLn;  
    WriteString (" given the current and resistance.");  
    WriteLn;
```

```
END DisplayInfo;
```

```
(*****)
```

```
VAR
```

```
    voltage, current, resistance : REAL;  
    answer : CHAR;
```

```
BEGIN
```

```
    DisplayInfo; (* invoke the above procedure *)  
    (* Gather the information from the user *)  
    WriteString ("What is the current in amperes? ==");  
    ReadReal (current);  
    SkipLine;  
    WriteLn;  
    WriteString ("What is the resistance in ohms? ==");  
    ReadReal (resistance);  
    SkipLine;  
    WriteLn;  
  
    (* Now, compute the voltage. *)  
    voltage := current * resistance;  
  
    WriteString ("A current of ");  
    WriteFixed (current, 2, 0);  
    WriteString (" amps running through a resistance of ");  
    WriteFixed (resistance, 2, 0);  
    WriteLn;  
    WriteString (" ohms produces a voltage of ");  
    WriteFixed (voltage, 2, 0);  
    WriteString (" volts. ");
```

```
WriteLn;  
WriteLn;  
  
WriteString ( "Press any key to continue.");  
ReadChar (answer);
```

END OhmsLaw1.

Here is a sample run:

OhmsLaw1 was written by R.J. Sutcliffe
to illustrate a simple procedure

This program computes the voltage in a circuit
given the current and resistance.

What is the current in amperes? ==> **14.2**

A current of 12.30 amps running through a resistance of 14.20
ohms produces a voltage of 174.66 volts.

Press any key to continue.

NOTES: 1. PROCEDURE is a reserved word. It marks the commencement of a subordinate section of code in much the same way as the word MODULE marks the start of the main program.

2. User defined procedures are named with the usual identifier rules. It is customary to begin a procedure name with an upper case letter, and to use for its name a verb that describes its action.

3. A procedure is defined by placing the heading and block of code in the declaration section of a program module, along with the variable declarations.

4. A procedure defined in a program is run (or *invoked*) by using its name, just as for the built-in ones.

5. Since it has a named "block" of code (like a Module), the procedure must also have that same name with its END.

Example: (second type; data used, but none produced.)

Write a module that converts percentages into letter grades and prints them out.

Discussion:

Most of the work in the program below has been pushed off into the procedure that accepts as data the average mark and selects and prints the letter grade. To shorten things down somewhat, this program writes no header at all.

MODULE LetterGrade;


```

(* Written by R.J. Sutcliffe *)
(* to illustrate a data consuming procedure *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 02 25 *)

FROM STextIO IMPORT
  WriteString, WriteLn, WriteChar, SkipLine;
FROM SRealIO IMPORT
  ReadReal, WriteFixed;
FROM SIOResult IMPORT
  ReadResult, ReadResults;

VAR
  theMark : REAL;
  ok : BOOLEAN;

(*****)
PROCEDURE PrintGrade (average : REAL);

VAR
  grade : CHAR;

BEGIN
  IF average < 40.0
    THEN
      grade := "F"
    ELSIF average < 50.0 THEN
      grade := "D"
    ELSIF average < 73.0 THEN
      grade := "C"
    ELSIF average < 86.0 THEN
      grade := "B"
    ELSE
      grade := "A"
    END;
  WriteString ("an average mark of ");
  WriteFixed (average, 2, 0);
  WriteString (" percent earns a letter grade of ");
  WriteChar (grade);
  WriteLn;
  WriteLn;

END PrintGrade;

```

```
( ***** )
```

```
BEGIN
  WriteString ("This program converts averages into grades. ");
  WriteLn;
  WriteString ("Enter an invalid real when done. ");
  WriteLn;

  REPEAT
    WriteString ("enter the average mark to convert here ==");
    ReadReal (theMark);
    ok := (ReadResult() = allRight);
    SkipLine;
    WriteLn;
    IF ok
      THEN
        PrintGrade (theMark);
      END;

  UNTIL NOT ok;

END LetterGrade.
```

Here is an output run:

```
This program converts averages into grades.
Enter an invalid real when done.
enter the average mark to convert here ==> 34.0
an average mark of 34.00 percent earns a letter grade of F

enter the average mark to convert here ==> 78.0
an average mark of 78.00 percent earns a letter grade of B

enter the average mark to convert here ==> aaa
```

NOTES: 1. In the example above, *average* is a temporary variable to which the value of *theMark* is assigned. This allows the value of *theMark* to be manipulated inside the procedure.

2. No variables declared inside a procedure (like *grade*) have any meaning outside it. If the procedure is re-entered later, its variables must be re-initialized if they are to have a specific value. This also means that variables in one procedure may have the same names as those in another one, or as in the main program. The compiler will not become confused by this, but the programmer may.

3. A consequence of this last point is that, when a reference to some identifier is made inside a procedure, a check is first made to see if that name is defined in the procedure. If so, that is the one used; if not, the enclosing procedure is checked next, and so on out to the main module. See a later section on *scope* for a

further discussion of this.

Example: (third type; data produced only.)

Write a procedure that can be used in various programs to obtain data from the keyboard with some error checking.

Discussion:

Procedures of this kind are often used to avoid writing out the same (or a very similar) statement sequence several times in a single program. Instead, the code is made *generic* and then is called upon from all the appropriate places in the program. Examples in this chapter will use variations on the following:

```
PROCEDURE GetNum (VAR theNum : REAL);

VAR
    ok : BOOLEAN;

BEGIN
    REPEAT
        WriteString ("Type the number here ==");
        ReadReal (theNum);
        ok := (ReadResult() = allRight); (* save result here *)
        SkipLine; (* because this will change ReadResult() *)
        IF NOT ok (* use saved value for testing *)
            THEN
                WriteLn;
                WriteString ("error in input number; try again.");
                WriteLn;
            END;
    UNTIL ok;
END GetNum;
```

When the main program invokes this procedure, it may print part of the prompt itself, then call on *GetNum* with a variable name of its own. Data assigned to a local variable whose name in the procedure heading is preceded by VAR will be passed back to the program variable used when the procedure is invoked. That is, if this procedure is invoked by the line

```
GetNum (sideLength);
```

then the program variable *sideLength* will be assigned whatever value is read by the procedure into the

variable it calls *theNum*.

Example: (fourth type; data used and produced.)

Write a procedure to round off real numbers to the nearest integer.

Discussion:

This problem is not a new one; the code to do this came up during the course of [section 3.12](#), program *DateCalc*, where it was observed that there is no such round off procedure built in to Modula-2 (though there is one in the library for integers.) Here is one way to write the code:

```
PROCEDURE Round (input : REAL; VAR output : CARDINAL);  
  
BEGIN  
    output := TRUNC (input + 0.5);  
END Round;
```

This procedure would be invoked in a program by writing

```
Round (numToRound, roundedNum);
```

where *numToRound* is a real variable, and *roundedNum* is a cardinal variable. Notice, however, that this procedure does not take into account the fact that the real input may be negative. One of the exercises asks the reader to make the necessary modifications.

This introduction to procedures is concluded by giving some additional formal definitions.

The list of variables in parentheses after the name of a procedure is called a parameter list.

An invocation of a procedure is the occurrence of its name somewhere in a program. The invocation instructs the processor to execute the code of the procedure.

4.2 Writing and Calling Procedures

A procedure is a miniature of a program in that it may have its own constants, variables and procedures. In fact, almost anything that can be part of a program module except imports can also be included in a procedure. Here is the general form:

```
PROCEDURE NameOfProcedure  ( <parameter list with types < local declarations  
including local procedures go here <parameter list
```

The full definition of the parameter list can be found in figure 4.5. Here are some partial declarations with only the procedure headings:

```
PROCEDURE Power (base : REAL; exponent : CARDINAL; VAR ans : REAL);  
PROCEDURE Max (num1, num2, num3 : INTEGER; VAR result : INTEGER);  
PROCEDURE WriteReal (realNum : REAL; fieldLength : CARDINAL);
```

with some corresponding calls in a program:

```
Power (theBase, theExp, theAns);  
Power (15.0, 6, theAns);  
Max (first, second, third, answer);  
Max (3, -2, 18, answer);  
WriteReal (2.75, 15);
```

Here is another complete, if somewhat short example. This procedure accepts for its parameters two numbers and calculates and prints the percentage that the first is of the second.

```
PROCEDURE PrintPercent (firstNumber, secondNumber: REAL);  
BEGIN  
  WriteReal (firstNumber, 10);  
  WriteString (" is ");  
  WriteReal (100.0 * firstNumber/secondNumber, 10);  
  WriteString (" percent of ");  
  WriteReal (secondNumber, 10);  
  WriteLn;  
END PrintPercent;
```

Two calls to this procedure are:

```
PrintPercent (mark, total);  
PrintPercent (15.0, 74.3);
```

As can be seen from these brief examples, when a procedure is declared, the names and the types of all the parameters employed by the procedure must be stated as part of that declaration. When the procedures are actually called, the parameters must be stated, but the types of those parameters are not given again. Here are two definitions:

The list of parameters provided in the declaration of a procedure is known as a formal parameter list. The list employed when the procedure is invoked is the actual parameter list.

WARNING: The actual parameters used when a procedure is called must match the formal parameters both in the order in which they are given and in their type.

For instance, given the declaration above, one could not use either of the following calls:

```
Power (15, 6, theAns); (* tries to pass cardinal to real *)
Power (15.0, 6.2, theAns); (* tries to pass real to cardinal *)
Power (15, theAns); (* missing a parameter *)
```

It is also worth pointing out that:

```
PROCEDURE Max (num1, num2, num3 : INTEGER; VAR result : INTEGER);
```

is equivalent to

```
PROCEDURE Max (num1: INTEGER; num2: INTEGER; num3 : INTEGER; VAR result : INTEGER);
```

Here is an example of a module to calculate and print the area and perimeter of either a square or a circle given the side length or radius as the case may be. It is formulated with separate procedures for the two calculations, and for obtaining the data.

```
MODULE Areas;
```

```
(* Written by R.J. Sutcliffe *)
(* to illustrate procedures *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 02 25 *)
```

```
FROM STextIO IMPORT
```

```
  WriteString, ReadChar, WriteLn, SkipLine;
```

```
FROM SRealIO IMPORT
```

```
  WriteReal, ReadReal;
```

```

FROM SIOResult IMPORT
    ReadResult, ReadResults;

VAR
    dimension, mArea, mPerim : REAL;
    which, ans : CHAR;
    again : BOOLEAN;

(*****)

PROCEDURE GetNum (VAR theNum : REAL);
VAR
    readOK : BOOLEAN;

BEGIN
    REPEAT
        WriteString ("Type the number here ==");
        ReadReal (theNum);
        readOK := (ReadResult() = allRight);
        IF NOT readOK
            THEN
                WriteLn;
                WriteString ("error in input number; try again.");
                WriteLn;
            END;
        SkipLine;
    UNTIL readOK;
END GetNum;

PROCEDURE CalcSquare (side : REAL; VAR area, perim: REAL);

BEGIN
    area := side * side;
    perim := 4.0 * side;
END CalcSquare;

PROCEDURE CalcCircle (radius : REAL; VAR area, perim: REAL);

CONST
    pi = 3.141592653;
    twopi = 2.0 * pi;

BEGIN
    area :=pi * radius * radius;
    perim := twopi * radius;
END CalcCircle;

(*****)

BEGIN    (* main program *)
    WriteString ("This program calculates areas and perimeters of");
    WriteLn;
    WriteString ("squares from a side length ");
    WriteString ("or circles from the radius.");
    REPEAT
        WriteLn;
        (* Present menu, give user the choice. *)
        WriteString ("Do you want to work with ");
        WriteString ("a circle or a square? ");

```

```

WriteLn;
WriteString ('Type a "C" or an "S" here ==="What is the dimension of the figure
?");
WriteLn;
GetNum (dimension);

IF which = 'C'
  THEN      (* Now go do one. "S" is the default *)
    CalcCircle (dimension, mArea, mPerim);
  ELSE
    CalcSquare (dimension, mArea, mPerim);
  END;      (* if *)

WriteString ("The area is ");
WriteReal (mArea, 0);
WriteString (" square units and ");
WriteLn;
WriteString ("the perimeter is ");
WriteReal (mPerim, 10);
WriteString (" units.");
WriteLn;
WriteLn;

WriteString ("Do another (Y/N) ==");
ReadChar (ans);
SkipLine;
again := (CAP (ans) = "Y");
UNTIL NOT again;

END Areas.

```

Here is a run from this module:

```

This program calculates areas and perimeters of
squares from a side length or circles from the radius.
Do you want to work with a circle or a square?
Type a "C" or an "S" here ==> zz
error in input number; try again.
Type the number here ==>y
Do you want to work with a circle or a square?
Type a "C" or an "S" here ==> 2.7
The area is  7.2900004 square units and
the perimeter is 10.80000000 units.

Do another (Y/N) =="C" or an "S" here ==> 5.0
The area is  7.8539818E+1 square units and
the perimeter is 31.41593000 units.

Do another (Y/N) ==>n

```

Note that the necessary imports *WriteReal*, *WriteLn*, *WriteString* are all done in the main program module. These entities, and any others defined in the main Module (but

not ones hidden away inside another procedure) are all visible to and available for the use of any procedure in the program. Likewise, anything defined inside a procedure (including its own parameters) is visible and usable within the procedure itself, and within any procedures defined inside it.

A program module and a procedure both define a scope of visibility for the entities they define. Such entities are usable inside all other procedures defined within that scope, but not outside it.

That is, procedures exist in a kind of hierarchy of scopes. This text will not have much occasion to make use of this fact until later, and a detailed examination of scope, visibility, and their consequences will be undertaken in chapter ten.

Also note that, while in theory a procedure may have any number of parameters; in practice, it is wise to write procedures with, say, no more than four--or to use more procedures. Otherwise code becomes cumbersome, confusing and hard to maintain.

[Contents](#)

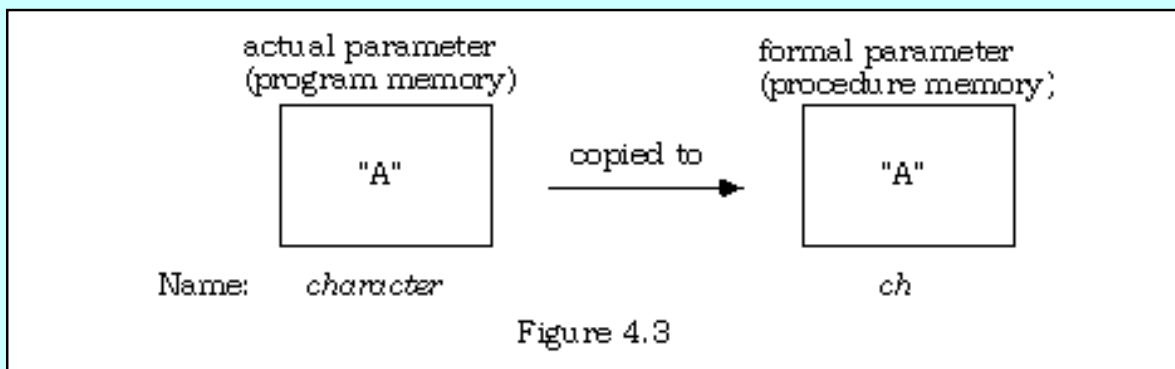
4.3 Value and Variable Parameters (Introduction)

There have been two kinds of parameters used thus far in the examples of procedures. One kind is used to input data into the procedure only. When the procedure is called, the value represented by the actual parameter is copied into the memory location set aside for the formal parameter. At this point, there are two copies of the information in memory. If any changes are made to the local copy owned by the procedure (with the formal parameter name), the actual object copied by the calling routine is unaffected. Figure 4.3 illustrates.

A value parameter is a formal parameter naming a memory location that is set aside while the procedure is active, and into which the contents of an actual parameter will be copied at the time the procedure is called. The two copies of the information are decoupled (independent of one another).

The Action of a Call Using a Value Parameter:

```
PROCEDURE Write (ch: CHAR); (* actual declaration *)  
.  
.  
.  
character := "A";  
Write (character); (*call to this in a program *)
```

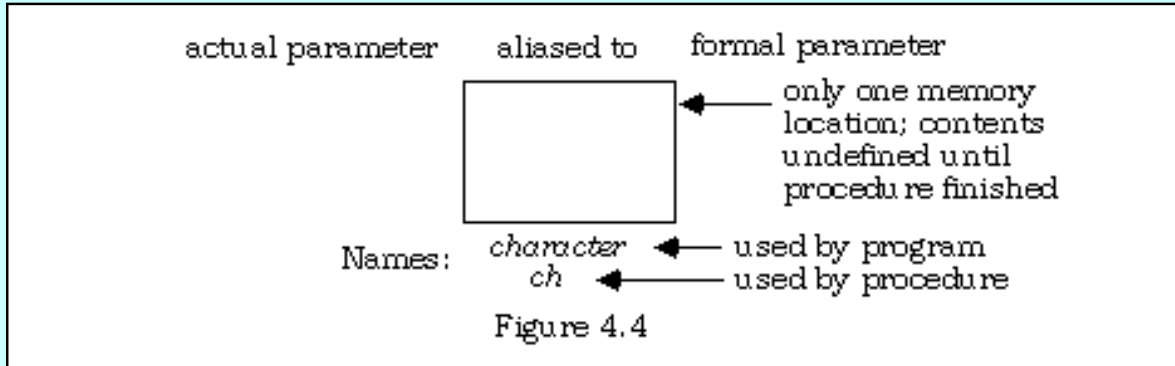


In some situations, one prefers to get things (often numbers) back from procedures, rather than have the procedure just take an action. To do this, one employs the second kind, a formal parameter whose name is simply attached to the same memory location that is already being used by the actual parameter. Any changes made to the formal parameter within the procedure body are automatically reflected in the actual parameter, because the two are simply different names for the same memory location. They are "tightly coupled". This action is illustrated in figure 4.4.

A variable parameter is a formal parameter whose name is attached to the same memory location as that named by the actual parameter at the time the call is made. The two names are coupled. These are distinguished from value parameters in the formal parameter list of the procedure heading by preceding them with the reserved word VAR.

The Action of a Call Using a Variable Parameter:

```
PROCEDURE Read (VAR ch: CHAR);           (* declaration *)
. . .
WriteString ("Do you want to do this again?");
Read (answer);           (* call to Read in a program *)
```



Here are some more examples:

```
PROCEDURE GetNextDay (VAR day : CARDINAL);
PROCEDURE Discrim (a, b, c : REAL;
                  VAR d : REAL; VAR ok : BOOLEAN);
```

The procedure *GetNextDay* can (and presumably does) directly modify whatever variable is aliased to *day* by assigning a new value to *day*. Since both the actual parameter and the formal parameter name the same memory location, both names get new values at once.

The procedure *Discrim* is capable of directly modifying whatever is assigned to *d* and *ok*, but it can only use whatever data is copied to *a*, *b*, and *c* when it is invoked. Any changes it might make to these inside the procedure are not reflected in changes to the original entities owned by the calling routine that were copied to them.

Assuming that *numberOfDay* is a *CARDINAL*, *w*, *x*, *y*, and *z* are *REAL* and *done* is of type *BOOLEAN*, here are some legitimate calls to these two procedures:

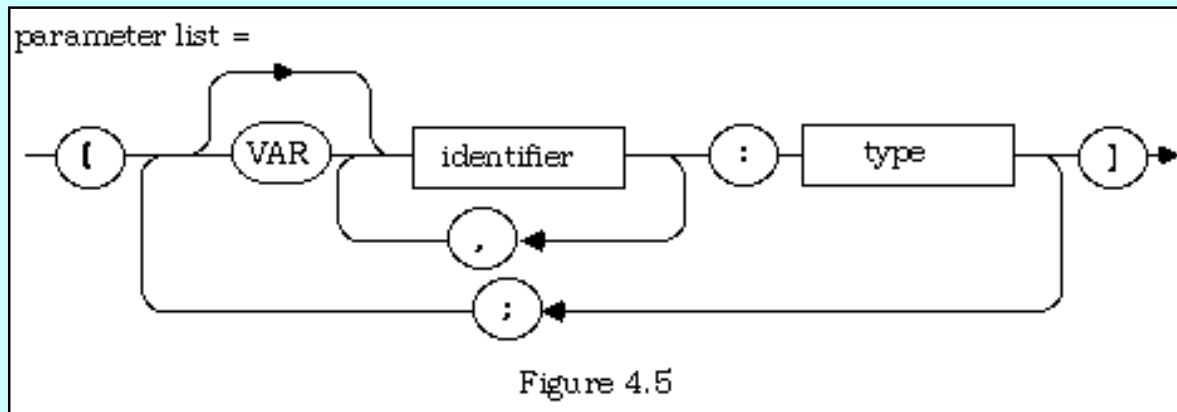
```
GetNextDay (numberOfDay);
Discrim (w, x, y, z, done);
Discrim (3.0, x, 2.5, z, done);
```

Here are some that are not correct, with the reasons:

```
GetNextDay (3)      (* 3 is not a variable *)
GetNextDay (x)      (* x is not a cardinal *)
Discrim (5, x, y, z, done);  (* 5 is not real *)
Discrim (w, x, y, 3.9, done); (* 3.9 is not a variable *)
Discrim (w, x, y, done, z);  (* wrong order of parameters *)
```

As the first and fourth of these incorrect examples illustrate, one must assign a variable to a variable parameter.

After all, one cannot expect that a literal value will be somehow changed. (Can a 3 become not a 3?) It is now possible (figure 4.5) to diagram the parameter list completely.



Here is something else that might seem to be permitted, but that is not. Suppose *cardNum* is a **CARDINAL**, and in fact *cardNum* happens to equal 5, and that one has:

```
PROCEDURE Fetch (VAR intNum : INTEGER)
```

Then a call `Fetch (cardNum)` produces a compiler error, because the types of an actual and formal **VAR** parameter must be *identical*, not merely assignment compatible, as are **CARDINAL** and **INTEGER**. Value parameters on the other hand, need only be assignment compatible.

To further illustrate the use of variable parameters, here is a procedure that swaps the values of two variables that are passed to it.

```
PROCEDURE Swap (VAR firstNum, secondNum : REAL);
```

```
VAR
    temp : REAL;
```

```
BEGIN
    temp := firstNum;
    firstNum := secondNum;
    secondNum := temp;
END Swap;
```

Notice the use of the local variable *temp* to hold the value of *firstNum* while the value of *secondNum* is assigned the name *firstNum*. The procedure *Swap* would be called by naming, as parameters, the two real variables whose values one wanted interchanged. The following sequence in the main program:

```
rVar1 := 2.5;
rVar2 := 7.8;
Swap (rVar1, rVar2);
WriteReal (rVar1, 0);
```

would output 7.8 as the value of *rVar1*.

Example:

Write a program module that will sort and print in ascending order three reals input from the keyboard.

Restatement:

- Ask for and accept three numbers from keyboard, reading them into REAL variables.
- Sort them into ascending order.
- Print them out in the order in which they now are.

Refinement:

```
Ask for a number
Read it and assign to first variable
Ask for a number
Read it and assign to second variable
Ask for a number
Read it and assign to third variable

Sort the numbers
  if first is greater than second then
    swap the two
  if second is greater than third then
    swap the two
  if first is now greater than second then
    swap the two

Print the numbers out in order
```

As one looks this refinement over, it is apparent that reading and assigning are done three times with three different numbers and that there are potentially three swaps. Both subtasks (reading/assigning, and swapping) should therefore be procedures, so one makes this:

Further Refinement:

```
procedure 1
  Read a number
  Assign it to a variable parameter
procedure 2
  Swap two variable parameters
Main Program
  Call procedure 1 for the three numbers in turn.
  continue as outlined in the first refinement, but the swap is a procedure
  invocation.
```

Data Table:

Variables Required:

procedure 1: only the single VAR parameter, and a boolean for recycling.

procedure 2: two VAR parameters, and a temporary real.

Main Program: three reals, and a character variable for carriage returns.

Imports Needed:

ReadReal, WriteFixed, Done

WriteString, WriteLn, ReadChar, SkipLine

ReadResult, ReadResults

Here is the code:

```
MODULE SortThree;
```

```
(* Written by R.J. Sutcliffe *)
```

```
(* using P1 MPW Modula-2 for the Macintosh computer *)
```

```
(* last revision 1993 02 26 *)
```

```
FROM STextIO IMPORT
```

```
  WriteString, WriteLn, ReadChar, SkipLine;
```

```
FROM SRealIO IMPORT
```

```
  ReadReal, WriteFixed;
```

```
FROM SIOResult IMPORT
```

```
  ReadResult, ReadResults;
```

```
VAR
```

```
  num1, num2, num3 : REAL;
```

```
  key : CHAR;
```

```
PROCEDURE GetNum (VAR theNum : REAL);
```

```
VAR
```

```
  ok : BOOLEAN;
```

```
BEGIN
```

```
  REPEAT
```

```
    WriteString ("Type the number here ==");
```

```
    ReadReal (theNum);
```

```
    ok := (ReadResult() = allRight);
```

```
    SkipLine;
```

```
    IF NOT ok (* use saved value for testing *)
```

```
      THEN
```

```
        WriteLn;
```

```
        WriteString ("error in input number; try again.");
```

```

        WriteLn;
    END;
UNTIL ok;
END GetNum;

PROCEDURE Swap (VAR firstNum, secondNum : REAL);

VAR
    temp : REAL;

BEGIN
    temp := firstNum;
    firstNum := secondNum;
    secondNum := temp;
END Swap;

BEGIN    (* main program *)
    WriteString ("This program sorts three numbers ");
    WriteString ("into ascending order.");
    WriteLn;
    WriteLn;
    (* obtain the three numbers *)
    WriteString ("First number: ");
    GetNum (num1);
    WriteString ("Second number: ");
    GetNum (num2);
    WriteString ("Third number: ");
    GetNum (num3);

    (* now sort the numbers. *)
    IF num1 > num3    (* if second, third out of order *)
    THEN
        Swap (num2, num3);    (* trade them *)

        IF num1 & 2 *)
        THEN
            Swap (num1, num2);    (* trading if necessary *)
        END;    (* inner if *)

    END;    (* outer if *)

    (* numbers are sorted, so do print out *)

    WriteString ("In ascending order, the numbers are: ");
    WriteLn;
    WriteFixed (num1, 5,0);
    WriteLn;
    WriteFixed (num2, 5,0);

```

```
WriteLn;  
WriteFixed (num3, 5,0);  
WriteLn;  
WriteString ("Press a key to continue ==");  
ReadChar (key);
```

END SortThree.

A run follows:

This program sorts three numbers into ascending order.

First number: Type the number here ==> **11.98**

Third number: Type the number here ==> q

[Contents](#)

4.4 Predicates

To this point, little regard has been paid to the documentation of procedures. Comments have been included where appropriate, but nothing has been said about any special need to document the role and function of procedures. This needs attention, for procedures are not just portions of a program, or miniature programs on their own, but are pieces of re-usable code that may be called upon under a variety of circumstances. It is important therefore, that each procedure be clearly documented as to:

- its purpose
- any assumptions it makes about the data passed to it (*preconditions*)
- the nature of any data produced by it (*postconditions*)

Such documentation will allow the programmer to avoid errors caused by using a poorly remembered piece of code for some purpose for which it was never intended; the proper use will be outlined in comments that remain with the code. Consider, for instance, a procedure designed to compute what percentage one number is of another:

```
PROCEDURE Percent (num, denom : REAL; VAR result : REAL);  
  
BEGIN  
    result := 100.0 * num / denom;  
END Percent;
```

What will happen if this code is called with *denom* equalling zero? There are two ways to avoid the *divide-by-zero* run-time error lurking in indiscriminate use of this procedure. The first is to be up-front about the limitations of the procedure by documenting them:

```
PROCEDURE Percent (num, denom : REAL; VAR result : REAL);  
(* calculates what percentage num is of denom *)  
(* pre : denom # 0.0  
    post: result = 100.0 * num / denom *)  
  
BEGIN  
    result := 100.0 * num / denom;  
END Percent;
```

If the procedure is written this way, it becomes the responsibility of the portion of code invoking it to check that the actual parameter passed to *Percent* is nonzero before going ahead with the procedure. On the other hand, the responsibility for avoiding a run time error could be given to the procedure itself.

If this is done, the main program must be informed when the result is not valid. The procedure could be written:

```
PROCEDURE Percent (num, denom : REAL;  
                    VAR result : REAL; VAR ok : BOOLEAN);  
(* calculates what percentage num is of denom *)  
(* pre : none  
   post: if denom # 0.0  
         then result = 100.0 * num / denom and ok = true  
         else result is undefined and ok is false *)  
  
BEGIN  
  IF denom # 0.0  
    THEN  
      result := 100.0 * num / denom;  
      ok := TRUE  
    ELSE  
      ok := FALSE  
    END  
END Percent;
```

Preconditions and postconditions specified in the documentation of a procedure are called predicates.

Example:

Write a program module that can examine two lines, each determined from the coordinates of two points entered in from the keyboard, and determine whether the lines are parallel, perpendicular, or neither.

Discussion:

When two points are available in coordinate form P1(x1, y1) and P2(x2, y2), the slope of the line through the two points is given by

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Two lines are parallel if they have the same slope, and they are perpendicular if their slopes are negative reciprocals. A horizontal line has zero slope, and the slope of a vertical line is undefined (it has no slope.) Thus, the failure of a procedure *Slope* to compute a result is not a bad thing; it just indicates that the line in question is vertical.

```
MODULE Slopes;
```

```
(* Written by R.J. Sutcliffe *)
(* using P1 MPW Modula-2 for the Macintosh computer *)
(* last revision 1991 02 26 *)
```

```
FROM STextIO IMPORT
```

```
  WriteString, WriteLn, ReadChar, SkipLine;
```

```
FROM SRealIO IMPORT
```

```
  ReadReal, WriteFixed;
```

```
FROM SIOResult IMPORT
```

```
  ReadResult, ReadResults;
```

```
VAR
```

```
  line1X1, line1Y1, line1X2, line1Y2,
```

```
  line2X1, line2Y1, line2X2, line2Y2,
```

```
  line1Slope, line2Slope: REAL;
```

```
  line1Ok, line2Ok : BOOLEAN;
```

```
  key : CHAR;
```

```
PROCEDURE GetNum (VAR theNum : REAL);
```

```
(* gets a number input from the keyboard
```

```
Pre: none
```

```
Post: if ok is true then return
```

```
      else ask for input to be re-entered *)
```

```
VAR
```

```
  ok : BOOLEAN;
```

```
BEGIN
```

```
  REPEAT
```

```
    WriteString ("Type the number here ==");
```

```
    ReadReal (theNum);
```

```
    ok := (ReadResult() = allRight);
```

```
    SkipLine;
```

```
    IF NOT ok (* use saved value for testing *)
```

```
      THEN
```

```
        WriteLn;
```

```
        WriteString ("error in input number; try again.");
```

```
        WriteLn;
```

```
      END;
```

```
  UNTIL ok;
```

```
END GetNum;
```

```
PROCEDURE Slope (x1, y1, x2, y2: REAL;
```

```

                VAR m: REAL; VAR ok: BOOLEAN);
(* computes the slope of a line joining (x1, y1) and (x2, y2)
Pre: none
Post: if x1#x2 then m is the slope and ok is true
      else m is set to the maximum possible real; ok = false *)

```

```

VAR
    deltaX, deltaY : REAL;

```

```

BEGIN
    deltaX := x2 - x1;
    deltaY := y2 - y1;
    IF deltaX # 0.0
        THEN
            m := deltaY/deltaX;
            ok := TRUE;
        ELSE
            m := MAX (REAL);
            ok := FALSE;
        END;

```

```

END Slope;

```

```

BEGIN      (* main program *)
    WriteString ("This program computes the slopes of 2 lines");
    WriteString (" from their coordinates.");
    WriteLn;
    WriteLn;
    (* obtain the coordinates *)
    WriteString ("First line:");
    WriteLn;
    WriteString ("First point:");
    WriteString ("x = ");
    GetNum (line1X1);
    WriteString ("y = ");
    GetNum (line1Y1);
    WriteString ("Second point:");
    WriteString ("x = ");
    GetNum (line1X2);
    WriteString ("y = ");
    GetNum (line1Y2);
    WriteString ("Second line:");
    WriteLn;
    WriteString ("First point:");

```

```

WriteString ("x = ");
GetNum (line2X1);
WriteString ("y = ");
GetNum (line2Y1);
WriteString ("Second point:");
WriteString ("x = ");
GetNum (line2X2);
WriteString ("y = ");
GetNum (line2Y2);

(* compute the slopes *)
Slope (line1X1, line1Y1, line1X2, line1Y2, line1Slope, line1Ok);
Slope (line2X1, line2Y1, line2X2, line2Y2, line2Slope, line2Ok);

(* now output the data. *)
WriteString ("The slope of the first line is ");
IF line1Ok
    THEN
        WriteFixed (line1Slope, 6, 0)
    ELSE
        WriteString ("undefined.")
    END;
WriteLn;
WriteString ("The slope of the second line is ");
IF line2Ok
    THEN
        WriteFixed (line2Slope, 6, 0)
    ELSE
        WriteString ("undefined.")
    END;
WriteLn;
WriteString ("The 2 lines are ");
IF NOT (line1Ok OR line2Ok) OR (line1Slope = line2Slope)
    THEN
        WriteString ("parallel.");
    ELSIF((line1Slope = 0.0) AND NOT line2Ok) OR
        ((line2Slope = 0.0) AND NOT line1Ok) OR
        (ABS (line1Slope + (1.0 / line2Slope)) < 0.000001) THEN
        WriteString ("perpendicular.");
    ELSE
        WriteString ("neither parallel nor perpendicular.")
    END;

WriteLn;

```

```
WriteString ("Press a key to continue ==");  
ReadChar (key);
```

END Slopes.

NOTES: 1. MAX is a new standard identifier. It returns the maximum value of a type. This has not been done in this case to simulate a value of "infinity," but so that a line with no slope has some value attached to it for purposes of comparison.

2. Observe the comparison ($\text{ABS}(\text{line1Slope} + (1.0 / \text{line2Slope})) < 0.000001$). It would be easier to write ($\text{line1Slope} = - (1.0 / \text{line2Slope})$). However, this is unlikely to work in practice because of rounding off effects, so one must instead rely on the two values being close.

3. Observe also the order of checking for perpendicularity. If either of the first two conditions is true, the third will not be checked--a good thing if it could produce a divide-by-zero error.

Here is a single run from this module:

This program computes the slopes of 2 lines from their coordinates.

First line:

First point:x = please type here ==> 3.0

Second point:x = please type here ==> 8.0

Second line:

First point:x = please type here ==> 4.0

Second point:x = please type here ==> 2.0

The slope of the first line is 2.500000

The slope of the second line is -0.4000000

The 2 lines are perpendicular.

Press a key to continue ==>n

[Contents](#)

4.5 Function Procedures

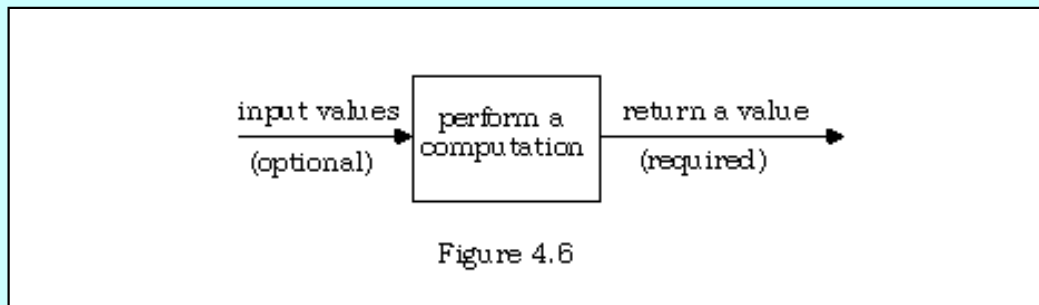
Some of the built-in procedures that have been used thus far can be employed in such a way that they return a value directly to the part of the expression that references them. For instance, programs in this text have contained things like:

```
ck := CAP (ch);  
num1 := ABS (num2);  
firstReal := FLOAT (someInteger) + secondReal;  
m := MAX (REAL);
```

and so on. (More of these will be covered in later chapters.) These have been referred to these all along as *functions* or, more correctly, as *function procedures*, but it should be clear that they work somewhat differently than do the regular procedures considered so far in this chapter, because they are oriented toward the return of a single result rather than to the taking of action. Moreover, that result is returned (in effect) in the procedure's own name, rather than in a variable parameter, as any result sent back by a regular procedure must be. The identifier of the procedure is itself being used in the expression as though it were a variable. Note the following definition:

A function procedure is a program unit that optionally accepts input data in its parameter list, but that always assigns an output value to the function name according to a fixed rule or formula determined by the statements in the procedure.

The usual mathematical notation for a function is $y = f(x)$, where the f is the name of the function. In Modula-2 one would write $y := f(x)$ to assign to y the results of calling the function procedure called f with the data x . This time the black box looks a little different, as shown in figure 4.6:



As has already been suggested, Modula-2 employs a variant of the PROCEDURE statement to implement functions. Here is the general form for a function procedure:

```
PROCEDURE name ( < formal parameter list <result type<local declarations<statement  
sequence
```

There are four differences between function procedures and regular procedures:

First: the type of result to be returned must be specified by placing it after a colon following the closing parenthesis of the formal parameter list.

Second: a specific RETURN with the result (of the correct type) that is to be sent back must actually be executed somewhere in the body of the function procedure.

Third: the call of a function procedure must always include a parameter list, even

if the latter is empty.

Fourth: the call of a function procedure appears in an expression, not as a separate command on a line by itself as does the call of a regular procedure.

These details are illustrated with some examples. First, here are some valid declarations:

```
PROCEDURE Largest (firstReal, secondReal : REAL) : REAL;
PROCEDURE Strange (VAR firstReal : REAL,
                  secondReal : REAL) : CARDINAL; (* poor code *)
PROCEDURE Round (realToRound : REAL) : CARDINAL;
PROCEDURE GetResult ( ) : CARDINAL;
PROCEDURE CAP (ch : CHAR) : CHAR;
```

The last of these is just the (implied) heading of one of the built-in pervasive function procedures. The second one does seem a little strange, because it returns both a variable parameter and a result type. In actual practice, it is not likely that anyone will do this very often--indeed, this practice may be legal, but it is very poor code. However, it does illustrate that function procedures are a specialization of regular procedures and share all the capabilities of the latter. To illustrate both this and the third one, here are two ways to write the code for a procedure that rounds off a REAL to the nearest whole number.

```
PROCEDURE Round (num : REAL; VAR numOk : BOOLEAN) : CARDINAL;
BEGIN (*poor version *)
  IF (num <= FLOAT (MAX (CARDINAL))) AND
      (num <take evasive action<= FLOAT (MAX (CARDINAL))) AND
      (num <= FLOAT (MAX (CARDINAL)))
  AND (numToRound <take evasive action<value> x
  THEN
    Swap (x, y)
  END;

  (* compute the GCD *)
  REPEAT
    result := y; (* let the gcd be the smaller of the two *)
    y := x MOD y; (* replace the smallest with new remainder *)
    x := result; (* & larger with smaller one from last step *)
  UNTIL y = 0;

  RETURN result; (* and send it back *)
END Gcd;

VAR (* main program variables *)
  num, denom, gcd : CARDINAL;
  goodData, again : BOOLEAN;
  answer : CHAR;
```



```

BEGIN
  WriteString ("This program reduces fractions to lowest terms.");
  WriteLn;

  REPEAT (* main repeat loop *)
    (* get the numbers *)
    WriteString ("Enter the numerator ==");
    ReadCard (num);
    SkipLine;
    WriteLn;
    WriteString ("And now, the denominator ==");
    ReadCard (denom);
    SkipLine;
    WriteLn;

    (* set a flag if the numbers weren't zero *)
    goodData := (num # 0) AND (denom # 0);

    (* write the result *)
    WriteString ("The lowest terms fraction is");
    WriteLn;
    IF goodData
      THEN
        gcd := Gcd (num, denom);
        (* call the procedure to do the calculation *)
        num := num DIV gcd; (* compute new fraction *)
        denom := denom DIV gcd;
        WriteCard (num, 0);
        WriteLn;
        IF denom # 1 (* denominators of one are not printed *)
          THEN
            WriteString ("-----");
            WriteLn;
            WriteCard (denom, 0);
            WriteLn;
          END;
        ELSIF num = 0 THEN
          IF denom = 0
            THEN
              WriteString ("indeterminate; both parts are zero");
            ELSE (* only numerator is zero, so ok *)
              WriteCard (num, 0);
            END;
          ELSE (* only denominator was zero *)
            WriteString ("undefined because the denominator is zero");
          END;
        WriteLn;
        WriteString ("Do another? Y/N ");
        ReadChar (answer);
        SkipLine;
        again := (CAP (answer) = "Y");
      UNTIL NOT again;

```

END ReduceFraction.

Here is a run from this program module:

This program reduces fractions to lowest terms.

Enter the numerator ==> **17**

The lowest terms fraction is

3

Do another? Y/N **Y**

Enter the numerator ==> **1001**

The lowest terms fraction is

1

11

Do another? Y/N **Y**

Enter the numerator ==> **0**

The lowest terms fraction is

undefined because the denominator is zero

Do another? Y/N **Y**

Enter the numerator ==> **0**

The lowest terms fraction is

indeterminate; numerator and denominator both zero

Do another? Y/N **N**

A wide variety of simple functions can be implemented in typical Modula-2 programs.

Here is one that squares the supplied real parameter.

```
PROCEDURE Sqr (numToSquare : REAL) : REAL;  
  (* Pre: numToSquare <= sqrt (MAX (REAL)))  
  Post: the square of numToSquare is returned *)
```

```
BEGIN
```

```
  RETURN numToSquare * numToSquare;
```

```
END Sqr;
```

Here is one based on a built-in procedure (ODD) that has not previously been mentioned:

```
PROCEDURE Even (num : CARDINAL) : BOOLEAN;  
  (* Pre : none  
  Post : returns true if num is even and false if num is odd *)
```

```
BEGIN
```

```
    RETURN NOT ODD (num);  
END;
```

A little more work is required for some others:

Problem:

Write a procedure to raise a real number to a real exponent.

Discussion:

What is desired here is the solutions to equations such as $x = a^b$. This is done as follows:

Take natural logarithms on both sides, obtaining

$$\ln (x) = b \times \ln (a)$$

Now, raise both sides to the power e (the base of the natural logarithms)

$$x = e^{b \ln(a)}$$

This is expressed in Modula-2 by the assignment

```
x := exp (b * ln (a) )
```

where both *exp* and *ln* must be imported from the library module *RealMath*. This produces the following code:

```
PROCEDURE Power (base, exponent : REAL) : REAL;  
  (* pre: base raised to the exponent is not greater than MAX (REAL) and base > 0  
    post: base raised to the exponent is returned *)  
  
BEGIN  
  RETURN exp (exponent * ln (base))  
END Power;
```

This function is actually provided as part of *RealMath*, but in a similar manner, it might be necessary to compute logarithms with some other base than e. (Non ISO versions may provide base ten, or common logarithms in a library). If one starts with

$$x = \log_b(a)$$

and wishes to find x; one can rewrite changing to exponent notation

$$b^x = a$$

and taking natural logarithms on both sides, yields

$$x * \ln(b) = \ln(a) \quad \text{or,} \quad x = \ln(a) / \ln(b)$$

It is now fairly easy to write:

```
PROCEDURE LogBaseB (base, number : REAL) : REAL;  
(* Pre : both base and number are positive reals  
Post : the log of number with the base base is returned. *)  
  
BEGIN  
    RETURN (ln (number)) / (ln (base))  
END LogBaseB;
```

The mathematics library mentioned above also contains a procedure for computing square roots, but this computation is not difficult, and is demonstrated below:

Problem:

To write a function to compute square roots of real numbers.

Solution:

1. Check to see if the number provided is negative.
2. If it is, return without going further.
3. Otherwise, apply a square root algorithm.
4. Return the answer.

Refinement of 3: (The divide and average method)

Basis: start with a guess at the square root; call it g . Now if $g^2 = x$ exactly, then $g = x/g$ and the calculation is finished. If $g^2 < x$ so take the average of x/g and g as the next guess and try the test again. When the result is close enough, return it.

- 3.1 Decide how close the result must be, say $\text{epsilon} < .00001$.
- 3.2 Guess an initial value. (say, the first guess is set to half the number.)
- 3.3 While guess and $\text{number}/\text{guess}$ are not within epsilon of each other
Take the average of guess and $\text{number}/\text{guess}$ for a new guess.

The code is formulated in such a way that if the precondition that requires a non-negative parameter is ignored by the client code invoking this routine, it will still be safe. The value returned will be invalid, however.

```
PROCEDURE Sqrt (num : REAL) : REAL;
```

```
(* precondition: num < 0 the answer is invalid,  
    otherwise the square root is returned.  
    method: divide and average *)
```

```
CONST
```

```
    epsilon = 0.00001; (* error limit *)
```

```
VAR
```

```
    guess : REAL;
```

```
BEGIN
```

```
    IF num <= 0.0
```

```
    THEN
```

```
        RETURN num;
```

```
    ELSE
```

```
        guess := num/2.0;
```

```
    END;
```

```
    WHILE ABS (num/guess - guess ) > epsilon
```

```
    DO
```

```
        guess := 0.5 * (num/guess + guess )
```

```
    END;
```

```
    RETURN guess;
```

END Sqrt;

NOTE: The precision of the result could be improved by making the type of the variable LONGREAL and then making epsilon much smaller. Such decisions depend on the range of the two real types in the implementation being used.

[Contents](#)

4.6 Some Standard Procedures

Most of the standard procedures that are built-in to Modula-2 have already arisen in the course of various examples thus far in the text. The purpose of this section is to provide a summary in one place of all the standard procedures, together with a few examples of code utilizing them.

(i) **ABS** (number) returns the "absolute value" of "number".

The absolute value of a number is its distance from zero, without regard to direction. More technically, it is the square root of the square of a number.

Either way one looks at it:

```
ABS (5) = 5
ABS (-10) = 10
ABS (2.7) = 2.7
ABS (-3.51) = 3.51
```

The actual parameter type and the result type must either be both **INTEGER** or both **REAL**. There is a little magic in this, in that no user-defined procedure can take or return more than one type, even through the built-in function procedure may do so. If one had *firstInt* and *secondInt* of type **INTEGER** and *firstReal* and *secondReal* of type **REAL**, the following would be valid:

```
firstInt := ABS (secondInt);
firstReal := ABS (secondReal);
firstInt := TRUNC (ABS (secondReal));
firstReal := FLOAT (ABS (secondInt));
```

Recall that there is a potential problem with loops that fail to terminate or comparisons that fail to yield equality because a **REAL** might never be identically equal to zero. **ABS** allows this to be avoided by writing, say:

```
REPEAT
    statement sequence;
UNTIL ABS (realVar) < epsilon; (* set to a small number, say .0000001 *)
```

The loop terminates when the *realVar* becomes sufficiently close to zero.

(ii) **CHR** (number) returns the character whose associated numerical value is the **CARDINAL** *number*. This number is sequential number of the character, in the underlying national character set (often ISO/ASCII.)

Example:

If one wrote the assignment `ch := CHR (65)` then *ch* would be equal to A

(iii) **DEC** (*s*) and **INC** (*s*)--subtract or add one (respectively) to the scalar *s*.

(iv) **DEC** (*s*, *number*), **INC** (*s*, *number*)--subtract *number* or add *number* (respectively) to the scalar *s*.

Examples:

```
VAR
    num : CARDINAL;
    ch  : CHAR;
BEGIN
    num := 1;
    DEC (num);      (* it's now zero *)
    INC (num, 8);    (* now it's eight *)
    ch := "Z";
    INC (ch);       (* holds next char after "Z" *)
    INC (num, num-1) (* now it's 15 *)
    (* INC allows expressions for the increment *)
```

Care must be taken not to **INC** or **DEC** a variable beyond its defined limits. For instance, if *num* were zero, then **DEC** (*num*) would leave *num* undefined. As noted, the increment or decrement need not be a literal; it can be any expression that evaluates to a number compatible with the type of variable being used. *INC* (*ch*, *b-a*) is allowed, provided *b-a* is a suitable **INTEGER**. This in turn means that it is correct to **INC** or **DEC** by a negative number.

NOTES: 1. When there is a choice between `x := x + 1` and **INC** (*x*), always use the latter, as in most systems the code generated executes more quickly. The same applies to **DEC**.

2. The two versions each of **INC** and **DEC** are the only regular procedures described in this section. The others are all function procedures.

(v) **CAP** (*ch*) converts the contents of the **CHAR** variable to uppercase.

If *ch*, *ck* are both of type **CHAR** then:

```
ch := 'a'    (* or, ch := 'A' *)
ck := CAP (ch) would leave ck holding the value 'A'.
```

This was used to avoid writing:

```
IF (ch = 'Y') OR (ch = 'y').
```

Example:

```
WriteString ("Are you sure you want to do this? ");
WriteLn;
WriteString ("Answer Y or N ===");
Read (ans);
```



```

IF CAP (ans) = 'Y'
  THEN
    Execute (Program)
  ELSE
    Execute (User)
  END;

```

(vi) **FLOAT** (n) and **LFLOAT** (n) convert any real, integer, or cardinal into the type REAL or LONGREAL, respectively. Thus:

```

realVar := FLOAT (5); (* produces 5.0 *)
lrealVar := LFLOAT (realVar); (* produces longreal 5.0 *)
lrealVar := LFLOAT (-10); (* produces -10.0 in longreal format *)

```

(vii) **ODD** (n) is a BOOLEAN function that returns TRUE if the INTEGER or CARDINAL expression *n* is odd and FALSE if it is even.

Example:

```

VAR
  flag : BOOLEAN
  number : CARDINAL
BEGIN
  number := 5;
  flag := ODD (number);    (* flag is now true *)
  INC (number);
  (* flag is still true; it has not been re-evaluated *)
  flag := ODD (number);    (* flag is now false *)

```

Example:

Here is a section of code that raises the quantity (-1) to the power *exponent* and places the result in the variable *answer*.

```

PROCEDURE PowerOfOne (theExponent : INTEGER) : INTEGER;
BEGIN
  IF ODD (exponent)
    THEN
      RETURN -1
    ELSE
      RETURN 1
    END;
END PowerOfOne;

```

(viii) **MAX** (type) and **MIN** (type) take as a parameter the name of any scalar type (including **REAL**) and return the maximum and minimum values possible for items of that type (respectively). **MAX** and **MIN** are most useful, for example, when applied just before attempting to **TRUNC** a **REAL**. Assume that *integerNum* is **INTEGER** and *realNum* is **REAL**.

```
IF (realNum < FLOAT (MAX (INTEGER)) )  
  THEN  
    integerNum := TRUNC (realNum)  
  ELSE  
    ErrorRoutine  
  END
```

(ix) **TRUNC**(num) and **INT**(num) will convert the supplied numeric parameter to a cardinal or integer value, respectively, with truncation as needed. Thus:

```
TRUNC (ABS(-7.245));    (* produces 7 *)  
TRUNC (8.1);            (* produces 8 *)  
INT (2.3);              (* produces 2 *)  
INT (-4.78);            (* produces -4 *)
```

[Contents](#)

4.7 Some Stylistic Considerations

In giving names to procedures, the following suggestions provide recommended rules-of-thumb.

1. Use a noun to identify a function procedure returning a value other than Boolean--it names the result of the function. (e.g. CAP, ABS, Gcd)
2. Use an adjective to identify the result of a function procedure that returns a BOOLEAN. (e.g. ODD, Even) It tells if a characteristic of the noun being acted upon is present (is true).
3. Use a verb to identify a regular procedure--it stands for the action being taken. (e.g. WriteChar, WriteString, ReadChar)
4. Start the names of procedures with an upper case letter. This will distinguish them from the names of variable, which should commence with a lower case letter.

Following these suggestions makes programs more understandable--especially when making modifications at a later time.

[Contents](#)

4.8 Recursion

[Section 3.8](#) introduced the idea of a sequence of numbers defined by a formula that could be applied to the numbers 1, 2, 3, 4, 5, ... in succession to produce the terms of the sequence. Not all sequences are defined in this manner; some are given in terms of a rule that works from previous terms to construct succeeding ones. For example, the *Fibonacci* sequence is defined as

$a_n = 1$ if $n = 1$ or 2

$a_n = a_{n-1} + a_{n-2}$ otherwise.

Thus, the first few terms are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Likewise, the mathematical function *factorial* is defined on non-negative integers as:

$n! = 1$ if $n = 0$

$n! = n(n - 1)!$ if $n \neq 0$

$2! = 2 * 1! = 2$

$3! = 3 * 2! = 6$

$4! = 4 * 3! = 24$

$5! = 5 * 4! = 120$ and so on.

Such definitions, and indeed any definition that uses itself as part of itself, is said to be *recursive*. Since Modula-2 procedures must be constructed to model the calculations they perform, they too may have occasion to use themselves.

Any routine that employs itself during execution is said to be recursive.

This idea will be introduced here with some simple examples. Much more sophisticated use will be applied later in the text in a variety of contexts.

For the first, consider a procedure that implements a computation of the factorial function as it is defined, that is, recursively.

```
PROCEDURE Fac (num : CARDINAL) : CARDINAL;  
BEGIN  
  
  IF num = 0  
  THEN  
    RETURN 1  
  ELSE  
    RETURN num * Fac (num - 1);  
  END;  
END Fac;
```

Suppose the flow of control in a program enters this procedure via the call *Fac (3)*. The statement under the ELSE would execute, causing $3 * \text{Fac}(2)$ to be computed. But this re-enters *Fac*, this time with *num* equal to 2. Since *num* is a *value* parameter, a new memory location is assigned to *num* on every entry. The statement under the ELSE executes again since *num* $\neq 0$, and a computation of $2 * \text{Fac}(1)$ is called for, so *Fac* is entered a third time asking for *Fac(1)*. This causes a fourth entry to ask for *Fac(0)* which is dutifully returned back up the chain one step at a time. The successive returns are 1, 1, 2 and 6 and at the last return the calculation from the very first execution of ELSE concludes and the procedure ends.

There is one potential problem though. As in the writing of loops, one must be very careful not to specify a stopping condition on the chain of recursive calls that can never be achieved. (Here the stopping condition is $n=0$). Recursive procedures must be carefully designed, and not every programming language allows them, this being a technique that Modula-2 inherits from Pascal.

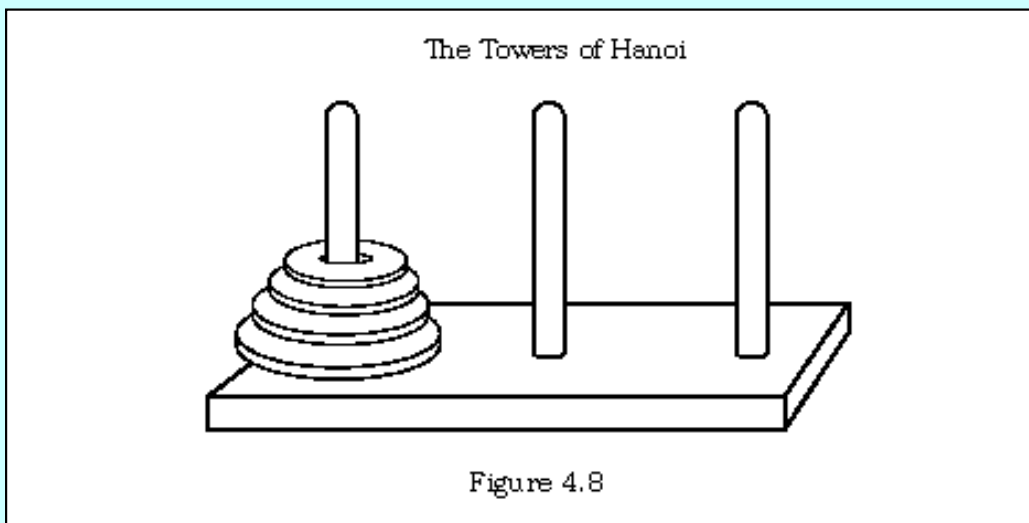
Even though the code given above does provide a simple and easily understood illustration of how recursion works, it is not difficult to see that calculating factorials does not have to be done recursively, for the solution is easily reformulated non-recursively as follows:

```

PROCEDURE FactorialNonRecursive (n : CARDINAL) : CARDINAL;
VAR
    count, result : CARDINAL;
BEGIN
    result := 1;
    count := 0;
    WHILE count < n
    DO
        INC (count);
        result := result * count
    END;
    RETURN result
END FactorialNonRecursive;

```

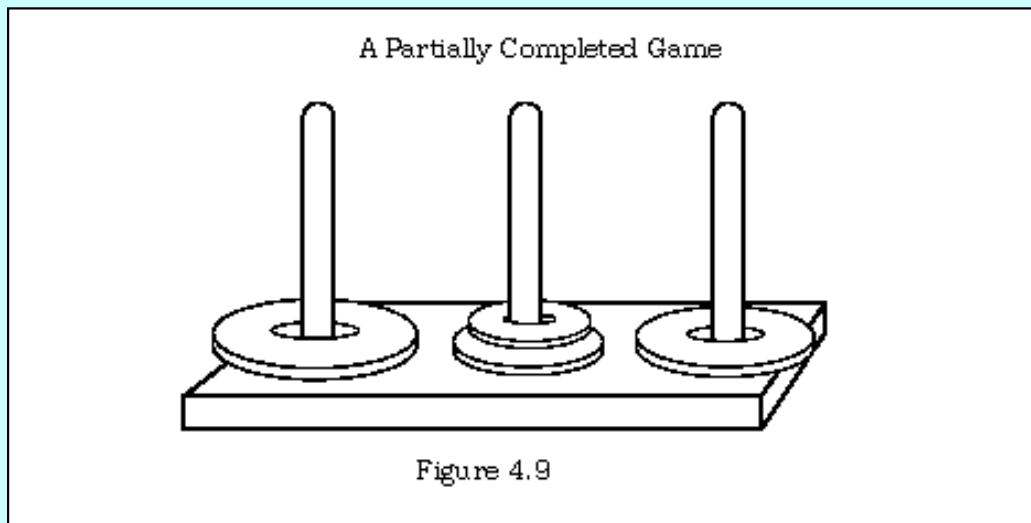
On the other hand, recursion *is* inherent in a wide variety of problems, and some of these are rather more difficult to follow than the example above. In particular, it is not always so easy to reformulate a recursive idea in a non-recursive fashion. An ancient problem of intermediate difficulty whose solution is more naturally recursive is called the *Tower of Hanoi*. Often built as a child's toy, this puzzle consists of a rigid base into which are mounted three upright pegs, and a series of concentric disks of various sizes that can be placed over the pegs. To start with, all the disks are on one of the pegs, arranged with the largest at the bottom, in descending order of radius to the smallest at the top. The idea is to move all the disks from the first peg to the second, while at all times maintaining the correct size order on each of the pegs. That is, in making a move, one must always place a disk on top of one that is larger in diameter (not necessarily the next larger, but it must be larger).



This problem is best solved by building up a solution from the simplest case. Suppose that the three pegs are labelled *origin*, *destination*, and *temporary*. If there is only one disk involved in the game, then one need only to remove it from peg *origin* and place it on peg *destination* and the game is over.

If two disks are involved, the top one can be taken from *origin* and placed on *temporary*, and the game is then reduced to two single disk moves, namely of the second disk to *destination* and then the first one from *temporary* to *destination* on top of the larger disk.

If three disks are involved, one has to move two (one less than the total number) to the *temporary* peg (use the moves for a two-disk game as above), but using the *destination* peg as a *temporary*. Then one moves the bottom disk on the *origin* peg to the (now empty) *destination* peg and finally, using the *original* peg as a *temporary*, moves the two remaining ones to the final *destination*, again using a two-disk subgame. Likewise, if one starts with four disks, one first moves three to the *temporary* peg, using the *destination* peg as a *temporary* and the rules for a move of three disks (including the sub-games for two disks and one disk) then move the last disk to the *destination*, then again use the steps above for a game of three disks to transfer those disks now on the *temporary* peg to the *destination* peg using the origin peg as the *temporary*. That is, the game of four disks includes two sub-games of three disks, each with its sub-games of two disks and one disk.



That is, at any degree of complexity for the game, one can formulate the required steps to transfer a given number of pegs in terms of two sub-games involving one fewer disk, one before and one after a move of one disk. The overall solution is therefore formulated recursively in this fashion:

To move *numberToMove* disks from the origin peg to the destination peg:

- move *numberToMove* - 1 disks to the temporary peg using the destination peg as a temporary
-
- move the last disk to the destination
-
- move the *numberToMove* - 1 disks again, this time to the destination, using the origin peg as the temporary

The procedure can call itself to do the moves, with the stopping point in each series of calls being the one at which only one disk needs to be moved. Each time the procedure is called, the pegs to be regarded as *origin*, *destination*, and *temporary* will depend on the current stage of the game.

This recursive procedure is formulated within a module that is capable of testing the entire operation and repeating the game several times. Assuming that the end user of the program has a physical model of the game onto which the moves can be translated as they are provided by the machine, instructions will be given as to which move to make at each step, with the top disk on a given peg always being the one being referred to.

MODULE TryHanoi;

```
(* Written by R.J. Sutcliffe *)
(* to illustrate recursion *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 02 26 *)
```

FROM STextIO **IMPORT**

WriteString, WriteChar, WriteLn, ReadChar, SkipLine;

FROM SWholeIO **IMPORT**

ReadCard, WriteCard;

VAR

origin, destination, temporary, key: **CHAR**;

userNumber: **CARDINAL**;

PROCEDURE TowerOfHanoi (numToMove : **CARDINAL**;

origin, destination, temp : **CHAR**);

BEGIN

IF numToMove = 1 (* only then do we move anything *)

```

THEN
    WriteString (" Move a disk from peg ");
    WriteChar (origin);
    WriteString (" to peg ");
    WriteChar (destination);
    WriteLn
ELSE (* else make a recursive call to this *)
    TowerOfHanoi ((numToMove - 1), origin, temp, destination);
    TowerOfHanoi (1, origin, destination, temp );
    TowerOfHanoi ((numToMove - 1), temp, destination, origin )
END
END TowerOfHanoi;

BEGIN    (* Main program starts here *)
    WriteLn;
    WriteString ('Before starting, please label the pegs of your ');
    WriteLn;
    WriteString ('Tower of Hanoi game as "A", "B", and "C".');
    WriteLn;
    WriteString ('Now, place the disks in order on peg "A".');
    WriteLn;
    WriteString ("Enter number of disks to move ");
    WriteString ("(use zero to end the program instead) ==");
    ReadCard (userNumber);
    SkipLine;
    WriteLn;
    IF (userNumber "A" and end at "B" *)
        TowerOfHanoi (userNumber, "A", "B", "C")
    END;
    WriteString ("press any key to continue");
    ReadChar (key);

END TryHanoi.

```

One run of this program produced the following output in the selected file for the choice of four disks:

```

Before starting, please label the pegs of your
Tower of Hanoi game as "A", "B", and "C".
Now, place the disks in order on peg "A".
Enter number of disks to move (use zero to end the program instead) ==
The binary equivalent of the decimal numeral is now read from the remainders in
reverse order.  It is 100110011000.

```

Restatement:

A solution to this problem can be formulated in terms of two procedures. The first will divide the remaining number by two and call the second to print the remainder. The second will examine the quotient and actually print the remainder if this quotient is zero. Otherwise, it will call the first procedure again before printing.

Library use:

Only STextIO and SWholeIO are used. Procedures to obtain the number to convert, and to print cardinals (zero and one) are all that are required.

Problem refinement:

Two procedures that call each other present a little difficulty. Ordinarily, one follows a "declare-it-before-using-it" rule in Modula-2, at the very least for the sake of clarity and readability. The Modula-2 *language* rules do not require this be done, envisioning that the code can be scanned as many times as necessary to resolve these circular references. However, some Modula-2 compilers only scan the source code once, so they do have "declare-it-before-using-it" as an inflexible rule. The difficulty is, if two procedures call each other, which one ought to be written out first? This is solved in such versions by adding a special reserved word FORWARD following the declaration of the *syntax* of one of the procedures at a point before its code is produced. When this is done, the compiler can check that *syntax* in the invocation by the second procedure, and postpone looking for actual code. The body of the procedure can then be elaborated somewhere else in the program, as long as the parameter list is repeated exactly at the time that is done. Note that if the compiler is of the type that makes several passes through the code, it can resolve such references *without* needing to have one procedure declared as FORWARD.

If two procedures each invoke the other, or control can otherwise pass through a chain of procedure calls and reach the original calling procedure, this is called mutual recursion or indirect recursion.

NOTES: 1. Although the multiple pass technique initially defined standard Modula-2, Wirth himself later began to write one-pass compilers in order to gain speed at compile time. As a result, FORWARD came to be regarded as an optional standard reserved word, rather than as a non-standard one. The ISO standard accepts this and allows compilers to conform regardless of which model is used, though multiple pass compilers should accept and deal correctly with FORWARD declarations, even if all they do is ignore them. That is, in the ISO standard, FORWARD is a reserved word; whether its use causes the compiler to do anything or not is optional.

2. The programmer must, as in all recursive situations, ensure that progress is actually made toward an eventual goal, or the program could end up executing in an infinite loop.

3. If the procedure being declared as FORWARD has a parameter list and/or a return type, this heading must be duplicated exactly when the procedure code is actually elaborated.

Data table:

The only variables that are needed in the main program are a cardinal to convert, a character variable keyboard responses to questions, and a boolean for recycling the entire program. Necessary imports include ReadCard, WriteCard, WriteString, WriteLn, SkipLine, and ReadChar.

Sample I/O:

An input of 2456 will produce an output of 100110011000. No padding with blanks will be undertaken; the Write will take the amount of room necessary for the output number.

Code:

```
MODULE CardBinConvert;

(* Written by R.J. Sutcliffe *)
(* to illustrate mutual recursion *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 02 26 *)
(* This program drives a procedure that prints a cardinal in binary form *)

FROM STextIO IMPORT
    WriteString, WriteLn, ReadChar, SkipLine;
FROM SWholeIO IMPORT
    WriteCard, ReadCard;

(*
PROCEDURE WriteRem (whatsLeft, RemToPrint : CARDINAL); FORWARD;
*)      (* uncomment if one-pass version *) (* code comes later *)

PROCEDURE WriteCardBin (x : CARDINAL);
(* This procedure writes the supplied cardinal in binary form
Pre: none
Post: The binary representation is written to the standard output *)
VAR
    quotient, rem : CARDINAL;

BEGIN
    quotient := x DIV 2;
    rem := x MOD 2;
    WriteRem (quotient, rem);
END WriteCardBin;

PROCEDURE WriteRem (whatsLeft, remToPrint : CARDINAL);
(* writes out the remainder from the division
Pre: whatsLeft is the current quotient from dividing by two
    remToPrint is the last remainder obtained
Post: if the quotient supplied was zero, the remainder is printed
```

else, WriteCardBin is first called with the current quotient *)

BEGIN

```
IF whatsLeft "This program tests a procedure to print ";
WriteString ("cardinals in binary form");
WriteLn;
```

REPEAT (* main repeat loop *)

```
  (* get the number *)
  WriteString ("Enter the number to be changed ");
  WriteString ("to binary form ==");
  ReadCard (theNumber);
  SkipLine;
  WriteLn;
  WriteString ("The cardinal ");
  WriteCard (theNumber, 0);
  WriteString (" converted to binary form is: ");
  WriteCardBin (theNumber);

  WriteLn;
  WriteString ("Do another? Y/N ");
  ReadChar (answer);
  SkipLine;
  again := (CAP (answer) = "Y");
UNTIL NOT again;
```

END CardBinConvert.

Here is one run from this program:

```
This program tests a procedure to print cardinals in binary form
Enter the number that is to be changed to binary form ==> 15
The cardinal 15 converted to binary form is: 1111
Do another? Y/N y
Enter the number that is to be changed to binary form ==> 0
The cardinal 0 converted to binary form is: 0
Do another? Y/N y
Enter the number that is to be changed to binary form ==> 8
The cardinal 8 converted to binary form is: 1000
Do another? Y/N n
```

[Contents](#)

4.9 An Extended Example (Compound Amounts)

This section consists not of a single example, but of a number of related ones. It culminates in a program to compute mortgage amortizations, but includes discussions of a number of topics closely related to that kind of problem.

Example 1:

Find the amount owing on \$1000 at 6% compounded annually for 10 years.

Discussion:

Since at each step of the computation the current amount owed can be computed as $A = 1.06A$, this could be solved in Modula-2 with a loop:

```
amount := 1000.0;
numYears := 10;
year := 1;
WHILE year <= numYears
DO
    amount := amount * (1.0 + intRate)
END;
```

Note, however, that this loop can be replaced with a closed formula, for what has been computed by it is just $A(1 + \text{rate})^{\text{years}}$. Using a function developed earlier, the Modula-2 code becomes:

The compound interest formula is $A_t = A_0(1+r)^t$ where A_t is the amount after t years, A_0 is the original principal, r is the annual rate and t is the number of years. If interest is compounding more often than annually, let n be the number of times per year and replace r by r/n and t by $n*t$ to get $A_t = A_0(1+r/n)^{nt}$

```
amount := amount * APowerB (1.0 + intRate, numYears);
```

Alternately, the function procedure power could be imported from *RealMath* (or a similar procedure from a non-standard library) and this line written:

```
amount := amount * power (1.0 + intRate, numYears);
```

Answer:

For the figures given in the original question, this yields \$1790.85

Example 2:

Suppose a fixed amount is deposited into a compound interest bearing account each year. How much will there be in the account after a number of years?

Discussion:

Since at each step of the computation the current amount on deposit can be computed as $\text{amount} = \text{amount}(1 + \text{rate}) + \text{payment}$, this could be solved in Modula-2 with the loop:

```

year := 1;
amount := payment;
WHILE year <= numYears
  DO
    amount := payment + amount * (1.0 + intRate)
    INC (year);
  END;

```

This is a little harder to express in a closed form. It is a sum with two parts--the periodic interest, and the periodic addition to the account. The individual sum can be thought of as a sequence of partial sums: ($R = \text{payment}$, $i = \text{rate}$)

R
 $R + R(1 + i)$
 $R + R(1 + i) + R(1 + i)^2$
 $R + R(1 + i) + R(1 + i)^2 + R(1 + i)^3$
 $R + R(1 + i) + R(1 + i)^2 + R(1 + i)^3 + R(1 + i)^4$, and so on.

The discussion on replacing loops with closed forms in section 3.8 included ways of finding partial sums of *arithmetic* sequences, but the one here is of a different kind. It has the form:

$a_1, a_1r, a_1r^2, a_1r^3, \dots, a_1r^{n-1}$

which is called a *geometric* sequence. (Each term is a fixed multiple of the one before.) In this particular instance, the first term of the sequence is the payment and each subsequent term that is added to this is $\text{payment}(1 + \text{intRate})^{n-1}$.

A sequence a_1, a_2, a_3, \dots is called geometric provided that each term is a fixed multiple of the preceding term.

The fixed multiplier so used is called the common ratio of the sequence.

The partial sums of geometric sequences can be found in a manner similar to that employed for arithmetic sequences. One writes down the product of the partial sum with r , then writes down the partial sum underneath, and subtracts.

$$\begin{array}{rcl}
 rS_n & = & a_1r + a_1r^2 + a_1r^3 + \dots + a_1r^{n-2} + a_1r^{n-1} + a_1r^n \\
 -S_n & = & a_1 + a_1r + a_1r^2 + a_1r^3 + \dots + a_1r^{n-2} + a_1r^{n-1}
 \end{array}$$

$$(r-1)S_n = -a_1 \qquad \qquad \qquad + a_1r^n$$

which is to say that, for geometric sequences in general:

$$S_n = \frac{a_1(r^n - 1)}{r - 1}$$

Applying this with the payment R and the common ratio r set to $1 + i$ as it is in the problem under discussion produces:

$$A = \frac{R \{ (1+i)^n - 1 \}}{(1+i) - 1}$$

or,

$$A = R \left[\frac{(1+i)^n - 1}{i} \right]$$

This bracketed factor is called the *series compound interest factor*, and provides a closed formula for the amount of money on deposit at a given time. The amount A calculated by this formula is also known as the *future value* of the deposit. In Modula-2:

```

futureValue := payment * (power (rate + 1.0, numYears) - 1.0) / rate

```

Example 3:

What amount must be deposited into a compound interest bearing account each year in order to accumulate a fixed amount in the account after a number of years?

Discussion:

The last formula above can also be solved for R to learn how much must be deposited at each time period to obtain a certain fixed total amount. The part of the formula in brackets is called the *sinking fund factor* and is just the reciprocal of the series compound interest factor. The amount computed is called the *size* of the annuity.

$$R = A \left[\frac{i}{(1 + i)^n - 1} \right]$$

Example 4:

Compute the monthly payments that can be made out of an interest bearing account that starts with a fixed sum and is to be emptied.

Discussion:

This is also called the *present value* of an annuity, and is the amount equal to a series of future payments. The most common use of such funds is to provide retirement fixed monthly payment out of a fund generated from contributions before retirement. However, this process also describes the way in which mortgage or other loan amounts are paid off. At each time period, interest is applied, and a payment is subtracted from the balance until nothing is left owing on the loan.

Suppose the principal amount of the loan or annuity is P. If this money were placed at compound interest for n years, it would become an amount $A = P(1 + i)^n$. Thus, $P = A(1 + i)^{-n}$ can be thought of as the present value of some amount A. If this present value P is accumulated by regular payments (either paid in or paid out,) it is a sum of the kind mentioned above, so that:

$$P = S_n(1 + i)^{-n}$$

or, substituting the compound interest factor from above for the accumulation S_n

$$P = \frac{R \left((1 + i)^n - 1 \right)}{i} \times (1 + i)^{-n}$$

and then simplifying, one obtains:

$$P = R \left[\frac{1 - (1 + i)^{-n}}{i} \right]$$

The expression in brackets is called the *series present worth factor* and the amount calculated here is the *present worth* or *present value* of the series of payments. This in turn means that the payments required to empty the annuity account (or to pay off the loan, as the case may be) can be computed by solving this last formula for R, obtaining:

$$R = P \left[\frac{i}{1 - (1 + i)^{-n}} \right]$$

The part in brackets, which is the reciprocal of the series present worth factor, is called the *capital recovery factor*. This last formula may be used to compute mortgage payments and then print an amortization table as in the following:

Problem:

Write a program to print loan amortization tables.

Comment:

The material presented here was originally written by a student. It was then edited and reorganized to fit the problem solving outline used in this text. In the process, it was shortened by about 25%.

Suitability:

See the discussion above. The problem requires that complex and repetitive computations be performed and is ideal for computer solution.

Restatement:

Given a principal, annual interest rate, and number of months for payback on a loan, calculate the monthly payment required to pay off the loan within the designated time. The user will be able to specify the month and the year in which the first payment will be made. Print out a payment schedule showing, month by month, the payment, the amount applied to the interest, the amount applied to the principal, and the new balance. Upon completion of printing, the user will be able to run the program again or terminate.

Refinement:

The opening credits, waiting for a keypress, request and validation of the starting month and year, the computation of the monthly payment required, the rounding off of dollar amounts to the nearest cent, and the printing of the amortization table can all be factored out as subtasks. Some of these are further refined as follows:

Refinement of Amortize: (Main Program)

1. Amortize will print an informative heading and instructions.
2. Amortize will obtain from the user the following data:
 - a. The principal of the loan
 - b. The annual interest rate of the loan
 - c. The number of months for payback on the loan
 - d. The month in which the first payment on the loan will be made
 - e. The year in which the first payment on the loan will be made
3. Amortize will compute the monthly payment necessary using:

$$R = P \left[\frac{i}{1 - (1 + i)^{-n}} \right]$$

4. Amortize will print a schedule listing, month by month:
 - a. the payment
 - b. the amount applied to the interest
 - c. the amount applied to the principal, and
 - d. the new balance.

4.1 In order to do this, the month and (possibly) the year must be incremented accordingly. The amount applied to the interest is based on a percentage (the monthly interest rate) of the current balance. The amount applied to the principal, therefore, is the payment minus the amount applied to the interest. The balance is the current value of the balance minus the amount applied to the principal.

4.2. The final payment on the loan has to be adjusted so that the balance is \$0.00. Therefore, the amount applied to the interest must be calculated first, and then added to the monthly payment.

4.3 All amounts will be rounded to the nearest cent in an effort to avoid an

accumulation of round-off errors.

5. When Amortize has finished listing the payment schedule, it gives the user the option of running another calculation again or terminating.

Refinement of the procedure RoundToCent:

1. Split the real number provided into two parts - dollars and cents.
2. Round off the cents by multiplying by 100 and truncating.
3. Recombine the two parts and return the result.

NOTE: Rounding is done this way so as to avoid multiplying the entire number by 100 before truncating it, thus severely limiting the size range of numbers that can be rounded off correctly.

Refinement of the information procedures: GetPrincipal, GetRate, GetNumMonths, GetStartMonth, and GetStartYear:

1. Each of these prints a prompt asking for the required information about the loan.
2. If the amount is outside reasonable limits, an error message is printed and the user is prompted for the information again.
3. The valid amount is returned.

Refinement of the procedure CalcPayment:

1. The parameters principal, intRate, and numMonths are passed, and the formula in 3 above is used to do the calculation.

Refinement of procedure PrintOut:

1. PrintOut is passed: firstMonth, firstYear, numOfMonth, payment, intRate, and principal.)

Pseudocode:

```
curMonth = startMonth
curYear = startYear
count = 1;
while curMonth <= numOfMonth
    Print out curMonth and curYear
    If month is December then
        Increment curYear
        curMonth = January
    Otherwise
        Increment curMonth
    end the if
amountAppliedToInterest = (intRate * principal)
If curMonth = numOfMonth then
    payment = principal + amountAppliedToInterest
Print the payment
Print amountAppliedToInterest
```

```

    amountAppliedToPrincipal = (payment - amountAppliedToInterest)
    Print amountAppliedToPrincipal
    principal = principal - amountAppliedToPrincipal
    Print principal in a right-justified column
end the while

```

Data Table:

```

the main program Amortize:
    principal, intRate, payment : reals
    numOfMonths, firstMonth, firstYear : cardinals
    response, cr : chars
the procedures HoldScreen, PrintAuthor, PrintOpeningMessage, APowerB
    none
the procedure RoundToCent:
    dollars, cents : real
    roundcents : integer
the procedure GetPrincipal:
    princOk : a boolean
    thePrinc : a real
the procedure GetRate:
    RateOk : a boolean
    theRate : a real
the procedure GetNumMonths:
    MonthsOk : a boolean
    theNum : a cardinal
the procedure GetStartMonth:
    month : a cardinal
    monthOk : a boolean
the procedure GetStartYear:
    year : a cardinal
    yearOk : a boolean
the procedure CalcPayment:
    payment : a real
the procedure PrintOut:
    lineNumber, count, curMonth, curYear : cardinals
    interestPaidOff, principalPaidOff : reals

```

Library Use:

NOTE: For the benefit of students who have traditional versions of Modula-2, this example has been left with the library use employed when it was submitted. One of the exercises will be to convert it into standard Modula-2.

```

From InOut:
    WriteLn, WriteString, ReadCard, Done, WriteCard, Write, ReadString
From RealIO:
    WriteReal, ReadReal, Done
From MathLib:
    Exp, Ln

```

Sample I/O:

For an input principal of \$100.00, an interest rate of 10%, repayment of one month starting September 1991, Amortize will print a chart such as:

Month	Payment	Amount	Amount	Balance
		Applied to Interest	Applied to Principal	
September 1991	\$100.83	\$.83	\$100.00	\$ 0.00

User's Manual:

When you first start up the program Amortize, a screen will appear that gives information about Amortize's author and the class for which Amortize was written. Continue by pressing any key.

New information will appear giving basic information about the required input and functions of Amortize. Read this over carefully so you are fully aware of all your options in Amortize. When you have finished reading the text on this screen, press any key.

You now need to type in the principal, annual interest rate, and number of months for payback on your loan. Once you have entered this information, you will then need to specify the month and the year in which the first payment will be made. (Amortize can then print the date and the month of each payment in your schedule). Press any key to continue.

Now you will be presented with a numbered list of the twelve months of the year. Select the number corresponding to the month in which the first payment will be made. Then specify the year in which the first payment will be made. Amortize will then display, month by month, the payment, the amount applied to the interest, the amount applied to the principal, and the new balance. If there are too many months to fit on one screen, Amortize will pause so you can view each screen at your own leisure. Press any key to continue the listing.

Once Amortize has finished printing your payment schedule, you can run Amortize again or you can quit. Press "y" to run the program again, or hit any other key to terminate Amortize.

Error-handling Messages And Procedures:

Amortize allows principals of up to \$500000.00. Specifying a principal that is greater than \$500000.00 (or less than \$0.01, for obvious reasons) will cause Amortize to print the following error message:

```
Sorry, but that is an invalid principal.
Please specify a principal between $0.01 and $500000.00.
Try again here ==>
```

Amortize will continue to print this error message until a valid annual interest rate (between 0 and 1) is entered.

Amortize assumes 1,000 to be the maximum number of months for payback on a loan. (Since there are 12 months in a year and a person lives an average of 70 years, 1,000 months for payback on a loan is an ample amount of time). Specifying the number of months to be greater than 1,000 (or less than 1, for obvious reasons) will cause Amortize to print the following error message:

```
Sorry, but that is an invalid number of months.
Please specify the number of months to be between 1 and 1,000.
Try again here ==>
```

Amortize will continue to print this error message until a valid year (between 1985 and 2050) is entered.

Code:

```
( * Name : Stephanie L. McLeod *)
( * Student Number : 880664 *)
( * Cmpt 360 Programming Languages Fall 1990 *)
( * Assignment #2 : "Amortize" *)
( * Printing Out Payments on a Loan *)
( * edited and revised by R. Sutcliffe 1991 03 28 *)
```

```
MODULE Amortize;
```

```
IMPORT InOut; (* to use Done from both InOut and RealInOut*)
FROM InOut IMPORT
    WriteLn, WriteString, ReadCard, Done, WriteCard, Write, Read;
FROM RealIO IMPORT
    WriteReal, ReadReal;
FROM MathLib IMPORT
    Exp, Ln;
```

```
VAR
    principal, intRate, payment : REAL;
    numMonths, firstMonth, firstYear : CARDINAL;
    response, cr : CHAR;
```

```
PROCEDURE HoldScreen; (* called whenever one wants to wait for a keypress *)
```

```
BEGIN
    WriteString ("Press carriage return to continue ==");
    Read (cr);
    WriteLn;
END HoldScreen;
```

```
PROCEDURE PrintAuthor; (* Prints out information about the author *)
```

```
BEGIN
    WriteString ("*****");
    WriteLn;
    WriteString ("                Amortize");
    WriteLn;
    WriteString ("                Was Created By");
    WriteLn;
    WriteString ("                Stephanie L. McLeod, Student # 880664");
    WriteLn;
    WriteString ("                For");
    WriteLn;
    WriteString ("                Cmpt 360, Fall 1990");
    WriteLn;
    WriteString ("                Assignment #2");
    WriteLn;
    WriteString ("                Printing Out Payments on a Loan");
    WriteLn;
    WriteString ("*****");
    WriteLn;
    WriteLn;
END PrintAuthor;
```

PROCEDURE PrintOpeningMessage;

(* Informs the user of the input required for Amortize and describes the functions of Amortize*)

BEGIN

```
WriteString ("Amortize has been designed to print a schedule of ");
WriteString ("monthly payments");
WriteLn;
WriteString ("for your loan.  When Amortize first begins, you ");
WriteString ("will be required to");
WriteLn;
WriteString ("enter the principal, the annual rate of interest, ");
WriteString ("and the number ");
WriteLn;
WriteString ("of months for payback on the loan.  You will also ");
WriteString ("specify the ");
WriteLn;
WriteString ("month and the year in which the first payment ");
WriteLn;
WriteString ("will be made.");
WriteLn;
WriteLn;
WriteString ("Then Amortize will print a schedule showing, ");
WriteString ("month by month,");
WriteLn;
WriteString ("the payment, the amount applied to the interest, ");
WriteString ("the amount ");
WriteLn;
WriteString ("applied to the principal, and the new balance.");
WriteLn;
HoldScreen;
```

END PrintOpeningMessage;

PROCEDURE APowerB (a, b : **REAL**) : **REAL**;

(* pre: a raised to the b is not greater than **MAX** (**REAL**)
post: a raised to the b is returned *)

BEGIN

```
RETURN Exp (b * Ln (a) )
```

END APowerB;

PROCEDURE RoundToCent (theAmount : **REAL**) : **REAL**;

(* can be used to round off any real number to two decimal places
Pre : theAmount is less than the maximum integer
Post : the number returned is theAmount rounded to two decimal places *)

VAR

```
dollars, cents : REAL;  
roundCents : INTEGER;
```

BEGIN

```
dollars := FLOAT (TRUNC (theAmount));  
cents := theAmount - dollars;  
roundCents := TRUNC (100.0 * (cents + 0.005));
```

```
    RETURN dollars + (FLOAT (roundCents) / 100.0);  
END RoundToCent;
```

```
PROCEDURE GetPrincipal (): REAL;
```

```
VAR
```

```
    princOk : BOOLEAN;  
    thePrinc : REAL;
```

```
BEGIN
```

```
    REPEAT
```

```
        WriteString ("What is the principal (or amount) of your loan? ==");  
        ReadReal (thePrinc);  
        Read (cr); (* consume line feed *)  
        princOk := Done AND (thePrinc < 500000.00);
```

```
    IF NOT princOk
```

```
        THEN
```

```
            WriteString ("Sorry, but that is an invalid principal.");  
            WriteLn;  
            WriteString ("Please specify a principal between $0.01 and ");  
            WriteString (" $500000.00. Try again here ==");
```

```
        END;
```

```
    UNTIL princOk;
```

```
    WriteLn;
```

```
    RETURN thePrinc;
```

```
END GetPrincipal;
```

```
PROCEDURE GetRate (): REAL;
```

```
VAR
```

```
    rateOk : BOOLEAN;  
    theRate : REAL;
```

```
BEGIN
```

```
    REPEAT
```

```
        WriteString ("What is the annual interest rate on your loan?");
```

```
        WriteLn;
```

```
        WriteString ("For example, type in 6% as 0.06 ==");
```

```
        ReadReal (theRate);
```

```
        Read (cr); (* consume line feed *)
```

```
        rateOk := Done AND (theRate < 1.0);
```

```
    IF NOT rateOk
```

```
        THEN
```

```
            WriteString ("Sorry, but that is an invalid interest rate.");  
            WriteLn;  
            WriteString ("Please specify an interest rate between 0 and 1.");  
            WriteLn;
```

```
        END;
```

```
    UNTIL rateOk;
```

```
    WriteLn;
```

```
    RETURN theRate;
```

```
END GetRate;
```

```
PROCEDURE GetNumMonths (): CARDINAL;
```

VAR

monthsOk : **BOOLEAN**;
theNum : **CARDINAL**;

BEGIN

REPEAT

WriteString ("How many months do you have to pay back the loan?");

WriteLn;

WriteString ("Type in the number of months here ==");

ReadCard (theNum);

Read (cr); (* skip linefeed *)

monthsOk := InOut.Done **AND** (theNum < 1000);

IF NOT monthsOk

THEN

WriteString ("Sorry, but that is an invalid number of months.");

WriteLn;

WriteString ("Please specify the number of months to be ");

WriteString ("between 1 and 1000.");

WriteLn;

END;

UNTIL monthsOk;

WriteLn;

RETURN theNum;

END GetNumMonths;

PROCEDURE GetStartMonth (): **CARDINAL**;

(* Pre - None

Post - Returns the number of the month for the first payment. *)

VAR

month : **CARDINAL**;
monthOk : **BOOLEAN**;

BEGIN

REPEAT (* checks for valid input *)

WriteString ("In which month will the first payment be made? 1 - 12");

WriteLn;

WriteString ("Type in the number of the month here ==");

ReadCard (month);

Read (cr); (* skip linefeed *)

monthOk := InOut.Done **AND** (month <= 12);

IF NOT monthOk

THEN

WriteLn;

WriteString ("1. January");

WriteLn;

WriteString ("2. February");

WriteLn;

WriteString ("3. March");

WriteLn;

WriteString ("4. April");

WriteLn;

```

        WriteString ("      5.  May");
        WriteLn;
        WriteString ("      6.  June");
        WriteLn;
        WriteString ("      7.  July");
        WriteLn;
        WriteString ("      8.  August");
        WriteLn;
        WriteString ("      9.  September");
        WriteLn;
        WriteString ("     10.  October");
        WriteLn;
        WriteString ("     11.  November");
        WriteLn;
        WriteString ("     12.  December");
        WriteLn;
        WriteLn;
    END; (* if *)
UNTIL monthOk;
WriteLn;
RETURN month
END GetStartMonth;

PROCEDURE GetStartYear (): CARDINAL;
(* Pre  -  None
   Post -  Returns the year in which the first payment is made *)

VAR
    year : CARDINAL;
    yearOk : BOOLEAN;

BEGIN
    REPEAT
        WriteString ("In what year will the first payment be made? ==");
        ReadCard (year);
        Read (cr); (* skip linefeed *)
        yearOk := InOut.Done AND (year < 2050);
        IF NOT yearOk
            THEN
                WriteString ("Please specify a year between 1985 and 2050.  ");
            END;
        WriteLn;
    UNTIL yearOk;
    RETURN year
END GetStartYear;

PROCEDURE CalcPayment (princ, rate : REAL;
    numPayments : CARDINAL): REAL;

(* Pre  -  princ is the principal amount, rate is the monthly interest, numPayments
is the number of monthly payments
   Post -  Returns the monthly payment necessary to pay off the loan rounded to the
nearest cent *)

```

```

VAR
    payment : REAL;

BEGIN
    payment := (princ * rate) / (1.0 - (1.0 / APowerB (1.0 + rate, FLOAT(numPayments))
) );
    RETURN RoundToCent (payment);
END CalcPayment;

PROCEDURE PrintOut (startMonth, startYear, numPayments : CARDINAL;
    monthlyPay, percentage, princ : REAL);

(* Prints a schedule showing, month by month, the payment, the amount applied to the
interest, the amount applied to the principal, and the new balance.
Pre - startMonth, startYear, numPayments, monthlyPay, percentage and princ have all
been set up in the main program
Post - none *)

VAR
    lineNumber, count, curMonth, curYear : CARDINAL;
    interestPaidOff, principalPaidOff : REAL;

BEGIN (* print a title and column headings for the schedule *)
    WriteString ("                        Amount                Amount");
    WriteLn;
    WriteString ("                        Applied to            Applied to");
    WriteLn;
    WriteString ("Month                Payment            Interest            Principal");
    WriteString ("    Balance");
    WriteLn;
    WriteString ("-----                -----                -----                -----");
    WriteString ("    -----");
    WriteLn;
    lineNumber := 12; (* print a year at a time *)

    (* print the data for each month with appropriate spacing*)
    curMonth := startMonth;
    curYear := startYear;
    count := 1;
    WHILE count <= numPayments
    DO
        IF curMonth = 1
        THEN
            WriteString ("January    ");
        ELSIF curMonth = 2 THEN
            WriteString ("February  ");
        ELSIF curMonth = 3 THEN
            WriteString ("March      ");
        ELSIF curMonth = 4 THEN
            WriteString ("April      ");
        ELSIF curMonth = 5 THEN
            WriteString ("May        ");
        ELSIF curMonth = 6 THEN

```

```

        WriteString ("June      ");
ELSIF curMonth = 7 THEN
        WriteString ("July      ");
ELSIF curMonth = 8 THEN
        WriteString ("August   ");
ELSIF curMonth = 9 THEN
        WriteString ("September ");
ELSIF curMonth = 10 THEN
        WriteString ("October  ");
ELSIF curMonth = 11 THEN
        WriteString ("November ");
ELSIF curMonth = 12 THEN
        WriteString ("December ");
END;  (* if *)

```

```

WriteCard (curYear, 0);
WriteString ("  $ ");
interestPaidOff := RoundToCent (princ * percentage);
(* calculate the various amounts for the printout *)
IF count = numPayments (* is final payment? *)
    THEN (* adjust last payment for rounding off *)
        monthlyPay := RoundToCent (princ + interestPaidOff);
    END;
principalPaidOff := RoundToCent (monthlyPay - interestPaidOff);
princ := RoundToCent (princ - principalPaidOff);
(* now do the printout for this month *)
WriteReal (monthlyPay, 7, 2);
WriteString ("    $ ");
WriteReal (interestPaidOff, 7, 2);
WriteString ("    $ ");
WriteReal (principalPaidOff, 7, 2);
WriteString ("    $ ");
WriteReal (princ, 7, 2);
WriteLn; (* done one line *)

```

```

IF count = lineNumber (* up to number of lines set? *)
    THEN (* yes, so let the person have a look first *)
        WriteLn;
        HoldScreen;
        INC (lineNumber, 12); (* then print another year *)
    END;  (* if *)

```

```

(* now reset the counters for the next round *)
IF curMonth = 12 (* was december this time? *)
    THEN
        INC (curYear); (* yes, reset to January of next year *)
        curMonth := 1
    ELSE
        INC (curMonth) (* no, so increment month, leave year alone *)
    END;  (* if *)
INC (count);

END  (* while *)

```



```

END PrintOut;

BEGIN (*main program *)
  PrintAuthor;
  PrintOpeningMessage;
  REPEAT (* main repeat loop around whole thing *)
    principal := GetPrincipal ();
    intRate := GetRate()/12.0; (* Obtain & convert to monthly interest rate *)
    numMonths := GetNumMonths ();
    firstMonth := GetStartMonth ();
    firstYear := GetStartYear ();
    payment := CalcPayment (principal, intRate, numMonths);
    PrintOut (firstMonth, firstYear, numMonths, payment, intRate, principal);

    WriteLn;
    WriteString ("Would you like to run Amortize again?  If so, ");
    WriteString ('then type "y".');
    WriteLn;
    WriteString ("If not, then press any other key to quit ==");
    Read (response);
    Read (cr); (* skip linefeed *)
  UNTIL (CAP (response) # "Y")

END Amortize.

```

Sample Run:

```

*****
                        Amortize
                        Was Created By
Stephanie L. McLeod, Student # 880664
                        For
                        Cmpt 360, Fall 1990
                        Assignment #2
                        Printing Out Payments on a Loan
*****

```

Amortize has been designed to print a schedule of monthly payments for your loan. When Amortize first begins, you will be required to enter the principal, the annual rate of interest, and the number of months for payback on the loan. You will also specify the month and the year in which the first payment will be made.

Then Amortize will print a schedule showing, month by month, the payment, the amount applied to the interest, the amount applied to the principal, and the new balance.

Press carriage return to continue ==> \$ 10000.00

What is the annual interest rate on your loan?

For example, type in 6% as 0.06 ==> 20

In which month will the first payment be made? 1 - 12

Type in the number of the month here ==> 1991

Month		Payment		Amount Applied to Interest		Amount Applied to Principal		Balance
-----		-----		-----		-----		-----
September	1991	\$	538.02	\$	70.83	\$	467.19	\$ 9532.81
October	1991	\$	538.02	\$	67.52	\$	470.50	\$ 9062.31
November	1991	\$	538.02	\$	64.19	\$	473.83	\$ 8588.48
December	1991	\$	538.02	\$	60.84	\$	477.18	\$ 8111.30
January	1992	\$	538.02	\$	57.46	\$	480.56	\$ 7630.74
February	1992	\$	538.02	\$	54.05	\$	483.97	\$ 7146.77
March	1992	\$	538.02	\$	50.62	\$	487.40	\$ 6659.37
April	1992	\$	538.02	\$	47.17	\$	490.85	\$ 6168.52
May	1992	\$	538.02	\$	43.69	\$	494.33	\$ 5674.19
June	1992	\$	538.02	\$	40.19	\$	497.83	\$ 5176.36
July	1992	\$	538.02	\$	36.67	\$	501.35	\$ 4675.01
August	1992	\$	538.02	\$	33.11	\$	504.91	\$ 4170.10

Press carriage return to continue == "y".
If not, then press any other key to quit ==> n

[Contents](#)

4.10 Chapter Summary

This chapter covered these topics:

- what a procedure is
- how to write and invoke a procedure
- the difference between value and variable parameters
- how to write preconditions and postconditions for procedures
- what a function procedure is
- about some standard procedures in Modula-2
- some rules of style and good taste in naming procedures
- recursion

It included discussion of the following Modula-2 built-ins:

Reserved Words

FORWARD

PROCEDURE

RETURN

VAR (new context)

Standard Identifiers

CHR

MAX

MIN

ODD

Imported Identifiers

RealMath:

exp

ln

sqrt

power

4.11 Assignments

Questions

1. What is a procedure, and under what circumstances is it better to use a procedure than to write all the code in the main block of the program module?
2. What are four different models for the action of a procedure on the data flow of a program?
3. What is meant by the *invocation* of a procedure?
4. What do you call a procedure that invokes itself? When should this technique be used?
5. What is the difference between a value parameter and a variable parameter? Illustrate with examples.
6. Comment on this statement: (Is it true or not, and why?) "*WriteCard* can write a compatible variable of type INTEGER, but *ReadCard* cannot read a compatible value to a variable of type INTEGER."
7. What are the differences between a proper procedure and a function procedure? Illustrate with examples.
8. What does RETURN do?
9. Classify each of the following as either built-in or imported procedures and as either a proper procedure or a function procedures (or, none of these.) WriteLn, ABS, FLOAT, ReadReal, MAX, ReadResult, ReadChar, CAP, MIN, exp, ln.
10. Explain why the recursive procedure Fac would not work if the parameter were a variable parameter instead of a value parameter.
11. What is the difference between simple and compound interest?
12. What is the difference between net present value and net future value?
13. Explain why mortgage payments against a debt can be handled like annuity payments out of an accumulated account.

Problems

14. Rewrite the procedure [Round](#) in section 4.1 to round off a REAL to an INTEGER instead of to a CARDINAL.
15. Extend the program [Areas](#) in section 4.2 to include the computation of the areas of triangles and rectangles, each with its own procedure.
16. Write and test a function procedure to convert radians to degrees, and a function procedure to convert degrees to radians.
17. The procedure *Swap* can be thought of as a special case of a *rotation permutation* (item 1 is moved to item two, which is moved to item three, etc, and item n is moved to item one.) Write and test a procedure that does a rotation permutation of four items, namely:

```
PROCEDURE PermuteFour (VAR item1, item2, item3, item4 : CARDINAL);
```

(* does a rotation permutation

Pre: none

Post: item#i becomes item #(i+1) and item#4 becomes item#1)

18. Write and test a procedure to generate the i th term of a Fibonacci sequence.

19. When two points are given in coordinate form $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$, the distance between the two points is given by the formula

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Write and test a procedure to make this calculation from point coordinates that are input from the keyboard.

20. Write a program to compute the compound interest on a deposit. The principal amount, interest rate, number of years, and the number of times per year the interest is compounded should all be read in from the keyboard. Note that in the formula $A = P(1 + i)^n$ for compound interest i will be the annual rate divided by the number of times per year the interest is compounded. Likewise, n is the total number of times the compounding is done, not just the number of years.

21. Rewrite the procedure [Round](#) to take into account that some reals input to the procedure may be negative.

22. Generalize the mortgage and annuity program so that the payment period can be other than monthly. Offer the choice of annual, quarterly, bi-weekly, and weekly payments.

23. The period T of a simple pendulum is given by the formula

$$T = 2\pi\sqrt{\frac{l}{g}}$$

where l is the length of the pendulum in metres and g is the gravitational constant 9.8m/s^2 . Write a procedure to do this calculation. Test the procedure in a program that prints out the period and frequency ($f = 1/T$) of pendulums from 10cm to 100cm in length in a table. Use increments of 10cm from one to the next.

24. A dynamic system that is being accelerated at a constant acceleration a has its initial velocity u , final velocity v and the distance travelled related by the formula $v^2 - u^2 = 2ad$. Write a program module that can calculate any one of these in terms of the other three. Each of the four possible calculations ought to be realized in a separate procedure.

25. The "greatest integer" or *Floor* function takes a real parameter and returns the greatest integer less than or equal to the supplied parameter. Note that this is the same as `INT` only for positive numbers, for `INT` (-5.7) produces -5, but *Floor* (-5.7) produces -6. Write and test the function procedure *Floor*.

26. Write and test the corresponding *ceiling* function that takes a real parameter and returns the smallest integer greater than or equal to the parameter.

27. Write a Program module that will dispense change for a dollar for some purchased item costing less than a dollar. The number and denomination of each coin dispensed should be given and the largest possible number of quarters, then of dimes, nickels and pennies should be dispensed. Use at least one procedure.

28. Write a program that can add fractions, finding the appropriate common denominator, doing the addition, and then reducing the fraction to lowest terms. Input should come from the keyboard. If the user inputs 2, 3, 4, 5 for the fractions $2/3$ and $4/5$ respectively, then the output should be in the form

2	4	7
-	+	-
3	5	17

29. Write and test a procedure to convert a cardinal to any base from 2 through 9, namely: PROCEDURE WriteCardInBase (num, destBase : CARDINAL)
30. Write a program that will accept a CARDINAL from the keyboard and print it out in words. (Thus 4625 comes out as four thousand six hundred twenty five.)
31. Write a program that will accept as input today's date and a birthdate and compute and print the number of days the person has been alive.
32. Write a program that will accept a CARDINAL x and a CARDINAL exponent y from the keyboard and then use a loop to compute x to the power y. The latter portion should be encapsulated in a procedure. Carefully trap any potential overflows.
33. Write a program to simulate a cash register. The screen should start by giving a message such as: "Please enter the amounts." It should then prompt for each dollar amount with a sign like ">". The user will type in one dollar amount after another until an invalid entry is made (just a carriage return, say). At that point, the program should display on the screen the total amount thus far, then the amount of sales tax for your State or Province (this could be a constant), and then the grand total. It should then display the "?" prompt and wait to see how much cash is tendered by the customer, and then calculate and print the amount of change. A sample "tape" as it might appear on the screen is shown below:

```

Please enter the amounts:
> 1.35
> 4.72
> 11.51
> 9.63
>-----
      27.21  subtotal
      1.90  sales tax 7%
-----
      29.11  total sale
?      30.00  cash tendered
*****
      .89   change

```

34. The equation $ax^2 + bx + c = 0$ (a, b, c are real numbers with a
The roots of the equation are real only if the portion $b^2 - 4ac$ (the discriminant) is positive. Write a program that will accept the values of a, b, and c, with a procedure to determine whether the roots are real and if they are, another procedure to solve for x.
35. A prime number is one that has exactly two different positive divisors--itself and one. Write a BOOLEAN valued function that returns TRUE if the parameter is prime, and FALSE otherwise. Test your procedure by calling it from some main program.

36. Write a function procedure that displays the prime factorization of the CARDINAL parameter that it is passed. For instance, if the number input is 60, the display should read: $60 = 2 * 2 * 3 * 5$ Test this procedure by including it in a program that obtains the numbers to factor from the keyboard.
37. A *perfect number* is one that is the sum of its divisors. Six is perfect because $6 = 1 + 2 + 3$. Write a procedure that determines if a number is perfect and prints it along with a message if it is. Test this by writing an encapsulating Module that calls the procedure successively for 1, 2, 3, ... someLargeCardinal. Are there very many perfect numbers?
38. Rewrite the mortgage amortization program [Amortize](#) in ISO standard Modula-2 and test it carefully.
-

[Contents](#)

Chapter 4

Program Organization and Procedures

[4.0 Chapter Goals](#)

[4.1 What is a Procedure, and Why Use It?](#)

[4.2 Writing and Calling Procedures](#)

[4.3 Value and Variable Parameters \(Introduction\)](#)

[4.4 Predicates](#)

[4.5 Function Procedures](#)

[4.6 Summary of Built-in Procedures](#)

[4.7 Some Stylistic Considerations](#)

[4.8 Recursion](#)

[4.9 An Extended Example \(Compound Amounts\)](#)

[4.10 Chapter Summary](#)

[4.11 Assignments](#)

[Contents](#)

5.0 Chapter Goals

The purpose of this chapter is to examine data types that have some kind of numeric ordering, whether that ordering is explicit or only implicit. Such types may consist of a sequence of discrete items that supply the possible values for variables of that type. These are called ordinal types, and include the whole number, character, and boolean types already encountered, as well as some new ones introduced in this chapter. On the other hand, the type may support variables that themselves consist of an ordered sequence of values with an explicit indexing scheme. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

ordinal data and indexed data

Realized in the Modula-2 notation:

ordinal types and their subranges, enumerated types, and arrays

Data Manipulation Abstractions

General:

a means of associating specific cardinals with the position a value occupies in an enumeration and vice-versa

Realized in the Modula-2 notation:

enumeration/cardinal conversion

Programming Abstractions

General:

iterations, nested loops

Realized in the Modula-2 notation:

the FOR loop as a variation on top-of-loop tested repetition for iterations

5.1 Abstract and Transparent Data Types in Modula-2

In chapter one, it was pointed out that there are a variety of levels of abstraction associated with the details of a task. (Cutting down trees can be viewed as a single task by an experienced logger, or as a series of subtasks in varying detail, depending on the need to elaborate). This is also true of programs--they can be thought of as single entities solving a single task (once they are written), but during their construction attention to the individual details is necessary.

Similar remarks may be made about data. Up to this point, the only data types employed by this text have been those built into Modula-2. It has not been necessary for the programmer to be concerned about specifying in the program any of:

- a) how the data is represented in the computer,
- b) what the possible values for the type were to be, or,
- c) what operations will work with the data type in question.

These characteristics have, in every case seen thus far, been prearranged and the details of the data type's specification hidden from the program. In using an entity of type REAL or CARDINAL, for example, one need not ordinarily be concerned with the way in which the numbers are stored in the computer. Each such entity is considered and used as an organic and indivisible whole. Likewise, while one understands that not everything that can be a real number in mathematics can be one in a program (because infinite precision is not available), the values available for assignment to the type REAL are simply taken for granted. The operations indicated by +, -, *, and / operators are also built-in for all the numeric data types provided in most computing notations.

A data type whose representation (and operational) details are hidden from the user of entities of that type is said to be an abstract data type. This is sometimes abbreviated as ADT.

NOTES:

1. The programmer *may* know, or be able to discover the representation details. The type is still abstract if that knowledge is not *used* in a program, that is, if it is not depended on in any way.
2. Sometimes it is suitable to say more about such representation details. For instance, one could think of a REAL as consisting of two cardinals--one to specify the significant figures of the real, and the other to signify the exponent. On the other hand, the REAL could be thought of as the string of characters that would have to be printed on an output device for a user to view it. Thinking about it in either way would not say anything about how the data is actually stored in the computer, though both views might assist in working with the type REAL.

Of more importance to this discussion is the fact that a computing notation usually allows the user to create new data types, and to specify certain details about such types in the program. As indicated above,

three aspects of data types may be of interest. This chapter will be concerned only with making a transparent list of potential values for a new data type. In later chapters, the issue of specifying the structure or representation of the data (at some level of abstraction), and that of specifying operations on items of the type will be considered.

A data type whose representation details are visible to the user of entities of that type is said to be transparent.

[Contents](#)

5.2 Making One's Own Data Types

Although small in the sense that it employs only a limited number of pre-defined words, Modula-2 is very flexible and expressive notation. When a feature one needs is lacking, it can usually be added by the programmer. There are a variety of ways of doing this. Some procedures and data types are available in library modules supplied with the implementation and can be imported from them. As discussed in the next chapter, a programmer may even create custom libraries of such Modules if desired.

However, many needs are specific to a particular problem and are defined and used only within the confines of a single program module. As indicated above, a program can not only specify new functionality by defining procedures, (chapter 4) but if the built-in data types are insufficient for clear expression of a problem solution, one can invent new ones. Consider, for instance, a payroll problem oriented by the days of the week. It may be desirable to be able to write a loop like:

```
day := Monday;
WHILE day # Friday
  DO
```

This could be accomplished readily enough with a long **CONST** declaration that equated numbers to the days of the week and by declaring *day* to be a variable of type **CARDINAL**. To do this, one would have to write:

```
CONST
  Monday = 1;
  Tuesday = 2;
  etc.
VAR
  day : CARDINAL;
```

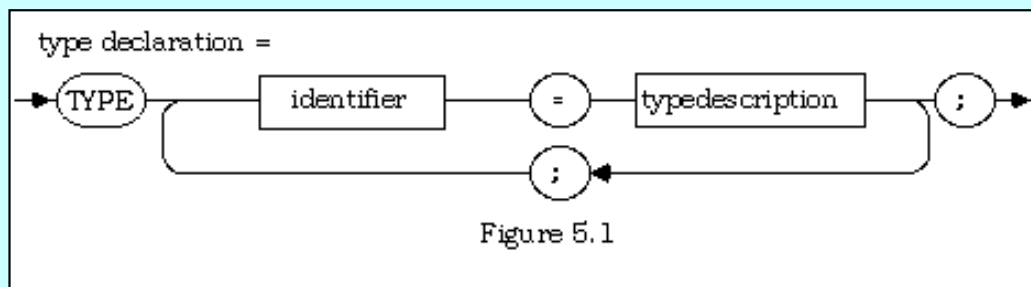
Conceptually, at least, the variable *day* is of a type that uses the names of the days of the week. Actually, it remains of type **CARDINAL**, and there is nothing to prevent some entirely inappropriate value from being assigned to *day*. Of course *day* could be restricted to [1..5] in the declaration, and this would improve things, but the declarations would still be rather clumsy.

5.2.1 Ordinal and Enumerated Types

There is a much better way: Create a new type of variable whose values can be the names of the days themselves, and specify *day* to be of this new type. Here are the appropriate declarations:

```
TYPE
  DayName = (Monday, Tuesday, Wednesday, Thursday, Friday);
VAR
  day : DayName;
```

The syntax diagram for a **TYPE** declaration is in figure 5.1:



Once a new type has been created in this way, variables can be declared to be of that type and manipulated very much as the built-in ones like INTEGER and CARDINAL. There are several other possibilities for creating new types; the one in this section is defined as follows:

An enumerated type has its possible values specifically listed by name in its declaration. There is a finite (Cardinal) number n of these values, and they can be thought of as being associated with the constants 0, 1, 2, 3, ... $n-1$.

NOTE: Unlike Pascal and Modula-2, some computing languages do not permit the declaration of enumerated types. That is, Modula-2 has already built-in the template for creating and using enumerated types. The representation is still hidden (abstract), though it is a sequential association of numbers with names. The names themselves constitute the potential values for variables of the new type; these are transparent. Once the new type has been declared and the variables created with that type, loops such as the one with which this discussion began are perfectly in order. Note, however, that such a type is not numeric, so that within the loop, the next value of the enumeration cannot be obtained by employing the addition operator. Instead, INC or DEC must be used:

Correct:

```
day := Monday;
WHILE day < Friday (* INC(Friday) doesn't work *)
  DO
    statement sequence;
    INC (day);
END;
```

Incorrect:

```
day := day + 1;
```

NOTES: 1. As was the case in using INC and DEC with numeric variables, care must be taken not to increment or decrement past the end of the range of the variable type. If either the value of *day* were *Monday* and one executed *DEC (day)*, or if the value of *day* were *Friday* and one executed *INC (day)*, the value of *day* would become undefined.

2. The alternate forms INC (day, n) and DEC (day, n) may also be used, subject to the stipulation in note (1) above.

3. One may *not*, however write INC (day, Tuesday) as if *Tuesday* were actually a cardinal value.

Even though it is not possible to directly treat a value in an enumeration as a cardinal, it is possible to convert back and forth between the value of the item and the value of the cardinal associated with its position in the enumeration. This is done in the following way.

If one has:

```
TYPE
  DayName = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);

VAR
  day : DayName;
```

```
cardNum : CARDINAL;
ch : CHAR;  (* this is a built-in enumerated type *)
```

then the following assignments return the indicated values.

```
cardNum := ORD (Sunday)           cardNum is now 0
cardNum := ORD (Thursday)         cardNum is now 4
day := VAL (DayName, 2)           day is now Tuesday
day := VAL (DayName, 7)           Range error. Last is #6.
cardNum := ORD ("A")              cardNum is now 65
ch := VAL (CHAR, 90)            ch is now "Z"
```

The built-in function ORD takes one of the names associated with the values of an enumerated type and returns the CARDINAL value corresponding to the position of the name in the enumeration.

The built-in function VAL takes the name of an enumerated type and a CARDINAL value of a position number in the enumeration and returns the corresponding value in the specified type.

Since the position of an item in a list ranges from 0 through (n-1) where n is the number of items, one might think that it would be proper to use VAL and ORD to convert only to and from CARDINALs, not INTEGERS. See the earlier discussion concerning the compatibility of these two. However, VAL has an extended meaning, and in standard Modula-2 can also be used to convert a value of any numeric type to an appropriate value in any other numeric type. Thus,

```
real := VAL (REAL, int);
```

is a way of writing

```
real := FLOAT (int);
```

and

```
lreal := VAL (LONGREAL, real);
```

converts from REAL to LONGREAL, and so on.

As the last two examples illustrate, the CHAR type is also an enumeration of the underlying national characters (often ISO/ASCII,) whose values could be obtained by writing a program to output them on a printer.

VAL and ORD are two more examples of built-in identifiers for standard functions. As with all standard identifiers, they are not reserved words. Rather, all such functions are automatically imported into every module (They are *pervasive identifiers*.) and should be regarded as unavailable for assignment by the programmer. It is syntactically correct, but in very bad taste to write:

```
VAR
  VAL : CARDINAL          (* bad bad bad *)
```

VAL and ORD are also inverse functions, in the sense that if *name* is of type T, and *num* is a CARDINAL, then

```
ORD (VAL (T, num));           yields num, and
VAL (T, ORD (name));          yields name.
```

The built-in type BOOLEAN is also enumerated. It may be thought of as having the definition:

```
TYPE
```

```
BOOLEAN = (FALSE, TRUE);
```

Thus,

```
VAL (BOOLEAN, 0);      yields FALSE,
ORD (TRUE);            yields the value 1,
INC (boolVar);         produces TRUE if boolVar was false.
```

Naturally, if *boolVar* were TRUE, incrementing it would render it undefined, as would decrementing it when it were FALSE. Here is a sample program illustrating some of these ideas. It calculates a person's pay for a week, given the number of hours worked and the hourly wage.

```
MODULE SimplePay;
```

```
(* Written by R.J. Sutcliffe *)
(* to illustrate the use of enumerated types *)
(* using ISO Standard Modula-2 *)
(* last revision 1996 12 03 *)
```

```
FROM STextIO IMPORT
```

```
  WriteString, WriteLn, ReadChar, SkipLine;
```

```
FROM SWholeIO IMPORT
```

```
  WriteCard;
```

```
FROM SRealIO IMPORT
```

```
  ReadReal, WriteFixed;
```

```
TYPE
```

```
  DayName = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

```
VAR
```

```
  day : DayName;
```

```
  num : CARDINAL;
```

```
  wage, hours, totHours : REAL;
```

```
  key : CHAR;
```

```
PROCEDURE WriteDay (day : DayName);
```

```
(* write out a string appropriate for the value of day
```

```
pre: none
```

```
post: a string is written but no line end is written; *)
```

```
BEGIN
```

```
  IF day = Monday
```

```
    THEN
```

```
      WriteString ("Monday");
```

```
    ELSIF day = Tuesday THEN
```

```
      WriteString ("Tuesday");
```

```
    ELSIF day = Wednesday THEN
```

```
      WriteString ("Wednesday");
```

```
    ELSIF day = Thursday THEN
```

```
      WriteString ("Thursday");
```

```
    ELSIF day = Friday THEN
```

```
      WriteString ("Friday");
```

```
    END;
```

```
END WriteDay;
```

```

BEGIN
  WriteString ("This program computes total weekly wages from ");
  WriteLn;
  WriteString ("a wage rate and daily hours worked");
  WriteLn;
  WriteLn;
  WriteString ("What is your hourly wage? ");
  ReadReal (wage);
  SkipLine;
  totHours := 0.0;      (* initialize total hours *)
  WriteLn;
  day := Monday;

  WHILE day <=Friday
    DO
      WriteString ("How many hours did you work ");
      WriteString ("on ");
      WriteDay (day);
      WriteString ("? ==");
      ReadReal (hours);
      SkipLine;
      totHours := totHours + hours;
      WriteLn;
      INC (day);
    END;      (* while *)

  WriteString ("Your total wages for the week are $ ");
  WriteFixed (wage * totHours, 2, 0);
  WriteLn;
  WriteString ("Press a key to conclude ==");
  ReadChar (key);
END SimplePay.

```

Sample Output:

This program computes total weekly wages from
a wage rate and daily hours worked

```

What is your hourly wage? 15.75
How many hours did you work on Monday? ==> 7.5
How many hours did you work on Wednesday? ==> 6.0
How many hours did you work on Friday? ==>

```

As this example illustrates, the ordinal value *Monday* and the string, "*Monday*" are not the same thing. A common student error is to code *WriteString (day)* in an attempt to output the string. The names *Monday*, *Tuesday*, *Wednesday*, *Thursday*, and *Friday* are for the use of the compiler only; the program sees them as abstract values, and not as strings. That is, the transparency of the type *DayName* to the program only goes as far as the listing of the possible values it can take on. The letters *M*, *o*, *n*, *d*, *a*, and *y* used to type the ordinal value *Monday* in the source code are not available as characters in a string "*Monday*" to the finished program; only the abstract value is.

The value *Saturday* is needed to prevent a run time error when the value *Friday* is incremented.

5.2.2 Subranges Of Existing Types

Still another way to create a new data type is to specify it as a range of consecutive values taken from some built-in or previously defined type (called the *host* type). The range is indicated by enclosing it in brackets.

A subrange of an ordinal or enumerated type is a sequence of consecutive values of the host type that is indicated by: [start of range .. end of range] where (start of range) <= (end of range.)

Here are a few examples:

TYPE

```
Capitals = ['A' .. 'Z'];
Peg = ['A' .. 'C'];
Digit = [0 .. 9];
Smallnum = [-5 .. 5];
Mistake = [5 .. 1]; (* compiler error as start "The ordinal number of a value of
a subrange shall be the same as the ordinal number which the value has in the host
type." Thus:
```

```
VAL (Weekday, 1)           produces Monday, and
num := ORD (Tuesday)      produces 2
```

It is also possible (and optional) to include the host type when declaring a range, in the following manner:

TYPE

```
SmallRange = INTEGER [0..99];
```

This overrides the compiler's automatic assignment of this range as a sub-type of the host type **CARDINAL**.

There are potentially three advantages to using a subranges. The first is that the subrange is a different abstraction than the original type. Use of the subrange may therefore make a program clearer. Second, some systems may store the values of a subrange more efficiently than those of the original type. This cannot be counted on, and makes a difference (if at all) only when the number of such variables is large, and memory is scarce. Third, an error is generated if any attempt is made to assign something to a variable that is not in the subrange. This occurs because the incorrect value is actually of a different type than the variable to which the assignment is being made. For instance, with the above declarations, all of the following assignment attempts are incorrect:

```
day := 5;           (* because 5 is not a DayName *)
wDay := Sunday;     (* not in the subrange *)
letter := "p";       (* lowercase not in this type *)
num := -5;           (* no negatives in this range *)
sNum := 10;          (* too large, out of range *)
```

Notice that the details of the declaration will determine whether two variables are expression compatible (the type or at least the host type must *exactly* match) or assignment compatible (i.e. the base types are assignment compatible). Consider the following declarations:

TYPE

```
ARange = CARDINAL [1 .. 10];  
BRange = INTEGER [1 .. 10];  
CRange = ARange;
```

VAR

```
aRangeVar : ARange;  
bRangeVar : BRange;  
cRangeVar : CRange;
```

Now,

```
cRangeVar := aRangeVar;  is legal (types are actually equal)  
aRangeVar := bRangeVar;  is legal (assignment compatible bases)  
cRangeVar := aRangeVar + cRangeVar;  is legal (expression compatible)  
aRangeVar := bRangeVar + aRangeVar;  is illegal (expression incompatible)
```

A variable of type *CRange* can be assigned to or used in the same expression as one of type *ARange* and vice versa. A variable of type *CRange* can be assigned to but cannot be used in the same expression as one of type *BRange* even though the same subrange of the same base type is employed to define both. As far as the compiler is concerned, *CRange* and *ARange* are *expression compatible*, but *BRange* is only *assignment compatible* with either. The program declaration says they are different types, does it not? There must have been some reason for having two separate types, so once they are declared that way, the programmer must use them that way.

It is also possible to directly assign a variable of type *ARange* to a variable of type *CARDINAL* or *INTEGER*, or to a variable in some range from which *ARange* is derived (a *super range*).

Here is some code that is erroneous, but the error cannot be caught by the compiler:

TYPE

```
eighties = [1980..1989];  
nineties = [1990..1999];
```

VAR

```
year1 : twenties;  
year2 : nineties;
```

BEGIN

```
...  
year1 := year2;
```

In this case, the two variables are of assignment compatible types because both are subranges of the same underlying type, so the compiler has nothing to complain about. However, because their respective ranges share no values in common, an actual assignment will always yield an error at run time.

The method described earlier of deriving a new type from a specified host type also applies when the host type is user-defined. One may write:

TYPE

```
ARange = [1..10];  
BRange = ARange [2..5];
```

and the values of variables of type *BRange* would be compatible with those of type *ARange* because *BRange* is specifically declared to be a *subtype* of *ARange*.

5.2.3 Summary of some Modula-2 compatibility issues:

1. Two variables are (expression) compatible if:

- they are of the same type
- the type of one was declared equal to the type of the other
- the type of one is a subrange of the other, or
- both types are subranges of the same type.

2. Two variables are assignment compatible if:

- their types are compatible
- one is INTEGER and the other CARDINAL or a subrange thereof, or
- one is CARDINAL and the other INTEGER or a subrange thereof.

3. Two variables are incompatible otherwise.

In view of these complications, and the fact that more assignments are likely to fail, one might ask why use subranges at all? The answer is that it is better to get a run time range error at the point at which an inappropriate assignment is *first* made, rather than much later in the program. The use of a subrange pinpoints the error to the place where remedial action must be taken. Without it, tracing the logic of faulty output back to the appropriate point might be very tedious indeed.

With both enumerations and subranges defined, it is possible to give the following:

5.2.4 Summary of some Modula-2 types:

1. *Whole number types:*

These are INTEGER, CARDINAL, non-standard long versions of either, if provided, and the type of whole number literals (such as the "5" in thumb := 5.) Items of the latter type may be assigned to either of the other two whole number types, provided they are in an appropriate range.

The name of the underlying type of whole number literals (whether signed or unsigned) is the Z-type, a supertype thought of as including all such whole number literals.

2. *Ordinal types:*

These are the whole number types, enumerations (built-in or user-defined) and subranges. An ordinal type is any type, all of whose values can be put into a one-to-one correspondence with a finite subset of the Z-type. That is, they are the ones that can be counted off with whole numbers.

3. *Scalar types:*

These include all the ordinal types together with the real types (REAL and LONGREAL). Because in practice there are only a finite number of different reals representable on a machine with finite precision, one could conceivably count the items of these two types with ordinal numbers. However, what would be two consecutive reals on one machine would not necessarily be on another (perhaps neither could even be represented exactly). Thus the counting of the real entities by a Modula-2 program would not be predictable. Moreover, in theory, a real type models the real numbers of mathematics, of which there are an infinite number between any two given reals. At the very least, there are almost certain to be more representable reals on a given machine than there are available cardinal numbers to count them. Thus the real types are distinguished from the ordinal (countable) types.

Real and long real literals and constants are said to be of the R-type, a supertype thought of as including all such real numbers.

4. *Number types:*

These include the whole number and real types, (and, as will be seen in a later section, complex number types) but not enumerations such as the type *DayName* used above.

5.2.5 Making Comparisons

All scalar types, including user-defined ones can be compared for equality or inequality. Using the less than and similar operators on them also makes sense because scalar types all have an ordering. Thus the line

```
WHILE day <= Friday
```

in the module *SimplePay* or

```
IF charVar <= "A"
```

both make sense. One may even write something like

```
IF booleanExpression1 <= booleanExpression2
```

which is true unless the left side is TRUE and the right side is FALSE.

[Contents](#)

5.3 Indexed Data Types--Arrays

Several examples and problems thus far have involved the use of two or more related identifiers. These can be distinguished by including a numeric character in the identifier. One can employ x_1 , x_2 , y_1 , y_2 , or num_1 , num_2 , num_3 , or item_1 and item_2 for example.

Some of the mathematical ideas that were being modeled also had numbered data items, even though it was possible to avoid a requirement for using this idea in programs. For instance, the terms of a sequence are numbered a_1 , a_2 , a_3 , ... a_j . This numbering scheme did not find its way into any of the examples because the abstractions of interest--the n^{th} term and n^{th} partial sum--could be computed by a closed form (formula) without needing to keep all the terms as program data items. There are times when this might be a good idea, especially if the data were to be stored in a file for later reference. For example, in the module *SimplePay* in [section 5.2.1](#), it might have been a good idea to have variables with which to refer to the hours worked, one day at a time. The desired Modula-2 entities would model $\text{pay}_{\text{Monday}}$, $\text{pay}_{\text{Tuesday}}$, $\text{pay}_{\text{Wednesday}}$, and so on.

Data that is referred to with a single name along with an ordinal subscript is said to be indexed.

Most computing notations, including Modula-2, have a means to group together and refer to such collections of related items under a single identifier in this manner. The data structure employed has an inherent order imposed by the indices, and the latter are enclosed in brackets, because programming notations are not designed to recognize and attach meaning to subscripts, nor can they be typed on many machines. One writes, say, $\text{item}[1]$, $\text{item}[2]$, $\text{item}[3]$, ... $\text{item}[n]$ to refer to the specific objects in such a data structure.

A Modula-2 array is a collection of objects of the same underlying data type that is indexed by an ordinal type.

One declares an array in a TYPE statement by giving the range of indices that are permitted for the items, and by stating the underlying type of the entities that can be entered into the array. A single array may be declared in a VAR statement, but this is not recommended. Here are some examples to demonstrate the correct syntax:

TYPE

```
SmallArray = ARRAY [0 .. 10] OF INTEGER;  
Data      = ARRAY ['A' .. 'D'] OF REAL;  
Range     = [1 .. 31];  
MonthPayGrid = ARRAY Range OF REAL;
```

VAR

```
smallArray1, smallArray2 : SmallArray;  
realArray : ARRAY ['A' .. 'D'] OF REAL; (* not a good idea *)  
dataArray1, dataArray2 : Data;  
payDays : MonthPayGrid;
```

and here are some assignments and expressions:

Correct:

```
smallArray1 [1] := 0;
payDays [12] := 12.34;
dataArray1 ['A'] := 4.5;
smallArray1 := smallArray2;    (* whole arrays can be assigned *)
```

Incorrect:

```
smallArray1 [11] := 5;          (* out of range, no item #11 allowed*)
dataArray1 ['A'] := realArray;  (* Error--Not of same type.*)
dataArray1 := realArray;        (* Error--Not of same type.*)
payDays [12] := 100;            (* wrong data type entered into array *)
IF smallArray1 = smallArray2    (* cannot be tested for equality. *)
```

NOTES: 1. The range of an array, which is enclosed in brackets in the declaration, gives the minimum and maximum index numbers or *selectors* that are allowed for the array type. A reference outside this range in the program generates an error.

2. All elements in an array must be of the same base type, but this can be any built-in or previously defined type, including another array. The complexity of a data type is up to the programmer trying to model a real-life situation.

3. When a variable is declared as an **ARRAY** without using a previously defined type name, its type is said to be *anonymous*. As the third example above shows, such a variable is not compatible with a named type, even if they are of the same underlying structure. This at least, is the Modula-2 rule; other notations have different rules.

4. The identifier of a range type can be used wherever a range is required. Such a range identifier is not enclosed in brackets, because it already has them.

Notice from the definition that any ordinal type, including an enumerated type, can be used as an array range. Thus,

TYPE

```
HoursArray = ARRAY [Mon .. Sat] OF REAL;
```

VAR

```
hours : HoursArray;
```

BEGIN

```
hours [Mon] := 4.51;
```

is a legal definition and use, provided that *Mon .. Sat* is a correct subrange of some previously defined enumerated type.

It is also worth noting that a function procedure cannot return an anonymous **ARRAY** type (one with no defined type). That is, one *cannot* write:

```
PROCEDURE ProcName(a : CARDINAL) : ARRAY [1 .. 5] OF CARDINAL; (* illegal *)
```

To achieve the desired result, it is necessary to write:

TYPE

```
CardArray = ARRAY [1 .. 5] OF CARDINAL;
```

```
PROCEDURE ProcName (a : CARDINAL) : CardArray;
```

5.3.1 A First Look at String Variables

Some computing notations have a built-in type for string variables. Modula-2 lacks this type (which is why string variables have not previously been used in this text). However, Modula-2 does allow the user to declare an **ARRAY** [0 .. n] **OF** **CHAR**, to assign string literals to such arrays, and to use such arrays with input/output procedures such as *ReadString* and *WriteString*. The minimum selector in the range should be zero, and the maximum selector is then one less than the maximum number of characters in the string. Given the declarations

TYPE

```
String  = ARRAY [0 .. 80] OF CHAR;  
Range  = [1 .. 10];  
SmallString = ARRAY [0 .. 5] OF CHAR;  
Paragraph = ARRAY Range OF String;
```

VAR

```
string1, string2 : String;  
sString : SmallString;  
ch : CHAR;  
para : Paragraph;
```

here are some assignments and expressions:

Correct:

```
string1 := "Hello there";  
string2 := string1;  
sString := "Hello";  
sString := "H"; (* see warning below *)  
sString := ch;  
ReadString (string1); (* allowed for any ARRAY range OF CHAR *)  
Write (string1 [2]); (* individual array items are type CHAR *)  
WriteString (para [3]);  
para [1] := string1;
```

Incorrect:

```
sString := "Hi there";          (* too long *)
```



```
sString := string1; (* not the same type; one is too long for the other *)
```

In ISO Standard Modula-2 an ARRAY [0..0] OF CHAR (string of length one) is compatible with the type CHAR. The following is correct:

TYPE

```
string = ARRAY [0 .. 50] OF CHAR;
```

VAR

```
key : CHAR;  
str : string;
```

BEGIN

```
str := "Hi there fellow, how are you doing?";  
WriteString (str);  
WriteChar (str [1]);  
WriteString (str [1]);  
key := str [5];  
WriteChar (key);  
WriteString (key);
```

WARNING: Even though it is in ISO Modula-2, a single CHAR is not interpreted as a string of length one by some older or non-standard versions of Modula-2, and may not be assignable to a string variable (ARRAY range OF CHAR) in this way in such versions.

String literals and constants are said to be of the S-type, a supertype thought of as including all strings, of whatever length. Much more detail about strings will be provided in chapter 7. The student should consult that material before attempting to manipulate strings in any extensive way. If the only needs are for declaration, assignment of string literals, input and output, the information given in this section will suffice.

[Contents](#)

5.4 The FOR Statement

A number of the examples thus far in the text have employed counting loops of the following general construction:

```
count := start;
WHILE count <= stop
  DO
    statement sequence;
    INC (count);
  END;
```

Such loops are likely to be common when dealing with indexed data, and in particular for arrays. They may be used for counting through the indices of an array, for counting the number of times to perform a computation, or both. So common are such tasks that a special name is given to them:

Any task that is repeated according to a numbered pattern is said to be iterative. The act of such counted repetition, whether of numbered items in a list such as an array, or of steps in a program, such as a loop, is called iteration.

In common with most computing notations, Modula-2 has a special shorthand loop syntax for performing iterations. This is called the FOR loop, and can be described with the following skeleton--a direct replacement for the code in the above WHILE loop:

```
FOR count := start TO stop
  DO
    statement sequence;
  END;
```

The initialization of the loop counter variable to its starting value, and the specification of the stopping point are both handled in the first line of the loop. There is no need for the programmer to write a statement for incrementing the loop counter, as this action is included in the FOR programming abstraction itself. When the END of the FOR loop is reached, the loop counter variable is increased by 1 and the new value checked to see if it is beyond the stopping value. If not, the statement sequence under the control of the loop is executed again. If the counter has gone past the stopping value, the loop exits to the next statement following it in the main program sequence.

NOTES: 1. FOR and TO are reserved words.

2. This code is prettyprinted like the IF .. THEN statement, with the END under the DO.

3. The loop control variable (*count* in this case) must have been previously declared as an ordinal type. It need not have been previously assigned, however, because the FOR statement itself starts with the initial assignment for the sake of starting the loop.

4. Variables or expressions are allowed to control the FOR, but must be of a countable (ordinal) type. That is, CARDINAL, CHAR, and enumerated or subrange types can be used, but REAL is not allowed.

Here are some examples:

Allowed:

```
FOR count := start TO index + 3
  (* start, index are expression compatible, and are assignment compatible with count *)
  DO
    Statement Sequence; (* semicolon optional here as usual*)
  END;
```

```

FOR ch = 'A' TO 'Z' (* ch previously declared of type CHAR *)
  DO
    Write (ch);
    WriteLn;
  END;

```

Not Allowed:

```

FOR realNum := 1.0 TO 5.0 (* uses a real *)
FOR count := 1 to 10.5 (* same problem *)

```

May cause a problem:

In the discussion that follows 32767 is being suggested as MAX(INTEGER). This low limit is unlikely on current machines; however there will always be a value of which the point made here does become relevant.

```

FOR count := 32465 TO 70000 (* outside CARDINAL range on some machines *)
FOR integerTypeCounter := 1 TO 32767

```

The latter will not work in some versions, because after *count* becomes 32767, it would next be 32768, and in many implementations this is past the limit of the INTEGER type. What happens next in such cases depends on the implementation. An overflow like this *ought* to generate an error at run time, but if it does not, 32768 may instead "wrap around" to MIN (INTEGER) and be interpreted as -32768 when the test for exit from the loop is made. Since this is less than 32767, the loop would continue forever. The same logic applies for other implementations of INTEGER with the number 32767 replaced by MAX (INTEGER). The logic of an actual program may require that the loop skip numbers, count in reverse, or both. If using the WHILE loop, these were accommodated by using INC (count, increment), DEC (count), or DEC (count, decrement). To do the same thing in a FOR loop, the increment or decrement is specified in the opening line along with the starting and stopping values with the reserved word BY followed by a whole number amount. Thus,

```

count := start;
WHILE count <= stop
  DO
    statement sequence;
    INC (count, increment);
  END;

```

becomes

```

FOR count := start TO stop BY increment
  DO
    statement sequence;
  END;

```

and likewise,

```

count := start;
WHILE count

```

Notice that if a constant BY step is included, it must be a constant expression of a whole number type.

Correct:

```
FOR day := Friday TO Monday BY -1
DO
```

Incorrect:

```
FOR day := Friday TO Monday BY Thursday (* can't use non-numeric *)
DO
```

```
FOR day := Friday TO Monday BY integerValue (* can't use a variable *)
DO
```

```
FOR count := 1 TO 5 BY 1.5 (* increment must be whole *)
DO
```

Care must be taken when using any loop with a variable of a subrange type. Suppose one had:

```
TYPE
    Digit = [0..9];
VAR
    digitCount : Digit;

BEGIN
    FOR digitCount := 0 TO 9
        DO
            statementSequence;
        END;
```

On the last pass through this loop, an attempt might be made to set *digitCount* to 10, a value that is out of range for this type. Consequently, a run time error could be generated. Some compilers *may* produce correct code (that would not fail) but situations like this may have to be avoided.

5.4.1 The FOR Loop and the WHILE Loop

Like the WHILE loop, the FOR loop has a top-of-loop test. As has been observed above, any FOR loop is equivalent to (and can therefore be replaced by) an equivalent WHILE construction. However, the FOR loop can only count up or down in constant

increments, whereas the WHILE construction places no limitations on re-assigning the loop counter variable. Thus,

```
count := 100;
WHILE count > stop
  DO
    statement sequence1;
    IF certainCondition
      THEN
        count := stop - 1
      END;
    INC (count);
  END;
```

When the test for the stopping condition is made at the top of the loop, no distinction is made between the two ways in which the value of *count* might have reached *stop* and be indicating that it is time to exit the loop code.

However, this kind of chicanery is not permitted with a FOR loop. Indeed, reassigning the loop control variable in the code under its control is an error. (It is also quite confusing, because the first line of the FOR statement ought to give reliable information about the number of times the loop will execute.) The Modula-2 rule is:

The loop control variable of a FOR loop may not be threatened by code within the loop.

Thus, the compiler ought to flag as an error any code such as:

```
FOR count := 100 TO 0 BY -2 (* error *)
  DO
    statement sequence
    count := count DIV 2
  END;
```

or

```
FOR count := 1 TO 10
  DO
    count := 2 * count + 1;    (* do not do this either *)
    (* more statements here *)
  END;
```

The variable can be used, even though it cannot be assigned within the loop, as in

the following simple example:

```
MODULE AlphabetWriter; (* revised 1993 02 26 *)

FROM STextIO IMPORT
    WriteChar, WriteLn;
FROM SWholeIO IMPORT
    WriteCard;

VAR
    ch : CHAR;
CONST
    skip = 2;

BEGIN
    FOR ch := 'A' TO 'Z'
        DO
            WriteChar (ch); (* writes whole uppercase alphabet *)
        END;
    WriteLn;
    FOR ch := 'z' TO 'a' BY -1
        DO
            WriteChar (ch); (* writes lowercase letters backwards *)
        END;
    WriteLn;
    FOR ch := 'A' TO 'Z' BY skip
        DO
            WriteCard (ORD (ch), 3); (* writes every second ordinal *)
        END;
    END AlphabetWriter.
```

This program produced the following when run:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
zyxwvutsrqponmlkjihgfedcba
 65 67 69 71 73 75 77 79 81 83 85 87 89
```

A compiler that conforms to the ISO standard Modula-2 specification will catch and report as errors any threats to a loop control variable during compilation. Classical standard Modula-2 had the same rule, but very few of its compilers enforced the rule. However, there are other, more subtle, ways to *threaten* the value of the loop control variable than simple reassignment and even the experts argue about whether the standard covers all the possibilities. Because checking for all these is rather difficult, some sub-standard compilers do not bother with the issue at all, and permit erroneous code to be written. Programmers ought to take note, however, that if they do get away with such a trick, the meaning of the program will then depend on code that is incorrect. The program will not compile properly under a standard conforming compiler unless changes are made.

Some people claim that since a WHILE loop is a more general form of which the FOR loop can be regarded as a special case, there is no need for the latter. As can be seen, there is something to be said for this view. A few teachers go further, and forbid their students to use the FOR construction altogether. This text is sympathetic to that view to a small extent, in that it has postponed the use of the FOR loop to a point at which it is hoped that the student understands the (more general) WHILE construction.

However, the two kinds of loops are not interchangeable, for some WHILE loops cannot be replaced by a FOR loop. Thus, while one could dispense with FOR loops, the two are actually different programming abstractions. The FOR loop has been specialized for incremental counting (iterations), and ought to be used for that purpose, leaving the WHILE and REPEAT constructions for loops involving logical tests, non-incremental counting, and situations where the loop control variable is REAL.

There is yet another difference between the Modula-2 FOR loop and WHILE loop, and indeed between the Modula-2 FOR loop and FOR loops in other computing notations. It is expressed in the following rule:

At the conclusion of a Modula-2 FOR loop, the value of the loop control variable is undefined.

This does not mean that the loop control variable cannot be re-used once the loop has exited. It simply means that it cannot be relied upon to have any particular value, and so it must be re-initialized before being used for anything else.

Correct:

```
FOR count := 1 to 10
  DO
    statement sequence1;
  END; (* first for *)
FOR count := 1 TO 100
  DO
    statement sequence2;
  END; (* second for *)
count := 2;
statement sequence 3; (* one that depends on count being 2 *)
```

Incorrect:

```
FOR count := 1 to 10
  DO
    statement sequence1;
```

```

    END; (* first for *)
IF count = 11 THEN (* count is actually undefined here *)
    DO
        statement sequence3;
    END

FOR count := 1 TO 100
    DO
        statement sequence2;
    END; (* second for *)
count := count + 2; (* count is undefined here *)
statement sequence 3;

```

The two statement sequences in the second group are not *syntactically* incorrect; they will compile without any error being reported by the compiler. However, they are *logically* incorrect, in that they apparently rely on the loop control variable being set to some particular value after the loop is exited, and Modula-2 says nothing about what such a value will be, so different implementations might produce different results. A program whose correctness depends on the way a particular compiler happens to treat such a situation is incorrect. Even this assumes that a given implementation treats this problem consistently, and that is not required either. Very good Modula-2 compilers will even generate a message warning the programmer of such ill-advised usage.

5.4.2 The FOR Loop in Use

To further illustrate some of these ideas, consider the following:

Problem:

A class set of marks is to be entered in the form of a student's name on one line, followed by a line of real percentages separated by spaces. At the beginning of the data entry are two cardinals. The first is the number of student records to be entered, and the second is the number of marks per student. Write a program to read these records one at a time, print the student's name, grades, and average mark, followed by the class average on the composite marks.

Sample Input:

```

3
4
Janet
87.5 92.8 89.0 90.0
Fred
45.8 81.9 78.0 88.5

```


Laurie
0.0 15.8 32.5 53.5

Code:

```
MODULE ClassMarks;
(* Written by R.J. Sutcliffe *)
(* to illustrate the use of the for loop *)
(* using ISO standard Modula-2 *)
(* last revision 1991 02 27 *)

FROM STextIO IMPORT
  WriteLn, WriteString, ReadString, SkipLine, ReadChar, WriteChar;
FROM SWholeIO IMPORT
  ReadCard, WriteCard;
FROM SRealIO IMPORT
  ReadReal, WriteFixed;

TYPE
  NameString = ARRAY [0..10] OF CHAR;

VAR
  theMark, studSum, studMean, classSum, classMean : REAL;
  numMarks, numStudents, markCount, classCount : CARDINAL;
  name : NameString;
  key : CHAR;

BEGIN
  (* initialize variable for sum of marks *)
  classSum := 0.0;
  (* write information *)
  WriteString ("This program opens a set of class records");
  WriteLn;
  WriteString ("and computes student and class averages. ");
  WriteLn;
  WriteLn;
  (* find out how many students *)
  WriteString ("How many students are in the group? ");
  ReadCard (numStudents);
  SkipLine;
  WriteLn;
  (* find out how many marks *)
  WriteString ("How many marks does each have?");
  ReadCard (numMarks);
  SkipLine;
  WriteLn;
  (* get class info one student record at a time *)
  FOR classCount := 1 TO numStudents
    DO
      WriteString ("Student Name, Please ==");
```

```

ReadString (name);
SkipLine;
WriteLn;
studSum := 0.0;
WriteString ("Please enter the");
WriteCard (numMarks ,0);
WriteString (" marks for ");
WriteString (name);
WriteString (" , separated by spaces on a single line");
WriteLn;
WriteString ("==");
FOR markCount := 1 TO numMarks
    DO (* all marks for each student *)
        ReadReal (theMark); (* get a number from the list *)
        studSum := studSum + theMark; (* reset the sum *)
    END; (* for markCount *)
SkipLine;
studMean := studSum / FLOAT (numMarks);
WriteString ("Average = ");
WriteFixed (studMean, 2, 6);
WriteLn; (* done one student *)

    classSum := classSum + studMean;
END; (* for classCount, done whole class when exit here *)

classMean := classSum / FLOAT (numStudents);

WriteString ("The average of the ");
WriteCard (numMarks, 0);
WriteString (" marks for the class of ");
WriteCard (numStudents, 0);
WriteString (" students was ");
WriteFixed (classMean, 2, 0);
WriteString (" percent.");
WriteLn;
WriteString ("Press a key to continue ==");
ReadChar (key);

END ClassMarks.

```

The program was run with the sample data above with the following results:

This program opens a set of class records
and computes student and class averages.

How many students are in the group? **3**

How many marks does each have?**4**

Student Name, Please ==> **87.5 92.8 89.0 90.0**

Average = 89.82

Student Name, Please ==> **45.8 81.9 78.0 88.5**

Average = 73.55

```
Student Name, Please ==> 0.0 15.8 32.5 53.5
Average = 25.45
The average of the 4 marks for the class of 3 students was 62.94 percent.
Press a key to continue =="*)
    END;
```

will print a line of stars on the screen. The loops that follow place on the screen ten lines consisting of blanks, except for the first and last characters, which are stars.

```
FOR countA := 1 TO 10 (* ten lines in total *)
DO
    Write ("*"); (* start with star *)

    FOR countB := 2 TO 79 (* then the blanks *)
    DO
        Write (" ")
    END; (* for countB *)

    Write ("*") (* end with star *)

END; (* for countA *)
```

Following this with another copy of the first loop would complete an 80 column by 10 row box surrounded with stars.

```
*****
*
*
*
*
*
*
*
*
*
*
*
*****
```

5.5 Manipulating Arrays

It is when it is used to manipulate arrays that the FOR loop comes into its own. Consider the oft-encountered problem of the initialization of variables. An array of numbers often must have all its elements set initially to a particular value (frequently zero). To illustrate, suppose a program had the declarations:

```
CONST
    length = 50;
TYPE
    MediumArray = ARRAY [1 .. length] OF INTEGER;
```

The desired procedure could be written:

```
PROCEDURE Init (VAR theArray : MediumArray);
VAR
    count : CARDINAL;
BEGIN
    FOR count := 1 TO length
    DO
        theArray [count] := 0;
    END;
END Init;
```

Likewise, if at some point the values in the array were all to be added, one could write:

```
PROCEDURE Add (VAR theArray : MediumArray) : INTEGER;
VAR
    count : CARDINAL;
    sum : INTEGER;
BEGIN
    sum := 0;
    FOR count := 1 TO length
    DO
        sum := sum + theArray [count];
    END;
    RETURN sum;
END Add;
```

NOTE: The details of using arrays as parameters are discussed in [Section 5.6](#).

Problem:

Write a program module that counts the occurrences of the letters in text obtained from input from the keyboard. Have it also count the number of words and calculate an average to the nearest whole number. The program should print a chart showing the frequency of each printable character as well as the total number of characters, and the average number per word.

Discussion:

A complete refinement is not provided, but some of the recently introduced ideas are used. Notice that the occurrences of a letter are kept track of by incrementing the number in an array indexed by the character itself. Thus, the array element *lets['e']* contains the number of times the letter e has been encountered. The solution makes use of four facts about the ISO/ASCII character sequence that may not be true of other character sets:

1. The sequence is numbered from 0 to 128.
2. Characters 0 to 31 and 127 are non-printing control characters.
3. Character number 32 is the space.

It also uses two values for *ReadResults**endOfLine*, and *endOfInput* the latter not previously employed

Algorithms:

The end of a word is handled as follows:

```
If one or more consecutive spaces or end of input then
    increment word counter
```

This is adequate, but notice that because punctuation marks are counted along with all the other printed characters, they are included in the average word length. The student is invited to improve on this program as an exercise.

```
MODULE LetterCounter;
```

```
(* Written by R.J. Sutcliffe *)
(* to illustrate the use of the for loop *)
(* using ISO Modula-2 *)
(* last revision 1991 02 27 *)
```

```
FROM STextIO IMPORT
    ReadString, ReadChar, WriteChar, WriteString, WriteLn, SkipLine;
FROM SWholeIO IMPORT
    WriteCard;
FROM SIOResult IMPORT
    ReadResults, ReadResult;
```

```
CONST
    space = CHR (32);
    cr = CHR (13);
    period = ".";
    min = 33; (* Set limits of printable ASCII characters *)
    max = 126;
    maxOnLine = 5;
```

```
TYPE
    LetArray = ARRAY CHAR OF CARDINAL;
```

```
VAR
    letterCount, wordCount, numOnLine, avPerWord : CARDINAL;
    lets : LetArray;      (* Examples:  lets ['A'], lets [",","] *)
    ch, last : CHAR;
    lastResult : ReadResults;
    userDone : BOOLEAN;
```

```

BEGIN
  FOR ch := CHR (min) TO CHR (max) (* initialize totals to zero *)
    DO
      lets [ch] := 0;
    END;    (* for *)
  userDone := FALSE;

  WriteString ("Please type in text you want analyzed.");
  WriteString (" End with period at start of line.");
  WriteLn;

  wordCount := 0;
  letterCount := 0;
  last := space; (* now leading space won't be seen as words *)

  REPEAT    (* main loop to read text by characters *)
    ReadChar (ch);    (* reads whatever is next *)
    lastResult := ReadResult ();
    IF lastResult = endOfLine (* translate end of line state *)
      THEN (* into 'carriage return character read' *)
        ch := cr;
        SkipLine;
      END;
    IF (lastResult = allRight) AND (ch = "."); (* leave some space; make columns *)
      INC (numOnLine );
      IF numOnLine MOD maxOnLine = 0
        THEN
          WriteLn;
        END;
    END;    (* for *)
  WriteLn;
  WriteLn;
  WriteString ("# of words = ");
  WriteCard (wordCount, 0);
  WriteLn;
  WriteString ("# of letters = ");
  WriteCard (letterCount, 0);
  WriteLn;
  IF wordCount # 0
    THEN
      avPerWord := TRUNC ((FLOAT (letterCount) / FLOAT (wordCount)) + 0.5);
      WriteString ("# of letters/word (nearest whole number) = ");
      WriteCard (avPerWord, 0);
      WriteLn;
    END;

  WriteString ("Press a key to end ==");
  ReadChar (ch);
END LetterCounter.

```

When this program was run, the text file given it to analyze was its own source code. (This was done by using a macro program to type out the contents of the file when the prompt appeared asking for input.) Here are the results it produced:

!	0	"	18	#	8	\$	0	%	0
&	0	'	5	(62)	62	*	42
+	1	,	17	-	1	.	0	/	2
0	11	1	4	2	5	3	4	4	0
5	3	6	1	7	1	8	0	9	2
:	19	;	63	<	0	=	33	>	2
?	0	@	0	A	23	B	2	C	40
D	22	E	25	F	15	G	1	H	14
I	24	J	1	K	0	L	33	M	9
N	29	O	34	P	10	Q	0	R	46
S	23	T	20	U	4	V	1	W	29
X	0	Y	2	Z	0	[5	\	0
]	5	^	0	_	0	`	0	a	90
b	5	c	45	d	42	e	174	f	16
g	13	h	35	i	81	j	1	k	5
l	65	m	21	n	94	o	76	p	21
q	0	r	125	s	79	t	156	u	47
v	7	w	19	x	11	y	8	z	3
{	0		0	}	0	~	0		

of words = 515
 # of letters = 2022
 # of letters/word (nearest whole number) = 4

[Contents](#)

5.6 Arrays as Parameters

As indicated by two of the examples in the last section, entire arrays (or their individual elements, as appropriate) may be passed to procedures as parameters. As in all other such cases, an actual value parameter passed must be assignment compatible with the formal value parameter found in the procedure declaration. If using variable parameters, the types of the actual and formal parameters must be the same.

These limitations are not onerous in most cases, but by themselves they would prevent the writing of general-purpose code in some situations. For instance, suppose one had two arrays of the same type and wished to add them element by element and place the sum in a third array. That is, given the declarations:

```
CONST
    size = 3;
TYPE
    Vector = ARRAY [0 .. size - 1] OF CARDINAL;
VAR
    vectA, vectB, ansVector : Vector;
```

and the assignments

```
vectA [0] = 5; vectA [1] = 3; vectA [2] = 7;
vectB [0] = 4; vectB [1] = 8; vectB [2] = 10;
```

and, one wished to end up with:

ansVector [0] <-- 9, ansVector [1] <-- 11, and ansVector [2] <-- 17

one could write a routine procedure like:

```
PROCEDURE AddArrays (firstVector, secondVector : Vector; VAR resultVector : Vector);
VAR
    count : CARDINAL;
BEGIN
    FOR count := 0 TO size - 1
        DO
            resultVector [count] := firstVector [count] + secondVector [count];
        END;      (* for *)
    END AddArrays;
```

This procedure would be called as usual by including the line *AddArrays (vectA, vectB, ansVector)* in the main program. However, suppose there were two vector types in the main program, one of length three, and one of length two.

```
TYPE
    Vector2 = ARRAY [0 .. 1] OF CARDINAL;
    Vector3 = ARRAY [0 .. 2] OF CARDINAL;
VAR
    vect2A, vect2B, ansVector2 : Vector2;
    vect3A, vect3B, ansVector3 : Vector3;
```


In order to add the vectors of type *Vector2*, one would have to have a different procedure than the one above for adding vectors of type *Vector3*, because the vectors to be added must be passed to the formal parameter of a compatible type, and a vector of two components is clearly not compatible with one of three components. There is nothing to assign to the third component.

However, there is a way to make the above code more generic or multi-purpose in Modula-2 by using an array parameter that does not specify the length of the array formally, but calculates the number of components being used when an actual assignment to the parameter is made. For instance, the above procedure would be re-written as:

```
PROCEDURE AddArrays2 (vect1, vect2 : ARRAY OF CARDINAL;  
                      VAR ansVector : ARRAY OF CARDINAL) ;  
VAR  
    count : CARDINAL ;  
  
BEGIN  
    FOR count := 0 TO HIGH (vect1)  
        DO  
            ansVector [count] := vect1 [count] + vect2 [count] ;  
        END ;  
END AddArrays2 ;
```

and this could be used to add vectors of either type discussed here, or of any similar type defined as ARRAY range OF CARDINAL.

NOTES: 1. The anonymous type ARRAY OF CARDINAL replaces ARRAY range OF CARDINAL or a named type in the formal declaration.

2. When the procedure is called, any actual parameter that is an ARRAY range of CARDINAL is assignment compatible with the formal parameter.

3. Within the operating procedure, the actual range of the parameters is always [0 .. HIGH (parameter)].

An open array is an array of some base type without any range specified. It is written as ARRAY OF type and its implied definition is: ARRAY [0 .. HIGH (name)] OF someType.

HIGH is a standard function procedure that computes the highest cardinal index employed when the actual parameter was assigned upon the procedure being invoked.

All three arrays passed to this particular procedure ought to be of the same base type, or else the use of HIGH applied only to one of them would be rather misleading. This stipulation should be added to the code as a precondition in the form of a comment. If there were any possibility that this might not be the case, some action would have to be undertaken. There are two possibilities:

```
PROCEDURE AddArrays3 (vect1, vect2 : ARRAY OF CARDINAL;  
                      VAR ansVector : ARRAY OF CARDINAL) ;  
(* adds the first two arrays and places the result in the third component by  
component  
Pre: none  
Post: If all three vectors are the same size then the third is the sum of the first  
two.  
All components of any of the vectors that is longer than any of the others are  
ignored. An overflow error may be generated on an individual component at run time  
if the values being added are too large. *)  
  
VAR
```

```
count, size : CARDINAL;
```

```
BEGIN
```

```
  (* pick the minimum of the three sizes *)
```

```
  size := HIGH (vect1);
```

```
  IF HIGH (vect2) < size
```

```
    THEN
```

```
      size := HIGH (vect2);
```

```
    END;
```

```
  IF HIGH (ansVector) < size
```

```
    THEN
```

```
      size := HIGH (ansVector);
```

```
    END;
```

```
  FOR count := 0 TO size
```

```
    DO
```

```
      ansVector [count] := vect1 [count] + vect2 [count];
```

```
    END;
```

```
END AddArrays3;
```

This approach detects an error and deals with it by ignoring it and computing as much of a sum as is meaningful under the circumstances. A second approach detects and reports the error, and refuses to do the sum, leaving it to the calling code to decide what to do about the error:

```
PROCEDURE AddArrays4 (vect1, vect2 : ARRAY OF CARDINAL;
```

```
                      VAR ansVector : ARRAY OF CARDINAL;
```

```
                      VAR addOk : BOOLEAN);
```

```
(* adds the first two arrays and places the result in the third component by  
component
```

```
Pre: none
```

```
Post: If all three vectors are the same size then the third is the sum of the first  
two and addOk returns true.
```

```
If any of the vectors is longer than any of the others, addOk returns false and  
ansVector is undefined. An overflow error may be generated on an individual  
component at run time if the values being added are too large. *)
```

```
VAR
```

```
count, size : CARDINAL;
```

```
BEGIN
```

```
size := HIGH (vect1);
```

```
IF (size # HIGH (vect2)) OR (size # HIGH (ansVector))
```

```
  THEN
```

```
    addOk := FALSE; (* bad data; don't do addition *)
```

```
  ELSE
```

```
    FOR count := 0 TO size
```

```
      DO
```

```
        ansVector [count] := vect1 [count] + vect2 [count];
```

```
      END;
```

```
    addOk := TRUE;
```

```
  END;
```

```
END AddArrays4;
```

Here are two first procedures from section 5.5 modified to use open arrays:

```
PROCEDURE Init2 (VAR theArray : ARRAY OF INTEGER);  
(* sets all elements of the array passed to zero *)  
  
VAR  
    count : CARDINAL;  
  
BEGIN  
    FOR count := 0 TO HIGH (theArray)  
        DO  
            theArray [count] :=0;  
        END;  
END Init;  
  
PROCEDURE Add (VAR theArray : ARRAY OF INTEGER) : INTEGER;  
  
VAR  
    count: CARDINAL;  
    sum : INTEGER;  
  
BEGIN  
    sum := 0;  
    FOR count := 0 TO HIGH (theArray)  
        DO  
            sum := sum + theArray [count];  
        END;  
    RETURN sum;  
END Add;
```

NOTES: 1. If an **ARRAY OF CHAR** is passed an empty literal string, **HIGH** would return zero.

2. Within the procedure, no assignment can be made to the entire open array. Its type is anonymous, so such entities are incompatible with anything else when taken as a whole. One can assign to it only element by element. So, *ansVector* := *vect1* would be illegal if *ansVector* were an open array.

HIGH always returns a cardinal value. It is quite proper to assign any actual parameter array of the correct type, however indexed, to an open array. However, in the procedure, the indexing will be by cardinal values starting at zero, and not by the values of the formal index parameter type. Thus, given the declarations

```
TYPE  
    Days = (Mon, Tue, Wed, Thu, Fri);  
    PayList = ARRAY [Mon .. Fri] OF REAL;  
    OddVector = ARRAY [-3 .. 4] OF REAL;  
  
VAR  
    pay : PayList;  
    theVector : OddVector;  
  
PROCEDURE FindMax (theReals : ARRAY OF REAL) : REAL;  
(* appropriate code here *)
```

the procedure *FindMax* can be invoked in the program with either *pay* or *theVector* as actual parameter. Inside the procedure, the formal parameter is indexed starting at zero in both cases. *HIGH* (*theReals*) would therefore produce 4 in the former case, and 7 in the latter.

In the examples given thus far, all arrays being returned by procedures have been handled as variable (or in-out) parameters. Modula-2 also permits function procedures to return an array.

```
TYPE
  RVector = ARRAY [0 .. 1] OF REAL;

PROCEDURE AddRVector (v1, v2 : RVector) : RVector;

VAR
  z : RVector;

BEGIN
  z [0] := v1 [0] + v2 [0];
  z [1] := v1 [1] + v2 [1];
  RETURN z;
END AddVector;
```

NOTE: The type *RVector* contains reals, and is therefore not compatible with an open array containing type cardinal as used in previous examples.

It is not permitted to use the syntax of an open array anywhere except in the parameter list of a procedure. Thus the following are both

Incorrect:

```
TYPE
  openArray = ARRAY OF CARDINAL;

PROCEDURE AddArray (v1, v2 : ARRAY OF REAL) : ARRAY OF REAL;
```

WARNING: In most cases when an array is passed as a parameter, and the array itself is not being changed, one would use a value parameter (passing data in only). However, arrays can consume a large amount of memory. If several arrays are passed as value parameters, the space each takes is doubled while the procedure is active. In some such cases, it might be better to use variable parameters even though the data is not in/out. This practice also saves the time taken to make the extra copies when the procedure is entered, and this time may be substantial if the array is large.

[Contents](#)

5.7 Multi-Dimensional Arrays

It was noted in section 5.3 that any type of data may be indexed by being placed in a Modula-2 array, including another array. Thus far, the only instance of doing the latter was in an array of strings.

TYPE

```
String  = ARRAY [0 .. 80] OF CHAR;  
Paragraph = ARRAY [1 .. 10] OF String;
```

VAR

```
para : Paragraph;
```

No mention was made at the time of how to refer to an individual character within the array *para*, but it seems reasonable to suppose that

```
ch := para [1] [2];
```

would be a legitimate assignment, and so it is. The type *Paragraph* can be thought of at one level of abstraction in the way it is defined, that is, as an *array of strings*. Looking a little more closely, one might also want to think of it as an *array of array of characters*. Likewise, if one desired, it would be quite proper to define

TYPE

```
DataBlock = ARRAY [1 .. 5] OF ARRAY [1 .. 4] OF INTEGER;
```

VAR

```
theData : DataBlock;
```

and to write assignments such as:

```
theData [2] [4] := -200;
```

to assign a value to a single item in one of the five arrays of four integers,
or

```
theData [2] := theData [4];
```

to assign one of the four-integer arrays to another in its entirety.

It is convenient to think of such data as if it were arranged in five rows each of four *columns*.

1	5	-6	9
2	0	4	-2
1	0	2	-6
-8	9	3	1
4	1	-3	5

In terminology previously used for several examples, the array consists of four *column vectors*, each of length 5. This closely parallels the mathematical construction known as a matrix. Here are some samples:

$$A = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & -5 \\ 2 & 9 & 7 \\ 1 & 6 & 8 \end{bmatrix} \quad B = \begin{bmatrix} 1 & -4 \\ 0 & 5 \end{bmatrix}$$

A two-dimensional matrix is a collection of data arranged in a rectangular fashion. An individual element is referenced by its row and its column as $A_{i,j}$ and $B_{i,j}$. Both rows and columns are numbered starting from one. If the maximum indices for the row and column are m and n , it is called an m by n matrix (written $m \times n$.)

NOTE: This text will follow normal mathematical conventions and capitalize the first letter of a matrix identifier. This is not a rule; but a question of taste. The identifier of a two-dimensional array in a Modula-2 program will be capitalized only if it represents a matrix, and not otherwise.

By convention, the row index is always given first. In the above examples, A is a 4 x 3 matrix, and B is a 2 x 2 matrix. Likewise, $A_{3,2}$ is the number 9, and $A_{2,3}$ is the number -5. Sample declarations for A and B are given below.

TYPE

```
Matrix43 = ARRAY [1 .. 4], [1 .. 3] OF INTEGER;
Range = [1 .. 2];
Matrix22 = ARRAY Range, Range OF INTEGER;
```

VAR

```
A : Matrix43;
B : Matrix22;
```

Modula-2 requires separate brackets for each range, unlike some computing notations, which allow all the ranges to be listed inside a single pair of brackets. As with the one-dimensional or ordinary array, no brackets are needed if the name of a range type is given in place of a literal range.

When referring to the individual two-dimensional array elements in a Modula-2 program, both index references may be placed inside a single pair of brackets. Thus, one might write:

```
A [1, 3] := 5;
B [2, 2] := A [3, 1];
```

rather than

```
A [1] [3] := 5;
B [2] [2] := A [3] [1];
```

reflecting the fact that the data being abstracted is a single two-dimensional array with a unified indexing scheme, rather than an array of arrays, as the one with which this discussion began.

NOTE: Modula-2 does not actually distinguish between:

```
ARRAY range1, range2 OF type
```

and
ARRAY range1 **OF** **ARRAY** range2 **OF** type.

One or the other may be a better description of the actual structure of the data being abstracted, though the former is more compact and is easier to write. However, it makes no difference to the compiler or to the eventual program which of the two one uses, for the interpretation of both declarations produces exactly the same code. Having constructed such a data type and assuming *rowCount* and *colCount* to be of type **CARDINAL** and to have values in the correct range, loops can be constructed with references to individual elements in the form:

```
int := B [rowCount, colCount]
```

or

```
int := A [rowCount, colCount]
```

Also, notice that the assignment

```
A [1, 4] := B [2, 1]
```

is perfectly legal, because both individual elements are of the same type, whereas the assignment

```
B := A;
```

is illegal because the arrays themselves are not of the same type, and the assignment

```
B [1, 3] := A [1, 3];
```

is also illegal because the indexing of *B* in this statement exceeds the correct bounds for the type. The first of these two errors is detected by the compiler, whereas the latter is normally found at run time when array index values are checked against the maximum and minimum allowable bounds. If it were done literally this way with *1* and *3*, the compiler would catch the error. Normally, variables are used, however, and their values are not known until run time.

5.7.1 Arrays of More than Two Dimensions

It is also possible to have arrays of matrices, that is, three dimensional arrays. These are harder to write down on paper, and four, five, or six dimensional ones are even more difficult to visualize. A six dimensional array of integers that is, say, three by three by three by three by three by three would have 729 elements. If each dimension were five instead of three, 15625 storage locations would be required for a single one of these entities. It is possible that insufficient room will be available for such variables in some machines. The compiler would allow the program to be compiled with such a dragon in it, but it might not be able to execute.

If the programmer has a sufficiently large memory store, it may not be necessary to be concerned about the size and number of large arrays. Some language implementations have extensive library facilities for array manipulation, and simply assume that the large quantities of memory sometimes necessary will in fact be available. However, there is always some limit on these things; one cannot consume memory space indefinitely.

Here's one (slightly unusual) way to declare that three by three by etc., along with one reference to it that does not

use the same abstraction as the declaration, but is still correct:

TYPE

```
Range = [0 ..2];  
MatrixA = ARRAY Range, Range OF INTEGER;  
BigStructure = ARRAY Range, Range, Range OF MatrixA;
```

VAR

```
theArray : BigStructure;
```

BEGIN

```
theArray [1, 1, 2, 0, 1, 2] := -4
```

5.7.2 Multidimensional Open Array Parameters

It was earlier noted that one can write more generic code using open array parameters that can accept a one-dimensional array of any length as an actual parameter. Likewise, a multidimensional actual array can be assigned to a multidimensional open array formal parameter provided the number of dimensions matches.

To use, say, a three-dimensional open array we need only write something like:

```
PROCEDURE MatrixOperation (VAR m1, m2 : ARRAY OF ARRAY OF ARRAY OF INTEGER);
```

Within the procedure, *HIGH (m1)* will return the highest index assigned on the *first* dimension. To find the highest index on one of the other dimensions, we observe that all the *m1[i]* are themselves open arrays, so that *HIGH (m1[0])* in particular will return the highest index assigned on the second dimension. Likewise, *HIGH (m1 [0, 0])* will return the highest index assigned on the third dimension.

To illustrate further, suppose one has

TYPE

```
DayRange = (Mon, Tue, Wed);  
LetIndices = ['A' .. 'D'];  
OddMatrix = ARRAY DayRange, LetIndices, [1 .. 5] OF INTEGER;
```

VAR

```
day : DayRange;  
let : LetIndices;  
num: CARDINAL;  
odd1, odd2 : OddMatrix;
```

and the PROCEDURE MatrixOperation described above was invoked by

```
MatrixOperation (odd1, odd2);
```

then, within the procedure the following are

Correct: (but only inside a procedure where m1 is an open array)


```
num := HIGH (m1);           would assign num the value 2
num := HIGH (m1 [0]);       would assign num the value 3 and,
num := HIGH (m1 [0, 0]);    would assign num the value 4
```

Incorrect:

```
day := HIGH (m1);
let  := HIGH (m1 [Mon]);
```

NOTES: 1. This also illustrates that the indexing *inside* the procedure on the formal open array parameter always starts at the zeroth position, regardless of the indexing on the actual parameter type. (1 .. 5 is mapped to 0 .. 4)

2. Older or non-standard versions of Modula-2 may not have multidimensional arrays.

3. One could use other than the zeroth element, so HIGH (odd1, [2, 3]) would also do, but in most cases, the zeroth element will be the array of choice upon which to operate with HIGH.

[Contents](#)

5.8 Manipulating Multi-Dimensional Arrays

This is done in much the same way as single-dimensional arrays, using FOR loops. Here, one makes use of the fact that repetition structures such as FOR loops can be nested one inside the other in a Modula-2 program. Suppose one had the declaration:

```
TYPE
    Matrix = ARRAY [0 .. rowmax], [0 .. colmax] OF CARDINAL;
VAR
    A : Matrix;
```

and wished to initialize all the elements to some value, *initNum*. Two FOR loops, one nested inside the other are written in the following manner:

```
FOR rowCount := 0 TO rowmax
DO
    FOR columnCount := 0 TO colmax
    DO
        A [rowCount, columnCount] := initNum;
    END;          (* for column *)
END;            (* for row *)
```

The outer of the two loops (*FOR rowCount*) steps through the rows, one at a time. For each of the rows, the inner (*FOR columnCount*) loop steps through the columns one at a time. In this fashion, each element is initialized in a systematic manner. This routine can be made generic as a procedure by using a two-dimensional open array as follows:

```
PROCEDURE InitMatrix (VAR theMatrix : ARRAY OF ARRAY OF CARDINAL; initNum :
CARDINAL);

BEGIN
    FOR rowCount := 0 TO HIGH (theMatrix)
    DO
        FOR columnCount := 0 TO HIGH (theMatrix [0])
        DO
            theMatrix [rowCount, columnCount] := initNum;
        END;          (* for column *)
    END;            (* for row *)
END InitMatrix;
```

Multidimensional arrays, though not open ones, are used in the following

Problem:

Design a program that employs a data structure that will record the number of male and female freshmen, sophomores, juniors and seniors enrolled in each of the Sciences, Humanities, Social Sciences, Education, Physical Education,

Business, and Aviation divisions at a typical small university. Write the declarations, the code to fill the data structure, and the code to print it out in tabular form. Print also the breakdown of the total number of students by sex, year, and division.

Restatement:

Given: data in the 56 distinct categories
Required to do: prepare summaries by major categories
Desired output: a complete listing of the data with summaries

Discussion of Data Structure:

At some stage during the refinement of the problem (close to the beginning), the programmer must make a decision on the kind of data structure to be used. Since this is a section on multi-dimensional arrays, it takes little imagination to arrive at a suitable way of doing it in this case.

Pseudocode:

```
Obtaining Data from user:
For each class
  for each division
    ask the user for number of men and women
    enter this into the data structure

Generating summaries:
For each class
  sum the men and women in all divisions
For each division
  sum the number of men and women in all classes
For each sex
  sum the number from all classes and divisions

Data output:
For each class
  print an informative heading
  for each division
    print an informative heading
    print the number of men
    print the number of women
```

Data Table:

Enumerated types:

- names of classes, divisions and sexes

Array Types:

- an array of cardinals indexed by class, division, and sex names to hold the raw data

- an array of cardinals indexed by class names to hold class totals
- an array of cardinals indexed by division names to hold division totals
- an array of cardinals indexed by sex names to hold sex totals
- a string type to hold names of all the categories

Main variables:

- a main data array
- an array for each of the sets of totals
- a counter variable for loops on each of the three index types
- an array indexed by class names to hold corresponding strings
- an array indexed by division names to hold corresponding strings
- an array indexed by sex names to hold corresponding strings

Imports required:

From STextIO:

- WriteString, WriteLn, ReadString

From SWholeIO:

- ReadCard, WriteCard

From SIOResult:

- ReadResults, ReadResult

Organization:

No attempt has been made to abstract separate tasks from the whole into procedures, as all the work is closely tied to a single array variable that holds all the program data.

Code:

```
MODULE EnrollData;

FROM STextIO IMPORT
    WriteString, WriteLn, ReadString;
FROM SWholeIO IMPORT
    ReadCard, WriteCard;
FROM SIOResult IMPORT
    ReadResults, ReadResult;

TYPE
    ClassName = (Freshman, Sophomore, Junior, Senior);
    DivName   = (Science, Humanities, SocialScience, Education,
                 PhysEd, Business, Aviation);
    SexName   = (male, female);
    ClassTot  = ARRAY ClassName OF CARDINAL;
```

```

DivTot    = ARRAY DivName OF CARDINAL;
SexTot    = ARRAY SexName OF CARDINAL;
MainData  = ARRAY ClassName, DivName, SexName OF CARDINAL;
String    = ARRAY [0 .. 18] OF CHAR;

```

```

VAR
  (* arrays *)
  classTotals : ClassTot;
  divTotals   : DivTot;
  sexTotals   : SexTot;
  mainDataArray : MainData;
  (* counters *)
  classCount : ClassName;
  divCount   : DivName;
  sexCount   : SexName;
  (* strings for program use *)
  classNames : ARRAY ClassName OF String;
  divNames   : ARRAY DivName OF String;
  sexNames   : ARRAY SexName OF String;
  (* could have done types for last three too *)

```

```

BEGIN  (* main program *)
  (* First initialize all strings. *)
  classNames [Freshman] := "freshman ";
  classNames [Sophomore] := "sophomore ";
  classNames [Junior] := "junior ";
  classNames [Senior] := "senior ";
  divNames [Science] := "Science ";
  divNames [Humanities] := "Humanities ";
  divNames [SocialScience] := "Soc Science";
  divNames [Education] := "Education ";
  divNames [PhysEd] := "PhysEd ";
  divNames [Business] := "Business ";
  divNames [Aviation] := "Aviation ";
  sexNames [male] := "males ";
  sexNames [female] := "females ";

  (* Now initialize the main array *)

  FOR classCount := Freshman TO Senior
    DO
      FOR divCount := Science TO Aviation
        DO
          FOR sexCount := male TO female
            DO
              mainDataArray [classCount, divCount, sexCount] := 0;
            END;      (* for sexCount *)
          END;      (* for divCount *)
        END;      (* for classCount *)
      END;
    END;

  (* Now write the main loop to gather this data. *)

```

```

FOR classCount := Freshman TO Senior
DO
  FOR divCount := Science TO Aviation
  DO
    FOR sexCount := male TO female
    DO
      REPEAT
        WriteLn;
        WriteString ("Please give the number of ");
        WriteString (classNames [classCount]);
        WriteLn;
        WriteString (sexNames [sexCount]);
        WriteString (" in the ");
        WriteString (divNames [divCount]);
        WriteString (" division ==");
        ReadCard (mainDataArray [classCount, divCount, sexCount]);
      UNTIL ReadResult () = allRight;
      (* keep trying until correct entry *);
      WriteLn;
    END;      (* for sexCount *)
  END;      (* for divCount *)
END;      (* for classCount *)

(* data is all here now, so do summaries *)

FOR classCount := Freshman TO Senior      (* by class *)
DO
  classTotals [classCount] := 0;      (* initialize to 0 *)
  FOR divCount := Science TO Aviation
  DO
    classTotals [classCount] := classTotals [classCount]
      + mainDataArray [classCount, divCount, male]
      + mainDataArray [classCount, divCount, female];
  END;      (* for divCount *)
END;      (* for classCount *)

(* include code here to total by division and by sex in the same way *)

WriteString ("Summary  by class:");
WriteLn;
WriteString ("class      number");
WriteLn;
FOR classCount := Freshman TO Senior
DO
  WriteString (classNames [classCount]);
  (* write class name headings *)
  WriteCard (classTotals [classCount], 10);
  WriteLn;
END;      (* for classCount *)

(* include code here to print the appropriate headings and final data by faculty

```

and by sex in the same way *)

END EnrollData.

NOTES: 1. The completion of this example is left for the exercises at the end of this chapter, as sufficient detail has been presented for illustrative purposes. What is present here is free from syntax errors, but since it has not been completed and run, it is not guaranteed to be free of logical errors, and the logic of the ReadCard statement needs work. Why?

2. A Modula-2 implementation is likely to have a fixed maximum permissible degree of loop nesting. This affects any combination of FOR, WHILE, REPEAT and any other similar programming structures. This limitation is rarely a problem for the student, whatever the figure may be.

3. Notice the painstaking initialization of all those strings. For the purpose to which they are put here, there is no alternative to this, as one cannot use strings (or any but ordinal types) in an enumeration. However, there are other ways of initializing arrays (including strings) and these are covered in [section 11.6.1](#)

[Contents](#)

5.9 An Extended Example (Finding Prime Numbers)

In previous discussion on the greatest common divisor, it was mentioned that the GCD could be found by factoring the two numbers into products of primes, that is numbers with exactly two different positive divisors. The sequence of prime numbers 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, and so on has long fascinated mathematicians because of their usefulness and their unique properties. Although this is a sequence, because it can be numbered off in order with the natural numbers, it is possible to prove that there does not exist a formula that can be used to compute the i^{th} term of the sequence. In order to obtain a list of primes, it is therefore necessary to resort to computational methods that produce them one at a time. A possible algorithm for determining if a number is prime is:

```
let a trial divisor have the value 2
set a boolean Prime to true
while Prime or the trial divisor ≤ the square root of the test number
  divide the number being tested by the trial divisor
  if there is no remainder, set Prime to false
  increment the trial divisor
```

Once the square root of the test number has been reached, there is no point in checking for more divisors, as there can be a divisor larger than this only if there was also one smaller. Here is some Modula-2 code to implement and test this algorithm:

```
MODULE CheckPrime;

(* Written by R.J. Sutcliffe *)
(* using ISO Modula-2 *)
(* last revision 1991 03 01 *)

FROM STextIO IMPORT
  WriteString, WriteLn, ReadChar, SkipLine;
FROM SWholeIO IMPORT
  ReadCard, WriteCard;
FROM RealMath IMPORT
  sqrt;

VAR
  theNum : CARDINAL;
  ans : CHAR;

PROCEDURE IsPrime (testNum : CARDINAL) : BOOLEAN;

VAR
  prime : BOOLEAN;
  trialDiv, stop : CARDINAL;

BEGIN
  IF (testNum = 0) OR (testNum = 1)
  THEN
    prime := FALSE
  ELSE
```



```

prime := TRUE;
trialDiv := 2;
stop := TRUNC (sqrt (FLOAT (testNum)));
WHILE (trialDiv <= stop) AND prime
DO
  IF testNum MOD trialDiv = 0
  THEN
    prime := FALSE
  END;
  INC (trialDiv);
END; (* while *)
END; (* if *)
RETURN prime;
END IsPrime;

BEGIN (* main *)
REPEAT
  WriteString ("The number you type will be checked ");
  WriteString ("to see if it is prime ==");
  ReadCard (theNum);
  SkipLine;
  WriteLn;
  WriteString ("the number ");
  WriteCard (theNum, 0);
  WriteString (" is ");
  IF NOT IsPrime (theNum)
  THEN
    WriteString ("not");
  END;
  WriteString (" prime.");
  WriteLn;
  WriteString ("Do another? Y/N ");
  ReadChar (ans);
  SkipLine;
UNTIL CAP (ans) = "N";
END CheckPrime.

```

This code will do, if all that is required is to test a single number. However, it has two drawbacks. First, it is inefficient. It is not necessary to use *every* number from 2 through the square root of the number being tested as a trial divisor; only the primes in that range. If a way can be found to do this, the number of computations is greatly reduced. Second, no record or list of primes is kept by this program. A number that is given to it twice must be tested twice. Although this algorithm cannot be replaced by a simple call to a closed formula, it can be improved upon considerably.

A much more efficient method of finding primes is often given in elementary mathematics textbooks. It is called the *Sieve of Eratosthenes* (after the Greek mathematician) and when done by hand it works like this:

1. Write down the numbers to be tested in convenient rows

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

2. Cross out 1 and circle the number 2--it is prime.
3. Cross out all multiples of 2; they cannot be prime.

1	<u>2</u>	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

4. The next number not crossed out is a prime; circle it and cross out all its multiples that are not already crossed out. When looking for multiples, start with the square of the number just circled; any lesser multiples have already been crossed out.

1	<u>2</u>	<u>3</u>	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

5. Repeat steps 3 and 4 until all the numbers less than the square root of the largest number that was written down are either circled (prime) or crossed out (composite).

1	<u>2</u>	<u>3</u>	4	<u>5</u>	6	<u>7</u>	8	9	10
<u>11</u>	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

6. All remaining numbers of those written down are prime, and may now be circled.

1	<u>2</u>	<u>3</u>	4	<u>5</u>	6	<u>7</u>	8	9	10
<u>11</u>	12	<u>13</u>	14	15	16	<u>17</u>	18	<u>19</u>	20
21	22	<u>23</u>	24	25	26	27	28	<u>29</u>	30
<u>31</u>	32	33	34	35	36	<u>37</u>	38	39	40
<u>41</u>	42	<u>43</u>	44	45	46	<u>47</u>	48	49	50
51	52	<u>53</u>	54	55	56	57	58	<u>59</u>	60
<u>61</u>	62	63	64	65	66	<u>67</u>	68	69	70
<u>71</u>	72	<u>73</u>	74	75	76	77	78	<u>79</u>	80
81	82	<u>83</u>	84	85	86	87	88	<u>89</u>	90
91	92	93	94	95	96	<u>97</u>	98	99	100

A count of the circled numbers reveals that there are 25 primes less than 100. This hand method works very well for checking up to a few hundred numbers, but becomes rather tedious beyond that point. The calculations are, however, very simple to computerize. Using the numbers above for the steps:

1. Create an array of booleans and set them all to *true* at first. (*true* = prime)

2. Set array element 1 to *false*. Now 2 is prime.
3. Set the values whose index in the array is a multiple of the last prime found to *false*.
4. The next index where the array holds the value *true* is the next prime.
5. Repeat steps 3 and 4 until the last prime found is greater than the square root of the largest number in the array.

The program that follows tested the numbers to 1000 and printed out the primes it found, ten to a line.

Code:

```

MODULE Sieve;

(* Written by R.J. Sutcliffe *)
(* using ISO Modula-2 *)
(* last revision 1993 03 01 *)

FROM STextIO IMPORT
    WriteString, WriteLn, ReadChar;
FROM SWholeIO IMPORT
    WriteCard;
FROM RealMath IMPORT
    sqrt;

CONST
    size = 1000;

VAR
    primeFlags : ARRAY [1 .. size] OF BOOLEAN;
    numFound,
    numToTest, (* the candidate for primality *)
    count, lineCount,          (* Loop counters *)
    mul,      (* for multiples *)
    max      (* largest to test *) : CARDINAL;
    key : CHAR;

BEGIN
    numFound := 0;
    max := TRUNC (sqrt (FLOAT(size)));
    primeFlags [1] := FALSE;
    FOR count := 2 TO size
        DO      (* Initialize variables *)
            primeFlags [count] := TRUE
        END;

    (* Find primes *)
    FOR numToTest := 2 TO max
        DO
            IF primeFlags [numToTest]
                THEN (* got the next prime here *)
                    INC (numFound); (* count them as we go *)
                    mul := numToTest * numToTest; (* start at its square *)
                    WHILE mul <= size

```

```

        DO
            primeFlags [mul] := FALSE;    (* Cross out *)
            INC (mul, numToTest)    (* Do all multiples *)
        END    (* while *)
    END    (* if *)
END;    (* for *)

(* now, count the rest *)
FOR numToTest := max + 1 TO size
    DO
        IF primeFlags [numToTest]
            THEN    (* got the next prime here *)
                INC (numFound);    (* count them as we go *)
            END    (* if *)
        END;    (* for *)

(* Now print them out 10 to a line *)

lineCount := 0;
FOR count := 2 TO size
    DO
        IF primeFlags [count]
            THEN
                WriteCard (count, 6);
                INC (lineCount)
            END;
        IF lineCount = 10
            THEN
                lineCount := 0;    (* reset line count *)
                WriteLn;    (* and start new line *)
            END
        END;    (* for *)
WriteLn;
WriteString (" ... a total of ");
WriteCard (numFound, 6);
WriteString (" primes ");
WriteString ("Press a key to continue ==");
ReadChar (key);
END Sieve.

```

Results:

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541

547	557	563	569	571	577	587	593	599	601
607	613	617	619	631	641	643	647	653	659
661	673	677	683	691	701	709	719	727	733
739	743	751	757	761	769	773	787	797	809
811	821	823	827	829	839	853	857	859	863
877	881	883	887	907	911	919	929	937	941
947	953	967	971	977	983	991	997		

... a total of 168 primes

- NOTES:** 1. The value of *size* could have been set much higher, but there was no need to print any more primes here.
2. In some implementations of Modula-2, the maximum array index range is limited to a number much less than MAX (CARDINAL), and if *size* were to be greater than this, the program would not compile. This is due to limitations on the total size of a variable.
3. A file of primes could be kept on hand for use in other problems.
-

[Contents](#)

5.10 Chapter Summary

This chapter covered these topics:

- abstract and transparent data types
- whole number, ordinal, and scalar types
- how to declare new variable types
- how to declare and use enumerated types
- subranges of ordinal and enumerated types
- new standard procedures and functions for ordinal types
- an introduction to the ARRAY OF CHAR as a string type
- the FOR structure for iterated loops
- nested loops
- arrays
 - how to declare them
 - how to assign them
 - how to manipulate them
 - how to pass them as parameters (including open arrays)
- multi-dimensional arrays
 - how to declare them
 - how to manipulate them

It included discussion of the following Modula-2 built-ins:

Reserved Words

ARRAY

BY

FOR

OF

TO

TYPE

Standard Identifiers

HIGH

ORD

Symbols:

() for enumerated type declarations

[] for subranges

5.11 Assignments

Questions

1. What is the difference between an abstract data type and a transparent data type?
2. What must the specification of an abstract data type include?
3. What is the difference between a built-in ordinal data type and a user-defined ordinal data type? Illustrate with an example of each, and indicate what kind of operations can be used with each.
4. Explain why one cannot write *WriteString (January)* where *January* is a value of an enumerated type *MonthNames*.
5. Consider the module [SimplePay](#) in section 5.2.1. Why is it necessary to have the value *Saturday* in the enumeration even though it is not used?
6. Describe some advantages of using subranges.
7. Which of the following ranges is incorrectly specified, and why?
 - a) [10 .. 5]
 - b) ["0" .. 9]
 - c) [a .. e]
 - d) ["a" .. "Z"]
8. What is the difference between an iterated repetition, and the more general counted repetition?
9. Why are the WHILE and FOR loops not equivalent?
10. Which of the following are incorrect and why?
 - a. FOR count = 1 TO 15 DO
 - b. FOR count := 15 TO 1 DO
 - c. FOR count := "a" TO "z" DO
 - d. FOR boolVar := FALSE TO TRUE DO
 - e. FOR day := Mon TO Fri BY 2 DO
 - f. FOR count := 1.5 TO 10 DO
 - g. FOR count := 1 TO 10 BY 1.5 DO
 - h. FOR count := 1 TO 10 BY count DO
11. When the following code is compiled, all is well. However, when it is run, there is always an out-of-range error. Why?

TYPE

```
countRange = [1 .. 10];
```

VAR

```
count : countRange;  
count := 0;
```

BEGIN

```

WHILE count <= 10
  DO
    statement sequence;
    INC (count);
  END;

```

12. Using *DayName* as defined in the chapter: (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday); consider the following statement sequences. If an error arises, indicate the nature of the error. Assume that *cardVar* is of type **CARDINAL** and has the value of 2 at the beginning of each statement sequence, and answer with the final value of *cardVar* or *day* in each case.

a) *cardVar* := *cardVar* + **ORD** (Wednesday);

b) *day* := Monday;

day := *day* + 2;

c) *day* := **VAL** (DayName, 3);

INC (*day*, **ORD** (Monday));

d) *cardVar* := **ORD** (**VAL** (DayName, 2)) - 5;

e) *day* := Friday;

INC (Friday, 2);

13. Both assignments to the variable *count* in the following code are incorrect. In each case, why?

```

FOR count := 1 TO 100 DO
  statement sequence1;
  count := count + 2;
END;
count := count + 2;
statement sequence2;

```

14. What is wrong with the following nested loop:

```

FOR count := 1 TO 10
  DO
    FOR count := 2 TO 10
      DO
        statement sequence;
      END;
    END;
  END;

```

15. The following loop constructions each contain the statement **WriteString** ("Hi") at their innermost point to represent the working code of an entire statement sequence. In each case, state the number of times the sample statement executes.

a.


```
FOR count := 2 TO 100 BY 3
DO
  WriteString ("Hi");
END;
```

b.

```
FOR count := 200 TO 100 BY -13
DO
  WriteString ("Hi");
END;
```

c.

```
FOR outerCount := 2 TO 100
DO
  FOR innerCount := 2 TO 10
  DO
    WriteString ("Hi");
  END;
END;
```

d.

```
FOR outerCount := 2 TO 100 BY 3
DO
  FOR innerCount := 2 TO 10 BY 5
  DO
    WriteString ("Hi");
  END;
END;
```

e.

```
start := 5;
FOR count := start TO 1
DO
  WriteString ("Hi");
END;
```

16. Indicate what will be the output from each program segment. Pay attention to carriage returns and spacing.

a.

```
GString := "chord";  
FOR i := 0 TO 4  
  DO  
    FOR k := 1 TO 4  
      DO  
        WriteChar (GString [i]);  
      END;  
    END;  
  END;
```

b.

```
REPEAT  
  i := 5;  
  WriteCard (i, 10);  
  WriteLn;  
  INC (i, 2);  
UNTIL i = 7;
```

c.

```
FOR i := 1 TO 5  
  DO  
    FOR j := i TO 3  
      DO  
        WriteCard (i, 0);  
        WriteCard (j, 2);  
      END;  
    END;  
  END;
```

17. What is the mathematical name for a two-dimensional array?

18. Show how you would (i) declare, (ii) fill with zeros, one row at a time an n by m two-dimensional array.

Problems

19. Show how to modify the code of the module [LetterCounter](#) in Section 5.5 to ignore any non alphabetic characters and to regard uppercase and lowercase versions of a letter as the same letter.

20. Modify the module [SimplePay](#) in section 5.2.1 to take into account the following additional considerations:

- pay time-and-a-half for over eight hours per day or Saturday work
- pay double time- for over ten hours per day or Sunday work
- use a two week cycle rather than a one week cycle
- deduct 25% for taxes and \$31.50 per week for medical coverage.

21. Modify the module [ClassMarks](#) in section 5.4.2 to allow the individual marks to be weighted. The weights should be expressed in decimal form, and should add to 1. The data input should have the weights of the mark categories on the third line before the first student's name.

22. Write your own short program to print out the ASCII character values and their corresponding ordinals. You should start with CHR (32) and end at CHR (126) as the ones below 32 and above 126 are control characters and may affect your display, program, or computer in rather unexpected ways, and the ones above 127 are not defined for the purposes of Modula-2. (They may be graphics characters on your machine--want to try?) Print out the material and put it in your notebook for future reference.

23. Consider the letter counting and word averaging program in [section 5.5](#). Improve on it so that starting and ending punctuation is not included in the length of a word. Thus, when the program encounters "hello," all eight printable characters are recorded in the frequency array, but only the five alphabetic characters are used in computing the average word length. If the word is a qualified identifier such as *STextIO.WriteChar*, that is, the punctuation mark is in the middle of a letter group, it should be included for purposes of computing the average word length. Numbers should be included if they are part of an alphabetic word such as *identifier1*, or *goody2shoes*, but not when they are part of literals such as *234* or *15.3*.

24. Write a program that will take an input word (say, with about four different letters) and write out all possible "words" (anagrams) based on permutations (rearranging the order) of the letters of the given word. Make sure you don't use a letter twice. Use only *STextIO.WriteChar* for the output, not *STextIO.WriteString*.

Sample Input:

two

Sample Output: (Your algorithm may produce a different order.)

two tow wot wto otw owt

25. Write a program that will test a selection of text and determine the average number of syllables per word. You may take as a simple definition of a syllable that it is a consonant followed by a sequence of one or more vowels. Some exceptions to this rule are words that start with vowels or that end with y, which is then a vowel.

26. Write a program that will analyze a selection of text and create a table of frequencies for the letters of the alphabet only. The results should be printed out in order of the most frequently occurring to least frequently occurring letter. (This could be used to decode "secret" messages sent in simple letter-for-letter substitution cyphers.)

27. Consider the PROCEDURE [AddArrays3](#) in section 5.6 and suppose that HIGH (vect1) does not have to equal HIGH (vect2). This leaves some additions undefined, so assume that any extra elements supplied are zero and any extra elements in the result are set to zero. Modify the procedure to work this way and test it with two different test sets of data, each involving two arrays of different sizes. (Six different sizes altogether, please; twice with *vect1* larger, twice with *vect2* larger, and twice with

ansVector larger.)

28. Suppose you have a one-dimensional array of up to ten CARDINALs. Write a procedure to find the smallest one and swap it with the first element. You may want to use a *Swap* procedure as well. Now write a Module that sorts the whole array this way with successive calls to this procedure scanning elements 2 .. N, 3 .. N, and so on, each time selecting the smallest remaining one. Test with the input data 2, 84, 1, 5, 63, 89, 12, 15. Have the results printed. (This is called a selection sort.)

29. In [section 5.6](#) a procedure was developed for adding two vectors of indefinite length. Write and test a similar procedure for finding the dot product of two vectors of indefinite (but equal) length.

```
PROCEDURE dotProduct (vect1, vect2 : ARRAY OF INTEGER;
```

```
VAR dotOk : BOOLEAN; VAR result : INTEGER);
```

The procedure should return an error in the manner of *AddArrays4* if the lengths of the two open array parameters do not agree. To compute the dot product, multiply the corresponding components together, and add up all the products. Thus if $a = (a_1 \ a_2)$ and $b = (b_1 \ b_2)$, then $a \cdot b = a_1 b_1 + a_2 b_2$.

30. Write and test a procedure for determining the length of a two dimensional vector and the angle it makes with the x-axis. If $a = (a_1 \ a_2)$, r represents the length and θ is the angle, then $r^2 = x^2 + y^2$ and $\theta = \arctan (y/x)$, where *arctan* may be imported from *RealMath*.

31. Complete and test the example [EnrollData](#) in section 5.8.

32. Write a program to record the marks of an entire class of students and calculate their letter grades. Feel free to use the example [ClassMarks](#) of section 5.4.2 for part of this. What you will also need is an array to store student names and another one for their marks, say on four tests, each out of one hundred. Provision should be made for weighting as in problem 16 above. You also need sections for entering the data and for printing it out along with the final letter grades. Make good use of procedures.

33. *Twin primes* are primes that differ by two. Thus 3 and 5, 5 and 7, 11 and 13, and so on are twin primes. Write a program to examine a file of primes created by a program like the example in [section 5.9](#) for twin primes, and print out all the twins it finds.

34. The *Fibonacci sequence* is defined by $a_1 = 1$, $a_2 = 1$, and for i --if it goes first it should always win or draw, regardless of the user's strategy.

40. A number is divisible by nine if the sum of its digits is divisible by nine. Write a program to test a number that is entered as a string of digits, one character at a time, to see if it is divisible by nine. (This problem would be trivial if you were allowed to enter the number as a CARDINAL.)

Chapter 5

Iterations, Enumerations, and Arrays

[5.0 Chapter Goals](#)

[5.1 Abstract and Transparent Data Types in Modula-2](#)

[5.2 Making One's Own Data Types](#)

[5.2.1 Ordinal and Enumerated Types](#)

[5.2.2 Subranges Of Existing Types](#)

[5.2.3 Summary of some Modula-2 compatibility issues:](#)

[5.2.4 Summary of some Modula-2 types](#)

[5.2.5 Making comparisons](#)

[5.3 Indexed Data Types--Arrays](#)

[5.3.1 A First Look at String Variables](#)

[5.4 The FOR Statement](#)

[5.4.1 The FOR Loop and the WHILE Loop](#)

[5.4.2 The FOR Loop in Use](#)

[5.5 Manipulating Arrays](#)

[5.6 Arrays as Parameters](#)

[5.7 Multi-Dimensional Arrays](#)

[5.7.1 Arrays of More than Two Dimensions](#)

[5.7.2 Multidimensional Open Array Parameters](#)

[5.8 Manipulating Multi-Dimensional Arrays](#)

[5.9 An Extended Example \(Finding Prime Numbers\)](#)

[5.10 Chapter Summary](#)

[5.11 Assignments](#)

[Contents](#)

6.0 Chapter Goals

The purpose of this chapter is to elaborate on the module abstraction as a means of organizing programs. The library modules already used (*STextIO*, *SRealIO*, *SWholeIO* and *RealMath*) are explained in more detail, as will some of their variations. In addition, user-written library modules are covered. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

No new data types are taken up in chapter 6.

Realized in the Modula-2 notation:

Data Manipulation Abstractions

General:

entity encapsulation in libraries, I/O streams

Realized in the Modula-2 notation:

abstract data types encapsulated in modules, redirection of I/O

Programming Abstractions

General:

Top-down design and planning is further extended to the encapsulation of related entities in a library that can be imported from by any program.

Realized in the Modula-2 notation:

library modules

middle-of-loop tested repetition structure

6.1 What Did You Say a Module is?

Back in Chapter 2, the following definition was provided:

A module is a container to hold the items and information that constitute all or part of an executable program.

At the time, the concept was left deliberately vague. Thus far, this text has employed program modules and has pointed out that various imports were also derived from modules. As the definition implies, a module is simply a box, the function of which is to enclose the various entities that are defined or used therein. As ought to be evident already, the entities contained in a module may be variables, constants, types and procedures. Each of these may be defined inside the module in question or may be imported from some other module. One module can even contain another module in this way; and this nesting can go on for as many levels as the programmer desires, or the system permits.

The closest analogue among the other things seen thus far may be the procedure. An entity such as a variable is either visible and useful in a procedure or it is not. Likewise, an entity (via its identifier) is either visible and usable within the confines of a module or it is not. If it is not, any attempt to employ it in that module will result in an "Undeclared Identifier Error."

None of this is entirely new, for much of it has been stated or hinted at already--but it does allow some refinement of the definition into a more useful form.

A module is an enclosing device or container to delimit the visibility and use of entities, whether these be internally defined or imported.

This version of the definition implies that the entities in question may constitute all or part of a program. If the module encloses types, constants, and procedures only (as does a library module) then these entities are not part of an actual program until they are imported and used by one. Until that time, they are only abstractly (or more correctly, potentially) portions of programs.

As remarked both here and in [section 4.2](#), a module defines a scope of visibility, much as does a procedure. Indeed, that is all it does. One might be tempted to extend the analogy between these procedures and modules, but such an attempt quickly fails, for modules are not themselves subprograms; they are only enclosures into which programs, subprograms, and other entities can be placed. They are either used within their own enclosure (as with a program module) or they are extracted from there for use inside another one.

Like procedures, modules are themselves entities, so they are named. This name must appear both in the declaration and with the end of the module as in all program modules thus far in the text. Unlike procedures, modules in the base ISO standard language have no parameter lists and are not generally subprograms themselves, so it is safest to regard a module as a set of walls and nothing more.

These concepts will be discussed in more detail in chapter eleven in the sections on scope and visibility rules for local modules. For now, more practical matters demand some attention.

[Contents](#)

6.2 Libraries--How to Borrow a Module and Sign it Out

Certain tools (input/output, for example) are needed in any programming environment in order to create useful code. From the very first program module presented in chapter two, this text has made use of some of the many modules that are available as part of a typical Modula-2 system. These contain a variety of entities that can be imported into a program without having to be re-written every time they are needed. When Niklaus Wirth defined the Modula-2 notation, he was, for the most part, quite clear and specific about the way its various elements were to be used, and how they would behave. However, when it came to the libraries, he suggested rather than legislated. Thus, for instance, the input/output modules that appear in his book are described as "typical." Wirth says:

"We postulate some operations that are to be available in every implementation of Modula, although we wish to specify rigidly neither the names of the modules containing them nor the set of remaining operations included in these modules." *Programming in Modula-2, third edition p.105*

He went on to describe the module *InOut* as one of these typical modules, and to make suggestions for several others as well. Because these were only suggestions, they were not always followed by implementors. For instance, although the *functionality* of a *WriteInt* was always needed, it was not always supplied by vendors in the same way. Likewise, there is always a *ReadString*, but the name of the module from which it was imported may vary. In the case of *WriteReal*, even the syntax often varies, as has been seen already in this text. In the absence of any other guidance, however, and because the first compilers were produced by Wirth himself and then licensed to commercial vendors, most early implementations of Modula-2 had a very similar core library.

The (classical standard) Modula-2 library is a collection of previously defined modules whose entities form a part of the operating environment of Modula-2. These are not a part of the language proper (i.e. neither reserved words nor standard identifiers), but are available for import by other modules.

The many variations in interpretation even of details of the language, and the limitations and inconsistencies in the various implementations of these suggested I/O libraries were part of the motivation that led to the formation of the ISO standards committee in 1987. It was charged with the production of an international standard for the language, and also with the definition of a standard set of library modules. The latter had reached what appeared to be a final form by 1992, and most implementations provided the new libraries by 1996 when the standard became official.

The (ISO standard) Modula-2 library is a collection of previously defined modules whose entities form a part of the operating environment of Modula-2. These are not a part of the language proper (i.e. neither reserved words nor standard identifiers), but are available for import by other modules. All versions of Modula-2 claiming compliance with the ISO standard and requiring the functionality of one of these libraries (input and output, for instance) must provide that library exactly as specified in the standard, with no additions, deletions, or changes in meaning.

Even though the ISO library has been widely adopted, the older classical core library is still supplied by some vendors, for compatibility with older code.

The case of input and output is a particularly difficult one to deal with, because every computer system in which a programmer works may handle these things differently. That is the reason why standard input and output are not handled as built-ins, but are found in the library. This is done in order to keep the language itself both small and more portable across many systems.

*S*TextIO, *S*RealIO, and *S*WholeIO (or *In*Out and *RealIn*Out) form a suite of high level modules, in the sense that their facilities can be used by a program without regard to very many system details. At some point, however, they must in turn call on lower level code (to access data streams, for instance) in order to translate standard procedure invocations into something the particular system can use. That is, there is a hierarchical ordering of the modules involved in I/O, with the higher ones the most general and furthest abstracted from the actual system, and the lowest ones the most detailed and system specific. Naturally, a programmer who desires to do the extra work required can tap into this hierarchy at a lower level lower to define different styles of or destinations for input and output than those used by the highest level modules.

The lower one goes in this hierarchy, the more power that is gained, but the modules also become progressively harder to use. Again, here is a flexibility not available in other languages; one could implement entirely different *WriteCard* or *WriteString* procedures than the ones provided if this were thought desirable. The details of this organization are not essential to a beginning programmer and a good Modula-2 implementation will hide most of this detail away from beginning programmers so only those who have become more skilled can take advantage of some of the lower-level library modules. More detail will be provided on these lower level modules in later chapters; in section three of this one high level I/O will be more fully described.

Besides those for I/O, there are a number of other modules available in the standard library. All one has to know in order to use any of them is:

1. the name of the module containing the entity
2. the name of the entity
3. the correct syntax for using the entity, that is
 - if a procedure, what are the types of the parameters?
 - if a data type, what is its structure?
 - if a variable, what are its allowed values?

In other words, one can treat standard library entities almost as if they were extensions of the language. Any library entities known to be available can be treated as extensions of a program. Once one knows they are there, one may just write:

```
FROM FantasticModule IMPORT  
    GreatRoutine, SuperVariable, MagicType;
```

or,

```
IMPORT FantasticModule
```

and then use them in ones own program module. In the latter case, the entity names must be *qualified* by the module name whenever they are referred to, and this may be necessary to avoid name conflicts, or to conform to local style rules. These methods apply both to existing library modules supplied by the vendor of the Modula-2 implementation, and to any that the programmer might write for private use. User-designed modules are covered in section five of this chapter. The next two sections contain an examination of most of the higher level modules in the standard library, first considering I/O routines, and then some of the others.

[Contents](#)

6.3 The Standard Library (1)--I/O

6.3.1 ISO Standard I/O Modules

The prefix *S* in the module names *S*TextIO, *S*RealIO, and *S*WholeIO stands for *simple*. All I/O controlled by these modules is directed to/from a standard I/O channel (normally the keyboard/screen.) A call to *S*WholeIO.*WriteCard* probably results in a call to *WholeStr.CardToStr* to convert the cardinal to a string, followed by a call to *S*TextIO.*WriteString*. However, vendors are free to implement the ISO standard without this relationship among the various modules, as long as they provide the standard items themselves. The module *SIOResults*, for determining the outcome of a *Readxx* operation from one of these modules must also be present. Here are the full listings of some of these modules (see [Appendix 5](#) for syntax):

```
DEFINITION MODULE STextIO;
```

```
PROCEDURE ReadChar (VAR ch: CHAR);
PROCEDURE ReadRestLine (VAR s: ARRAY OF CHAR);
PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
PROCEDURE ReadToken (VAR s: ARRAY OF CHAR);
PROCEDURE SkipLine;
PROCEDURE WriteChar (ch: CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteString (s: ARRAY OF CHAR);
```

```
END STextIO.
```

NOTE: The exact meaning of the reserved word DEFINITION will be made clear later in the chapter. It is enough to know that these are just lists of the entities that are importable from these modules.

The *Readxx* procedures produce the *ReadResult allRight* if everything goes well. Other possible results are *endOfLine* or *endOfInput*. In the case of those procedures taking an ARRAY OF CHAR parameter, another possible outcome is *outOfRange* if the supplied parameter is not big enough to hold the input.

ReadRestLine removes the characters from the input until the next *endOfLine* state is reached, copying as many as possible to the string parameter. *ReadString*, on the other hand, copies as many characters as possible to the string parameter until it either runs out of room or gets to the end of the line. Thus, it might return with the value *allRight*, but with some characters still remaining as available input before the next *endOfLine* state. *ReadToken* will skip spaces before reading other characters and then read as far as a space or an *endOfLine* state, whichever comes first.

NOTE: What is here called *ReadToken* is in non-standard versions usually called *ReadString*, and the other two are not usually provided in such versions.

SkipLine removes characters from the input to the next *endOfLine* state and discards them. It also clears the end of line state.

```
DEFINITION MODULE SWholeIO;
```

```
PROCEDURE ReadInt (VAR int: INTEGER);
PROCEDURE WriteInt (int: INTEGER; width: CARDINAL);
PROCEDURE ReadCard (VAR card: CARDINAL);
PROCEDURE WriteCard (card: CARDINAL; width: CARDINAL);
```

```
END SWholeIO.
```

```
DEFINITION MODULE SRealIO;
```

```
PROCEDURE ReadReal (VAR real: REAL);
```

```

PROCEDURE WriteFloat (real: REAL; sigFigs: CARDINAL; width: CARDINAL);
PROCEDURE WriteEng (real: REAL; sigFigs: CARDINAL; width: CARDINAL);
PROCEDURE WriteFixed (real: REAL; place: INTEGER; width: CARDINAL);
PROCEDURE WriteReal (real: REAL; width: CARDINAL);
END SRealIO.

```

```

DEFINITION MODULE SLongIO;

```

```

PROCEDURE ReadReal (VAR real: LONGREAL);
PROCEDURE WriteFloat (real: LONGREAL; sigFigs: CARDINAL; width: CARDINAL);
PROCEDURE WriteEng (real: LONGREAL; sigFigs: CARDINAL; width: CARDINAL);
PROCEDURE WriteFixed (real: LONGREAL; place: INTEGER; width: CARDINAL);
PROCEDURE WriteReal (real: LONGREAL; width: CARDINAL);

END SLongIO.

```

If the procedures *ReadInt*, *ReadCard*, *SRealIO.ReadReal* or *SLongIO.ReadReal* are given input that cannot be integer, cardinal, real, or longreal respectively, the *ReadResult* returned is *wrongFormat*. This might happen, for instance, if a *ReadCard* were executed and then a negative whole number or a real were found on the actual input.

Except for the *SLongIO* procedures, (which all compare to those in *SRealIO*) all the rest have been well used thus far in the text. Notice that in the ISO standard, there is no lower level module called *Terminal* or *Screen*. Rather, it is assumed that the normal origin and destination of I/O from these *S*-modules is the standard terminal for the system.

6.3.2 Classical I/O--The InOut Family

The reader who has the ISO standard I/O modules only may skip most of this section unless interested in the historical development of Modula-2. The high level modules *InOut* and *RealInOut* (or *RealIO*) have been referred to a few times in this text, but without a complete list of all the things available for import from them. Here they are--with parameters where applicable:

```

DEFINITION MODULE InOut;
CONST
    EOL = CHR (13) (* may be system dependent *)

VAR
    Done : BOOLEAN;
    termCh : CHAR;

PROCEDURE OpenInput (defext : ARRAY OF CHAR);
PROCEDURE OpenOutput (defext : ARRAY OF CHAR);
PROCEDURE CloseInput;
PROCEDURE CloseOutput;
PROCEDURE Read (VAR ch : CHAR);
PROCEDURE ReadString (VAR s : ARRAY OF CHAR);
PROCEDURE ReadInt (VAR x : INTEGER);
PROCEDURE ReadCard (VAR x : CARDINAL);
PROCEDURE Write (ch : CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteString (s : ARRAY OF CHAR);
PROCEDURE WriteInt (x : INTEGER; n : CARDINAL);
PROCEDURE WriteCard (x, n : CARDINAL);
PROCEDURE WriteOct (x, n : CARDINAL);
PROCEDURE WriteHex (x, n : CARDINAL);

```

END InOut.

NOTE: 1. These contents are only typical, and may vary considerably from vendor to vendor.

2. Some implementations also include the less typical procedure *ClearScreen* in this module. It may even be necessary to call this procedure before attempting any other screen output.

3. Others include the procedure *HoldScreen* to pause and wait for a key. This avoids the necessity of writing two lines of code for this purpose.

4. *OpenOutput* and *OpenInput* are procedures designed to allow the output to go to other than the screen and the input to come from other than the keyboard. An example will be given in the next chapter.

5. *defext* is short for "default extension" and is usually "TEXT," if it is anything at all. If the user of a program containing one of these redirecting procedures answers the prompt asking for a file name by typing, say, by "Mydisk:Superfile." that is, ending the name with a period, then the default extension "TEXT" is added, and the file that is actually looked for will be "Mydisk:Superfile.TEXT".

6. In (less typical) versions that take file names directly as parameters for "OpenOutput", no prompts are given, but it is then the responsibility of the programmer to ensure the correct file name syntax. As this may differ from one system to another, such programs are probably not portable.

DEFINITION MODULE RealInOut;

VAR

Done : **BOOLEAN**;

PROCEDURE ReadReal (VAR x : **REAL**);

PROCEDURE WriteReal (x : **REAL**; n : **CARDINAL**);

PROCEDURE WriteRealOct (x : **REAL**);

END RealInOut.

NOTES: 1. In some implementations the contents of *RealInOut* are contained inside the module *InOut*. In such cases, there is only one variable *Done*.

2. As previously observed, the meaning of *WriteReal* varies widely.

Which of these have not already been used in this text? There are *WriteOct* and *WriteHex* that output the **CARDINAL** type in a different format than as conventional decimal numerals, and *WriteRealOct* that similarly outputs a **REAL**. These abbreviations refer to the octal and hexadecimal formats for numbers that many programmers find useful when working with the computer at a very low level. (That's base eight and base sixteen notation for those who have already taken the Mathematics.) For now, experiment with these. They will be also discussed in a later Chapter.

NOTE: Not all systems bother with octal notation, as it is seldom used. The relevant procedures might be omitted in some implementations.

ReadString has not been seen before, though its purpose may seem obvious, especially given its parameter. It skips leading spaces and then reads characters from the input and constructs a string in the parameter, stopping when it comes to either a space or a line marker. The variable *termCH* is used to hold the character that caused *ReadString* to stop reading.

NOTES: 1. This behaviour is typical, but some versions supply a *ReadString* that does not stop until the string parameter is full or the end of line marker is reached, whatever comes first. Such versions (and this includes the ISO standard) supply a separate *ReadToken* with the meaning of the *ReadString* described here.

2. Different implementations vary as to whether a leading end of line marker is consumed by *ReadString*. Some do, and some do not. The latter require a specific *Read(cr)* or a call to a procedure *ReadLn* to consume the end of line marker before the next string can be read.

As mentioned earlier, there is a hierarchy of modules involved in I/O. There may also be a hierarchy of the procedures within a module. Thus it is typical, for example, for an invocation of *ReadReal* to involve several other procedures in the manner of the following pseudocode:

ReadReal:

Perform an InOut.ReadString (ReadToken) to get the input


```
    Use InOut.Read to obtain the first character and put it in the string
    while character read is neither a space or end-of-line
        read another character and put it in the string
    invoke a conversion procedure to convert the characters to a real
```

The conversion procedure employed to change the string representation of the real that was read into the appropriate internal representation for the system being used will normally be in yet another module. It is variously termed *Reals.Convert*, *Conversions.StringToReal*, or some similar name. The programmer must check the implementation in order to use these conversion routines directly; they are regarded as lower level, and vary considerably in non-standard implementations. Another aspect of this hierarchy is that input and output can either be connected to the terminal (screen and keyboard) or redirected somewhere else. When either is attached to the terminal, the procedures in *InOut* may act by calling those of a module by that name. Here is a typical definition:

```
DEFINITION MODULE Terminal;

PROCEDURE Read (VAR ch : CHAR);
PROCEDURE ReadLn (VAR s : ARRAY OF CHAR);
PROCEDURE BusyRead (VAR ch : CHAR);
PROCEDURE ReadAgain;
PROCEDURE Write (ch : CHAR);
PROCEDURE WriteString (s : ARRAY OF CHAR);
PROCEDURE WriteLn;

END Terminal.
```

NOTE: Some implementations also include the (even less standard) procedure *ClearScreen* in this module. When it is provided, the programmer can if desired use this module directly instead of *InOut* if that output is to go to the terminal and not elsewhere. Since some of these procedures have the same names as the ones in *InOut*, one must avoid importing any duplicates. As previously indicated, one imports either *Terminal* or *InOut* as a whole by using a statement like:

```
IMPORT Terminal;
```

and then using qualified identifiers such as:

```
Terminal.Read (ch);
InOut.Write (ch);
```

and so on. In this manner, one may redirect the output of *InOut* (and therefore of *RealInOut* which uses it) by employing *OpenOutput* and sending it to a file, while continuing to write prompts to the screen using *Terminal.WriteString*. See 6.3.4 for an example.

There are a few new ones in this collection, though. *BusyRead* (where supplied) returns character zero (also called NULL) if something has not already been typed at the keyboard when it is invoked. If some character is waiting in the input from the keyboard, it returns that character.

NOTE: In most implementations providing this module, *BusyRead* (and sometimes *Read* also) will not echo what they fetch to the screen as do the read procedures in *InOut*. This could be useful if one wanted a program to ask the user to type in a password before carrying on with business, and didn't want some one reading that secret access code from the screen as it was typed. An even simpler use for *BusyRead* is to check on whether *any* key has been pressed. In many programs, after writing out a message to the user, and just before filling the screen with new information, the programmer may wish to print the message "Press any key to continue =" and then wait for the user to read and digest the information on the screen before carrying on. Examples so far have used:

```
WriteString ("Press any key to continue => ");
```

```
Read (ch);  
Read (cr);
```

which has the advantage of simplicity, but the disadvantage that if the user presses the <RETURN> key, the program is thrown off by one character in its interpretation of the input stream. A better way is:

```
PROCEDURE HoldScreen ();  
  
VAR  
    Ch : CHAR;  
  
BEGIN  
    WriteString ("Press any key to continue =");  
    REPEAT  
        BusyRead (Ch);  
    UNTIL (Ch # CHR (0));  
  
END HoldScreen;
```

where the character read will not be examined, and can be anything that can be typed.

A few implementations have a procedure already present in *InOut* to achieve this same goal. It may be called *HoldScreen*, *WaitForKeypress*, or some similar name.

ReadAgain (where supplied) takes the last character read and places it back into the part of the computer's memory from which input characters are read (called a *buffer*). The next invocation of *Read* will obtain the same character that was read the last time. This may seem a little obscure, and simple programs would not use it, but one application for this could be if the item being read necessitated a transfer of control to another module that did not have access to the variable used for the read. It could find out what character was read by looking for itself.

ReadLn (often written *ReadLine* to distinguish it from a *ReadLn* discussed above) reads a line of characters into a string variable. It (usually) echoes characters to the screen as they are read, and stops when it gets to a carriage return--which is not a part of the line read. In most cases, editing of the input is allowed before the <RETURN><ESC><ESC RET><ESC

In figure 6.1, the solid lines represent channels that are presently in use, and the dashed lines channels that are potentially available. There may be many of these channels. Each may represent access to a device, such as a printer, screen, keyboard, or mouse. Or, a channel may be an access to a file stored on a device such as a tape or disk drive.

A channel has one of three conditions. It may be:

closed: only potentially available to a program,

open: attached to a program and immediately available for its use,

in use: open and with a data stream actually flowing in it.

A channel has one of three modes. It may be:

read: a channel from a source,

write: a channel to a sink,

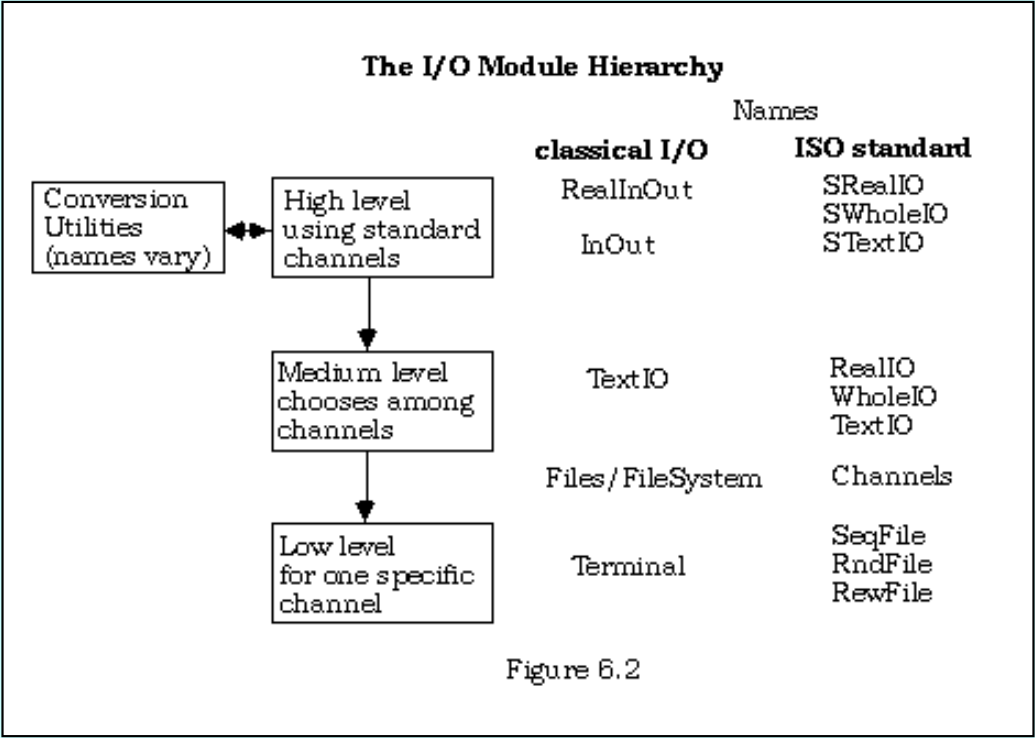
read/write: a channel connection with something that is both a source and a sink.

Thus, the connection between the program and the source or sink can be thought of as a *channel* in which runs a *stream* of characters. Individual characters are inserted into (output) or plucked from (input) these streams by the I/O procedures. Modules like *STextIO*, *SWholeIO*, and *SRealIO* employ *Writexx* procedures to send data streams to a sink, and *Readxx* procedures to obtain data from a source. Typically, a procedure like *WriteCard* calls on a conversions module to convert the Cardinal data to a string, and then employs *WriteString* to do the output. The input/output modules will always mark a single special input channel and a single special output channel as open. These are called the standard I/O channels.

When these modules first start up, the default console channels are automatically opened for standard I/O. Data can be redirected elsewhere by attaching the open mode to some other channel(s), but these simple modules can only have one input and one output channel open at a time. More sophisticated modules that will be studied later can have several of each open

simultaneously. Furthermore, since a single processor can only do one thing at a time, only one of the *open* channels can actually be *in use* at a time.

Without going into further detail at this time, there are more modules that can also control the flow of such information at a lower level, adding the power to open, close, and manipulate such streams from a program in more detail than can the *S-*modules (or *InOut*). The typical module hierarchy these considerations produced may be represented in part as in figure 6.2. The module names are positioned beside the diagram in their approximate place in the hierarchy. Additional details, including how to use the middle and lower levels of these modules, will be discussed in Chapter 7.



6.3.4 Redirection (optional)

WARNING: All information in this section is non-standard implementation specific and may not be available in the form given on the reader's own system.

In order to accomplish the re-direction of standard output to some other device (such as a printer or a file) one needs the following entities:

OpenOutput redirects the output away from the screen.

CloseOutput redirects output back to the console.

Here is a brief sample to illustrate the use of these:

```
OpenOutput;    (* system will ask user where to send output *)
WriteString ("The answer is ");    (* output will go there *)
WriteReal (ans, 15);
WriteLn;
CloseOutput;   (* now output to console *)
```

If using classical Modula-2, *OpenOutput* and *CloseOutput* are both imported from *InOut*. In this event, the syntax for *OpenOutput* is usually something like:

```
OpenOutput ("TEXT")
```

or, just

```
OpenOutput ( "" )
```

When a program containing an `InOut.OpenOutput` statement is executed, the system is supposed to place a message like

```
Output file?
```

or, perhaps

```
out"PRN:").
```

2. If the user response is "ThisProg." (i.e. it ends with a period), the output will be sent to "ThisProg.TEXT" in the current disk directory (or folder.) The suffix "TEXT" is called the default extension and is obtained from the `InOut.OpenOutput` statement as it was used in the program. This could as easily have been "TXT" or "CODE," or whatever is appropriate. It is added only if the file name ends in a period.

3. Other systems will search for the given file name first, then append the default extension and try again.

4. A few versions of `InOut.OpenOutput` use a radically different approach. In these, `OpenOutput("MyFile")` is an instruction to find a particular file, and no prompts are given to the user at all.

5. Pressing just the <ENTER><RETURN><ESC><RETURN

- Only one file can be opened by `OpenOutput` for output at a time.
- Only one file can be opened by `OpenInput` for input at a time.
- A program cannot close a file with `CloseOutput`, write to the screen, reopen it with `OpenOutput`, and continue writing where it left off. The insertion (writing) point is repositioned to the beginning upon opening, and the write operation following reopening would then erase and write over the top of what was there before. For a more sophisticated file handling technique that does allow this, see chapter 7.
- Some systems allow the user to open up a special log file before running the program. When this is done, all the normal input and output at the console is copied to the log file, without the program having to do anything itself to achieve this. Depending on the system, use of this facility may prevent use of redirection.

NOTE: Each system has its own rules in this regard, and all the possible details for correct syntax, device names, path names, and so on, cannot be covered here. System operating manuals must be consulted.

WARNING: Failure to close a channel to a file using `CloseOutput` and/or `CloseInput`, as appropriate, before the program terminates could result in some of the data being lost or the file damaged. The operating system is not responsible for (and indeed may not even "know" about) files that are opened by a program.

To illustrate some of the points made in this section, here is a program in classical-style Modula-2 that reads a sequence of real numbers from the keyboard and stores them into a data file on the disk for later use by another program. It determines that the sequence is finished when it fails to read a valid real or the number typed equals zero.

```

MODULE NumsToDisk;

(* Written by R.J. Sutcliffe *)
(* in non-ISO classical Modula-2 *)
(* to illustrate the use of redirection *)
(* using Metropolis Modula-2 for the Macintosh computer *)
(* last revision 1991 02 22 *)

FROM InOut IMPORT
    OpenOutput, CloseOutput, WriteLn, WriteString;
FROM RealInOut IMPORT
    ReadReal, WriteReal, Done;

VAR
    num : REAL;

BEGIN
    WriteString ("This program creates a disk file and enters ");
    WriteLn;
    WriteString ("the real numbers you type into the file. ");
    WriteLn;
    WriteLn;
    WriteString ("Type the numbers, separated by returns ");
    WriteLn;
    WriteString ("When finished, type a letter, or the number 0");
    WriteLn;
    WriteLn;
    OpenOutput ("TEXT");

    REPEAT
        ReadReal (num);
        IF Done
            THEN
                WriteReal (num, 0);
                WriteLn;
            END;
    UNTIL NOT Done OR (num = 0.0);

    CloseOutput;

END NumsToDisk.

```

This program was run and some numbers entered. A picture of the screen is reproduced in figure 6.3 below:

```

This program creates a disk file and
enters the real numbers you type into the file.

Type the numbers, separated by carriage returns
When finished, type a letter, or the number 0

out> test
test overwrite? y

12.2
15.678
11.9
0.345
17.1
2.5
a

```

Figure 6.3

Following this, the file test contained:

```

12.20000
15.67800
11.90000
0.3450000
17.10000
2.500000

```

The use of *OpenInput* and *CloseInput* closely parallels that of *OpenOutput* and *CloseOutput*. The following module was used to open the file created above, examine the numbers in it, find the maximum, sum, and average and then display the results. It, however, is written in ISO Standard Modula-2, and employs the module *RedirStdIO* shown above.

Pseudocode:

```

Print an informative message
Set the maximum, number read, and sum to zero
Invoke OpenInput
Repeat
  read the next number
  If read was successful then
    add to the sum
    increase the count of those read successfully
    If the number is greater than the maximum then
      set maximum to the number
Until the read is unsuccessful
Print the results

```

```

MODULE StatsFromDisk;

(* Written by R.J. Sutcliffe *)
(* to illustrate the use of redirection on input *)
(* using ISO Modula-2 for the Macintosh computer *)
(* and RedirStdIO as implemented by R. Sutcliffe *)
(* last revision 1993 03 02 *)

FROM STextIO IMPORT
    WriteLn, WriteString, ReadChar, SkipLine;
FROM SWholeIO IMPORT
    WriteCard;
FROM RedirStdIO IMPORT
    OpenInput, CloseInput;
FROM SRealIO IMPORT
    ReadReal, WriteFixed;
FROM SIOResult IMPORT
    ReadResult, ReadResults;

VAR
    numRead, max, sum, mean : REAL;
    readOK : BOOLEAN;
    howMany : CARDINAL;
    key : CHAR;

BEGIN
    howMany := 0; (* initialize variables *)
    max := 0.0;
    sum := 0.0;
    WriteString ("This program opens a disk file of real numbers");
    WriteLn;
    WriteString ("and computes their sum, average, and maximum. ");
    WriteLn;
    WriteLn;
    OpenInput; (* no default extension used in this version. *)

    REPEAT
        ReadReal (numRead); (* get a number from the file *)
        readOK := (ReadResult () = allRight);
        SkipLine;
        IF readOK
            THEN
                sum := sum + numRead; (* if ok, add to sum *)
                howMany := howMany + 1; (* and increment how many read *)
                IF numRead > max
                    THEN
                        max := numRead; (* reset maximum if needed *)
                    END;
            END;
    UNTIL NOT readOK;

    CloseInput;
    mean := sum / FLOAT (howMany);

```

```

(* report results *)
WriteString ("There were ");
WriteCard (howMany, 0);
WriteString (" numbers read, of which the largest was ");
WriteFixed (max, 5, 0);
WriteLn;
WriteString ("The sum of these numbers was ");
WriteFixed (sum, 5, 0);
WriteString (" and the mean was ");
WriteFixed (mean, 5, 0);
WriteLn;
WriteString ("Press a key to continue ==> ");
ReadChar (key);

```

END StatsFromDisk.

There follow two pictures of the screen taken when this program was run. In the first, the Macintosh operating system has brought up a standard dialog box used to find existing files. The user of the program has found the correct folder, has highlighted the correct file, and is about to press the return key to allow the file to be opened.

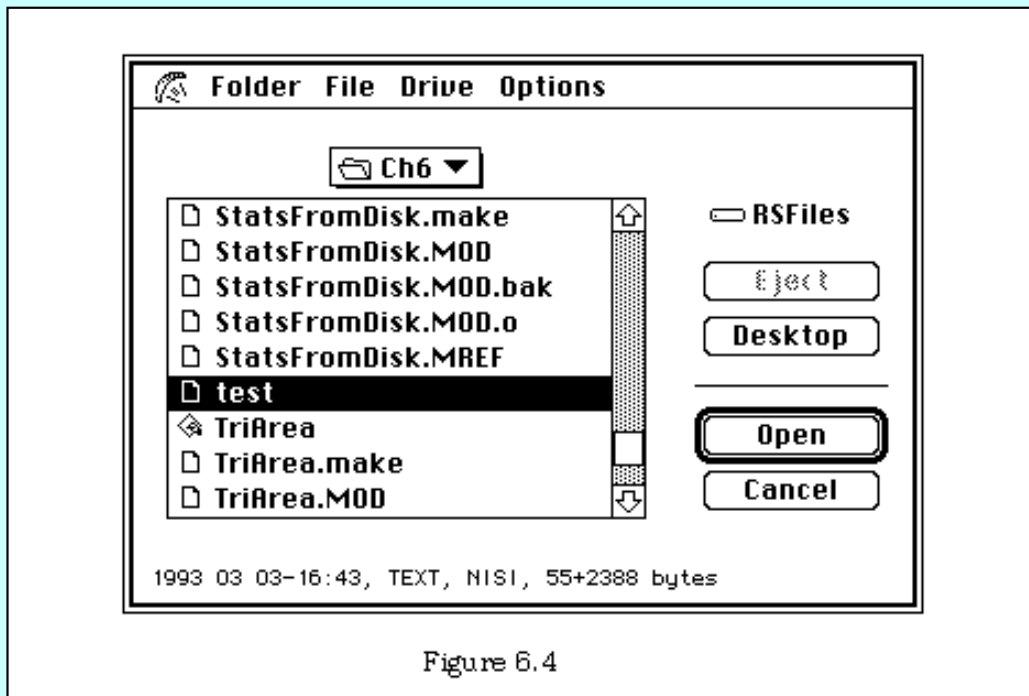


Figure 6.4

The following is picture of the screen taken when this program was run.

```

This program opens a disk file of real numbers
and computes their sum, average, and maximum.

There were 6 numbers read, of which the largest was 17.10000
The sum of these numbers was 59.72300 and the mean was 9.95383
Press a key to continue ==> █

```

Figure 6.5

A common use of *OpenInput* and *OpenOutput* is to give the user the choice of where to obtain or send data. Subsequent program statements are then executed in the light of the user choice. The following module is a modification of the *DateCalc* example of chapter 3 so as to employ the last of the refinements discussed there and also to allow for a series of input data to come from a file. Notice that the flag *usingFile* is reset according to the results of the attempt to do the redirection.

```

MODULE DateCalcB;

FROM STextIO IMPORT
    WriteString, WriteLn, ReadChar, SkipLine;
FROM SWholeIO IMPORT
    ReadCard, WriteCard;
FROM RedirStdIO IMPORT
    OpenInput, CloseInput, OpenResults, OpenResult;
FROM SIOResult IMPORT
    ReadResult, ReadResults;

VAR
    day, month, year, result, adjMonth, daysSinceMarch : CARDINAL;
    usingFile, again, leap, readOK: BOOLEAN;
    response : CHAR;

BEGIN
    (* header information left out*)

    REPEAT (* main repeat loop *)
        (* check to see if user wants to use a file *)
        WriteString ("Do you wish the information to come");
        WriteString (" from a file? (Y/N) ====> ");
        ReadChar (response);
        SkipLine;
        WriteLn;
        WriteLn;
        usingFile := CAP (response) = "Y";
        (* all other responses mean we are not using file input *)

        IF usingFile
            THEN
                OpenInput;
            END;
        usingFile := usingFile AND (OpenResult () = opened);

        (* now get the data, either from file or keyboard *)
        REPEAT (* process until bad data *)
            IF NOT usingFile (* prompts printed only if no file used *)

```

```

    THEN
        WriteString ("Enter the year number here ====>");
    END;
ReadCard (year);
readOK := (ReadResult() = allRight);
SkipLine;
IF readOK
    THEN
        IF NOT usingFile
            THEN
                WriteLn;
                WriteString ("Enter the month (1 - 12) here ====>");
            END;
        END;
ReadCard (month);
readOK := (ReadResult() = allRight);
SkipLine;
IF readOK
    THEN
        IF NOT usingFile
            THEN
                WriteLn;
                WriteString ("Enter the day number here ====>");
            END;
        END;
ReadCard (day);
readOK := (ReadResult() = allRight);
SkipLine;
IF readOK
    THEN
        (* next section computes number of days *)
        leap:= (year MOD 400 = 0)
            OR ((year MOD 4 = 0) AND (year MOD 100 # 0));
        adjMonth := month + 12 * (( 12 - month) DIV 10);
        daysSinceMarch := TRUNC(30.6 * FLOAT (adjMonth - 3) + 0.5);
        result := (59 + daysSinceMarch + day);
        IF result -->> 2)
            THEN
                INC (result);
            END;

        (* Output the result in the required form *)
        WriteCard (year, 4);
        WriteCard (month, 3); (* ensure one space between *)
        WriteCard (day, 3);
        WriteString (" is day number ");
        WriteCard (result, 4);
        WriteString (" in that year.");
        WriteLn;
        WriteLn;
    END; (* if *)
UNTIL NOT (readOK);

```



```

(* close the file if it was open *)
IF usingFile
THEN
    CloseInput;
ELSE
    WriteLn;
END;
WriteString ( "Do you wish to do another group? Y or N ==

```

Notice the two places at which input errors are trapped and the series terminated.

6.3.5 Writing Special Characters To a Terminal

It is sometimes important to be aware that certain *control characters* have rather standard actions when sent to a terminal device. These considerations apply regardless of whether the I/O library is of the classical style or is ISO standard, and regardless of whether Modula-2 is used, or some other programming notation. Rather, the actions of these characters when they arrive at an output device like a screen or other terminal, is *hard wired* into that device and not under the control of the program or the user. There may be others as well, but the following are likely (but not guaranteed) to have the effect indicated:

1. CHR (30), also called EOL (End-Of-Line)--Sets the writing position to the beginning of the next line. (Use *WriteLn* instead.)
2. CHR (13), also called CR (Carriage Return)--Sets the writing position to the beginning of the current line.
3. CHR (10), also called LF (LineFeed)--Moves the writing position down to the same position on the next line.
4. CHR (8), also called BS (Backspace)--Moves the writing position back by one position leaving the character backspaced over intact.
5. CHR (127), also called DEL (Delete)--Backspaces and deletes the character.
6. CHR (12), also called FF (Form Feed)--Clears the screen, setting the cursor in the top left corner.

NOTE: On some terminals and on many printers an EOL, CR, CR/LF or LF/CR all do the same thing as a carriage return followed by a line feed. It is because of this variation from one system to another that the ISO standard regards the reaching of the end-of-line as a *state* or condition rather than as the reading of a particular character. (It may be more than one character in character based systems.)

Other control codes may be implemented on some terminals, but the above are the most common. All of these can be defined as constants, or if a module such as *ASCII* is provided (See [Appendix 4.7](#)), the names (EOL, CR, etc.) may be imported instead. Some such modules may spell these with lower case letters.

6.4 The Standard Library (2)--Mathematical Functions

Every implementation of Modula-2, and of every computer language that is to be used in a scientific or academic environment, must also provide a number of standard mathematical functions. A name and contents of the module that contains these were suggested by Wirth--he called it *MathLib0*. However, in some versions, vendors called it either *MathLib* or *MathLib1*, and in a few the procedure names started with an uppercase letter. The ISO standard version is called *RealMath*. It makes available the two real constants:

CONST

```
pi    = 3.1415926535897932384626433832795028841972;  
exp1  = 2.7182818284590452353602874713526624977572;
```

to as many decimal places as the implementation allows. In addition, the following sections detail the basic functions that are exported by *RealMath*, with their parameter lists, some comments and examples. (A few have been used before.)

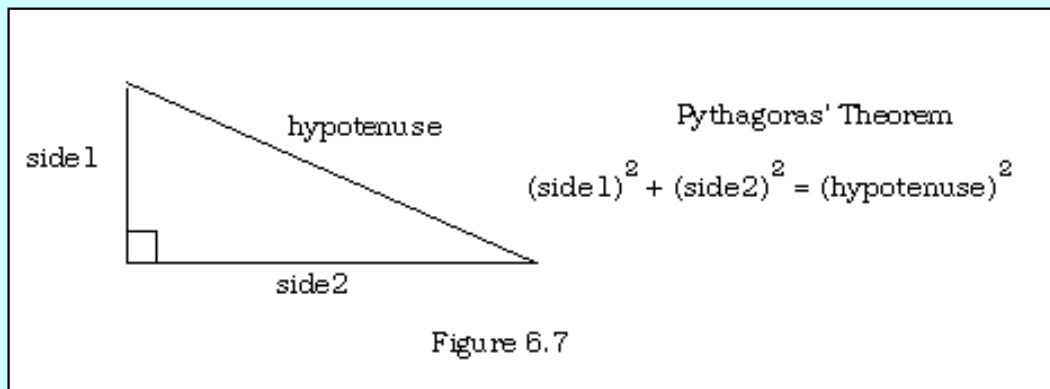
6.4.1 Square Root

PROCEDURE sqrt (x : **REAL**) : **REAL**;

This function procedure returns the square root of a positive real number. Naturally, there will be an error generated if one attempts to take the square root of a negative number.

Problem:

Given two sides of a right triangle, compute the hypotenuse.



Refinement:

```
Ask the user for two numbers, the sides' lengths  
  Read each into a real variable  
Compute the hypotenuse using Pythagoras' Theorem  
  hyp = sqrt (a * a + b * b)  
Print out the result
```

Data Table:

Variables:

Two reals to hold the side lengths, one for the hypotenuse

Imports

WriteString, WriteLn, ReadReal, WriteReal, sqrt

Code:

```
MODULE Pythagoras;  
  
(* Written by R.J. Sutcliffe *)  
(* using ISO Modula-2 *)  
(* to illustrate RealMath.sqrt *)  
(* last revision 1993 03 01 *)  
  
FROM STextIO IMPORT  
    WriteString, WriteLn, SkipLine, ReadChar;  
FROM SRealIO IMPORT  
    ReadReal, WriteFixed;  
FROM SIOResult IMPORT  
    ReadResult, ReadResults;  
FROM RealMath IMPORT  
    sqrt;  
  
VAR  
    side1, side2, side3 : REAL;  
    key : CHAR;  
  
PROCEDURE GetReal (VAR numToGet : REAL);  
VAR  
    tempResult : ReadResults;  
  
BEGIN  
    REPEAT  
        WriteString ("Please type in a real number ===");  
        ReadReal (numToGet);  
        tempResult := ReadResult ();  
        SkipLine; (* swallow line marker *)  
        WriteLn;  
    UNTIL tempResult = allRight;  
END GetReal;  
  
BEGIN    (* main *)  
    WriteString ("What is the first side length? ");  
    WriteLn;  
    GetReal (side1);  
    WriteString ("What is the second side length? ");  
    WriteLn;  
    GetReal (side2);  
    side3 := sqrt (side1 * side1 + side2 * side2);  
    WriteString ("The hypotenuse is ");
```

```

WriteFixed (side3, 2, 0);
WriteString (" units long.");
WriteLn;
WriteString ("Press any key to continue");
ReadChar (key);
END Pythagoras.

```

Results of a run:

```

What is the first side length?
Please type in a real number ==> 21.0
The hypotenuse is 29.00 units long.

```

6.4.2 Exponential and Logarithmic Functions

It was observed in [section 4.9](#) that an amount A placed at compound interest *rate* for *time* years would grow to $A(1 + \text{rate})^{\text{time}}$. Notice that if the interest is compounded n times a year, the amount will be higher than if it is compounded annually, even though the interest rate at each compounding period must be divided by n . For instance, the first example in that section found that \$1000 at 6% compounded for 10 years would grow to 1790.85. If the interest is computed monthly, the rate at each application of interest is .06/12, or .005, and the number of applications becomes $10 \cdot 12$ or 120. Modifying the formula above yields $A(1 + \text{rate}/n)^{n \cdot \text{time}}$ and in this case the \$1000 grows to \$1819.40. If the compounding is done daily, the result is 1822.20, not much of a difference from monthly compounding. One might ask whether continuing to increase the number of compounding periods indefinitely would yield an indefinitely large (infinite) amount, or if there is some limit beyond which the amount will not grow.

The latter turns out to be the case. To see that this is so, consider a principal amount of \$1.00 placed at 100% for a year, and increase the number of compounding periods indefinitely. In mathematical terms, this computes:

$$\lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n$$

As the number of periods grows and the interest rate per period shrinks, this converges to a definite limit at about 2.7182818. (It has a non-repeating, non-terminating decimal representation.) This number, denoted e arises naturally in a variety of situations in mathematics, and mathematical libraries provide the exponential function to compute $y = e^x$. The inverse function, to compute the exponent, or logarithm of x given the number y is the logarithm function $x = \ln(y)$. These were mentioned in [section 4.5](#) in connection with writing the function procedure *APowerB* and are found in *RealMath* as:

```
PROCEDURE exp (numb : REAL) : REAL;
```

and

```
PROCEDURE ln (numb : REAL) : REAL;
```

In addition, *RealMath* exports the related function procedure:

```
PROCEDURE power (base, exponent : REAL) : REAL;
```

NOTE: Some non-standard mathematical library modules also export some or all of the following related function procedures:

For Base 10 Logarithms:

```
PROCEDURE log (numb : REAL) : REAL;
```

```
PROCEDURE TenToX (numb : REAL) : REAL;
```

For other power and magnitude operations:

```
PROCEDURE ipower (numb1 : REAL; numb2 : INTEGER) : REAL;  
  (* Both return numb1 to the numb2 power *)
```

```
PROCEDURE Magnitude (numb : REAL) : INTEGER;  
  (* returns the order of magnitude of numb, namely the largest integer less  
  than or equal to the scale factor or log10 of numb *)
```

One application of the logarithmic and exponential functions is to compute radioactive (and other) decay processes. Under normal conditions, a quantity of radioactive material decays over time according to the formula

$$A = A_0 e^{kt}$$

where A_0 is the amount of the substance present at time zero, A is the amount at the time being examined, t is the elapsed time in appropriate units, and k is a constant that is a property of the substance.

In the standard literature, one often finds the constant k expressed indirectly as the half-life, that is, the time it would take for half of any given quantity of the substance to decay.

Problem:

A lab is gathering data from experiments done on radioactive samples and determines experimentally the amount of a radioactive substance present at time zero and also at some subsequent time. Write a program to calculate the half-life of the substance from this data. (This is often one way of identifying an unknown radioactive material.)

Discussion:

The formula $A = A_0 e^{kt}$ may be rewritten as

$$\frac{A}{A_0} = e^{kt}$$

and, upon taking natural logarithms on both sides and solving for k , one obtains

$$k = \frac{\ln\left(\frac{A}{A_0}\right)}{t}$$

In the case where half of the material is supposed to have decayed, the right hand side of the latter formula becomes

$$k = \frac{\ln(0.5)}{t}$$

or, solving for t ,

$$t = \frac{\ln(0.5)}{k}$$

With these variations on the initial formula, all the tools are at hand to write the code to do the computation.

Code:

```
MODULE HalfLife;  
  
(* Written by R.J. Sutcliffe *)  
(* using ISO Modula-2 *)  
(* to illustrate RealMath.ln *)  
(* last revision 1994 08 30 *)  
  
FROM STextIO IMPORT
```

```

    WriteString, WriteLn, ReadChar, SkipLine;
FROM SRealIO IMPORT
    ReadReal, WriteFixed;
FROM SIOResult IMPORT
    ReadResult, ReadResults;
FROM RealMath IMPORT
    ln;

PROCEDURE GetReal (VAR numToGet : REAL);
VAR
    tempResult : ReadResults;

BEGIN
    REPEAT
        WriteString ("Please type in a real number ===");
        ReadReal (numToGet);
        tempResult := ReadResult ();
        SkipLine; (* swallow line marker *)
        WriteLn;
    UNTIL tempResult = allRight;
END GetReal;

VAR
    initialAmount, laterAmount, timePassed,
    constant, halfLife : REAL;
    key: CHAR;

BEGIN
    WriteString ("What was the initial amount? ");
    GetReal (initialAmount);
    WriteString ("How much time elapsed til the second reading? ");
    GetReal (timePassed);
    WriteString ("And, how much material was left then? ");
    GetReal (laterAmount);
    constant := ln (laterAmount / initialAmount) / timePassed;
    halfLife := ln ( 0.5) / constant;
    WriteString ("The half life of this material is ");
    WriteFixed (halfLife, 6, 10);
    WriteLn;
    WriteString ("Press a key to continue ==");
    ReadChar (key);
END HalfLife.

```

Trial Run:

```

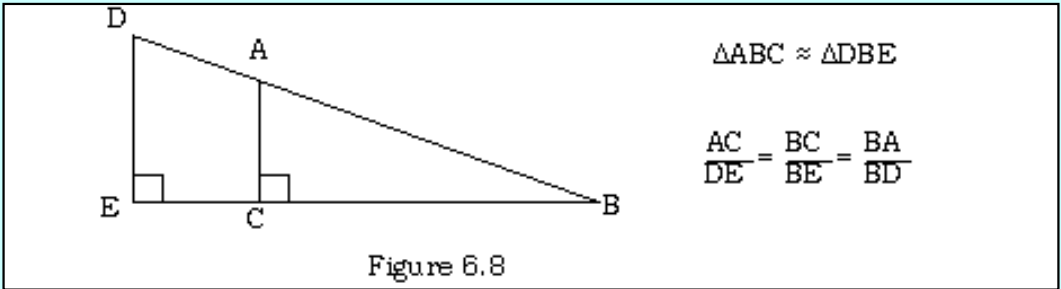
What was the initial amount? Please type in a real number ==> 10.0
And, how much material was left then? Please type in a real number ==>

```

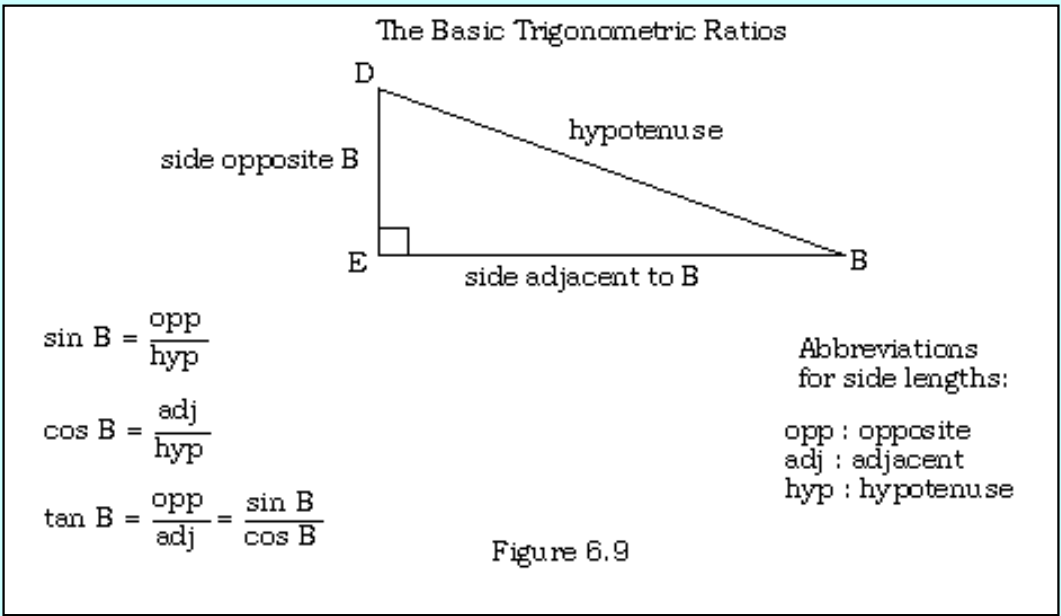
Naturally, the units for the half life will be the same as those of the elapsed time given by the user. Exponential growth works in much the same way, except that the constant k is positive instead of negative.

6.4.3 Trigonometric Functions

It is discovered in elementary geometry that in two similar triangles, the sides are in proportion.



In particular, for right triangles, one may relate these fixed ratios to a particular acute angle, such as angle B in figure 6.8. When this is done, and the right triangle labelled as in figure 6.9, the trigonometric ratios are defined as follows:



The symbols *sin*, *cos*, and *tan* are themselves abbreviations for *sine*, *cosine*, and *tangent*, respectively. Auxiliary to these three are their inverse functions *arcsin*, *arccos*, and *arctan* for producing an angle given one of the fixed ratios. Wirth suggested that *MathLib0* provide only three of these functions; the minimum necessary for work in trigonometry, but the ISO library *RealMath* supplies all six. They are:

```
PROCEDURE sin (x : REAL) : REAL;  
PROCEDURE cos (x : REAL) : REAL;  
PROCEDURE tan (x : REAL) : REAL;  
PROCEDURE arcsin (x : REAL) : REAL;  
PROCEDURE arccos (x : REAL) : REAL;  
PROCEDURE arctan (x : REAL) : REAL;
```

- NOTES:** 1. The first three require the angle to be in radians, and return the sine, cosine, and tangent, respectively, of the angle supplied. The last three takes the sine, cosine, and tangent of an angle and returns the principal value of the angle measure (in radians). For arcsine and arctangent, the values returned are in the range $-\pi/2 < \phi < \pi/2$. For arccosine, the values returned are in the range $0 < \phi < \pi$. (Recall that one radian is $180/\pi$ degrees.)
2. Many non-standard implementations are much less generous in their supply of trigonometric functions than this, and may omit as many as three of these.
3. While angles larger than 2π (about 6.28) will work correctly, specific implementations may have a maximum argument for trigonometric functions that is much smaller than $\text{MAX}(\text{REAL})$.

Here is another useful procedure:

```
PROCEDURE degToRad (x : REAL) : REAL;  
  (* converts degrees to radians *)  
BEGIN  
  RETURN (x * pi / 180.0);  
END degToRad;
```

Basic trigonometric identities and formulas can be employed to extend the scope of the available mathematical functions.

Example:

Write a module that computes the area of any triangle given two adjacent sides and an included angle.

Discussion:

Consider the triangle ABC in figure 6.10 below. If the base b (here AC) and the altitude h (here BD) is known, the area of the triangle is given by

$$S = .5bh$$

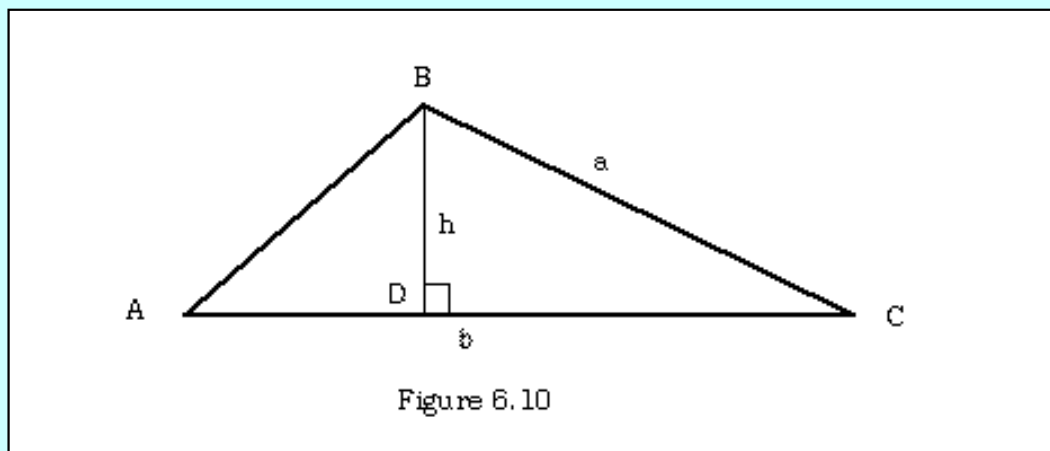
However, since BD is the side opposite angle C, in the right triangle BCD, and $\sin(C) = BD/BC$,

$$h = BC \sin(C) = a \sin(C).$$

That is, provided the given data consists of two sides a and b of the triangle and an included angle C, its area can be computed by using the formula

$$s = .5ab \sin(C)$$

which reduces to the original formula when C is a right angle, because the sine of 90° is one.



Code:

```
MODULE TriArea;  
  
(* Written by R.J. Sutcliffe *)  
(* using ISO Modula-2 *)  
(* to illustrate RealMath.sin *)  
(* last revision 1993 03 01 *)  
  
FROM STextIO IMPORT  
  WriteString, WriteLn, ReadChar, SkipLine;
```



```

FROM SRealIO IMPORT
    ReadReal, WriteFixed;
FROM SIOResult IMPORT
    ReadResult, ReadResults;
FROM RealMath IMPORT
    sin, pi;

PROCEDURE GetReal (VAR numToGet : REAL);
(* prompts for a real number, reads it, loops until a correct one is typed, swallows
the end-of-line state and returns the number read *)

VAR
    tempResult : ReadResults;

BEGIN
    REPEAT
        WriteString ("Please type in a real number ===");
        ReadReal (numToGet);
        tempResult := ReadResult ();
        SkipLine; (* swallow line marker *)
        WriteLn;
    UNTIL tempResult = allRight;
END GetReal;

PROCEDURE degToRad (x : REAL) : REAL;
(* converts degrees to radians *)
BEGIN
    RETURN (x * pi / 180.0);
END degToRad;

VAR
    angleC, sideA, sideB, area: REAL;
    key : CHAR;

BEGIN (* main *)
    (* obtain triangle data *)
    WriteString ("What is the first side length? ");
    GetReal (sideA);
    WriteString ("What is the second side length? ");
    GetReal (sideB);
    WriteString ("Now, what is the included angle ");
    WriteString ("in degrees? ");
    GetReal (angleC);
    (* do calculation *)
    angleC := degToRad (angleC);
    area := 0.5 * sideA * sideB * sin (angleC);
    (* inform user of result *)
    WriteString ("The area is ");
    WriteFixed (area, 5, 0);
    WriteString (" square units ");
    WriteLn;
    WriteString ("Press a key to continue ==");
    ReadChar (key);

```

```
END TriArea.
```

First run:

```
What is the first side length? Please type in a real number ==> 8.0
Now, what is the included angle in degrees? Please type in a real number ==> 10.0
What is the second side length? Please type in a real number ==> 90.0
The area is 100.0000 square units
```

NOTE: Some nonstandard versions of the math library module use *atan* instead of *arctan* and may not export *asin*, or *acos*. Others provide the hyperbolic functions *sinh*, *cosh* and *tanh*.

6.4.4 Conversions

RealMath also offers the useful conversion function:

```
PROCEDURE round (x: REAL): INTEGER;
```

which rounds off a real to the nearest integer.

Two function procedures that *may* be in the traditionally named *MathLib0* that are of more specialized interest are the following integer/real conversions.

```
PROCEDURE real (m : INTEGER) : REAL;
PROCEDURE entier (x : REAL) : INTEGER;
```

The first of these is essentially the same as `FLOAT` except that it only operates on the type `INTEGER` (and assignment compatible cardinals.) This is important in older versions of Modula-2 where `FLOAT` works only on `CARDINAL` (not on either one as in the ISO standard.)

The second one is sometimes called the greatest integer function. It takes a real argument, and returns the greatest integer less than or equal to the real. Note that this is not the same as `TRUNC` even in those versions where both can return integers.

Compare the following:

`entier (5.7)` produces 5 and `TRUNC (5.7)` also produces 5, but

`entier (-4.3)` produces -5 while `TRUNC (-4.3)` yields -4

That is, for positive numbers, the result is the same, but for negative ones, it will be different, because in those cases `entier` gives the nearest integer less than the argument and `TRUNC` simply "hacks off" the decimal fractional portion. Notice that an order of magnitude function would be written using `entier` rather than `TRUNC`.

```
PROCEDURE Magnitude (num : REAL) : INTEGER;
(* uses non-ISO functions *)
```

```
BEGIN
  RETURN (entier (ln (num) / ln (10.0) ));
END Magnitude;
```

This procedure returns -6 when given 4.5E-6 and 2 when given 3.8E2, having computed the base ten logarithms as approximately -5.346 and 2.579 respectively. Notice that a base ten logarithm of a number (or one in any other base) is computed by dividing the natural logarithm of the number by the natural logarithm of the base, for if

$x = \log_{10} y$ then $10^y = x$,

so that, taking natural logarithms on both sides yields

$y \ln(10) = \ln(x)$

and therefore
 $y = \ln(10)/\ln(x)$
as used in the procedure.

6.4.5 Summary of RealMath

```
DEFINITION MODULE RealMath;

CONST
  pi    = 3.1415926535897932384626433832795028841972;
  exp1  = 2.7182818284590452353602874713526624977572;

PROCEDURE sqrt (x: REAL): REAL;
PROCEDURE exp (x: REAL): REAL;
PROCEDURE ln (x: REAL): REAL;
PROCEDURE sin (x: REAL): REAL;
PROCEDURE cos (x: REAL): REAL;
PROCEDURE tan (x: REAL): REAL;
PROCEDURE arcsin (x: REAL): REAL;
PROCEDURE arccos (x: REAL): REAL;
PROCEDURE arctan (x: REAL): REAL;
PROCEDURE power (base, exponent: REAL): REAL;
PROCEDURE round (x: REAL): INTEGER;
END RealMath.
```

6.4.6 Other Mathematical functions

A wide variety of other function procedures and error handling may be provided in some auxiliary modules associated with *RealMath*, or, in non-standard versions, added to *MathLib0*.

The ISO standard libraries, and some non-standard versions as well, include a second module that is identical to *RealMath* but that acts on and returns long types.

```
DEFINITION MODULE LongMath;

CONST
  pi    = 3.1415926535897932384626433832795028841972;
  exp1  = 2.7182818284590452353602874713526624977572;

PROCEDURE sqrt (x: LONGREAL): LONGREAL;
PROCEDURE exp (x: LONGREAL): LONGREAL;
PROCEDURE ln (x: LONGREAL): LONGREAL;
PROCEDURE sin (x: LONGREAL): LONGREAL;
PROCEDURE cos (x: LONGREAL): LONGREAL;
PROCEDURE tan (x: LONGREAL): LONGREAL;
PROCEDURE arcsin (x: LONGREAL): LONGREAL;
PROCEDURE arccos (x: LONGREAL): LONGREAL;
PROCEDURE arctan (x: LONGREAL): LONGREAL;
PROCEDURE power (base, exponent: LONGREAL): LONGREAL;
PROCEDURE round (x: LONGREAL): INTEGER;

END LongMath.
```

This second module (along with the built-in type LONGREAL itself) are provided because many systems have two or more real types of different precisions. ISO Modula-2 defines the precision of LONGREAL to be equal to or greater than that of REAL. Thus, if there is only one underlying type in the actual system being used, the programmer may use either or both of the Modula-2 logical types to refer to this actual type. Both *RealMath* and *LongMath* also include an error enquiry function not listed here but the use of this will be postponed to a later chapter.

It should be noted that the name and the contents of both modules in non-standard versions, can vary widely from one implementation to another. For further information, see the [Appendix](#) on standard module definitions or consult the manuals that are available with the system.

[Contents](#)

6.5 Starting Your Own Libraries

This section details one of the most powerful and unique features of Modula-2--how to create new library modules from which to import entities such as data types or procedures as desired in a particular program.

Library modules are different from program modules and there are special rules for writing and compiling them. That is, one cannot simply compile an ordinary program module, place it on a disk, and import from it. There are two main reasons for making this distinction between program and library modules.

1. There has to be some way for potential client modules to determine what entities are available for import. (Some procedures may be for the internal use of the library module only and not be appropriate for outsiders to know anything about.)
2. Client programs (and programmers for that matter) do not need to know what is the code by which some library procedure was implemented in order to use it. All that is required for use is the syntax. Moreover, the compiler also needs to have available only the syntax; the code can be collected and bound together with the program code by the linker at a later time.

Modula-2 addresses these considerations by partitioning the source material into two separate modules, one containing only the definitions (syntax) of the library entities, and the other containing the implementation (actual code) of those definitions.

These are called the DEFINITION and IMPLEMENTATION parts of a library module respectively (two new reserved words) and are indicated by having the appropriate keyword precede the word MODULE in the heading. Here is a simple example to illustrate how to set this all up.

Problem:

Write a library module to supplement *RealMath*.

Solution:

One would first have to decide what to include in such a module. For the sake of simplicity, this example will have only base ten logarithms, and a simple constant--the square root of two.

First write out a text file such as the following:

```
DEFINITION MODULE MyRealMath;  
  
CONST  
    root2 = 1.414213562;  
  
PROCEDURE log (x : REAL) : REAL;  
  
END MyRealMath.
```

This text file is saved and is then compiled. Since this is not a program module (it does not do anything) no code is produced. Instead, compilation yields a *symbol file* that can be checked whenever any other code is compiled that makes use of the definition. In fact, the definition part of a module is not allowed to contain a body--all code goes elsewhere. A definition module may have imports, such as data types, but these should only be what it uses itself,

perhaps by mentioning them in parameter lists.

The actual code for the procedures is embodied in a separate part of the library module, and stored in a different file. It is a good strategy to retain the definitions in the program editor, make such changes as needed, and then save the results as the following:

```
IMPLEMENTATION MODULE MyRealMath;  (* note same name *)

FROM RealMath IMPORT
    ln;

PROCEDURE log (x : REAL) : REAL;
(* returns the base 10 logarithm of x *)
BEGIN
    RETURN (ln (x)) / (ln (10.0) )
END log;

END MyRealMath.
```

Once this source file has been created and saved, it too is compiled, and the corresponding code file is generated. This step *must* be done after the definition module has been compiled to a symbol file, for the symbol file contains information that the compiler requires when building the code for the implementation part of the module. At this point a client program can be written that uses this library module in exactly the same way as it would a vendor supplied library. That is, one can now write:

```
MODULE Client;

FROM MyRealMath IMPORT
    root2, log;
```

and then write code making use of the imported items in the normal way.

In order to manage all the files that are needed to store the text and compiled versions of this code, most operating systems employ standard extensions to the basic name. These are typically some variation of the following:

File Contents	Typical File Names
text source of a program module Myfile	Myfile.MOD
compiled code of the program Myfile	Myfile.MOD.o or Myfile.OBM
text source of definition module Mylib	Mylib.DEF
symbol file compiled from Mylib	Mylib.SBM or Myfile.SYM
text source of implementation module Mylib	Mylib.MOD
compiled code of Mylib	Mylib.MOD.o or Mylib.OBM
linked executable final program	Myfile or Myfile.EXE

NOTES: 1. The syntax of implementation modules is identical to that of program modules, except that they may not contain export lists (these will be covered in [section 10.5](#)).

2. The parameter lists of their procedures should *exactly* repeat the ones declared in the corresponding definition module. (Although the parameters names may differ between definition and implementation, the types must exactly match).

3. Anything declared in the definition module is available to the implementation module, (thus, *root2* can be used in the implementation of *MyRealMath* without further mention) but anything just imported into the definition module is not; it

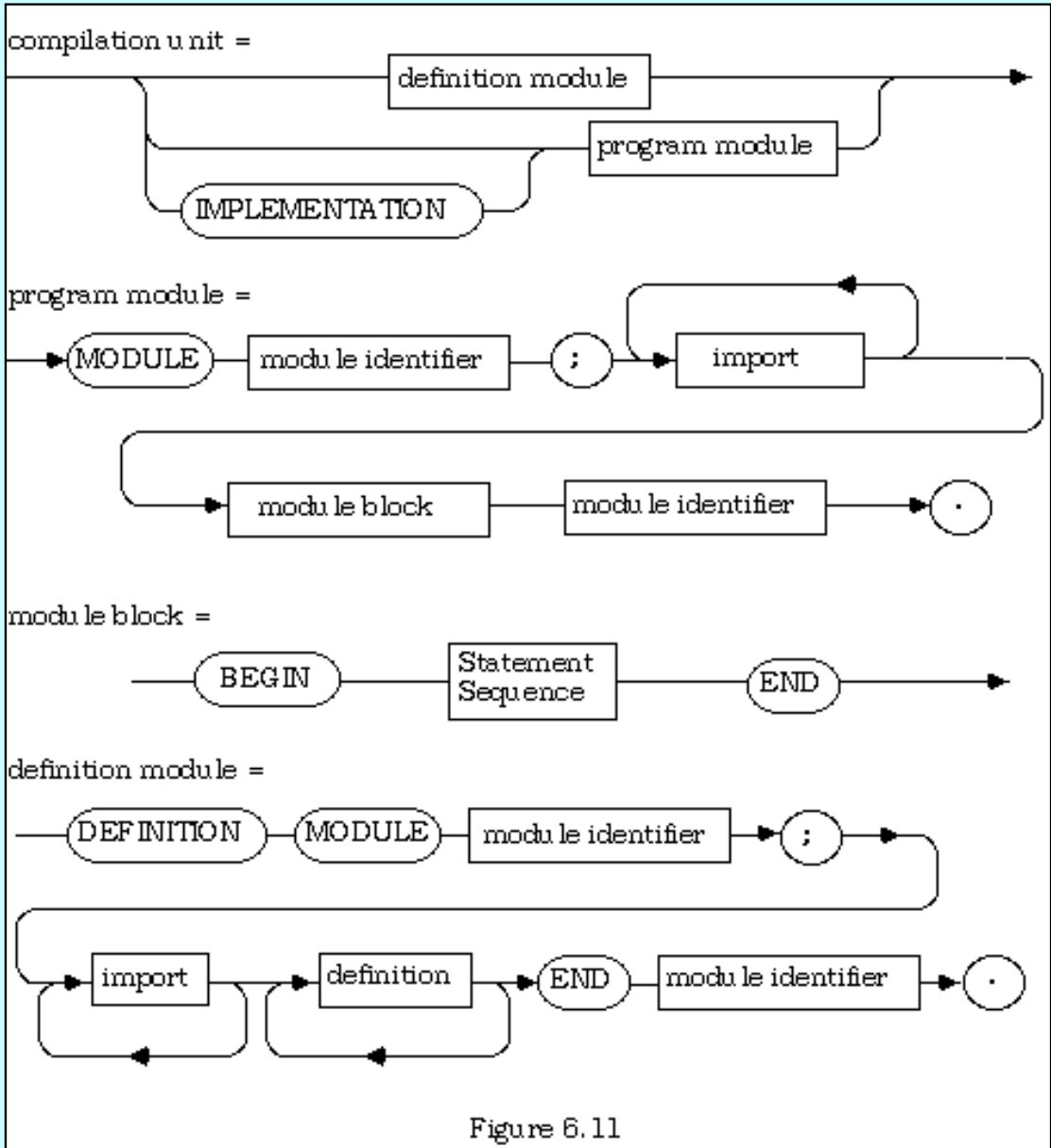
must be imported again into the implementation module if it is to be used there.

4. Like program modules, the implementation part of a library module may have a body. This body (when it exists) is executed at the point in the client program where the imports from the library are done. This is sometimes used for initializing variables needed for the correct operation of the module, but will not be employed in any examples until later in the text.

When a client program that does imports is compiled, only the syntax of the imported entities needs to be looked up. This can be done with the symbol file, so only the definition part of the library module needs to be compiled at this point. It is not until the system linker is invoked to tie all the code segments together into a single executable application that the output from compiling the implementation part is required. This allows the design of the program and its library support to be conducted in one phase of the job, and all the code implementation to be postponed until later.

The separation of the definition from the implementation is called module decoupling.

In all, Modula-2 has three kinds of modules that can be compiled. Each is termed a *compilation unit* and their general relationship and syntax is summarized in the diagram below. Note that the diagram does not say that an implementation module contain something that *acts* like a program module, but something that *looks* (syntactically) the same as a program module.



6.5.1 Modules and User-Defined Data Types

A common (indeed the most common) use of library modules is to encapsulate the representation of some abstract data type along with the procedures necessary to manipulate items of this type. What follows is a simple example of this technique. Another will be found in section 6.9, and several more throughout later chapters.

Consider the abstract data type *fraction*. Mathematically, items of this type are represented as $\frac{a}{b}$ where a and b are integers and $b \neq 0$.

The fraction bar is just a convenient symbol, and the representation could just as easily be done with an ordered pair (a, b) where the first component is the numerator and the second is the denominator. With this notation, and if (a, b) and (c, d) are fractions, some basic operations can be defined as follows:

Negation:

The opposite of a fraction can be found by negating the numerator alone. That is, $-(a, b)$ is just $(-a, b)$. It is a common error on the part of elementary school students to negate *both* the numerator and the denominator. That answer, of course, is actually equal to the original fraction.

Inverse:

$1/(a, b)$, that is $\frac{1}{\frac{a}{b}}$, can be inverted just by writing (b, a) , that is, $\frac{b}{a}$ at least, provided that $a \neq 0$ also.

Multiplication:

$(a, b) \times (c, d)$ is calculated by multiplying the numerators and denominators together, respectively. This yields (ac, bd) .

Addition:

To compute $(a, b) + (c, d)$, a common denominator bd is employed. The numerator can then be expressed as $ad + bc$. The result: $(ad + bc, bd)$. The reader should verify this by writing it in the more conventional notation.

At this point, subtraction and division can be written in terms of adding a negation and multiplying by an inverse, respectively. Another operation on fractions is reducing them to lowest terms. There is a choice here; this could be provided to the client of the library module, or it could be retained inside the implementation module and all answers automatically reduced to lowest terms. In the initial implementation that follows, it will be left out altogether, and the decision deferred to the reader in the exercises.

What will not be left out, however, even though one might argue that it is not necessary, is a procedure to assign two integers to an item of the type *fraction*. Since the representation will simply be

TYPE

```
Fraction = ARRAY [1 .. 2] OF INTEGER;
```

and assuming that *frac* is of type *Fraction*, one could do assignment transparently by having a client program contain the lines

```
frac [1] := a;
frac [2] := b;
```

without a separate procedure. However, it is best not to *use* the implementation details of the type in client programs, but to treat the data type abstractly at all times. Indeed, a means will be shown later in the text to deliberately hide the fact that a *Fraction* is implemented as an array at all. This will actually prevent client programs from knowing any details about the representation, and force them to access items of the abstract type solely through the provided procedures. With the means available so far, this abstract view cannot be enforced, but it ought to be disciplined. This is done to reduce the possibility of errors being introduced by client programs that manipulate parts of data items

incorrectly or inappropriately.

NOTE: It is incumbent on the writer of a library module encapsulating an abstract data type to provide all necessary services for that data type so as to eliminate any necessity for client programs to attempt to use items of the type in a transparent manner.

In order to be consistent with this design philosophy, procedures will also have to be included to return the numerator or denominator of a fraction without having a client program look at its components directly. These discussions lead to the following:

DEFINITION MODULE Fractions;

TYPE

Fraction = **ARRAY** [1 .. 2] **OF** **INTEGER**;

(* the first component is the numerator; the second the denominator *)

PROCEDURE Assign (num, denom : **INTEGER**) : Fraction;

(* Pre: denom is not equal to zero

Post: the fraction returned has num as numerator and denom as denominator *)

PROCEDURE Numerator (x : Fraction) : **INTEGER**;

(* Pre: none

Post: the numerator of the fraction is returned *)

PROCEDURE Denominator (x : Fraction) : **INTEGER**;

(* Pre: none

Post: the denominator of the fraction is returned *)

PROCEDURE Neg (x : Fraction) : Fraction;

(* Pre: none

Post: the fraction returned has the numerator negated *)

PROCEDURE Inv (x : Fraction) : Fraction;

(* Pre: the numerator of x is not equal to zero

Post: the fraction returned has numerator and denominator swapped*)

PROCEDURE Add (x, y : Fraction) : Fraction;

(* Pre: the fractions are initialized

Post: the fraction returned is the sum x plus y *)

PROCEDURE Sub (x, y : Fraction) : Fraction;

(* Pre: the fractions are initialized

Post: the fraction returned is the difference x minus y *)

PROCEDURE Mul (x, y : Fraction) : Fraction;

(* Pre: the fractions are initialized

Post: the fraction returned is the product of x and y *)

PROCEDURE Div (x, y : Fraction) : Fraction;

(* Pre: the fractions are initialized and the numerator of y is not equal to 0

Post: the fraction returned is the quotient of x by y *)

END Fractions.

What follows is only an initial implementation of this module. It can be finished and tested by the reader at her leisure once decisions have been made on such matters as how to handle reducing fractions.

```
IMPLEMENTATION MODULE Fractions;
```

```
PROCEDURE Assign (num, denom : INTEGER) : Fraction;
```

```
VAR
```

```
    temp : Fraction;
```

```
BEGIN
```

```
    temp [1] := num;
```

```
    temp [2] := denom;
```

```
    RETURN temp;
```

```
END Assign;
```

```
PROCEDURE Numerator (x : Fraction) : INTEGER;
```

```
BEGIN
```

```
    RETURN x [1];
```

```
END Numerator;
```

```
PROCEDURE Denominator (x : Fraction) : INTEGER;
```

```
BEGIN
```

```
    RETURN x [2];
```

```
END Denominator;
```

```
PROCEDURE Neg (x : Fraction) : Fraction;
```

```
BEGIN
```

```
    x [1] := -x [1];
```

```
    RETURN x;
```

```
END Neg;
```

```
PROCEDURE Inv (x : Fraction) : Fraction;
```

```
VAR
```

```
    temp : INTEGER;
```

```
BEGIN;
```

```
    temp := x [1];
```

```
    x [1] := x [2];
```

```
    x [2] := temp;
```

```
    RETURN x;
```

```
END Inv;
```

```
PROCEDURE Add (x, y : Fraction) : Fraction;
```

```
VAR
```

```
    temp : Fraction;
```

```

BEGIN
    temp [1] := x [1] * y [2] + x [2] * y [1];
    temp [2] := x [2] * y [2];
    RETURN temp;
END Add;

PROCEDURE Sub (x, y : Fraction) : Fraction;

BEGIN
    RETURN Add (x, Neg (y) );
END Sub;

PROCEDURE Mul (x, y : Fraction) : Fraction;

BEGIN
    RETURN Assign (x [1] * y [1], x [2] * y [2]);
END Mul;

PROCEDURE Div (x, y : Fraction) : Fraction;

BEGIN
    RETURN Mul (x, Inv (y) );
END Div;

END Fractions.

```

As above, once this second part of the library module has been compiled, client program modules can be written that import one or more of these items from the new library and then employ them in writing code.

Some readers might be wondering at this point why one would bother implementing this data type at all when the type REAL is available to handle fractional quantities. There are two reasons:

1. The type REAL is intended to represent the set of real numbers. The type *Fraction* represents just the rational numbers. As these are mathematically distinct, it makes sense to represent them differently in a computing notation as well. Of course, items of the type REAL actually can only take on a finite number of values, as storage space for significant figures and exponents is limited, but this is an implementation problem and ought not to be allowed to obscure the abstraction of reals by REAL.
2. Not only is it the case that some reals cannot be represented in a given implementation of REAL, some rational numbers cannot be represented exactly either. A fraction such as $1/2$ may have an exact representation in a particular implementation of REAL, whereas the fraction $121/349$ may not. Both have exact representations in the abstract type *Fraction*, as does any rational whose numerator and denominator do not overflow the INTEGER type in the implementation.

It ought also to be observed that this example follows a different design methodology than any previously employed in the text, though it is one that has been hinted at in general terms ever since chapter one. Here, the designer perceives the need to have a certain type of data, designs a representation and decides on the procedures that will act upon it. After the design is complete and the resulting definition module compiled, the data type is implemented in a separately compiled implementation module. The text has not at this point even suggested what application (client) program might have given rise to the need to make use of this data type. While one must presume that such an application would in fact surface *before* the decision to implement the data type, it is important to note that creating the abstract data type *Fraction* and its operations, is a task that can be handled independently of the original problem, whatever it may have been.

Once having decided on the need for a particular data type, the programmer is free to concentrate on its implementation, temporarily setting aside the original problem that brought about the need in the first place. The procedure is always as in this section:

1. Decide on what the data type will be and how it will be represented.
2. Write down all the procedures (with their parameter lists and types) that will be needed to use that data type effectively.
3. Write out the definition module in the syntax of Modula-2 with all the necessary types, variables, and procedures that will be available for import by client programs. No code is written, only declarations. Compile this module first.
4. Implement the types and procedures in an implementation module that contains the details of all types, the code for all procedures (the headings of which must be as defined in step three), and assign values to any variables in the body of the module. Compile this module last. The corresponding symbol file of the definition module from step three must be on-line (probably in the same disk directory) when this module is compiled, because it will be checked for matching syntax and some important information copied for library control. See section 6.7.

This section closes with a definition of the design process that has been introduced here:

The method of design that begins with the specification of a data type and all its associated procedures, and then implements these--all independent of the application problems that spawned the need for the data type in the first place--is called object-oriented design.

Some programming notations are themselves designed in such a way as to encourage this kind of design, or even to enforce it. These are termed *object oriented languages*. Modula-2 can be used as a vehicle for object oriented design, but object oriented notations such as Smalltalk take this design methodology much further. Some versions of Modula-2 have object oriented extensions to the language. These will be discussed in a later chapter.

[Contents](#)

6.6 Handling Errors in Library Modules

There are a number of common methods of detecting and handling errors that may take place in some specialized section of the program, particularly while some library module code has control of the program execution. Some of the simpler ones are:

Precondition checking

This method relies on checking for potential error conditions before taking the action which might otherwise cause the error to arise. Code to compute the tangent of an angle by the formula $\tan(x) = \sin(x) / \cos(x)$ could possibly look something like this outline:

```
IF (abs ( cos (x)) "Please type in a real number ===");  
  ReadReal (numToGet);  
  tempResult := ReadResult ();  
  SkipLine; (* swallow line marker *)  
  WriteLn;  
UNTIL tempResult = allRight;
```

or, perhaps:

```
try := 1;  
REPEAT  
  WriteString ("Please type in a real number ===");  
  ReadReal (numToGet);  
  tempResult := ReadResult ();  
  SkipLine; (* swallow line marker *)  
  WriteLn;  
  IF tempResult # allRight  
    THEN  
      WriteString ("Error in input. Try again");  
      WriteLn;  
      INC (try);  
    END;  
UNTIL (tempResult = allRight) OR (try = 5);  
(* handle more than 5 bad tries here somewhere *)
```

Alternately, the same thing could be achieved, but with more pressure on the programmer to actually

check the result if a "success" variable is included as one of the parameters in the procedure. Since an actual parameter must be provided at each call, the variable cannot be simply ignored.

REPEAT

```
WriteString ("Type in a Real here = = ");  
ReadReal (theReal, allOK); (* this ReadReal has two parameters *)  
UNTIL allOK;
```

This type of error handling (in either style) can easily be added to library modules created by the programmer. The first can be achieved using an enumerated type with or without an enquiry function, or a boolean. The first value in the enumeration represents a no-error condition, and the others represent specific problems that have been encountered. For instance, the definition part of the module *Fractions* could have:

TYPE

```
FracStatus = (fracOk, undefined, divideByZero);
```

```
PROCEDURE FracState (): FracStatus;
```

The implementation module maintains a variable, say, *fracState* to store the last error state. With these in place, the procedure *Assign* would be written to set *fracState* to the value *fracOk* whenever the denominator being set was non zero, and to *undefined* if it were zero. The procedure *Div* would set *fracState* to *divideByZero* if the second parameter it were passed had a zero numerator, and to *fracOk* otherwise. The other procedures would not give rise to such errors and could all set *fracState* to *fracOk*. (An overflow in the integer type during *Add* or *Mul* could also lead to an error, but that would be more difficult to handle in this manner). If a client of this module wanted to determine the value of this variable after an operation it would use the procedure *FracState* to do so.

Automatic Error Handling

Methods that take this approach invoke some procedure that automatically handles errors whenever they take place, without any boolean or enumerated type having to be set or checked. A detailed discussion of such methods cannot be included here, but will be taken up later in the text. One uses procedure variables, and will be mentioned as an application of these once they are introduced. The other makes use of *exceptions*, and is only available to the programmer if some changes have been made to the Modula-2 notation itself. Such changes have been incorporated into the ISO standard for the notation, but are not available in classical Modula-2 as defined by Wirth. Exception handling will be covered in chapter 10. Which of the methods is best? Some would respond that this is merely a matter of taste and style, but others would say that the question is a fundamental one that goes to the heart of proper program design philosophy. This text takes the position that precondition checking is generally preferred, on the assumption that errors are better caught before they can happen, but that postcondition checking and exception handling are sometimes necessary. The most important thing is that all errors be caught and

handled in *some* way, and that whatever method is used be clearly documented in the definition and implementation parts of library modules and in client programs.

[Contents](#)

6.7 Modules and Design Considerations

In order to understand the points being made about design in this section, it is necessary to know something about what happens when the various kinds of modules are compiled--for each of the three compiles in a somewhat different way.

As previously noted, when a definition module is compiled, a *symbol file* of the various entities with their syntax is created so that compilations of client programs can reference the entities defined. At the same time, a unique identification key is generated by the compiler and placed in the symbol file. If that particular definition module is ever re-compiled for any reason, the new key generated will be different from the old one.

Exception: Some compilers ignore comments to the extent that changing only these does not produce a new key on recompilation. Do not count on this fact.

NOTE: The definition of Modula-2 does not *require* that the symbol file be separate from the definition module source code; an implementation might choose to keep the module key with the source and do nothing else. Then, the source would also be the symbol file.

When the implementation part of a library module is compiled, the disk is searched for the symbol file of the corresponding definition part. For the compilation to be successful, the syntax of the various entities must match. For instance, a procedure heading that is given in the definition part and then altered, say, as to the number and type of parameters, or omitted in the implementation, would cause compilation of the implementation to fail.

However, something else takes place at this time. The compiler also copies the previously generated key from the symbol file into the code file that is generated for the implementation part.

Likewise, when a program module is compiled, the syntax of program statements is checked against that defined in any modules being imported from, and the names and keys from all the definition parts of the library modules are collected and stored in the object code file from the program module. When the code files are all linked, the implementation modules are all read, and the keys are checked. If there are any differences, the system will refuse to link together the (presumed) incompatible modules, and the program will not link or run.

It must be understood that this checking of version compatibility is not implementation dependent, but is part of the very warp and woof of the notation. All Modula-2 systems must implement this library control behaviour, though the way in which they put it into practice may vary slightly.

The result is that compiling a program depends only on the definition modules for its success, but linking and running it requires that the implementation modules not only have been coded and compiled by this time, but that their keys agree with those of their corresponding definitions.

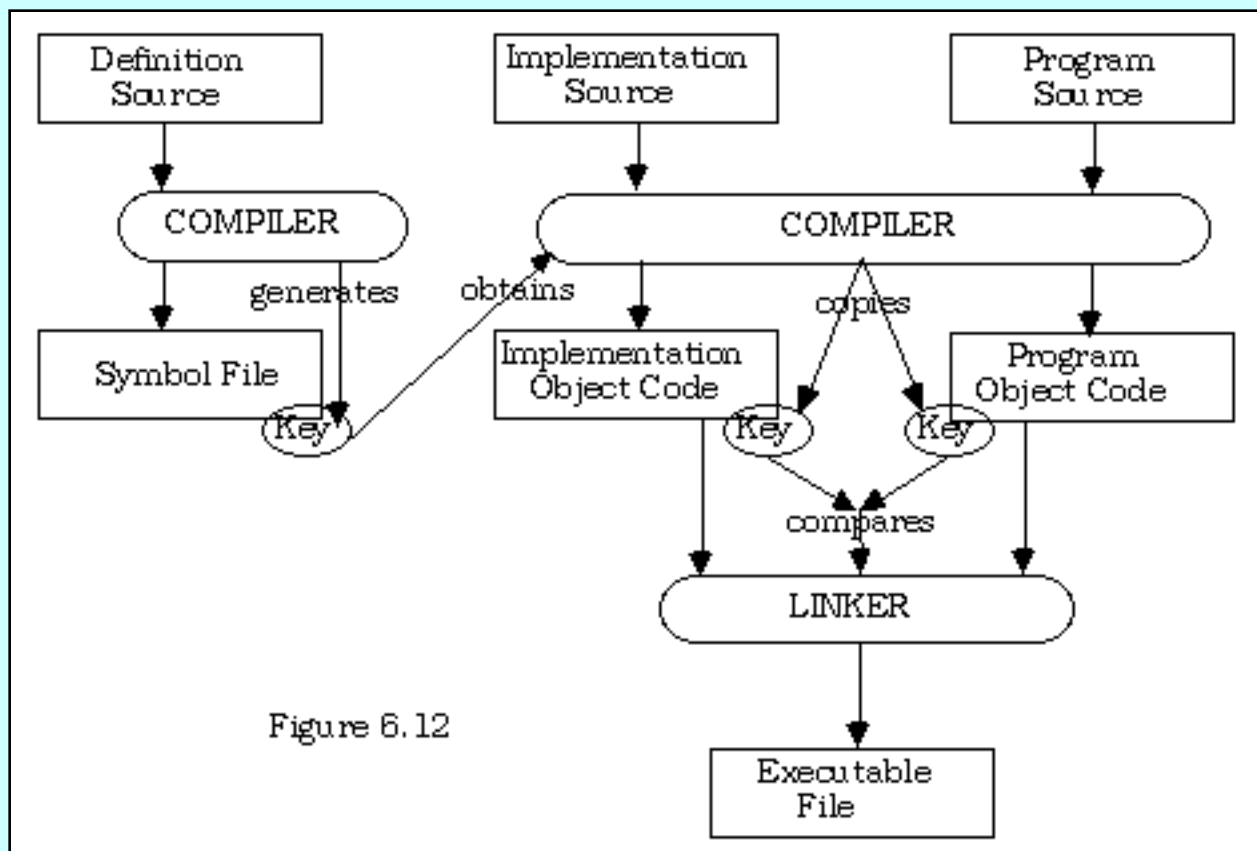


Figure 6.12

If it is later decided that there is a more efficient way to write the implementation of one of these portions of the package, this can be done without affecting either its definition, or any client programs.

On the other hand, any changes or recompiling of a definition module (with the possible exception of changes in comments in some implementations) requires that all clients depending on it also be recompiled before trying to link again. After all, changes at this level imply that there may now be syntax errors in both its corresponding implementation and in all client programs.

This behaviour of Modula-2 has two important implications for the designers of programs.

The first is that a program can be decomposed into a collection of modules, and the responsible team of programmers can agree collectively on the content of the various definition modules. These can be written and compiled so as to be available for all later stages of the project. Once this is done, the implementation modules and the program module can be assigned to various members of the team without any consultation on the details being necessary. This is because the definition modules contain the entire interface among the various parts of the final program, and are the only means by which they can obtain information from one another. Any changes one team proposes to a definition module have to be agreed to by all the teams, incorporated into the common interfaces, compiled and distributed to the teams before anyone uses them.

One consequence is that it does not matter what names each member of the team uses for variables and other local entities in the individual sections; there is no possibility for conflict among them. Modula-2 therefore completely eliminates side-effects caused by inadvertent re-use of entity names from one compilation unit to another. There is still a problem if a name is re-used within the same compilation module of course.

Furthermore, if someone finds a more efficient way of writing the implementation of a library module at a later date, this can be done, and the new version compiled, without having to re-compile the definition module or even any client program. (The keys all still agree.) However, if the definition module is re-

compiled for any reason (even if it has not actually been changed), a new key will be placed in its symbol file, and the corresponding implementation part of the module and any program modules using it will become incompatible and will have to be re-compiled before the next attempt to link the code together.

Second, it is important to keep these dependencies in mind even in small projects, because ignoring them could cause major problems. These could arise from one of two situations:

1. The programmer makes changes to the number and type of parameters for some user library routine. Here, the enforcement of version compatibility is of great benefit, for the definition, implementation, and client program modules must all be rewritten and recompiled to reflect the changes. If it were possible to change the first without changing the other two, then the (presumably correct) definition part would interface to an incorrect or obsolete implementation part. This would undoubtedly cause a system crash at run time if it were allowed.

2. The programmer inadvertently recompiles a definition module without actually making any changes in it. This time, unnecessary problems are created, for what appears to be an identical definition part is in fact different, because of the new key associated with it. Since the implementation modules get their key from the previously compiled definition part of the module with the same name, the actual code is now inaccessible to the linker. It will compare the key from the program module, obtained from the previous version of the definition, with the key in the implementation obtained from the new version of the definition, and refuse to do the link because they are different. The system cannot, after all, "know" that you have made no changes--the assumption is that the programmer must have had some reason for re-compiling the definition part.

If linking is done at run time, these problems will become evident the first time one tries to run a (perhaps previously successful) program that is a client of the incompatible module. If, on the other hand, the system does linking at a separate step and requires all code from library modules and elsewhere to be stored in a separate executable file (the most common method), the previously linked programs would still work, but a new attempt at linking would fail.

One can get into real trouble of the second type, because in some versions of Modula-2 the source codes for the system library definition modules are distributed on the disks along with the compiled versions. If one compiles any of these, access to the associated library module is cut off. Imagine not being able to use *STextIO* or any module that depends on it! Since few vendors also include the source code for the implementation parts of library modules for recompiling, it would now be necessary to restore the original from a back-up (there is one, isn't there?) or place an embarrassing call to the vendor for a new disk.

Careful housekeeping and cautious file management will prevent such problems from arising. Original disks should be copied and filed in a safe place, and the text files for definition modules should be printed out for reference and then deleted from operating disks to prevent their recompilation by a user of the final package. Likewise, text files for a programmer's own library modules should be backed up on separate disks than those from which the compiled versions will eventually be run.

6.8 Libraries--an Overview

There are potentially three different types of libraries of modules available to a programmer from which to import entities at any given time. These are:

The Supplied Library:

This is the collection of modules provided to the programmer as a part of the language package as it comes from the vendor. They include:

1. The Standard Library Modules:

These are the library modules defined by the International Organization for Standards (ISO). Some might include variations on the library modules that Wirth suggested be part of the operating environment in this package. This material includes:

- a. *system modules* to interface between Modula-2 and certain low level facilities that must exist on any computer
- b. *separate modules* which include (among others) those for
 - (i) input and output
 - (ii) text/numeric conversion
 - (iii) mathematical functions

The ISO standard also distinguishes between library modules that are *optional* and those that are *required* for certified compliance with the standard.

2. Nonstandard Utility Modules

These may include modules to provide communications capabilities, or certain useful data types or operations that programmers commonly use, but may not wish to write for themselves.

3. System Specific Modules

These support specialized facilities available only on a particular type or brand of

- a. computer (VAX, Apple, IBM PC, Sun, or a mainframe.)
- b. operating system (UNIX, the UCSD P-system, VMS, MS-DOS, Macintosh or other graphics user interface, etc.)

In some implementations, all three groups might be collected together formally into a single library file and in others they may be present as separate but previously compiled files. This text has already made use of several of these library modules, and shall have more to say about others in later chapters.

The User Library:

This is the set of library modules that the programmers working on the system have devised for themselves. As indicated already in this chapter, these are compiled and their entities used in exactly the same way as those in the supplied library.

Each module has a definition part and a (probably) separate symbol file created by the compiler and for its later use when compiling client programs to determine if the correct syntax has been used. It also has an implementation part that is keyed to the corresponding definition part and contains the actual run-time code for the linker to use in generating the executable version of a program module.

When the implementation part has been compiled in the presence of the previously compiled definition part, the defined entities are available for import in exactly the same way as the supplied ones.

NOTE: If the supplied library is a single file, instead of a scattering of files, there will probably be a supplied utility called a *library manager* that will allow the code of the user library collection also to be bound into a single file, or perhaps even added to the system library. If the latter is done, there will be a slight increase in linking speed, as the module loader will search in the supplied library before going through the other disk files for the desired name.

The Program Library:

This is the set of modules that are part of a program file itself, for a module can contain another module. This idea will not be pursued here, as this is a more advanced technique (see chapter eleven). However, note the following definitions:

A module that can be independently compiled, such as a program module, is called a global module. It may also be termed a compilation unit. One that is contained inside some other module, and is therefore not itself a compilation unit, is called a local module.

[Contents](#)

6.9 An Extended Example (Coordinate Geometry)

Example:

Write a library module to implement support for planar point coordinate geometry.

Discussion:

Points are represented in the plane by their horizontal and vertical coordinates (x, y). This representation can be rendered in Modula-2 as a one-dimensional array of length two.

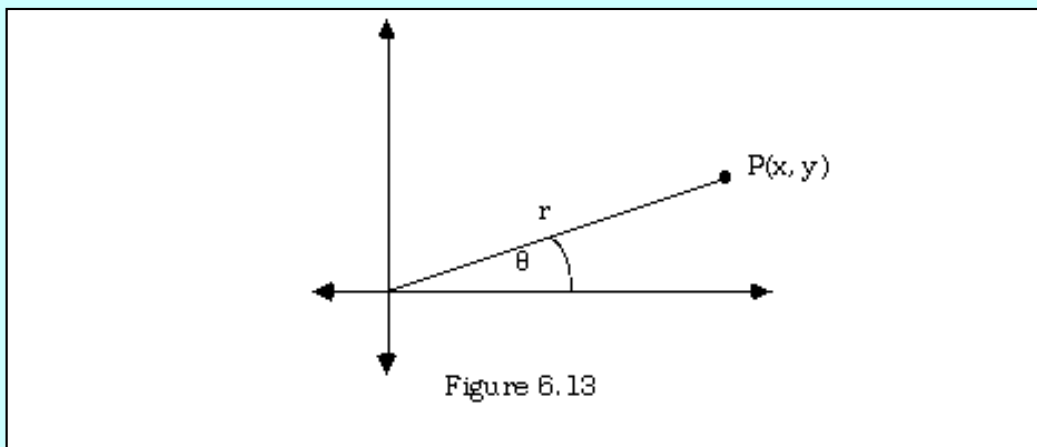
TYPE

```
Point = ARRAY [1 .. 2] OF REAL;
```

Note that the word *dimension* in this context refers to the array, not to the geometry. There are several useful operations relating to single points. These fall into two main categories:

Computations:

Three of these will be considered here; they arise in connection with the line segment joining the point with the origin.



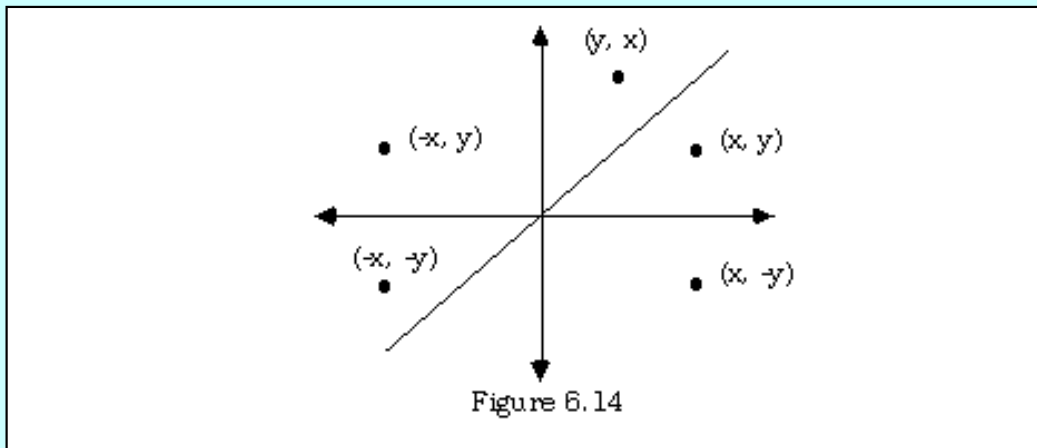
Both the length r of this line segment, and also the angle θ that it makes with the positive x-axis are of interest in various problems. It may also be necessary to construct the rectangular coordinates (x, y) of a point from a knowledge of the length r and the angle θ (also called its polar coordinates). These are expressed as follows:

$$r = \sqrt{x^2 + y^2} \quad \theta = \arctan \frac{y}{x}$$

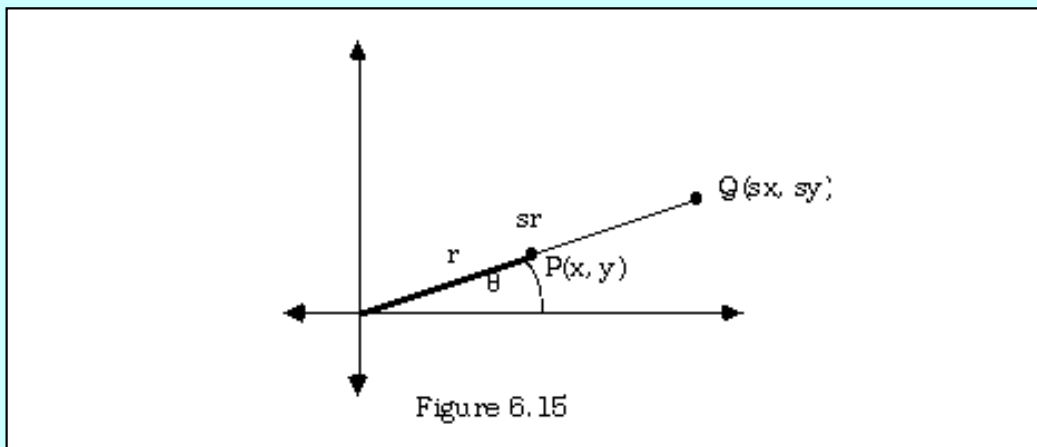
$$x = r \cos \theta \quad y = r \sin \theta$$

Transformations:

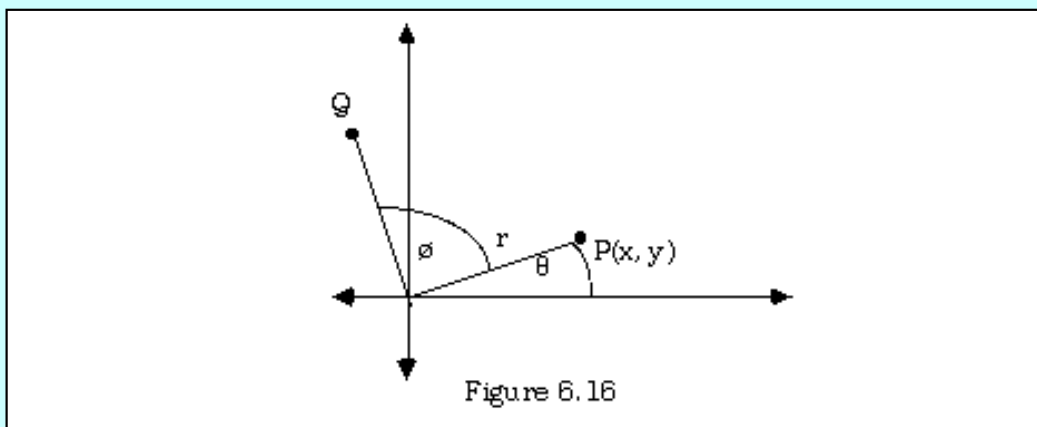
These involve moving the point P according to some determined pattern. The first such patterns involve various *reflections* or *symmetries*. These can be done in the x-axis to yield (x, -y), in the y-axis to yield (-x, y), across the origin yielding (-x, -y), or in the line $y = x$ producing (y, x). The possibilities are indicated in the figure 6.14 below:



The second type of transformation *scales* the point in such a way that the length of the line segment it defines is multiplied by some number, s called the *scale factor*. If this is done, the x -coordinate and y -coordinate of the new point Q are also multiplied by the same scale factor, but the angle θ is unchanged.



A third type of transformation on the point is via the segment it defines and consists of a rotation through an angle ϕ . The new point obtained is at the same distance r from the origin as the original, but its segment subtends an angle $\theta + \phi$ from the positive x -axis.



The fourth and last transformation is a simple shift or *translation* of the point by h units horizontally, and k units vertically. That is, the point that starts at (x, y) is transformed to the point $(x + h, y + k)$.

Finally, in order to maintain the integrity of items of type *Point* as an abstract data type, it is necessary to make provision for client programs to assign values to (and obtain values from) the coordinates of a point without directly using the knowledge of its structure as an array. This concern for integrity of the ADT necessitates three more procedures: *assign*, *abscissa*, and *ordinate*.

Thus the abstract data type *Point* and the operations and transformations on it may be defined in Modula-2 as follows:

DEFINITION MODULE Points;

(* angles are measured in radians counterclockwise from the positive x-axis *)

TYPE

Point = **ARRAY** [1 .. 2] **OF REAL**;

PROCEDURE assign (x, y : **REAL**) : Point;

(* returns the abstract point with coordinates x and y *)

PROCEDURE abscissa (p : Point) : **REAL**;

(* returns the first, or x-coordinate of the point *)

PROCEDURE ordinate (p : Point) : **REAL**;

(* returns the second, or y-coordinate of the point *)

PROCEDURE abs (p : Point) : **REAL**;

(* returns the distance from the point to the origin *)

PROCEDURE arg (p : Point) : **REAL**;

(* returns the angle to the positive x-axis subtended by a line segment from the origin to the point measured in the range 0 to 2π radians *)

PROCEDURE polarToRect (abs, arg : **REAL**) : Point;

(* returns the point with the given absolute value and argument *)

PROCEDURE reflectX (p : Point) : Point;

(* returns the reflection of the point in the x-axis *)

PROCEDURE reflectY (p : Point) : Point;

(* returns the reflection of the point in the y-axis *)

PROCEDURE reflect0 (p : Point) : Point;

(* returns the reflection of the point in the origin *)

PROCEDURE reflect45 (p : Point) : Point;

(* returns the reflection of the point in the line $y = x$ *)

PROCEDURE scale (p : Point; scaleFactor : **REAL**) : Point;

(* returns the point with the same argument as p and its absolute value multiplied by the scale factor *)

PROCEDURE rotate (p : Point; rotAngle : **REAL**) : Point;

(* returns the point with the same absolute value as p and with its argument increased by rotAngle *)

PROCEDURE translate (p : Point; deltaX, deltaY : **REAL**) : Point;

(* returns the point obtained by shifting the given point deltaX horizontally and deltaY vertically *)

END Points.

IMPLEMENTATION MODULE Points;

(* original by R. Sutcliffe
corrections by G. Tischer 1995 05 09 *)

FROM RealMath **IMPORT**

```
sqrt, arctan, sin, cos, pi;
```

```
PROCEDURE assign (x, y : REAL) : Point;
```

```
VAR
```

```
    temp : Point;
```

```
BEGIN
```

```
    temp [1] := x;
```

```
    temp [2] := y;
```

```
    RETURN temp;
```

```
END assign;
```

```
PROCEDURE abscissa (p : Point) : REAL;
```

```
BEGIN
```

```
    RETURN p [1];
```

```
END abscissa;
```

```
PROCEDURE ordinate (p : Point) : REAL;
```

```
BEGIN
```

```
    RETURN p [2];
```

```
END ordinate;
```

```
PROCEDURE abs (p : Point ) : REAL;
```

```
BEGIN
```

```
    RETURN sqrt (p[1] * p[1] + p[2] * p[2]);
```

```
END abs;
```

```
PROCEDURE arg (p : Point) : REAL;
```

```
(* if both coordinates are zero, the angle is not defined, but this procedure will  
return zero. No errors are generated. *)
```

```
VAR
```

```
    temp : REAL;
```

```
BEGIN
```

```
    IF p[1] = 0.0
```

```
        THEN
```

```
            IF p[2] < 0.0 THEN
```

```
                RETURN 1.5* pi (* case of point on negative y-axis *)
```

```
            ELSE
```

```
                RETURN 0.0;
```

```
            END;
```

```
        END;
```

```
    temp:= arctan (p[2]/p[1]); (* returns first and fourth quadrants only *)
```

```
    IF p[1] >= 0.0
```

```
        THEN
```

```
            RETURN temp;
```

```
        ELSE
```

```
            RETURN (2.0 * pi) + temp;
```

```
        END; (* IF *)
```

```
    ELSE
```

```
        RETURN pi + temp (* adjust for second and third quadrants *)
```

```
    END;
```

```
END arg;
```

```
PROCEDURE polarToRect (abs, arg : REAL) : Point;
```



```

VAR
    temp : Point;
BEGIN
    temp [1] := abs * (cos (arg));
    temp [2] := abs * (sin (arg));
    RETURN temp;
END polarToRect;

PROCEDURE reflectX (p : Point ) : Point;
BEGIN
    RETURN assign (p[1], -p[2]);
END reflectX;

PROCEDURE reflectY (p: Point ) : Point;
BEGIN
    RETURN assign (-p[1], p[2]);
END reflectY;

PROCEDURE reflect0 (p : Point) : Point;
BEGIN
    RETURN assign (-p[1], -p[2]);
END reflect0;

PROCEDURE reflect45 (p : Point) : Point;
BEGIN
    RETURN assign (p[2], p[1]);
END reflect45;

PROCEDURE scale (p : Point; scaleFactor : REAL) : Point;
BEGIN
    RETURN assign (scaleFactor * p[1], scaleFactor * p[2]);
END scale;

PROCEDURE rotate (p : Point; rotAngle : REAL) : Point;
BEGIN
    RETURN polarToRect (abs (p), arg (p) + rotAngle)
END rotate;

PROCEDURE translate (p : Point; deltaX, deltaY : REAL) : Point;
BEGIN
    RETURN assign (p [1] + deltaX, p [2] + deltaY);
END translate;

END Points.

```

Problem:

Write a routine to compute the distance between two points.

Discussion:

This problem can be solved, as suggested in exercise 4.19 using the formula

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

and employing the coordinates of the points directly. In the context of this discussion, having imported the type *Point* and the procedure *sqrt* into the surrounding module, the requested routine could be expressed as:

```

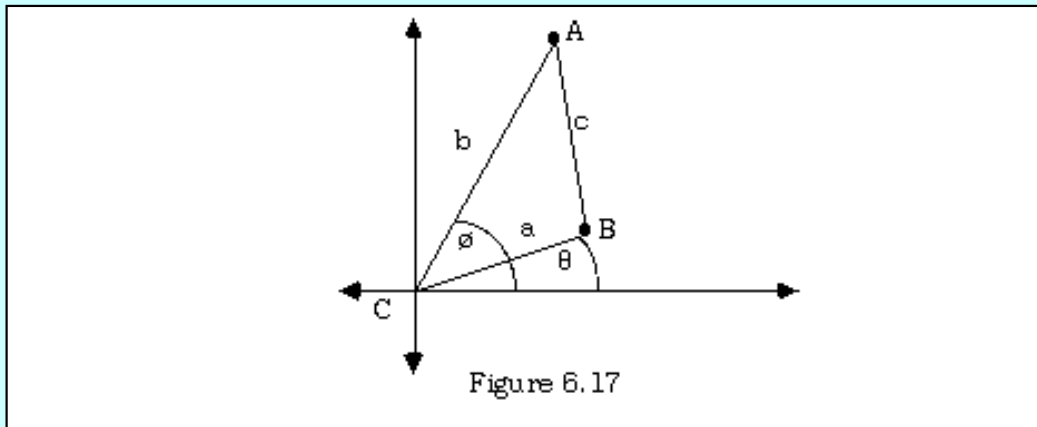
PROCEDURE distance (p1, p2 : Point) : REAL;

VAR
    deltaX, deltaY : REAL;

BEGIN
    deltaX := p2 [1] - p1 [1];
    deltaY := p2 [2] - p1 [2];
    RETURN sqrt (deltaX * deltaX + deltaY * deltaY);
END distance;

```

A second approach is more complicated in some ways, but trades off some additional mathematical steps for a more abstract view of the type *Point* within the procedure. Rather than have the procedure made direct use of the structure of the type *Point* by basing its computations on the coordinates of the point, it can make the calculation by using facilities provided by the module *Points* alone. This approach is consistent with the inclusion in the module of procedures to extract the coordinates without directly using the data structure. In this particular instance the law of cosines is exploited. (A proof of the law of cosines is not presented here).



The Law of Cosines: In any triangle ABC with sides opposite the angles labelled a, b, and c,

$$c^2 = a^2 + b^2 - 2ab \cos C$$

Here, the measure of angle C is $\theta - \phi$ and the distance between the two points is the length of the third side in a triangle whose other sides are the segments from the origins to the points A and B. Thus one could write

```

MODULE PointToPoint;

(* Written by R.J. Sutcliffe *)
(* to test the module Points *)
(* using ISO Modula-2 *)
(* last revision 1993 03 01 *)

FROM Points IMPORT
    Point, arg, abs;
FROM RealMath IMPORT
    cos, sqrt;
FROM STextIO IMPORT

```

```

    WriteString, WriteLn, ReadChar, SkipLine;
FROM SRealIO IMPORT
    ReadReal, WriteFixed;
FROM SIOResult IMPORT
    ReadResult, ReadResults;

PROCEDURE GetReal (VAR numToGet : REAL);
VAR
    tempResult : ReadResults;

BEGIN
    REPEAT
        WriteString ("Please type in a real number ===");
        ReadReal (numToGet);
        tempResult := ReadResult ();
        SkipLine; (* swallow line marker *)
        WriteLn;
    UNTIL tempResult = allRight;
END GetReal;

PROCEDURE DistByCosines (side1, side2, angle : REAL) : REAL;

BEGIN
    RETURN sqrt (side1 * side1 + side2 * side2 -
        2.0 * side1 * side2 * cos (angle));
END DistByCosines;

VAR
    point1, point2 : Point;
    a, b, c, C : REAL;
    key : CHAR;

BEGIN
    WriteString ("This program computes the distance ");
    WriteString ("between two points");
    WriteLn;
    WriteString ("in the coordinate plane.");
    WriteLn;
    WriteLn;
    WriteString ("First point,  X coordinate: ");
    GetReal (point1 [1]);
    WriteString ("                Y coordinate: ");
    GetReal (point1 [2]);
    WriteString ("Second point, X coordinate: ");
    GetReal (point2 [1]);
    WriteString ("                Y coordinate: ");
    GetReal (point2 [2]);
    a := abs (point1);
    b := abs (point2);
    C := arg (point1) - arg (point2);
    c := DistByCosines (a, b, C);
    WriteString ("The distance between the two points is ");
    WriteFixed (c, 5, 0);
    WriteString (" units.");

```

```
WriteLn;  
WriteString ("Press a key to continue ==");  
ReadChar (key);  
END PointToPoint.
```

Sample Run:

This program computes the distance between two points
in the coordinate plane.

```
First point,  X coordinate: Please type in a real number ==> 5.5  
Second point, X coordinate: Please type in a real number ==> 26.5  
The distance between the two points is 29.00000 units.
```

[Contents](#)

6.10 Chapter Summary

This chapter covered these topics:

- what a module is and how it differs from a procedure
- about libraries, and the hierarchy of I/O modules
- about mathematical functions in the library
- how to create libraries of your own
- the different kinds of libraries

It included discussion of the following Modula-2 built-ins:

Reserved Words	Standard Identifier
DEFINITION	none

IMPLEMENTATION

Imports--all contents of:

STextIO, *SWholeIO*, and *SRealIO*

or

InOut and *RealInOut* (many variations possible)

Terminal (where applicable)

Screen (where applicable)

RealMath and *LongMath*

or

MathLib0 and *MathLibLong* (many variations possible)

6.11 Assignments

Questions

1. What is a module?
2. List all the things that a module may contain.
3. What differences are there between modules and procedures?
4. (review) What is the difference between the two kinds of `IMPORT` from library modules?
5. Discuss the possible meanings of the word "standard" in the expression "standard library module."
6. Why are input and output among the most difficult library facilities to standardize?
7. What are the *character* input/output facilities in your implementation? Make detailed notes from the manuals on what their names and parameters are, what modules they come from (*STextIO*, *InOut*, *Terminal*, etc.) whether they skip spaces and/or end-of-line marks before or after reading the character, and what they do after reading the character (put it back in the input buffer or not.)
8. What are the *string* input/output facilities in your implementation? Make detailed notes from the manuals on what their names and parameters are, what modules they come from (*STextIO*, *InOut*, *Terminal*, etc.) whether they skip spaces and/or end-of-line marks before or after reading the string, and what they do when there is not enough room in their parameter for the characters read.
9. What are the *conversion* facilities (numeric/text) in your implementation? Make detailed notes from the manuals on what their names and parameters are, what modules they come from, and how they handle errors.
10. Explain why *STextIO*, *SWholeIO*, and *SRealIO* do not need separate redirection procedures (one for each) but a single `OpenOutput` or `OpenInput` can be written to serve all these. Alternately, explain why *RealInOut*, when a separate module, does not need an `OpenInput` and `OpenOutput` of its own.
11. What is meant by the expression "module hierarchy"? Illustrate with some examples.
12. What is a compilation unit?
13. Make a list of the differences among the different types of compilation units discussed in this chapter.
14. Document for your system the exact suite of file name extensions used by source files, and object files. Are there any differences between the treatment of program and library modules? Compare with the [chart](#) in section 6.5.
15. What are the main types of libraries, and how is each used by a programmer?
16. How does the module mechanism allow side effects to be reduced?
17. What is "module decoupling?"
18. If the definition part of a library module is changed, what modules need to be compiled or re-linked? Why?
19. If the implementation part of a library module is changed, what modules need to be compiled or re-linked? Why?
20. What advantage is there to not using the implementation details of a data type in the client programs that import the type?

21. Describe the major ways of handling library module errors and communicating these back to client programs.
22. Look up (or recall) the definitions of the three trigonometric functions cotangent, secant, and cosecant. Write these in terms of the functions in *RealMath*.

Problems

NOTE: Where facilities are available and it is appropriate, programs written for these exercises ought to have the option of taking input from a file or of directing output to a file.

23. (Requires redirection) Write a program module that will print out the cardinals from 0 to 99 in three columns, headed "decimal form", "binary form", and "hexadecimal form". Do this and obtain a printout for your reference. This question is easy to answer if *WriteHex* and *WriteOct* are available, so be sure to create your own procedures for this purpose.
24. (Requires redirection) Write a program that will take several lines of text from the keyboard and place it in the file *my.text* on a disk. (It is a good idea to use a non-system disk to avoid clobbering important files). Verify that you have done this correctly by then transferring the strings to the screen or, if possible to the printer from the disk file.
25. Write a program that will take an input line of text and write each letter twice as it comes in on both the screen and a file simultaneously (or at least in very close succession). Hint: Import more than one *Write*. A keyboard response of "How are you?" should output in both places as "HHooww aarree yyooouu??".
26. Use the exponential growth formula $A = A_0 e^{kt}$ and consider the growth of bacteria in this manner. Write a program to accept as data the number of organisms present at two times, and the amount of time elapsed for each measurement. (The first need not be zero). The program should compute and print the constant k and a "doubling time" (analogous to half life in the decay question.) It should then ask for a third time, and compute and print the number of bacteria expected to be present at that time.
27. Human populations also grow exponentially. Compute the doubling time for a human population assuming growth rates of 1%, 2%, ... 15% annually, and print the results in a table.
28. Suppose the human population began with two individuals at time $t = 0$ and had a growth rate that averaged 2%. How many people would there be after 1000, 2000, ... 7000 years? What if the growth rate had been 3% instead?
29. Write, enclose in a library module and test two procedures that compute k and the half-life from each other in processes using the formula $A = A_0 e^{kt}$.
30. Actually set up *MyRealMath* with the *log* function and a *tenToTheX* function as its inverse. Include also the *Entier* function found in *MathLib*, (or, if you are using *MathLib* already, include the *Round* function instead) and functions called *Max* and *Min* that return the largest (smallest) of two REAL parameters, respectively. Include at least one other function that you think may be useful. (*Magnitude*, and *Power* are possibilities).
31. Add to *MyRealMath* the other three trig functions, and their inverse functions.
32. The *entier* or *greatest integer* function has a companion function called the *ceiling* function. It takes a real parameter and returns the least integer greater than or equal to the parameter. Write and test a procedure to do this. Include it in your mathematical library.

33. Write a program to compute the area of a triangle given its three sides. Use any formulas from trigonometry or elsewhere that you wish, but include error trapping so that triples like 2, 2, 30 are rejected as impossible for triangle side measurements. Hint: The law of Cosines or Heron's formula may be applicable. Now enclose both this and a procedure based on the example in [section 6.4.3](#) in a library module, and write a program module that asks the user for data about a triangle and then selects the correct imported procedure to compute the area.
34. Notice that the computation given in the program module in section 6.4.1 can get into trouble under some circumstances. The expression $x^2 + y^2$ may, in whole or in part be larger than MAX (REAL) even though its square root is not. Suppose $x < y$ (if it isn't, swap them.) Now rewrite $\sqrt{x^2 + y^2}$ as $x\sqrt{1 + \left(\frac{y}{x}\right)^2}$ and there will be much less chance of the calculation overflowing the real type. Write and test a procedure for doing the calculation in this manner. It needs to contain code to test which is larger and a procedure for swapping when that becomes necessary.
35. Complete and test the module [Fractions](#) in section 6.5.1 and add to it the error handling routines suggested in [section 6.6](#). Alter the module in the following two ways: a. Denominators are always positive; the sign of a fraction is the sign of its numerator. b. All fractions are returned in lowest terms. c. Refine the *Add* procedure to reduce the possibility of overflowing the integer type by choosing the lowest common denominator for the addition instead of using the product of the two denominators. d. (optional) Also refine the *Mul* procedure by eliminating common factors between the numerator of the multiplier and the denominator of the multiplicand, and vice-versa before doing the operation.
36. In [section 6.4](#) in the module *Pythagoras* the procedure *GetReal* was employed to poll repeatedly for a real number from the keyboard. The same procedure was used later in the module *TriArea* in [section 6.4.3](#). Also, the non-standard procedure *HoldScreen* supplied in the Metropolis versions could be employed in place of the code:

```
WriteString ("Press a key to continue ==");
ReadChar (key);
```

These are therefore both good candidates for inclusion in a user library module. Write the module *Inputs* to encapsulate these two, along with similar procedures to the first for obtaining items of the types (where available) CARDINAL, INTEGER, LONGCARD, LONGINT, AND LONGREAL. There may be other useful procedures you can add to this module.

37. Write a *test harness* to test all aspects of the module *Points* to ensure that the various procedures work correctly. Choose sample data in such a way that it encounters a variety of different situations in the testing. Try to make things go wrong. Can you suggest any error handling techniques to prevent the problems you have uncovered?
38. A line segment is the set of all points on the straight line joining two specified endpoints. It can therefore be represented as

TYPE

```
Segment = ARRAY [0 .. 1] OF Point;
```


where *Point* is the abstract data type defined in [section 6.9](#). Write and test a library module *Segments* implementing the ADT *Segment* along with suitable procedures for manipulating items of this type. Include error handling as appropriate. Apart from specifying that *Point* must be imported from *Points*, the rest of the actual design is left to the student. If this problem is solved as an exercise for a course however, the definition module ought to be approved by the instructor before the implementation is done. Be sure to include a detailed explanation in the design stage for all the mathematics employed and a justification for each procedure included, as has been done in this chapter. The design ought also to include separate mention (with headers) of any procedures deemed to be necessary for inclusion in the implementation for its private use.

[Contents](#)

Second Interregnum--Nellie and the Pirates

I wrote this little piece in 1983, and it may appear a little dated in some respects today (the prices mentioned could be doubled, for instance) but the principles discussed are timeless, so I reprint the article here just as it originally appeared in my column.

-- R. Sutcliffe (a.k.a. The Northern Spy)

"Hey, sir, look at this!".

That is one thing that you can rely on about Nellie Hacker. When she has something to say she just bursts right in.

"I thought you were typing up one of my articles tonight." I shot back to the other end of my lab where Nellie sat hunched over the computer keyboard.

"Oh don't worry, I keep track of my time. I'm finished, so I thought I would boot up this disk I got from Alex the other day. It looks like a giant accountant's worksheet with rows and columns that you can put all sorts of formulas into. This is really nifty!"

"Sounds like a spreadsheet to me, Nellie. Where did you buy it and how much did you pay?" I solicited.

"O, Alex just ran me off a copy and photocopied the manual he was using; he didn't charge me anything."

I had finished marking the last of the labs by now - a good one too, and wouldn't you know it was hers - so I decided to return it there and then and to see what she had.

"You've got a rip-off of Busy-Calc, Nellie." I tried to make it sound like bad news.

"What's that?"

"It's a well known program of this type. They have sold thousands of copies on protected disks that you are not supposed to be able to duplicate. It retails for about \$250."

"You mean..."

"Yes, Nellie. You've got a hot program in your hot little hands; why, I could turn you in - some manufacturers give pretty big rewards."

"But, how could I know? There's no copyright notice in the manual." Nellie was indignant. "Wait until I see that Alex."

"This one I know because I once bought it myself," I responded. "Obviously, some enterprising soul has produced this brief manual in an attempt to make money from another person's program."

"Besides, Alex could never in his lifetime come up with a program like this; and you can bet there's nothing of this quality in the public domain either."

"That's the stuff anybody can copy, right?"

"Right, Nellie."

"But, who can afford to pay retail for this? I mean, like \$250 is pretty ridiculous. If you own a business or something, you pay the shot of course. But, what about us students? It's one thing to pay, maybe, thirty bucks like we would for a text, but this kind of price is just out of sight!"

"Well Nellie, the manufacturer would say that he needs that high price to justify the cost of production."

"That's absolutely ridiculous. A disk costs about two dollars, the manual no more than eight, and even if they pay that much out in overhead and split the same with the author as profit - it still comes to only thirty dollars."

"Actually, Nellie, advertising is pretty expensive, and they might have to pay the author of the manual - after all, a lot of programmers are illiterate." She gave me a savage look, so I quickly added; "Not you of course, your writing is as brilliant as always. Here, have your lab back," I added, hoping that she would be mollified by yet another perfect score.

"Furthermore," I continued without missing a beat, lest we start a new conversation before this one was half done, "even if it went to a wholesaler for, say, sixty dollars, he might as much as double the price, and the retailer could do the same. Add a few taxes along the way and you've got your \$250 in jig time. "I still say the whole thing is absurd. Let's set aside wholesale and retail rip-off pricing policies for a minute and look at it this way: Suppose as an author you are getting a twenty percent royalty. Now, wouldn't you rather have a fifth of 20,000 copies times a selling price of \$20 than a fifth of 500 copies times a price of \$250? After all, if you offer a good product for a sensible price aren't you going to sell it by the bag full?"

Nellie was not the sort of person who needed to waste much time on mental multiplication, and I could see that she obviously understood the business principles involved better than I had thought, so I replied: "I figure it that way too, girl, and so do a few software houses that are putting prices down from the \$200-\$300 range to a tenth of that. In fact some of them have ended up backordered by thousands of copies as a result.

"However, it seems that most of the people who have been in the business for a few years have grown up with the fat prices and are sticking to those high markups with the low volumes that they imply."

"I hope they go broke." Nellie put in.

"So do I, Nellie, so do I."

She gave a sort of "harrumph" that sounded suspiciously like one of my own as she dumped the offending "document" into the round file, along with the label from her diskette. The latter quickly went back in her box of blanks. I'm proud to say, that girl does have a well-tuned sense of what's right.

"By the way, I thought you said that these kinds of diskettes were uncopiable," Nellie proffered after a few moments. "How did Alex get one that he could copy for me?"

"O Nellie, there is no such thing as a really uncopiable disk. There are all sorts of programs around that are designed for just that. In fact, what with one group of programmers inventing new ways to lock diskettes, and another group working on ones to copy them, a tremendous amount of time and talent has been tied up on this problem alone. You should read the magazine ads more often."

"Haven't the time," she sulked. "You should see the English assignments Mr. M is giving us."

"And I'll bet", she added after the briefest of pauses, putting us abruptly back on the subject, "that they charge plenty for their picklock software and that their programs can't copy themselves."

"Of course they can't," I responded, answering her points in reverse order, "and they do charge between fifty and a hundred dollars, but people just buy two different kinds of copiers. In fact for that higher price, you can pick from several kinds of hardware devices which work by making a 'snapshot' of everything in memory and then putting it on a disk in an unlocked format. For those, you don't even need any software, it's all built into the hardware card."

"That's disgusting." Nellie was becoming quite agitated, as she always does when the issue under discussion offends her moral sensibilities. "If they really thought they were providing any kind of

legitimate service, they would sell their own stuff for a lot less and on copiable disks at that. They're no better than all the rest of them - just out to cheat us out of all the money the traffic will bear."

"Now, hold on a minute, kid. What if you do buy one of those expensive locked disks, because it just happens to be the best program available for your needs (and you also have the money to do it). Do you really want to get stuck without a backup?"

"Wouldn't you rather be able to make your own cheap copies for backup as soon as you buy a program, just in case? Wouldn't you like the freedom to have several copies of a word processor configured for your different writing needs? Wouldn't you also like to be able to save some three hour long game on a disk so that you can resume play at a later date?"

"Uck, you know I can't stand all those gamey things", she put in. "They are full of nothing but violence and fantasy. Give me concrete applications for computers any day over silly games."

"Besides, is even that sort of copying allowed? I've read some of the package notices and they can make it sound like you're a criminal if you even read the thing from the disk into the memory of the machine."

"I must have told you a million times, Nellie, don't exaggerate. But yes, it is allowed. As long as you do not sell or give the copies away, you can make as many as you want for your own use. What the vendors are most worried about is the large scale rip-off artists who run off hundreds of copies and sell them cheap or give them away at their club meetings."

"The worst offenders are schools. A lot of Districts buy a single copy of some major software package and then proceed to run off one for each machine in their jurisdiction. That can run into the thousands in some cases."

"The trouble is, in their zeal to stop these big thefts, the manufacturers are interfering with the legitimate rights of their honest customers. Why some of them include the most extravagant anti-copying and warranty disclaimer notices I have ever seen on any product. They try to be as intimidating as possible to protect their investment, I suppose."

"But, it's like I keep saying, sir. If they are going to ask outrageous prices like \$250, nobody is going to buy more than one copy, and precious few are even going to do that."

"They are just making it too tempting to be a crook. And can't the people who steal software in that way see that that is just what they are - crooks? Not only that, but that they are ruining the market for the people who want to write good programs, especially for the schools - if they are as bad as you say in this respect."

She continued her scattergun condemnation of all involved, both indignant and a little breathless:

"Besides, what are they supposed to be teaching us kids about honesty, and trust and all that stuff?."

"I'm glad to hear that you care too, Nellie, but let's be realistic: What fraction of the people in our society really do? As you say, the prices are far too high - they do make the temptation too great for most people to handle. Not only that, honesty in such matters is not even an issue with most people. They might not hold up a store; they might even hesitate about shoplifting, but if they can get away with stealing something in the privacy of their homes, they do not even give it a second thought. It takes a higher power than the law of the land to change people, so the problem is not going to be solved just by passing tougher laws. There isn't a program available today for any micro that can justify even a \$100 price tag at retail. "Games should be under twenty dollars, utilities up to thirty, spreadsheets and word processors up to sixty, and systems packages at the upper limit of a hundred."

"All software should be unlocked and you should be able to make as many copies as you want for yourself, with manuals selling for 25% of the package cost on the understanding that an organization can

place the program at one location per manual they buy.

"If prices were brought within reason, people would willingly foot the bill for the originals and piracy would not pay. Then the better software availability and lower prices would result in higher sales volumes for both programs and machines, and we would go into a spiral cycle of higher demand coupled with lower costs due to economies of scale. Ultimately the whole industry would benefit - we poor consumers most of all."

"That would help the individual owner, and schools too, but how would you deal with the clubs and others who just pass the stuff around to their friends for free?" Nellie asked.

"The same way as they do it in the music industry, Nellie--with media royalties."

"I can see how you could put a royalty on blank disks as they do on cassette tapes and then divvy it up among the programmers, but how do you decide on each person's fair share? You can't very well monitor the number of times a program is run," she added, with a look of puzzlement. "After all, there isn't any such thing as air time in the computer industry, you know."

"I wasn't talking about air time, Nellie. What I had in mind was just collecting the sales figures on the various commercial programs from the software houses. This could be done by a central agency which could then divide up the royalties to authors and publishers on a percentage basis calculated on their share of the total amount. They would go along with it because it would mean more revenue, and the disk manufacturers would too, because in the long run the general volume increases would result in them also selling more of their products."

"It sounds like a good idea, but do you really think they will take the first step by lowering prices and taking off the copy protection?"

"Nellie, girl, the ones who do are going to make money faster than they can count it, and the ones that do not are going to be watching from the bankruptcy courts."

"That seems fitting." Nellie's sense of justice seemed to be satisfied.

"It does, doesn't it?" I replied, and then added: "O, and Nellie, before I forget, when you see Alex tomorrow in Mr. H's class - tell him I would like to have a little chat with him in my office at noon."

"O.K., but you aren't going to yell at him too much are you?" she solicited in a concerned tone - her earlier irritation now evaporated. (Nellie never stays even a little angry for long.)

"No, Nellie, you know me better than that. Let's just say that I'm going to have another go at educating the boy. He's a valuable young man; he just has some growing up to do."

"Going home early?", I added as she started to pack up her books.

"Early?" she expostulated, "It's after five, sir."

And so it was. Time to pack it in for the day.

The Northern Spy--Nellie and The Pirates, Call A.P.P.L.E. September 1983, p.55-59

Assignment

In this chapter, the question of "professional ethics" was raised in a very minor way in connection with your potential services to a client. In this article, we have explored the issue of software piracy. There are many other ethical issues relating to Computer Science, such as job displacement, privacy, the quality of work and the workplace, the copying of patented designs, the amount of time spent on playing games, and the security of computer systems.

Whichever of the assignments below that you choose, be sure to make it clear where you stand; this is not

simply a survey of the problem, but an attempt to clarify one or more of your own values as a potential professional in this field.

1. Write a 2000 word essay giving a general overview of these ethical issues.
 2. Write a detailed examination of one or more of them.
 3. What does the word "ethics" mean, and where do ethical systems come from?
-

[Contents](#)

Chapter 6

Program Organization and Modules

[6.0 Chapter Goals](#)

[6.1 What Did You Say a Module is?](#)

[6.2 Libraries--How to Borrow a Module and Sign itOut](#)

[6.3 The Standard Library \(1\)--I/O](#)

[6.3.1 ISO Standard I/O Modules](#)

[6.3.2 Classical I/O--The InOut Family](#)

[6.3.3 Alternate Origins and Destinations](#)

[6.3.4 Redirection](#)

[6.3.5 Writing Special Characters To a Terminal](#)

[6.4 The Standard Library \(2\)--Mathematical Functions](#)

[6.4.1 Square Root](#)

[6.4.2 Exponential and Logarithmic Functions](#)

[6.4.3 Trigonometric Functions](#)

[6.4.4 Conversions](#)

[6.4.5 Summary of RealMath](#)

[6.4.6 Other Mathematical functions](#)

[6.5 Starting Your Own Libraries](#)

[6.5.1 Modules and User-Defined Data Types](#)

[6.6 Handling Errors in Library Modules](#)

[6.7 Modules and Design Considerations](#)

[6.8 Libraries--an Overview](#)

[6.9 An Extended Example \(Coordinate Geometry\)](#)

[6.10 Chapter Summary](#)

[6.11 Assignments](#)

[Second Interregnum--Nellie and the Pirates](#)

7.0 Chapter Goals

The purpose of this chapter is to provide the student with some experience in applying the programming techniques learned thus far to a few real life problems. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

No new data types are taken up in chapter 7.

Realized in the Modula-2 notation:

the details of string representation, strings as a semi-abstract data type

Data Manipulation Abstractions

General:

string operations

Realized in the Modula-2 notation:

length and concatenation of strings in detail, other string operations abstractly

Programming Abstractions

No new programming abstractions are taken up in chapter 7. The ones of previous chapters are applied to a variety of problems.

7.1 Introduction to Some Applications of Modula-2

Until now this text has concentrated on the details of *how* to program using the Modula-2 notation. The present chapter does not represent a radical departure from that pattern for it is principally concerned with examples of such programs. However, it will attempt to consider some of the *why* and *what* questions, that is, the motivation for writing programs. For instance, there are some facilities that are not a part of the Modula-2 notation proper, but that must (frequently) be devised or imported by the programmer. Likewise, there are certain common classes of applications problems that arise frequently in real life, and some samples of these are worth a close examination.

Two groups of applications:

A. Strings

B. Mathematical, Scientific and Financial Applications

have been selected, not because these are the only important ones, but because all are frequently present in some way as part of many applications. These three also present problems whose solutions are both interesting in themselves and illustrative of a number of important programming methods and language constructions.

For instance, most applications make use of strings. The writer of useful programs must not only create the code that performs the task at hand, but must also generate the interface between that task and the person who will operate the program. This human/machine interface will necessarily involve writing and presenting numerous screens of information for that operator. This in turn will require that the programmer pay close attention to the handling of the text that constitutes that information. In addition, the data processed by a program is often in the form of strings, and so the manipulation of these as entities in themselves also takes on some importance.

More important than that, computers compute. Whether one takes the traditional view and regards mathematics as God's universal notation to express His design of the Universe, or whether one thinks of mathematics as purely the invention of the human kind and not dependent on any prior reality for its validity, it is the indispensable tool for describing our surroundings. Without the language of mathematics, there would not only be no computers or modern technology, there would be no science at all--no means by which to express questions about the Universe, much less to obtain answers to those questions. There is, after all, that within mathematics which cries out to be applied.

So, this portion of the chapter will offer a few illustrative examples from among the many possible that are largely mathematical in nature. These have a wide range of applications that often belies their apparently abstract nature. In this section the techniques to compute some statistical tools such as the mean, and standard deviation. Random number generators will also be examined.

One section is devoted to a specific kind of problem in physics, and the solutions to these too, as it will soon be evident, depend heavily on the underlying mathematical principles.

The last section is devoted to expanding upon and then encapsulating the many procedures needed to make the financial computations that were discussed in [section 4.9](#).

The sections of the chapter (Part A) on strings is a single unit, but all the ones after that (Part B) are

independent of it and of each other. This will allow hard-pressed instructors and students to pick and choose somewhat to suit their interests.

[Contents](#)

Part A--Strings

7.2 Communicating in English

The data type String (an ARRAY [0 .. n - 1] OF CHAR) has been mentioned several times, and previous programs have made considerable use of string literals. This data type is neither a part of the Modula-2 language proper nor is it part of the predefined standard operating environment. However strings have a wide variety of uses, and all implementations of the language include a module implementing specific instances of this data type, as well as a number of procedures to act on such arrays.

Even if that were not the case, programs are free to use such arrays for their own purposes, and programmers may devise their own procedures to operate upon them.

Some things are built in. Any literal or constant string can be directly assigned (using the := operator) to an ARRAY [0 .. n - 1] OF CHAR provided that the length of the string (number of characters) being assigned is less than or equal to n. If it is equal, the array is filled entirely; if it is not, then a special *string terminator character* is automatically appended to the string when it is placed in the array. The latter is done so that programs using these arrays will know when they have encountered the last valid character.

NOTE: In classical versions of Modula-2 the string terminator character was always the null character (CHR (0)). In the ISO standard, the value of this character is implementation defined, but it is always equal to the character literal " or "" (i.e. two single or two double quotes with no characters between them). In most, however, it will still likely be CHR (0).

Once such an assignment to an ARRAY OF CHAR is done, a procedure like *WriteString* is able to write out the array just as it does a literal string. In anticipation of this, the code for *WriteString* is arranged so that it will terminate the output of characters if it should arrive at the string terminator character before it reaches the end of the array.

Naturally, the corresponding procedure such as *ReadString* also places a string terminator character after any string that it reads into a longer array. Thus, some string operations are included in the Modula-2 system (language and standard modules) and, as mentioned above there are always more in a utility module designed expressly for the purpose of manipulating this data type. Indeed, the ISO standard for the language mandates that conforming implementations shall supply such a utility module. Summarizing to this point, it is worth taking note of the following definitions:

A Modula-2 string is an ARRAY [0 .. n - 1] OF CHAR. Because the range must start at zero, they are said to be zero-based. Because an array that is not entirely full has a string terminator character to mark the last-used position, a Modula-2 string is said to be terminated.

When the basic structure of a data type is visible, and specific instances of it have to be instantiated, but it can otherwise be treated abstractly, perhaps because library modules or built-in routines are available for manipulations, it may be termed an implied abstract type.

That is, the whole collection of potential string types taken together can be thought of as constituting an implied abstract type STRING, even though, strictly speaking, Modula-2 does not have an abstract string type *per se*. One is permitted to define string constants, and to assign string constants and literals to string variables, provided the target being assigned to has at least as many components as the source being assigned from. Moreover, in the ISO standard version of Modula-2, there is a type for the entire collection of string literals.

All string literals are said to be of S-type.

Here are a few examples. Given the following declarations:

```

CONST
    mesg = "Hello there";
    name = "Fred";

TYPE
    String10 = ARRAY [0..10] OF CHAR;
    String11 = ARRAY [0..11] OF CHAR;

VAR
    string1, string2 : String10;
    string3 : String11;

```

then, in a program, the following assignments and statements have the indicated effect: (The spaces indicate positions in the string and the symbol \emptyset indicates the string terminator character.)

```

string1 := mesg;      

|   |   |   |   |   |  |   |   |   |   |   |
|---|---|---|---|---|--|---|---|---|---|---|
| H | e | l | l | o |  | t | h | e | r | e |
|---|---|---|---|---|--|---|---|---|---|---|

 (exactly filled)

string2 := name;      

|   |   |   |   |             |   |   |   |   |   |   |
|---|---|---|---|-------------|---|---|---|---|---|---|
| F | r | e | d | $\emptyset$ | - | - | - | - | - | - |
|---|---|---|---|-------------|---|---|---|---|---|---|

 (last part undefined)

WriteString (mesg);
ReadString (string1);

```

Suppose the person answered by typing the string "Yes."

string1 would now hold:

Y	e	s	\emptyset		t	h	e	r	e
---	---	---	-------------	--	---	---	---	---	---

Note that the assignment rule mentioned above means that

```
string3 := "Now is then";
```

is a valid assignment because the literal being assigned has eleven characters, but that all of

```

string2 := "Now is then";
string2 := string3;
string3 := string2;

```

are not valid, the first because the target type is too small, and the other two because the entities are of different types, and the normal rule for array assignment comes into effect.

In addition, zero character literals are regarded as strings according to this definition, and single character literals can be taken as either type **CHAR** or as type **ARRAY**[0..0] **OF** **CHAR**. The latter can also be defined by writing out their ASCII number in *Octal* (base eight) notation followed by the symbol *C*. For more details on number bases, see Chapter eight. On the other hand, constants declared with the **CHR** are just of the type **CHAR**, not strings. Thus, if in addition to the last set of declarations, we had:

```

CONST
    CR = 15C;
    space = " ";
    empty = "";
    LF = CHR (10);

```

then the following assignments are valid:

```

string 1 := CR;
string 1 := space;

```

```
string 1 := empty;
```

but

```
string 1 := LF;
```

is invalid, because LF is of type CHAR only.

Notice that the assignment to the string variable does not affect any positions in the array beyond those necessary to do the assignment. The characters after the string terminator are no longer of any interest, for even though the history of this particular variable's use does tell us what those characters are, they should be regarded as undefined insofar as the string variable is concerned.

The details just described affect the way that string operations are coded, whether these are imported from modules, or are user-devised.

To see how this is so, consider how a programmer could write some typical string operations. Two fairly easy things one might like to be able to do are:

1. compute the number of active characters in a string, and
2. join two strings together.

The number of characters in a string is called its length.

When two strings are joined end-to-end to make a new string whose length is the sum of the first two this process is called concatenation.

Example:

"HOW TO" is a string of length six and " PROGRAM" has length eight (note the spaces). The concatenation is "HOW TO PROGRAM" and has length fourteen.

When using *string literals* and *string constants* only (not variables) ISO standard Modula-2, implements within the language proper a concatenation operator "+" so that one may write:

CONST

```
CR = 15C
LF = 12C;
DOSLineEnd = CR + LF;
strConst = "Hi" + " There"
strConst2 = strConst + DOSLineEnd;
```

or an assignment such as

```
strVar := "Hello" + " world"
```

but may not write:

```
string1 := string1 + string2;
```

or even

CONST

```
return = CHR (13);
strConst = "Hello" + return;
```

because this last constant can not be used as a string of length one, but is a CHAR. Likewise, if in addition to the above, one has:

TYPE

```
String80 = ARRAY [0..79] OF CHAR;
```

VAR

```
str : String80;
```

then the assignment

```
str := strConst + str;
```

is also illegal, because the concatenation operator cannot be used with strings of a specified type, only with literals (that is, of the S-type.)

Otherwise, one may use + with the same meaning as indicated above for *Concat*. The function *Concat* must still be included in a library module for operations on string variables.

What follows is a portion of a library module that could implement an instance data type and these two operations.

The procedure *Length* works by examining each character in the array passed to it until it either reaches the last index of the array or to a string terminator, whichever comes first. The number of characters checked by the time this loop is exited is the length of the string.

The procedure *Concat* starts by placing the first string into the result; then, if there is room left, it puts as much of the second one in as possible. The result will either be entirely filled, or will end with the string terminator taken from the end of the second string put into it. If the concatenation of the two strings would contain too many characters to fit into the array being used to hold the result, the extra characters are quietly discarded with no error being generated. One says that it is "silently truncated."

DEFINITION MODULE Strings;

TYPE

```
String = ARRAY [0 .. 79] OF CHAR; (* convenience type *)
```

PROCEDURE Length (str : **ARRAY OF CHAR**) : **CARDINAL**;

(* returns the number of characters in a string up to a string terminator, or the end of the array, whichever is less *)

PROCEDURE Concat (str1, str2 : **ARRAY OF CHAR**;

VAR result : **ARRAY OF CHAR**);

(* This procedure concatenates two strings. It will use as much of the two as possible, silently truncating the result if there is not enough room. *)

END Strings.

IMPLEMENTATION MODULE Strings;

CONST

```
terminator = "";
```

PROCEDURE Length (stringVal: **ARRAY OF CHAR**): **CARDINAL**;

(* Returns the length of stringVal *)

VAR

```
count : CARDINAL; (* Counting Variable *)
```

```

hiStr : CARDINAL; (* hold high of string for comparisons *)

BEGIN
  hiStr := HIGH (stringVal);
  count := 0;
  WHILE (count <= hiStr) AND (stringVal[count] # terminator)
    DO
      INC(count);
    END;
  RETURN count;
END Length;

PROCEDURE Concat (str1, str2 : ARRAY OF CHAR;
                  VAR result : ARRAY OF CHAR);

VAR
  max, rcount, scount : CARDINAL;

BEGIN
  max := HIGH (result);
  (* max is the maximum number of places available in the result*)
  rcount := 0;    (* initialize result string count *)
  WHILE (rcount <= HIGH (str1))
    AND (str1 [rcount] # terminator)
    AND (rcount <= max)
  DO
    result [rcount] := str1 [rcount];
    INC (rcount)    (* Put in as much of str1 *)
  END ;    (* as will fit *)
  IF rcount <= max    (* room left? *)
  THEN    (* yes, so, reusing last position with terminator *)
    scount := 0;    (* set counter for second string *)
    WHILE (scount <= HIGH (str2))
      AND (str2 [scount] # terminator)
      AND (rcount <= max)
    DO
      result [rcount] := str2 [scount];  (* and put in as *)
      INC (rcount);    (* much of it as will fit too *)
      INC (scount);
    END;
  END;    (* if *)
  IF rcount <= max    (* still room left? *)
  THEN
    result [rcount] := terminator; (* put in terminator *)
  END;
END Concat;

END Strings.

```

Notice that **HIGH** (str1) and **HIGH** (str2) do *not* return a number corresponding to the *length* of the string. Instead (as always) they produce the highest index used when the actual parameter array is assigned to the open formal parameter array. If one passes a literal string instead of an array variable, then **HIGH** (str1) would be one less than the length of the string, but if one is passing objects of the type *String* (above) then eighty places are assigned to the formal parameter. This would be reflected by

HIGH (str1), which would therefore be 79 for this data type even if not all eighty CHARs are actually being used (there is a string terminator somewhere before position 79).

Example:

If one had:

```
VAR
    str : String;

PROCEDURE PrintMax (str : ARRAY OF CHAR);

BEGIN
    WriteCard (HIGH (str), 1)
END PrintMax;
```

then

```
PrintMax ("HELLO"); would print 4
```

whereas

```
str := "HELLO";
PrintMax (str);
```

would print 79, as it is assigned "str", not "HELLO" directly.

[Contents](#)

7.3 Library String Functions

So much for a few homemade attempts to implement some string operations. The reader will be asked to implement a few more of these in the exercises. This section will elaborate on what others consider to be a reasonable collection of string functions.

What follows is the definition of a module called *Strings* that is available in the ISO compliant versions of Modula-2. There are also notes on some variants for other versions. The explanation of the various procedures follows.

NOTES: 1. ISO standard versions of Modula - 2, implement within the language proper a standard procedure **LENGTH** with the syntax exactly as indicated above for *length*. The function *Length* is still included in the *Strings* module for the sake of compatibility with older versions.

2. Because some other type of string may also be required for operating system use, or for interface to other languages, many versions of Modula-2 have two separate string handling modules, and a third for conversion functions to move data between the two types. See the exercises for more information.

3. In non-ISO versions, the procedures made available are not always the same as these, and when they are, the order of the parameters varies from one implementation to another.

DEFINITION MODULE Strings;

TYPE

```
String1 = ARRAY [0..0] OF CHAR;  
  (* for constructing a value of a single-character string type to pass CHAR values  
to ARRAY OF CHAR parameters. *)
```

PROCEDURE Length (stringVal: **ARRAY OF CHAR**): **CARDINAL**;

```
  (* Returns the same value as would be returned by LENGTH *)
```

PROCEDURE Assign (source: **ARRAY OF CHAR**; **VAR** destination: **ARRAY OF CHAR**);

```
  (* Copies source to destination *)
```

PROCEDURE Extract (source: **ARRAY OF CHAR**; startPos,
 numberToExtract: **CARDINAL**;
 VAR destination: **ARRAY OF CHAR**);

```
  (* Copies at most numberToExtract characters from source to destination, starting  
at position startPos in source. *)
```

PROCEDURE Delete (**VAR** stringVar: **ARRAY OF CHAR**; startPos,
 numberToDelete: **CARDINAL**);

```
  (* Deletes at most numberToDelete characters from stringVar, starting at position  
startPos. *)
```

PROCEDURE Insert (source: **ARRAY OF CHAR**; startPos: **CARDINAL**;
 VAR destination: **ARRAY OF CHAR**);

```
  (* Inserts source into destination at position startPos *)
```

PROCEDURE Replace (source: **ARRAY OF CHAR**;
 startPos: **CARDINAL**;
 VAR destination: **ARRAY OF CHAR**);

```
  (* Copies source into destination, starting at position startPos. Copying stops  
when all of source has been copied, or when the last character of the string value in  
destination has been replaced. *)
```

```

PROCEDURE Append (source: ARRAY OF CHAR; VAR destination: ARRAY OF CHAR);
    (* Appends source to destination. *)

PROCEDURE Concat (source1, source2: ARRAY OF CHAR; VAR destination: ARRAY OF CHAR);
    (* Concatenates source2 onto source1 and copies the result into destination. *)

PROCEDURE CanAssignAll (sourceLength: CARDINAL; VAR destination: ARRAY OF CHAR):
BOOLEAN;

PROCEDURE CanExtractAll (sourceLength, startPos,
                        numberToExtract: CARDINAL;
                        VAR destination: ARRAY OF CHAR): BOOLEAN;

PROCEDURE CanDeleteAll (stringLength, startPos, numberToDelete: CARDINAL): BOOLEAN;

PROCEDURE CanInsertAll (sourceLength, startPos: CARDINAL;
                        VAR destination: ARRAY OF CHAR): BOOLEAN;

PROCEDURE CanReplaceAll (sourceLength, startPos: CARDINAL;
                        VAR destination: ARRAY OF CHAR): BOOLEAN;

PROCEDURE CanAppendAll (sourceLength: CARDINAL;
                        VAR destination: ARRAY OF CHAR): BOOLEAN;

PROCEDURE CanConcatAll (source1Length, source2Length: CARDINAL;
                        VAR destination: ARRAY OF CHAR): BOOLEAN;

(* The following type and procedures provide for the comparison of string values, and
for the location of substrings within strings. *)

TYPE
    CompareResults = (less, equal, greater);

PROCEDURE Compare (stringVal1, stringVal2: ARRAY OF CHAR): CompareResults;
    (* Returns less, equal, or greater, according as stringVal1 is lexically less than,
equal to, or greater than stringVal2. *)

PROCEDURE Equal (stringVal1, stringVal2: ARRAY OF CHAR): BOOLEAN;
    (* Returns Strings.Compare(stringVal1, stringVal2) = Strings.equal *)

PROCEDURE FindNext (pattern, stringToSearch: ARRAY OF CHAR;
                    startPos: CARDINAL;
                    VAR patternFound: BOOLEAN;
                    VAR posOfPattern: CARDINAL);

    (* Looks forward for next occurrence of pattern in stringToSearch, starting the
search at position startPos. If startPos < LENGTH(stringToSearch) and pattern is
found, patternFound is returned as TRUE, and posOfPattern contains the start position
in stringToSearch of pattern. Otherwise patternFound is returned as FALSE, and
posOfPattern is unchanged. *)

PROCEDURE FindPrev (pattern, stringToSearch:
                    ARRAY OF CHAR; startPos: CARDINAL;
                    VAR patternFound: BOOLEAN;
                    VAR posOfPattern: CARDINAL);

```

(* Looks backward for the previous occurrence of pattern in stringToSearch and returns the position of the first character of the pattern if found. The search for the pattern begins at startPos. If pattern is found, patternFound is returned as TRUE, and posOfPattern contains the start position in stringToSearch of pattern in the range [0..startPos]. Otherwise patternFound is returned as FALSE, and posOfPattern is unchanged. *)

PROCEDURE FindDiff (stringVal1, stringVal2: **ARRAY OF CHAR**;

VAR differenceFound: **BOOLEAN**;

VAR posOfDifference: **CARDINAL**);

(* Compares the string values in stringVal1 and stringVal2 for differences. If they are equal, differenceFound is returned as FALSE, and TRUE otherwise. If differenceFound is TRUE, posOfDifference is set to the position of the first difference; otherwise posOfDifference is unchanged. *)

PROCEDURE Capitalize (**VAR** stringVar: **ARRAY OF CHAR**);

(* Applies the function CAP to each character of the string value in stringVar. *)

END Strings.

The Procedures *Assign*, *Extract*, *Delete*, *Insert*, *Replace*, *Append*, and *Concat* have the property that if the string they are constructing has a length exceeding the capacity of the variable parameter, the result is silently truncated. That is, the extra characters are discarded and no error condition is created. Of course, if the string created is shorter than the capacity of the variable parameter, the string terminator is appended.

Because this behaviour might create a problem for the client of this module, the procedures *CanAssignAll*, *CanExtractAll*, *CanDeleteAll*, *CanInsertAll*, *CanReplaceAll*, *CanAppendAll*, and *CanConcatAll*, are provided to test whether the corresponding command would work correctly (whether there is enough room) if attempted. The writer of the client program must decide whether to use these or not, and what the program should do if the test produces FALSE, indicating that the intended destination lacks the room for the result.

The procedure *Assign* takes the content of one string variable and assigns it to a second (which could have a different underlying type). If *str1* and *str2* are of the same formal type of string, one could also write

```
str1 := str2;
```

instead of

```
Assign (str2, str1);
```

so this procedure is not always necessary. However, if one had an **ARRAY [2 .. n] OF CHAR** (i.e. any array not zero based) then *Assign* could be used to give its contents to a string identifier. (i.e. one defined as of a type [0 .. m]) Also, if one had:

TYPE

ShortStrings = **ARRAY [0 .. 10] OF CHAR**;

LongStrings = **ARRAY [0 .. 79] OF CHAR**;

VAR

name : ShortStringType;

fullName : LongStringType;

then, an assignment like:

```
fullName := name;
```

would be illegal in standard Modula-2, because the two underlying types are different. In this case also, one must, import *Assign* from *Strings* and use

```
Assign (name, fullName);
```

Note, however, that *Assign* is not needed in a few non-standard versions, which do allow direct assignments of the kind above as a language extension.

In general, it is best to use *Assign* and the other utility library procedures, treating string types as ADTs even though their structure is known to the program. An ADT *String* could be implemented that way. If it were, the parameters for these procedures would all be of type *String* rather than ARRAY [0 .. N - 1] of CHAR. *Assign* would then always be necessary to convert an ARRAY OF CHAR to a variable of type *String*, for one would not officially know how the type *String* is defined. In some ways this would be more in accord with sound Modula-2 programming practice than what is in fact done. One should therefore regard the implied type *String* as an anomaly--an exception to the general technique, rather than as a model to follow in devising user libraries implementing ADTs. (The details of how to create ADTs whose structure truly is invisible to client programs will be discussed in a later chapter.)

The procedure here called *Extract* is also known as *Copy* in some non-ISO versions. Its purpose is to assign a substring in the source to the destination.

Insert and *Delete* are closely related. One puts characters in to a string; the other takes them out. In both cases, the text following the insertion point is moved appropriately. *Replace*, on the other hand, copies new material into a destination string over the top of whatever was there. Suppose one had:

```
str1 := " the old";  
str2 := "Nellie Hacker";
```

Then,

```
Extract (str2, 3, 5, str3);
```

would result in str3 holding "lie H", while

```
Insert (str1, 6, str2);
```

would result in str2 holding "Nellie the old Hacker". At this point,

```
Delete (str2, 1, 14);
```

would result in str2 holding "Nacker". Now,

```
Replace ("Si", 0, str2);
```

would result in str2 holding "Sicker".

The procedure *Append* attaches the first string supplied to the end of the second, returning the second string in an altered condition, whereas *Concat* does this, but places the result in a third parameter, leaving the original strings untouched. *Concat* and *Length* are shown here just as they were defined in the last section. Of course, the programmer who imports from this library neither knows nor cares whether the writer of the implementation part of this particular module actually wrote them in the same way as shown here.

The function *Compare* makes an alphabetical comparison of the strings *str1* and *str2*. The result is *less* if *str1* < *str2*; it is *equal* if *str1* = *str2*, and it is *greater* if *str1* > *str2*. Examples: Compare ("Cat", "Hat"); Compare ("at", "atrocious"); Compare ("105", "108"); (* character, not numerical compare *) Compare ("Albert", "albert"); (* upper case before lower case *)

The procedure *FindNext* (sometimes called *Pos* in non-standard versions) returns as the value of *posOfPattern* the cardinal of

the starting position of the first string (*pattern*) if it is found in the second string (*stringToSearch*) after the given *startPos*. In this case, the BOOLEAN parameter *patternFound* is TRUE. It returns FALSE in *patternFound* if it is instructed to start at a point past the end of the target string, or if it does not find the target embedded in the source. In this case, *posOfPattern* is unchanged. Thus;

```
str1 := "Johanna Meier the next hacker";  
FindNext ("Me", str1, 0, found, where);
```

would set *found* to TRUE and *where* to 8, whereas, if the third parameter were given as 10, *found* would be FALSE. The procedure *FindPrev* acts in the same way as *FindNext* but searches backward from the given starting point instead of forward. The procedure *FindDiff* compares two strings for equality character by character. If they are equal the BOOLEAN parameter *differenceFound* is set to FALSE. If there is a difference, that parameter is set to TRUE and the place at which they first differ is the value of the parameter *posOfDifference*.

The procedure *Capitalize* capitalizes all the letters in a string. The (non-standard and occasionally supplied) procedures *FetchChar* and *AssignChar* allow the client program to manipulate individual characters in items of type *String* abstractly (i.e. without knowing the structure, or if knowing it, without using that knowledge).

It is a useful exercise in understanding the meaning of string data to implement all these procedures yourself.

NOTES: 1. Some non-standard implementations export two or more convenience types such as `String80 = ARRAY[0..80] OF CHAR`.

2. Strings are implemented in a variety of ways in different languages and implementations. Modula-2 itself uses, as seen here, a terminated `ARRAY [0 .. len - 1] OF CHAR`. However, the associated operating system may not use Modula-2 style strings for such things as, say, file names. Thus if the programmer decides to manipulate the file system on a level lower than what is provided by the utility module *Files*, (see the next chapter for details) it will probably be necessary to convert Modula-2 strings that name files to the appropriate form for the actual file system being used. Facilities to make these changes may be provided in an appropriate conversions module, or they may have to be programmed.

To illustrate some of the ideas presented in these two sections, here is a simple program module that will accept an input line from the standard input, and then print it out back to front in two ways--letter by letter, and word by word. Note that it is written in a way that depends on knowing what is the structure of string variables.

MODULE Backwards;

```
(* Written by R.J. Sutcliffe *)  
(* to illustrate string manipulation *)  
(* using ISO Modula-2  *)  
(* last revision 1996 12 04 *)
```

FROM STextIO **IMPORT**

```
  WriteLn, WriteChar, WriteString, ReadString, SkipLine;
```

TYPE

```
  String = ARRAY [0 .. 80] OF CHAR;
```

CONST

```
  blank = ' ';
```

VAR

```
  inStr : String;  
  count, startWord, endWord : CARDINAL;
```

BEGIN

```
  WriteString ("Please type in the line to process ");  
  WriteLn;  
  ReadString (inStr);
```

```

SkipLine;
WriteLn;
(* Write out original *)
WriteString (inStr);
WriteLn;

(* Write backwards letter by letter *)
FOR count := (LENGTH (inStr) - 1) TO 0 BY -1
DO
    WriteChar (inStr [count]);
END;
WriteLn;

(* Now get ready to print the word-reversed string *)
endWord := LENGTH (inStr) - 1;
REPEAT
    WHILE (endWord > 0) AND (inStr [startWord - 1] # blank)
    DO
        DEC (startWord)    (* move back to letter *)
        END;              (* following a blank *)
    (* startword and endword now delimit a word *)

    (* now print one word *)
    FOR count := startWord TO endWord
    DO
        WriteChar (inStr [count]);
    END;
    IF startWord # 0    (* except for the first word *)
    THEN                (* WriteChar a blank *)
        WriteChar (blank);    (* after the word *)
        endWord := startWord - 1    (* reset wordend, carry on *)
    ELSE
        endWord := startWord
    END;
UNTIL endWord = 0

END Backwards.

```

Sample run:

Now is the time for all good men to come to the aid of the party.
.ytrap eht fo dia eht ot emoc ot nem doog lla rof emit eht si woN
party. the of aid the to come to men good all for time the is Now

7.4 Comparing and Manipulating Strings

Some of the material in this section has already been touched upon in this chapter, but is here discussed in more detail. Suppose one has a variable of a *String* type, that is, an ARRAY OF CHAR of some length, that is holding a keyboard input. One then wishes to compare this with some other string--either a literal, or one of a (possibly) different potential length (i.e. of a different formal type). Say, for instance:

```
TYPE
String80  = ARRAY [0 .. 80] OF CHAR;
String10  = ARRAY [0 .. 10] OF CHAR;

VAR
str1, str3 : String80;
str2 : String10;
```

Now, such comparisons as are found in code like

```
IF str1 = str2 ... or
IF str2 = 'January' ...
```

will certainly result in a "Type conflict" error in the first case and ought to in the second one as well (though some non-standard versions have been known to relax the rules.)

Neither can the programmer expect

```
IF str1 = str3 ...
```

to yield meaningful results, even if an ISO compiler would pass the code as correct, which it does not, because it does not permit arrays to be compared. Even if it did, comparisons would need to involve the entire array, and two arrays are equal only if all their entries are equal (including ones after the string terminator in which we have no interest.) The extraneous characters present after the string terminator in the third comparison will usually cause these entities to be unequal as arrays even though one wishes to regard them as equal strings. (This is one of the problems with not using an abstract implementation of *String*; one simply knows too much about the structure, and that knowledge could get in the way.) There is also the difficulty that if one writes something like:

```
IF str1 < str2
THEN
    StringAlongWithUs
ELSE
    StringUpTheUser
END;
```

one will always get an error, as the comparisons "<" and "" are not defined for arrays. Several solutions to these difficulties are possible--most involve appealing to procedures such as *Equal* and *CompareString*, which are designed for the purpose of making such comparisons.

It would not be hard to write such procedures if the system lacked it. The comparison would proceed character by character through the two strings until it found one that was different, and at that point the function value would be returned. The key

comparison could be something like:

```
IF ORD (str1 [count]) < ORD (str2 [count]) ...
```

NOTE: In the ASCII character set that Modula-2 uses, uppercase letters have a lower ordinal value than do lower case letters.

The details involved in completing this are left to the student as an exercise.

Certain very specific situations arise frequently--such as comparing an input string to one of several possibilities in a small list. For these, one may make use of a specially tailored method as illustrated in the next example.

Suppose the need is to compare input keyboard data, say, to the names of months, in order to determine what action to take next. (That action might be based on the number of days in the month, for instance.) One might take advantage of the fact that the first one to three letters are sufficient to determine the month and that in some cases, one letter is sufficient.

Suppose:

TYPE

```
MonthName = (Err, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
```

VAR

```
month : MonthName;
```

```
answer : ARRAY [0 .. 10] OF CHAR;
```

Now, after accepting a keyboard input of the name of a month into the variable *answer*, one could determine which of the twelve enumeration constants it corresponded to in the following way:

```
PROCEDURE MonthEnum (mon : ARRAY OF CHAR) : MonthName;
```

VAR

```
ch : CHAR;
```

BEGIN

```
(* check for unique characters in third position *)
```

```
IF CAP(mon [2])="B"
```

```
THEN
```

```
    RETURN Feb
```

```
ELSIF CAP (mon[2]) = "C" THEN
```

```
    RETURN Dec
```

```
ELSIF CAP (mon[2]) = "G" THEN
```

```
    RETURN Aug
```

```
ELSIF CAP (mon[2]) = "L" THEN
```

```
    RETURN Jul
```

```
ELSIF CAP (mon[2]) = "P" THEN
```

```
    RETURN Sep
```

```
ELSIF CAP (mon[2]) = "T" THEN
```

```
    RETURN Oct
```

```
ELSIF CAP (mon[2]) = "V" THEN
```

```
    RETURN Nov
```

```
ELSIF CAP (mon[2]) = "Y" THEN
```

```
    RETURN May
```

```
END; (* if *)
```

```
(* check for unique characters in second position *)
```

```
IF CAP (mon [1]) = "P" THEN
```



```

    RETURN Apr
  ELSIF CAP (mon [1]) = "U" THEN
    RETURN Jun (* Jul and Aug are done already *)
  END;

(* look at remaining first letters *)
IF CAP (mon [0]) = "J" THEN
  RETURN Jan (* Jun and Jul are done already *)
ELSIF CAP (mon[0]) = "M" THEN
  RETURN Mar (* May is done already *)
ELSE (* any other second letter passes to next step. *)
  RETURN Err; (* anything else is an error *)
END;

```

```

END MonthEnum;

```

Mind you, this method is far from perfect. It will accept "Hug", "Jug" and "Rut" as correct under the first case selection, and it will accept "Hay", "Pay" and "Tadpole" under the second. In fact, more incorrect answers will be processed and assigned a scalar of type *MonthName* than there will be correct ones. Note the need to have an error result that can be returned in order to prevent a run time error in the event that none of the twelve possibilities are selected from the keyboard input. A more elegant approach involves the use of pattern matching and is enclosed in the following:

TYPE

```

MonthName = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);

```

```

PROCEDURE ConvertToMonth (name: ARRAY OF CHAR;

```

```

    VAR gotIt : BOOLEAN; VAR result: MonthName);

```

```

(* MonthName is the Type defined above. *)

```

VAR

```

length, (* Length of Name passed in *)
monthNameCounter, (* Counter on MonthNames *)
count (* Counter on Name passed in *) : CARDINAL;
NamesOfMonths : ARRAY [0 .. 35] OF CHAR;

```

BEGIN

```

NamesOfMonths := "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC";
(* First, get length of Name passed in, or use 3, whichever is less. Also,
capitalize it at the same time for easier matching *)

length := 0;
WHILE (length <= 2) AND (name [length] <"length" now holds the actual length or 3
*)
monthNameCounter := 0;
(* "monthNameCounter" will count through "MonthNames" *)
gotIt := FALSE;

WHILE (NOT gotIt) AND (monthNameCounter < 36)
(* last try is at letter #35 *)
DO

```

```

count := 0;    (* and "count" counts up to the length *)
WHILE (count < length) AND (name [count] = NamesOfMonths [monthNameCounter])
  DO    (* try to match "length" characters in a row *)
    INC (count);
    INC (monthNameCounter);
  END;
IF count = length    (* we did it *)
  THEN
    gotIt := TRUE;    (* tell outside world *)
    DEC (monthNameCounter); (* because start of next one *)
    result := VAL (MonthName, monthNameCounter DIV 3);
    (* and return month name *)
  ELSE    (* no, not yet, so skip to start of next month*)
    monthNameCounter := (monthNameCounter + (3 - count))
  END    (* if count *)
END;    (* first while *)

END ConvertToMonth;

```

NOTE: The parameter *gotIt* is provided, because the value returned is not defined when *gotIt* is false. This procedure is also not perfect, and may match things incorrectly if not used with care. For instance, if it is called with just "J" it will match the first string starting with that letter and return the value *Jan*. If it is called with the string "Ma" it will return *Mar* and never get to the letters describing *May*. It will not matter what is typed after the third letter if a match is possible. Thus, "*Jan* is a good kid" will produce Jan. However, if sufficient letters are provided in the input string to allow for a unique match, it will find one. Naturally, this same method could also be used to match somewhat longer strings.

[Contents](#)

7.5 An Application for Strings--Program Menus

Our major illustration on the use of strings is their application to the printing of menus for the users of various programs. A programmer of text-based applications frequently wishes to present screens of information such as the following:

```
Arjay Accounting Program

Menu
Do You wish to:

A. Set up a new account for payables
B. Set up a new account for receivables
C. Post an entry to an existing account
D. Rename an existing account
E. Close the books for the current month
F. Print a statement for the current month
G. Put everything away and quit the program

Please pick a letter (A ... G) here ==>
```

A pleasing screen layout like the one shown here gives the final product a professional look and feel, but takes time to achieve. (Of course, there are more sophisticated menus available under graphical user interfaces such as the Macintosh operating system, but employing these is a more advanced topic.) Since such menus commonly appear even in short programs, and there may be many of them in a larger piece of work, it is worthwhile to attempt to automate the process of creating such screens and obtaining the user's response.

Much of what is in the menu above is standard from one program to another or from one portion of a program to another. The layout of the information on the screen and the code to accept and check the validity of the user's choice does not change from one instance of a menu to another. Only the name of the program and the number and wording of the choices varies--otherwise the menus are handled in a very similar manner in each instance.

It is worthwhile, therefore to treat the printing of a menu as an abstract task in itself, encapsulating the necessary procedures in a utility module that takes the client programs' collection of menu strings, prints them in the form shown above, and then obtains and checks the user response. For the sake of simplicity, the implementation below makes the following assumptions:

- The screen is eighty characters wide and twenty-four lines high.
- The material is to be centred on the screen in a field whose width is determined by the length of the longest menu string passed to the menu printer and whose height is determined by the number of choices to print. The remaining space around all sides is to be blank.
- The user is required to keep trying if no valid input is given in response to the prompt.
- Data is fed to the main printing menu in the form of an appropriate array of strings, with the title of the menu first.
- The routine to accept the user's response and check it to ensure that it is correct is a separate procedure.

With all this in mind, here is the definition for a menu handler module:

DEFINITION MODULE MenuHandler;

```
(* by R. Sutcliffe
revision date 1993 04 06 *)
```

CONST

```
MaxMenuItems = 14;
```

TYPE

```
MenuStringType = ARRAY [0 .. 76] OF CHAR;
(* ensure there is room on screen for longest possible one *)
```

```
MenuType = ARRAY [0 .. MaxMenuItems] OF MenuStringType;  
(* leaves nine lines for the other material on the screen *)
```

```
PROCEDURE PrintMenu (VAR menu : ARRAY OF MenuStringType);  
(* prints a menu on the screen  
pre: the parameter has been initialized with the title (first) and the items  
following. If not all MaxMenuItems items after the title are used, the last one must  
be the empty string.  
post : the menu is printed *)
```

```
PROCEDURE GetUserChoice ( ) : CHAR;  
(* pre: none  
post: a character corresponding to a menu choice is returned *)
```

```
END MenuHandler.
```

The user interface has been kept very simple. Part of this simplicity is ensured by requiring that the user employ entities of the *MenuStringType* type. This means that it will be easy for the code in the implementation to determine the number of items in the menu.

Now, for the actual code. It is written so that the screen format outlined in the initial example is followed. That is, the sample menu illustrated above has been taken as specifying the behaviour of the procedures in this library module. Notice also that certain simple tasks have been off-loaded from the main procedures and implemented using small local (non-exported) routines.

```
IMPLEMENTATION MODULE MenuHandler;
```

```
(* by R. Sutcliffe  
revision date 1993 04 06 *)
```

```
FROM STextIO IMPORT  
  ReadChar, WriteString, WriteLn, WriteChar, SkipLine;
```

```
CONST  
  spc = " ";  
  cString1 = "Please pick a letter (A .. ";  
  cString2 = ") here ==";  
  screenHeight = 24;  
  screenWidth = 80;
```

```
VAR  
  numberOfItems: CARDINAL;  
  mString, dString, pString : ARRAY [0..42] OF CHAR;
```

```
(* First, a couple of local (non-exported) procedures. *)  
PROCEDURE WriteSpace (numOfSpaces: CARDINAL);  
(* This procedure writes the specified number of blank spaces. *)
```

```
VAR  
  count: CARDINAL;  
BEGIN  
  FOR count := 1 TO numOfSpaces  
    DO  
      WriteChar (spc)
```

```

    END
END WriteSpace;

PROCEDURE WriteLines (numOfLines: CARDINAL);
(* This procedure writes the specified number of blank Lines. *)

VAR
    count: CARDINAL;
BEGIN
    FOR count := 1 TO numOfLines
        DO
            WriteLn
        END
    END
END WriteLines;

PROCEDURE FindFieldWidth (VAR menu: ARRAY OF MenuStringType): CARDINAL;
(* examines all the strings in the menu set and determines which is the longest *)

VAR
    max, count: CARDINAL;
BEGIN
    (* first, set the maximum to the Length of the last or put away choice. *)
    max := LENGTH (pString);
    FOR count := 0 TO HIGH (menu)
        DO (* Then see if one of the ones passed is longer. *)
            IF LENGTH (menu [count]) <= MaxMenuItems) AND (LENGTH (menu [numberOfItems])
"Quit" plus headings, blank lines, and the line for the choice at the end. *)

            fieldWidth := FindFieldWidth (menu);
            spaceToLeave := (screenWidth - fieldWidth) DIV 2;

            (* Print the headings *)

            WriteLines ( (screenHeight - fieldHeight) DIV 2 );
            PrintCentred (mString);
            WriteLines (2);
            PrintCentred (menu [0]);
            WriteLines (2);
            PrintCentred (dString);
            WriteLines (2);

            (* Now print the menu items *)

            FOR count := 1 TO numberOfItems
                DO
                    WriteSpace (spaceToLeave);
                    WriteChar (CHR (ORD ("A") + count - 1));
                        (* letter for this choice *)
                    WriteString (". ");
                    WriteString (menu [count]);
                    WriteLn;
                END;

            count := numberOfItems + 1;

```

```

WriteSpace (spaceToLeave);
WriteChar (CHR (ORD ("A") + count - 1));
WriteString (". ");
WriteString (pString);    (* quit choice *)
WriteLines (2);
WriteSpace (spaceToLeave + 4);
WriteString (cString1);
WriteChar (CHR (ORD ("A") + count - 1));
WriteString (cString2);
END PrintMenu;

PROCEDURE GetUserChoice ( ) : CHAR;

VAR
    answer: CHAR;
BEGIN
    REPEAT
        ReadChar (answer);
        SkipLine;
        answer := CAP (answer);
    UNTIL (ORD (answer) = "A") AND
           (ORD (answer) <= ORD ("A") + numberOfItems);
    RETURN answer
END GetUserChoice;

BEGIN (* main body to initialize variables *)
    (* These are made variables because one cannot pass a
       constant as an array of char in a variable parameter *)
    mString := "MENU";
    dString := "Do you wish to:";
    pString := "Put everything away and quit this program.";
END MenuHandler.

```

Notice that in this version, if a letter choice is typed that is out of the range of the correct choices, it is ignored and no action is taken. *GetUserChoice* will simply wait until a correct choice is typed. This behaviour could be changed. The initial part of a suitable client program might include code like this:

```

MODULE ClientTestMenu;

FROM MenuHandler IMPORT
    MenuType, PrintMenu, GetUserChoice;

VAR
    actionToDo: CHAR;
    menu: MenuType;

BEGIN
    menu [0] := "Arjay Enterprises Word Processor";
    menu [1] := "Enter the Editor";
    menu [2] := "Enter the Runoff/Formatting subprogram";
    menu [3] := "Enter the Communications Subprogram";
    menu [4] := "Configure the Runoff/Formatting Subprogram";
    menu [5] := "Configure the Communications Subprogram";

```

```
menu [6] := "Print a Catalog of the Disk";
menu [7] := "";
```

REPEAT

```
PrintMenu (menu);
actionToDo := GetUserChoice ( );
CASE actionToDo OF
    'A':
        (* put all choices and actions here
        ..... *)
    | 'F':    (* last one for program *)
    | 'G' :   (* corresponds to choice quit program *)
        (* putEverythingAway; *)

    ELSE
    END;
UNTIL actionToDo = 'G'
    (* goes forever unless last choice picked *)
END ClientTestMenu.
```

Observe that this test program *is* just a shell. It does not (yet) do anything but run and put up the menu. However, this is exactly the way one would proceed to write a large project. The shell would be written and the user interface tested and perfected. Then, the sections of the program corresponding to the individual menu choices would be implemented one at a time, with the unimplemented sections remaining as stubs until they were completed. In this way, the programmer always has a working program on hand, and the highest level portions of it (the menu) are tested many times during the tests of the lower level parts.

[Contents](#)

7.6 Message Encoding and Cryptography

The encoding of messages into a form that a casual reader cannot decipher is more than just a schoolchildren's game. Sensitive data is often protected in such a manner, either for storage or for transmission to other parties. Intelligence gathering agents rely on such techniques to forward reports to their control authorities without the corresponding agencies of hostile nations having access to their contents. Counter intelligence operations simultaneously work hard to break such codes and read the messages after all. Lives and whole wars have been won or lost because of the quality of a nation's code making or code breaking techniques.

One of the simplest types of ciphers (and one of the easiest to break) is based on a simple substitution. Each letter in the original message is replaced by a corresponding letter in the coded version. The decoder must apply the substitution in reverse to render the code in plain text. The substitution could be a simple shift:

"ABCDEFGHIJKLMNOPQRSTUVWXYZ "

is coded by

"BCDEFGHIJKLMNOPQRSTUVWXYZA "

Or, it could be coded by a less regular replacement:

"QAZXSWEDCVFRTGBNHYUJMKIOLP "

The means for doing this is rather easy to render in Modula-2.

```
DEFINITION MODULE Substitution;
```

```
TYPE
```

```
  CodeString = ARRAY [0 .. 25] OF CHAR;
```

```
PROCEDURE Encode (source : ARRAY OF CHAR;
```

```
  VAR dest : ARRAY OF CHAR; key : CodeString);
```

```
(* encode the source string with a substitution cipher defined by the key into the  
destination string *)
```

```
PROCEDURE Decode (source : ARRAY OF CHAR;
```

```
  VAR dest : ARRAY OF CHAR; key : CodeString);
```

```
(* decode the source string with a substitution cipher defined by the key into the  
destination string *)
```

```
END Substitution.
```

```
IMPLEMENTATION MODULE Substitution;
```

```
(* several local procedures are useful *)
```

```
PROCEDURE AlphaPos (ch : CHAR) : CARDINAL;
```

```
(* returns the position of a character in the alphabet in the range 0 .. 25 *)
```



```

BEGIN
    RETURN ORD (CAP (ch)) - ORD ('A');
END AlphaPos;

PROCEDURE DecodeKey (skey : CodeString; VAR dKey : CodeString);
(* construct an inverse coding key for use in decoding *)

VAR
    count : CARDINAL;
    ch : CHAR;

BEGIN
    FOR count := 0 TO 25
        DO
            ch := skey [count];
            dKey [AlphaPos (ch)] := CHR (ORD ('A') + count);
        END;
    END DecodeKey;

PROCEDURE IsLetter (ch : CHAR) : BOOLEAN;
(* determines if the character passed is a letter or some other char *)

BEGIN
    IF ((ORD (ch) <= ORD ('z')))
        OR ((ORD (ch) <= ORD ('Z')))
    THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    END;

    END IsLetter;

(* now the exported procedures *)

PROCEDURE Encode (source : ARRAY OF CHAR;
    VAR dest : ARRAY OF CHAR; key : CodeString);

VAR
    max, count : CARDINAL;

BEGIN
    count := 0;
    max := LENGTH (source);
    IF max <= max - 1) AND (count <= HIGH (dest))
        DO
            IF IsLetter (source [count]) (* only encode letters *)
                THEN
                    dest [count] := key [AlphaPos (source [count])];
                ELSE
                    dest [count] := source [count];
            
```

```

        END;  (* if *)
        INC (count);
    END;  (* while *)
END;  (* if *)
IF count < HIGH (dest)
    THEN
        dest [count] := "";
    END;
END Encode;

PROCEDURE Decode (source : ARRAY OF CHAR;
                  VAR dest : ARRAY OF CHAR; key : CodeString);

VAR
    theKey : CodeString;

BEGIN
    DecodeKey (key, theKey); (* create the decode key *)
    Encode (source, dest, theKey );  (* and encode back with it *)
END Decode;

END Substitution.

```

Here is a brief module designed to test the code above by encoding and then decoding a string.

```

MODULE TestSubstitution;
(* By R. Sutcliffe.  Revised 1993 04 06 *)

FROM STextIO IMPORT
    WriteString, WriteLn, ReadString, SkipLine;

FROM Substitution IMPORT
    Encode, Decode, CodeString;

TYPE
    String = ARRAY [0 .. 255] OF CHAR;

VAR
    inStr, outStr, dStr : String;
    coder : CodeString;

BEGIN
    coder := "PLMOKNIJBHUVYGCTFXRDZESWAQ";
    WriteString ("enter message to encode.");
    WriteString (" End with a carriage return ");
    WriteLn;
    ReadString (inStr); (* get the message *)
    SkipLine;
    WriteString ("Original String:");
    WriteLn;
    WriteString (inStr);  (* echo input *)

```

```

WriteLn;
Encode (inStr, outStr, coder);
WriteString ("encoded string:"); (* write it out encoded *)
WriteLn;
WriteString (outStr);
WriteLn;
Decode (outStr, dStr, coder);
WriteString ("decoded string:");
(* write de-encoded message to check *)
WriteLn;
WriteString (dStr);
WriteLn;
END TestSubstitution.

```

Here is the output (except for the opening message) when this program was run.
Original String:

The quick brown fox jumped over the lazy dog's back.

Encoded String:

DJK FZBMH LXCSG NCW UZYTOK CEKX DJK VPQA OCI'R LPMH.

Decoded String:

THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG'S BACK.

This method of encoding is quite inadequate for any security purposes, for it is too easy to construct the coding string from the pattern of the message. The repeated triplet *DJK* can at once be guessed as *THE*, and the character after the apostrophe as an *S*. The positioning of the letters *P* and *C* strongly suggests that both are encoded vowels. From these clues, it would not take long to decode the message without the key. A frequency analysis of the letters in a longer message would open up its key even more rapidly.

Several more sophisticated methods are available and offer greater security. In one, the letters in the original are shifted ahead (mod 26) in the alphabet by an amount equal to the position of some special letter in a code word. When the letters of the code word are exhausted, they are used again. To decode, the letters need to be shifted backwards by the same amount as derived from the original code word. That is, for the message:

Mother has died. Bring the black lantern.

the code word

beaded

produces the shifts

1,4,0,3,4,3 derived from the alphabetic positions of b,e,a,d,e, and d.

and this pattern of shifts is then applied in a repeating way to each block of six letters throughout the message, yielding

Ostkiu lav gjid. Cviqk ule fobgk pdoxeur.

The result is that a given letter of the alphabet generally translates into a different letter every time it is encountered, making a frequency analysis of the letters worthless. The resulting code is much more difficult to break.

A secret agent would enter the field with a set of encoding words or phrases. Each would be used only once, then discarded. Even the number of letters in the encoding scheme would vary from one message to the next in a random fashion. The people receiving the messages would progress through the same series of encoders to decrypt the messages as they arrived. (This is called a one-pad system.) The pad of encoders might, perhaps be taken from the first lines of a sequence of pages of a popular novel. For even greater security, they might be a set of strings specially chosen to make decoding difficult.

Another use for this system would be to encode data files on a user-entered password. Any other password would produce nothing but gibberish when the file was read back. To make the result even more difficult to decode, the entire ASCII set of characters (128 in all) could be included in the coding scheme.

The code that follows merely re-implements the library module *Substitution*. Even in this version, not all the possible characters are encoded. The control characters, whose ASCII codes fall in the range 0..31 and 127, are skipped in the coding scheme so that only readable text is produced. Punctuation and spaces are encoded, however. Note that character coding and decoding are off-loaded into separate procedures.

IMPLEMENTATION MODULE Substitution;

(* several local procedures are useful *)

PROCEDURE CodeChar (**VAR** ch : **CHAR**; shifter : **CHAR**);

VAR

temp : **CARDINAL**;

BEGIN

IF (**ORD** (ch) < 127)

THEN (* does not alter control characters; just the rest *)

temp := **ORD** (ch) + **ORD** (shifter);

IF temp > 31) **AND** (**ORD** (ch) < 127)

THEN (* does not alter control characters; just the rest *)

temp := **ORD** (ch);

(* Now, if ord (ch) - ord (shifter) < 32, then 95 was subtracted *)

IF **ORD** (shifter) + 32 > 0

THEN

shiftCount := 0;

coderLen := **LENGTH** (key);

WHILE (count <= **HIGH** (dest)) **AND** (count <= max - 1)

DO

ch := source [count];

CodeChar (ch, key [shiftCount]);

shiftCount := (shiftCount + 1) **MOD** coderLen;

dest [count] := ch;

INC (count);

END;

END;

IF count <= **HIGH** (dest)

THEN

dest [count] := ""; (* terminator character *)

```

    END;
END Encode;

PROCEDURE Decode (source : ARRAY OF CHAR;
    VAR dest : ARRAY OF CHAR; key : CodeString);

VAR
    max, count, shiftCount, coderLen : CARDINAL;
    ch : CHAR;

BEGIN
    count := 0;
    max := LENGTH (source);
    IF max <= HIGH (dest)) AND (count <= max - 1)
        DO
            ch := source [count];
            DeCodeChar (ch, key [shiftCount]);
            shiftCount := (shiftCount + 1) MOD coderLen;
            dest [count] := ch;
            INC (count);
        END;
    END;
    IF count <= HIGH (dest)
    THEN
        dest [count] := ""; (* terminator character *)
    END;
END Decode;

END Substitution.

```

The client module used with the first version was re-linked and another sample run conducted with the result below. A better code string would result in an even more scrambled result. Note that if the restriction on coding to and from control characters were removed, the implementation would be easier to run faster (fewer computations) but the result would not be readable as plain text.

Original String:

The quick brown fox jumped over the lazy dog's back.

Encoded String:

EUSo]dSNNuKii_RtMhkde[ahGVp\dU^n^SHuUXtacYV`yXzHU[M

Decoded String:

The quick brown fox jumped over the lazy dog's back.

There are much more sophisticated coding schemes than these, of course, and for very secure messages, enormous computing power is required to decode, and this can only be done by brute force trial-and-error.

Contents

Part B--Other Applications

7.7 Some Statistical Tools

One of the most common applications of computing machinery is that of the analysis of data. It is often the case that a researcher is faced with large amounts of data describing the results of experiments. In order to make sense of this, certain standard tools are often applied, and in this section, two of the simplest are considered. The first of these is the *mean* or ordinary average:

The mean of a group of data is their sum divided by the number of items in the group.

Besides the mean itself, one often wants to know something about how closely clustered are the data points. For instance the mean of the data

13,14,13,15,16,13,15,16,14,15,14,14,13,12,13

is 14 and so is the mean of the data

1,1,1,1,1,1,1,1,1,1,1,1,1,1,196

yet these two data sets are very different, and there has to be a way to measure that difference.

One way of expressing the scattering of data from the mean is given by the standard deviation. It is found by taking the difference between the mean and each data item, squaring this number and adding all these squares. One then divides by the number of items and takes the square root of the final result.

$$\text{StdDev}^2 = \frac{(M - y_1)^2 + (M - y_2)^2 + (M - y_3)^2 + \dots + (M - y_n)^2}{n}$$

where M is the mean. That is, the standard deviation is the square root of the expression on the right.

NOTES: The expression above is used only when the standard deviation is computed on the entire population (all possible data.) If the statistics are gathered on a sample of the population, the denominator is changed to $n - 1$.

The quantity (standard deviation)² is called the variance of the data.

To construct a satisfactory Modula-2 routine for computing standard deviation, note that if the numerator of the above expression is expanded, one gets:

$$M^2 - 2My_1 + y_1^2 + M^2 - 2My_2 + y_2^2 + \dots + M^2 - 2My_n + y_n^2$$

or

$$nM^2 - 2M(y_1 + y_2 + \dots + y_n) + y_1^2 + y_2^2 + \dots + y_n^2$$

Substituting the formula for the mean, namely

$$M = \frac{y_1 + y_2 + \dots + y_n}{n}$$

yields

$$\text{StdDev}^2 = \frac{n \frac{(y_1 + y_2 + \dots + y_n)^2}{n^2} - 2 \frac{(y_1 + y_2 + \dots + y_n)^2}{n} + y_1^2 + y_2^2 + \dots + y_n^2}{n}$$

or

$$\text{StdDev}^2 = \frac{y_1^2 + y_2^2 + \dots + y_n^2}{n} - \frac{(y_1 + y_2 + \dots + y_n)^2}{n^2}$$

In the case of a *sample* population, the n in the main denominator becomes $n - 1$ and this formula is

$$\text{StdDev}^2 = \frac{y_1^2 + y_2^2 + \dots + y_n^2}{n - 1} - \frac{(y_1 + y_2 + \dots + y_n)^2}{n(n - 1)}$$

If a summation notation is employed, these two are written:

$$\text{StdDev} = \sqrt{\frac{\sum_{i=1}^n y_i^2 - \frac{\left(\sum_{i=1}^n y_i\right)^2}{n}}{n}} \quad \text{StdDev} = \sqrt{\frac{\sum_{i=1}^n y_i^2 - \frac{\left(\sum_{i=1}^n y_i\right)^2}{n}}{n - 1}}$$

Whichever of these (whole population or sample) is needed, this form is much easier to work with than the original definition, because the algorithm can operate by storing running totals of numbers as they are read or entered and also storing (separately) the running sum of their squares. The standard deviation can be determined at any point for the data entered thus far, or after all items have been entered, for the division by n or by $n - 1$ can be done at any time.

One could even decide to save storage space by not storing all the data items in an array as they are examined, but instead keeping track only of the two running totals and the number of items. While this suggestion is not in fact adopted in the code that follows, it could be an important one if the number of data entries in some disk file were very large and the amount of available memory comparatively small.

To obtain maximum flexibility, the statistical functions have been divided into two library modules, one a *low level* module whose only purpose is to accumulate the running number of items, sum, and sum of squares as data is fed to it, and to report the three statistical measures when desired. The second, higher level module drives the lower level one and does the computations of mean and standard deviation using results obtained from it.

DEFINITION MODULE LowStats;

(* Library of commonly used low level statistical functions
design by R. Sutcliffe & portions of implementation by Mark Harder
last revision 1993 04 06 *)

PROCEDURE Reset ();

(* Use this procedure before starting to call accumulating variables for a new calculation

Pre: none

Post: the number of items, sum, and sum of squares are all set to zero. Max is set to **MIN (REAL)** and min is set to **MAX (REAL)**

NOTE: The initialization code of the module calls Reset. *)

PROCEDURE Enter (x : **REAL**);

(* Pre: If this is the first call of this procedure for a new set of data, Reset must be called first.

Post: the number of items, running sum and running sum of squares are updated *)

PROCEDURE Size () : **CARDINAL**;

(* Pre: none

Post: returns the number of items accumulated since the last call to Reset. *)

PROCEDURE Sum () : **REAL**;

(* Pre: none

Post: returns the sum of the items accumulated since the last call to Reset. *)

PROCEDURE SumSquares () : **REAL**;


```

    (* Pre: none
    Post: returns the sum of the squares of the items accumulated since the last call
to Reset. *)

PROCEDURE Max ( ) : REAL;
    (* Pre: none
    Post: returns the largest of the items accumulated since the last call to Reset. *)

PROCEDURE Min ( ) : REAL;
    (* Pre: none
    Post: returns the smallest of the items accumulated since the last call to Reset.
    *)

END LowStats.

IMPLEMENTATION MODULE LowStats;

(* Library of commonly used low level statistical functions
design by R. Sutcliffe & portions of implementation by Mark Harder
last revision 1993 04 06 *)

VAR
    count : CARDINAL;
    sum, sumSq, max, min : REAL;

PROCEDURE Reset ( );

BEGIN
    count := 0;
    sum := 0.0;
    sumSq := 0.0;
    max := MIN (REAL);
    min := MAX (REAL);
END Reset;

PROCEDURE Enter (x : REAL);

BEGIN
    INC (count);
    sum := sum + x;
    sumSq := sumSq + x * x;
    IF x < min
        THEN
            min := x
        END;
END Enter;

PROCEDURE Size ( ) : CARDINAL;

BEGIN
    RETURN count;
END Size;

PROCEDURE Sum ( ) : REAL;

```

```

BEGIN
    RETURN sum;
END Sum;

PROCEDURE SumSquares ( ) : REAL;

```

```

BEGIN
    RETURN sumSq;
END SumSquares;

```

```

PROCEDURE Max ( ) : REAL;

```

```

BEGIN
    RETURN max;
END Max;

```

```

PROCEDURE Min ( ) : REAL;

```

```

BEGIN
    RETURN min;
END Min;

```

```

BEGIN (* initialization code *)
    Reset;

END LowStats.

```

Observe that no error handling has been done. There could be problems if the real type is overflowed, for example. In the exercises, the reader is asked to remedy this oversight. In the higher level module that follows, there is some redundancy, for the procedures *Largest* and *Smallest* return results that could be obtained by calling the lower level module directly. However, there are other possible higher end modules that could be written here; not all of them might need the maximum and minimum items.

```

DEFINITION MODULE Stats;

```

```

(* Library of commonly used statistical functions
design by R. Sutcliffe & portions of implementation by Mark Harder
last revision 1993 04 06 *)

```

```

PROCEDURE EnterData (items : ARRAY OF REAL;
                     numItems : CARDINAL);
    (* Pre: the items in use are numbered 0 .. numItems - 1
    Post: data is ready for analysis with statistical functions below *)

```

```

PROCEDURE Largest ( ) : REAL;
    (* Pre: none
    Post: The highest value in the last array submitted with EnterData is returned *)

```

```

PROCEDURE Smallest ( ) : REAL;
    (* Pre: none
    Post: The lowest value in the last array submitted with EnterData is returned *)

```

```

PROCEDURE Mean( ) : REAL;
    (* Pre: none

```

Post: The mean of all the values in the last array submitted with EnterData is returned *)

PROCEDURE VariancePop () : **REAL**;

(* Pre: none

Post: The population variance of all the values in the last array submitted with EnterData is returned *)

PROCEDURE VarianceSamp () : **REAL**;

(* Pre: none

Post: The sample variance of all the values in the last array submitted with EnterData is returned *)

PROCEDURE StdDevPop () : **REAL**;

(* Pre: none

Post: The population standard deviation of all the values in the last array submitted with EnterData is returned *)

PROCEDURE StdDevSamp () : **REAL**;

(* Pre: none

Post: The sample standard deviation of all the values in the last array submitted with EnterData is returned *)

END Stats.

IMPLEMENTATION MODULE Stats;

(* Library of commonly used statistical functions
design by R. Sutcliffe & portions of implementation by Mark Harder
last revision 1993 04 06 *)

FROM LowStats **IMPORT**

Reset, Enter, Size, Sum, SumSquares, Max, Min;

FROM RealMath **IMPORT**

sqrt;

PROCEDURE EnterData (items : **ARRAY OF REAL**; numItems : **CARDINAL**);

VAR

count : **CARDINAL**;

BEGIN

Reset;

FOR count := 0 **TO** numItems - 1

DO

Enter (items [count]);

END;

END EnterData;

PROCEDURE Largest () : **REAL**;

BEGIN

RETURN Max();

END Largest;

PROCEDURE Smallest () : **REAL**;

```

BEGIN
    RETURN Min();
END Smallest;

PROCEDURE Mean ( ) : REAL;

BEGIN
    RETURN Sum() / FLOAT (Size());
END Mean;

PROCEDURE VariancePop ( ) : REAL;

VAR
    size : REAL;

BEGIN
    size := FLOAT ( Size ());
    RETURN (SumSquares ( ) - (( Sum() * Sum()) / size)) / size;
END VariancePop;

PROCEDURE VarianceSamp ( ) : REAL;

VAR
    size : REAL;

BEGIN
    size := FLOAT ( Size ());
    RETURN (SumSquares ( ) - (( Sum() * Sum()) / size)) / (size - 1.0);
END VarianceSamp;

PROCEDURE StdDevPop ( ) : REAL;

BEGIN
    RETURN sqrt (VariancePop ());
END StdDevPop;

PROCEDURE StdDevSamp ( ) : REAL;

BEGIN
    RETURN sqrt (VarianceSamp ());
END StdDevSamp;

END Stats.

```

Notice how the work has been distributed so that most of the procedures have only a line or two of code. This makes easier to debug than it would be otherwise. In fact, when the test program below was compiled and run, the only errors found were in the client. The library modules needed no corrections after the initial compilation; all their functions worked correctly the first time.

```

MODULE TestStats;

(* by R. Sutcliffe
to test the statistics modules
last revision 1993 04 06 *)

```

```

FROM Stats IMPORT
    EnterData, Largest, Smallest, Mean, VariancePop, VarianceSamp,
    StdDevPop, StdDevSamp;
FROM SRealIO IMPORT
    ReadReal, WriteFixed;
FROM STextIO IMPORT
    WriteString, WriteLn;
FROM SWholeIO IMPORT
    WriteCard;
FROM RedirStdIO (* non-standard *) IMPORT
    OpenOutput, CloseOutput;

PROCEDURE WriteStats;

BEGIN
    WriteString ("largest is ");
    WriteFixed (Largest (), 2, 0);
    WriteLn;
    WriteString ("smallest is ");
    WriteFixed (Smallest (), 2, 0);
    WriteLn;
    WriteString ("mean is ");
    WriteFixed (Mean (), 2, 0);
    WriteLn;
    WriteString ("population variance is ");
    WriteFixed (VariancePop (), 2, 0);
    WriteLn;
    WriteString ("sample variance is ");
    WriteFixed (VarianceSamp (), 2, 0);
    WriteLn;
    WriteString ("population standard deviation is ");
    WriteFixed (StdDevPop (), 2, 0);
    WriteLn;
    WriteString ("sample standard deviation is ");
    WriteFixed (StdDevSamp (), 2, 0);
    WriteLn;
END WriteStats;

TYPE
    DataArray = ARRAY [0 .. 20] OF REAL;

VAR
    theData : DataArray;

BEGIN
    theData [0] := 13.0; theData [1] := 14.0; theData [2] := 13.0;
    theData [3] := 15.0; theData [4] := 16.0; theData [5] := 13.0;
    theData [6] := 15.0; theData [7] := 16.0; theData [8] := 14.0;
    theData [9] := 15.0; theData [10] := 14.0; theData [11] := 14.0;
    theData [12] := 13.0; theData [13] := 12.0;
    theData [14] := 13.0;
    OpenOutput;

```

```

EnterData (theData, 15);
WriteString ("          First Run:");
WriteLn;
WriteStats;
WriteLn;
theData [0] := 1.0; theData [1] := 1.0; theData [2] := 1.0;
theData [3] := 1.0; theData [4] := 1.0; theData [5] := 1.0;
theData [6] := 1.0; theData [7] := 1.0; theData [8] := 1.0;
theData [9] := 1.0; theData [10] := 1.0; theData [11] := 1.0;
theData [12] := 1.0; theData [13] := 1.0; theData [14] := 196.0;
EnterData (theData, 15);
WriteString ("          Second Run:");
WriteLn;
WriteStats;
CloseOutput;

```

END TestStats.

As can be seen, the data chosen for the two runs are the very collections with which this discussion began. The results are given below:

```

      First Run:
largest is  16.00
smallest is  12.00
mean is    14.00
population variance is  1.33
sample variance is  1.43
population standard deviation is  1.15
sample standard deviation is  1.20

      Second Run:
largest is  196.00
smallest is  1.00
mean is    14.00
population variance is  2366.00
sample variance is  2535.00
population standard deviation is  48.64
sample standard deviation is  50.35

```

[Contents](#)

7.8 Random Numbers

A number of applications in mathematics and computer science require collections of *random* numbers over some range. For instance, one may wish to take a political poll by choosing telephone numbers at random. A list of the possible exchanges could be made, and a household or business in one of them selected by generating a random number between 1 and 9999 for the last four digits of the call.

A random number in some range is chosen in such a way that every number is equally likely to be selected on each occasion.

In most cases, including the one just mentioned, a sequence of random numbers, rather than a single number is required. Not only ought each number to be random, but so should the order in which they occur in the sequence. A great deal of work has been done on the properties of such collections of numbers and it is possible to obtain carefully worked out tables of such numbers that satisfy the most stringent criteria of randomness.

What is desired here is to fashion such a sequence using the computer. One difficulty of doing this is the fact that is that once a given piece of code has been written for a computer, the same sequence of random numbers will be generated each time. The numbers produced may well pass various tests of randomness, but they will not be truly *random* in at least one important sense--they are predetermined.

Random numbers generated by formula in a pre-determined sequence are called pseudo-random numbers.

Some sequences of pseudo-random numbers are better than others. Suppose one were working over the range [0 ..99] and used the sequence 0, 1, 2, 3, .. 99 or the sequence 0, 2, 4, 6, .. 98, 1, 3, 5, ...99. In both cases, each number occurs only once (is equally represented), but in neither is any given number equally likely to be at any given position. Moreover, if the second did not start over (mod 99), only half the numbers would occur at all. A random number generator must be able to do better than that.

As already noted, it is difficult to generate a sequence of truly random numbers. However, the following function might do for many non-critical applications. It does generate pseudo-random numbers--and purists might argue that it does not do so very well, but it will serve for some purposes.

```
DEFINITION MODULE Generator;  
(* by R. Sutcliffe  revised 1993 04 06 *)
```

```
PROCEDURE Random ( ) : REAL;  
(* Returns a random number 0 .. 1 *)
```

```
END Generator.
```

```
IMPLEMENTATION MODULE Generator;
```

```

(* by R. Sutcliffe  revised 1993 04 06 *)

FROM RealMath IMPORT
    exp, ln, pi;

VAR
    seed : REAL;
    (* global variable maintained between calls to Random *)

PROCEDURE Random ( ) : REAL;
(* Returns a random number 0 .. 1 *)

VAR
    temp : REAL;

BEGIN
    temp := seed + pi;    (* scramble up digits of seed *)
    temp := exp (5.0 * ln (temp));    (* scramble them some more *)
    seed := temp - FLOAT (TRUNC (temp));
    (* remove whole number part *)
    RETURN seed;
END Random;

BEGIN    (* initialize seed in body of module*)
    seed := 4.0;

END Generator.

```

Notice that in true Modula-2 style the value of *seed* is maintained, hidden away from the main program inside the library module Generator until the next time Random is called. A brief test program follows:

```

MODULE TestGenerator;
(* by R. Sutcliffe  revised 1993 04 06 *)

IMPORT STextIO;
IMPORT SRealIO;
IMPORT Generator;
IMPORT RedirStdIO; (* non-standard *)

VAR
    count : CARDINAL;

BEGIN
    RedirStdIO.OpenOutput;
    FOR count := 0 TO 19
        DO
            IF count MOD 4 = 0 (* write them four per line *)

```



```

    THEN
        STextIO.WriteLine;
    END;
    SRealIO.WriteFixed (Generator.Random (), 10,15);
END; (* if *)
RedirStdIO.CloseOutput;
END TestGenerator.

```

Output:

```

0.0000000000    0.0198669434    0.8190307617    0.5805053711
0.3963012695    0.2736206055    0.6108093262    0.9552612305
0.1240234375    0.3860473633    0.2877807617    0.3228149414
0.0510864258    0.7236328125    0.7287597656    0.4659423828
0.0164794922    0.1307373047    0.2195129395    0.9540710449

```

These numbers *look* somewhat random, but the same sequence is obtained each time this program is run as things stand. This library module could be improved with the addition of:

```

PROCEDURE Randomize (newSeed : REAL);
(* reset the random number seed to a user supplied one *)

```

The most common technique for producing pseudo-random cardinal numbers, that satisfy most tests for randomness is called the *linear congruential* method. It generates a new random number by multiplying the previous one by a constant a , adding one, and take the remainder modulo (MOD) a second constant m . The result is a new random number that lies between zero and $m - 1$. That is,

$$r := (r * a + 1) \text{ MOD } m$$

Any initial value for r can be used as the seed to start things off and, as before, this seed will in fact determine the entire sequence.

Now, a close examination of this formula reveals a potentially serious problem, for if one assumes that r and a are of type CARDINAL, then unless both are kept very small, or the range for CARDINAL is large, the calculations are frequently going to overflow the data type, possibly on the very first attempt to obtain $r * a$. Suppose one were generating random numbers in the range $[0 .. 9999]$. To prevent overflows, say when the previous value of r had been close to 9999, the value for a would have to be less than six, but this is not too practical. Indeed, without going into the details here, it has been found that the best results are obtained if a has one digit less than m , ends with the digits "21" where x is even, and otherwise has no particular pattern in its digits. What the calculation needs to do, then, is $(r * a) \text{ MOD } 10\,000$ but without any possible overflow when $(r * a)$ is computed. To accomplish this, break the number into two parts. Since 10 000 is the square of 100, it is convenient to write

$$r = 100 * r_1 + r_0$$

$$a = 100 * 1 + 0$$

and express the multiplication as

$$r * a = (100 * r_1 + r_0) * (100 * 1 + 0) \\ = (10\,000 * r_1 * 1) + (100 * (r_1 * 0 + r_0 * 1)) + (r_0 * 0)$$

In this answer, one is now interested only in the third term and the rightmost two significant digits of the second term as the first one is already more than 10 000.

Example:

$$7245 * 6175 \\ = (100 * 72 + 45) + (100 * 61 + 75) \\ = 10000 (72 * 61) + 100 (72 * 75 + 45 * 61) + 45 * 75 \\ = 10000 (4392) + 100 (8145) + 3375 (* \text{ break up the second term } *) \\ = 10000 (4392 + 81) + 4500 + 3375 \\ = 10000 (4473) + 7875$$

Had the last number also overflowed 10 000, one step would be added to *carry the one* to the first term. Note, for later use, that this is a way to multiply two numbers in the range 0 .. 9999 and keep track of *all* the digits in the answer--in this case, store 7875 in one CARDINAL, and 4473 in another. (9999² is too large to fit into the cardinal type in many implementations. Even if not, a similar argument could be applied to 9999999, if that were the largest such number representable in the cardinal type.)

Once the whole process is finished in this situation, however, the focus is on only in the rightmost four digits (7875). Putting these observations together in a problem refinement and writing the code yields:

Problem:

Write a procedure to multiply two numbers in the range [0 .. 9999] and retain only the last four digits.

Solution:

Break the numbers into two parts of the form $100 * 1 + 0$.

for a number n

$$1 = n \text{ DIV } 100$$

$$0 = n \text{ MOD } 100$$

Multiply these as binomials as shown above.

The resulting terms are:

$$1 = r_1 * 1 \text{ (understood to be times } 10000\text{)}$$

$$2 = r_1 * 0 + r_0 * 1 \text{ (times } 100\text{)}$$

$$3 = r_0 * 0$$

Add together the parts affecting the rightmost digits.

$$\text{answer} = 100 (2 \text{ MOD } 100) + 3$$

Discard any portion over 10000.

$$\text{answer} = \text{answer} \text{ MOD } 10000$$

Here is the code for this procedure:

```
PROCEDURE Multiply (mplier, mcand : CARDINAL) : CARDINAL;
```

```
(* Pre: mplier and mcand are both in the range [0 .. 9999]
Post: Returns (mplier * mcand) mod 10000 *)
```

TYPE

```
parts = ARRAY [0 .. 1] OF CARDINAL;
```

VAR

```
x, y : parts;
```

BEGIN

```
x[1] := mplier DIV 100;    (* break numbers into two parts *)
```

```
x[0] := mplier MOD 100;
```

```
y[1] := mcand DIV 100;
```

```
y[0] := mcand MOD 100;
```

RETURN

```
((x[1] * y[0] + x[0] * y[1]) MOD 100) * 100
+ x[0] * y[0]) MOD 10000;
```

END Multiply;

With that, it is possible to re-write the procedure Random for cardinals.

PROCEDURE Random () : **CARDINAL**;

CONST

```
a = 73421;
```

BEGIN

```
seed := (Multiply (seed, a) + 1) MOD 10000;
```

```
RETURN seed;
```

END Random;

Finally, there are several ways to initialize r for the first time in the body of the library module containing this procedure. It could be read in from the keyboard, or an attempt could be made to generate it in some truly *random* fashion. So the body could look like this:

BEGIN

```
WriteString (" What is the random number seed");
```

```
ReadCard (r);
```

```
WriteLn;
```

END Randomizer;

Some versions of Modula-2 provide a procedure *BusyRead*. This procedure polls the keyboard for a character and immediately returns either CHR(0) if nothing has yet been typed, or the character typed if one was. So, perhaps randomizing the seed could be done like this:

```

PROCEDURE RandomizeK ( );

VAR
    ch : CHAR;

BEGIN
    STextIO.WriteString ( " Randomizing the seed number " );
    STextIO.WriteLine;
    STextIO.WriteString ( " Please press a key " );
    REPEAT
        BusyRead (ch);
        seed := Random ( );
    UNTIL ch # CHR (0)
END RandomizeK;

```

The idea here is that *seed* begins with whatever number happens to be in the memory at the time the procedure is entered. This depends on the sequence of events since start up of the computer. *Random* is called repeatedly until the user presses a key. Since everyone has a different reaction time and this loop would execute many times before the keypress, the value of the initial *seed* and thus of the whole sequence would be more genuinely random.

Still another method is available to those who have an intimate knowledge of the workings of the particular hardware on which their randomizer is running. Many systems change a particular memory location at regular (or irregular) intervals, and if a method of accessing such a location is known, its contents may be interpreted as a cardinal, and used for the seed.

NOTE: Forcible re-interpretation of the data in some memory location, initially named by a variable of some other type is done using *newName := CAST (newTypeName, variableName)*; where *CAST* is imported from the module *SYSTEM*. Use of any procedures or types in the module *SYSTEM* implies that the code is not portable to some other computing system.

A popular choice is a memory location representing the value held for a clock of some kind. The latter may be a real-time clock, whose contents are a representation of the time and (possibly) the date. It could be a pseudo-clock, sometimes called a *tick count*, which is a location automatically incremented by the operating system as part of various other tasks it performs. The contents of the *tick count* location, where applicable, are predictable for a given time and sequence of events after system start up, but as both of these may vary considerably before the randomizing function is called, the results may be quite satisfactory.

Note also that all references to 10000 and 100 in the above discussion could be replaced by some other numbers, say *m* and *n* with $n^2 = m$ as long as *m* is less than half of the largest possible *CARDINAL*. A suitable expansion is possible if the nonstandard *LONGCARD* is available or the type *CARDINAL* has a sufficiently large maximum value. It might be best to write the algorithm with references to 10 000 and 100 replaced by constants so that these can easily be changed.

It is also worth noting that the last digits of a pseudo-random number sequence cycle through a regular pattern (test this!) and that sequences of random numbers over some specified range that are produced from some generic generator should therefore use the first digits for the numbers in the sequence.

For this reason, most generators return a *REAL* between 0 and 1 (say, by dividing by 10 000 in this case). Given this assumption, a program would then acquire a random number from such a generator--say in the range $[0 .. j - 1]$ by executing the statement:

```
num := TRUNC (j * RealRandom ( ) )
```

The procedure RealRandom might then be:

```
PROCEDURE RealRandom ( ) : REAL;  
  
CONST  
    m = 10 000;  (* easy to change this way *)  
  
BEGIN  
    seed := (Multiply (seed, a) + 1) MOD m;  
    RETURN FLOAT (seed) / FLOAT (m)  
END RealRandom;
```

Here is a library module based on some of the ideas in this section. The implementation was written specifically for the Macintosh computer and uses a system utility to access the memory location containing the tick count, so the code is not portable to other systems.

```
DEFINITION MODULE Randoms;  
  
(* Written by Mark Harder  
revised by R. Sutcliffe 1993 04 15 *)  
  
(* Rnd and Random use the linear congruential generator method *)  
  
PROCEDURE Randomize ( );  
    (* Pre: none  
    Post: Sets the seed for random Rnd based on tick count. *)  
  
PROCEDURE SetSeed (seed : CARDINAL);  
    (* Pre: none  
    Post: Sets the seed to specified seed value. *)  
  
PROCEDURE Rnd ( ) : CARDINAL;  
    (* Pre: none  
    Post: Returns a pseudorandom number of the range of cardinal *)  
  
PROCEDURE RndInRange (low, high : CARDINAL) : CARDINAL;  
    (* Pre: none  
    Post: Returns a pseudorandom cardinal number in the range low .. high *)  
  
PROCEDURE Random ( ) : REAL;  
    (* Pre: none  
    Post: Returns a pseudorandom real number in the range of [0 .. 1) *)  
  
END Randoms.
```

IMPLEMENTATION MODULE Randoms;

(* Written by Mark Harder
for the Macintosh computer (system dependent)
revised by R. Sutcliffe 1993 04 15 *)

FROM SYSTEM IMPORT

CAST; (* means this is non-portable *)

FROM Events IMPORT (* Mac system only *)

TickCount;

FROM RealMath IMPORT

ln, sqrt, pi;

CONST

a = 94621; (* fits pattern of digits and is prime *)

maxcard = **MAX** (**CARDINAL**);

cardmodulus = **FLOAT** (maxcard) + 1.0;

(* can't express cardinal modulus in cardinal *)

VAR

gSeed : **CARDINAL**;

PROCEDURE Multiply (mplier, mcand : **CARDINAL**) : **CARDINAL**;

(* Returns (x * y) mod maxcard *)

TYPE

parts = **ARRAY** [0 .. 1] **OF CARDINAL**;

VAR

x, y : parts;

rtmxcard : **CARDINAL**;

temp : **REAL**;

BEGIN

rtmxcard := **TRUNC** (sqrt (cardmodulus) + 0.01);

(* but can its square root *)

x[1] := mplier **DIV** rtmxcard; (* break numbers into two parts *)

x[0] := mplier **MOD** rtmxcard;

y[1] := mcand **DIV** rtmxcard;

y[0] := mcand **MOD** rtmxcard;

temp := **FLOAT** (((x[1] * y[0] + x[0] * y[1])
 MOD rtmxcard) * rtmxcard + x[0] * y[0]);

IF temp < maxcard (* now do the add 1 and mod maxcard part *)

THEN

INC (gSeed);

ELSE

```

        gSeed := 0;
    END;
    RETURN (gSeed)
END Rnd;

PROCEDURE RndInRange (low, high : CARDINAL) : CARDINAL;

VAR
    range, result : CARDINAL;

BEGIN
    range := high - low;
    result := (Rnd () MOD (range + 1) ); (* in range 0 .. range *)
    INC (result, low); (* adjust range *)
    RETURN (result);
END RndInRange;

PROCEDURE Random () : REAL;

VAR
    temp : REAL;

BEGIN
    RETURN FLOAT (Rnd ())/ FLOAT (cardmodulus);
END Random;

BEGIN
    Randomize ();
END Randoms.

```

Note the changes to the multiplication procedure to make it operate over the entire range of cardinals. There are other ways of doing this, including converting both numbers to reals, performing a real multiplication, stripping the amount over *maxcard* and converting back to a cardinal. You will be asked to produce this and some other refinements of this material in the exercises at the end of this chapter. With the proper imports, a client program could itself employ the keyboard response method of randomizing the seed by changing the line in the procedure *RandomizeK* earlier in this section from

```
seed := Random ();
```

to

```
Randoms.Randomize ();
```

It is also worth observing at this point that some versions of Modula-2 provide random number generator functions provided in a library module. As these are not in any sense standard items, the names, syntax, and location of such items will vary from one implementation to another and they are likely to be machine

dependent. Check the documentation for details.

[Contents](#)

7.9 Longer Cardinals

It was noted in the last section that one could multiply two numbers in the range 0 .. 9999 and retain all the digits in two separate cardinals. Specifically, one can modify the procedure presented there to return the entire product, rather than just some of the digits:

```
PROCEDURE Multiply8 (mplier, mcand : CARDINAL; VAR ansHi, ansLo : CARDINAL);
```

```
TYPE
```

```
  parts = ARRAY [0 .. 1] OF CARDINAL;
```

```
VAR
```

```
  x, y : parts;
```

```
  t1, t2, t3 : CARDINAL;
```

```
BEGIN
```

```
  x[1] := mplier DIV 100;    (* break numbers into two parts *)
```

```
  x[0] := mplier MOD 100;
```

```
  y[1] := mcand DIV 100;
```

```
  y[0] := mcand MOD 100;
```

```
(* work out all three terms in the multiplication *)
```

```
  t1 := x[1] * y[1];
```

```
  t2 := x[1] * y[0] + x[0] * y[1];
```

```
  t3 := x[0] * y[0];
```

```
(* redistribute the contents of t2 to t1 and t3 *);
```

```
  t1 := t1 + t2 DIV 100;
```

```
  t3 := t3 + 100 * (t2 MOD 100);
```

```
(* and from this obtain the least significant digits of the answer. *)
```

```
  ansLo := t3 MOD 10000;
```

```
(* and transfer any part of t3 over 10000 to the high digits *)
```

```
  ansHi := t3 DIV 10000 + t1;
```

```
END Multiply8;
```

One could write out such eight digit numbers with:

```
PROCEDURE Write8 (hi, lo : CARDINAL);
```

```
VAR
```

```
  temp : CARDINAL;
```

```
BEGIN
```

```
  IF hi # 0 (* else write only the digits of lo part *)
```

```
    THEN
```

```
      WriteCard (hi, 1);    (* first group of digits *)
```

```

IF lo # 0
  THEN
    temp := lo;
  ELSE
    (* if=0 and this not done next part prints zeros forever *)
    temp := 1;
  END;

  (* now pad with zeros if needed *)
WHILE temp < 1000 (* four digits needed in second part *)
  DO
    WriteChar ("0");
    temp := temp * 10;
  END;
END; (* if hi # 0 *)
WriteCard (lo, 1); (* always *)
END Write8;

```

All of this is possible with far less work if the type `CARDINAL` has a suitable range. Expand the horizon, however. Suppose the desire were to be able to keep track of (up to) sixteen digit numbers, and be able to add or multiply two of these and still retain up to sixteen digits in the answer. If the range of the built in cardinal is already 8 digits, either use just two to get sixteen digits or use four to obtain thirty-two.

There are several ways that this could be achieved, and some will be examined later in the text. Here, it will be accomplished by using four cardinals, each retaining four digits. Typically, the type is specified in a library module, as abstractly as possible for the tools available thus far.

```

DEFINITION MODULE BigCards;
(* By R. Sutcliffe
  Modified 1993 05 10 *)

```

```

TYPE
  BigCard = ARRAY [0 .. 3] OF CARDINAL;
  (* least significant digits in lowest numbered component *)

```

```

VAR
  bigOK : BOOLEAN;
  (* will be true if last operation ok, false if overflow took place *)

```

```

PROCEDURE Add (first, second : BigCard; VAR result : BigCard);
  (* Pre: None
    Post: the result is the sum of the two numbers. If there is an overflow, the left
    digit is lost and bigOK is false. *)

```

```

PROCEDURE Mul (mplier, mcand : BigCard; VAR result : BigCard);
  (* Pre: None
    Post: the result is the product of the two numbers. If there is an overflow, the
    left digit(s) is/are lost and bigOK is false. *)

```

```

PROCEDURE WriteBigCard (theNum : BigCard);
  (* Pre: None
    Post: theNum is written to the standard output. *)

```

```

END BigCards.

```

The lowest indexed cardinal in variables of type *BigCard* holds the least significant digits. If one has two of numbers of type *BigCard*, one can begin at the rightmost side of the multiplier and multiply each component of the multiplier, carrying as needed. This mimics the algorithm for multiplying two cardinals in decimal notation:

```

      2567
    * 678      -----
    20536
   179690
  1540200      -----
 1740426
```

except that each "column" contains a number in the range 0 .. 9999 rather than a digit in the range 0 .. 9.

```

      256 1142 7895
x      1 5387
-----
      4253 0365
     615 1954 0000
    137 9072 0000 0000
      7895 0000
     1142 0000 0000
    256 0000 0000 0000
    394 0830 4102 0365
-----
multiply 5837 (right component)
multiply 1 (second component)
```

Another way to think about this algorithm is that one must shift the first number (the multiplicand) to the left by a number of columns equal to the position from the right of the component of the second number (the multiplier) currently being used. For instance, to multiply by the second multiplier component from the right, shift the multiplicand left by one component first, effectively splitting the multiplication up as:

```
(10000 (a[1]) + a[0]) * b
(10000 (a[1] * b) + (a[0] * b)
```

If one overflows the data type on the left, the result is invalid and this fact must be flagged. With all these ideas in mind, here are some procedures that could implement this module. Note that *bigOK* is a boolean that is global to all the procedures here but that must be checked by any client programs after calling these procedures if they are to rely on the results. Its role is essentially the same as *Done* in *InOut*.

IMPLEMENTATION MODULE BigCards;

```
(* By R. Sutcliffe
   Modified 1993 05 10 *)
```

```
FROM STextIO IMPORT
    WriteChar;
FROM SWholeIO IMPORT
    WriteCard;
```

```
(* local procedures *)
PROCEDURE Carry (VAR big : BigCard);
```

```
(* Pre: none
Post: Any excesses over 10000 in any component of the number are carried to the next
component on the left *)
```

```
VAR
    count, temp : CARDINAL;
```

```
BEGIN
    FOR count := 3 TO 1 BY -1
    DO
```

```

    temp := big [count];
    big [count] := temp MOD 10000;    (* leave part < 10000 *)
    big [count - 1] := big [count - 1] + (temp DIV 10000);
        (* carry excess *)
    END; (* for *)
IF big [0] > 0
    DO
        DEC (count); (* start at component 3 *)
        IF count + shift <= 3 (* shift only if place to go *)
            THEN
                big [count + shift] := big [count];
            ELSIF big [count] # 0 THEN
                bigOK := FALSE; (* if were supposed to shift, is bad *)
            END; (* if *)
        END; (* while *)
    WHILE count < shift (* now pad the back end with zeros *)
        DO
            big [count] := 0;
            INC (count);
        END; (* while *)

```

END Shift;

(* main procedure variables declared here *)

VAR

```

    mulAnsHi, mulAnsLo,
    count, comCount : CARDINAL;
    temp, prod : BigCard;

```

BEGIN (* main procedure *)

```

    Init (result);
    bigOK := TRUE;
    (* may be reset to false by either shift or carry *)
    FOR comCount := 0 TO 3    (* count on multiplier *)
        DO
            temp := mcand;    (* move multiplicand to a temporary *)
            Shift (temp, comCount);    (* shift multiplicand if needed *)
            FOR count := comCount TO 3    (* count in multiplier *)
                DO
                    Multiply8 (temp [count], mplier [comCount],
                        mulAnsHi, mulAnsLo);
                        (* do one component *)
                    result [count] := result [count] + mulAnsLo;
                        (* low part into result *)
                    IF count < 3    (* now put high part in *)
                        THEN
                            result [count + 1] := result [count + 1] + mulAnsHi
                        ELSIF mulAnsHi # 0 THEN
                            (* if count = 3 and mulAnsHi > 0

```

DO

```

    IF theNum [count] # 0 (* else do only digits of next part *)
        THEN

```

```

WriteCard (theNum [count], 0); (* this group of digits *)
IF theNum [count - 1] # 0
    THEN
        temp := theNum [count - 1];
    ELSE
        (* if=0 and this not done next part prints zeros forever *)
        temp := 1;
    END;

    (* now pad with zeros if needed *)
    WHILE temp < 1000 (* four digits needed in next part *)
        DO
            WriteChar ("0");
            temp := temp * 10;
        END;
    END; (* theNum [count] # 0 *)
DEC (count);
END; (* while *)
WriteCard (theNum [0], 0); (* always *)

END WriteBigCard;

END BigCards.

```

By means of such methods, one could implement a full range of operations on the type BigCard, and keep track of any overflows that might take place. The student is asked to extend this module slightly in the exercises, but a fuller treatment of such a data type will be postponed until Chapter 16 when a slightly different and more abstract implementation will be considered.

[Contents](#)

7.10 Matrices

In chapter 5 the matrix was defined and modelled by a two dimensional array. Matrices have a number of applications in various branches of mathematics, and are useful entities for study in their own right.

7.10.1 Matrix Operations

Addition and Subtraction:

If two matrices are the same size (have the same number of rows and columns) then addition is done component by component in the same fashion as for vectors. Subtraction is just adding the opposite. Thus, if

$$A = \begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & -5 \\ 2 & 9 & 7 \\ 1 & 6 & 8 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 3 & -2 \\ -3 & 0 & -5 \\ 6 & 9 & 9 \\ 4 & 6 & -5 \end{bmatrix}$$

Then

$$A + B = \begin{bmatrix} 3 & 7 & 4 \\ 0 & 1 & -10 \\ 8 & 18 & 16 \\ 5 & 12 & 3 \end{bmatrix} \quad \text{and} \quad A - B = \begin{bmatrix} 1 & -1 & 8 \\ 0 & 1 & 0 \\ -4 & 0 & -2 \\ -3 & 0 & 13 \end{bmatrix}$$

One Modula-2 procedure that can do the first of these operations is given below. It uses open array parameters and returns the result of checking for valid data in a boolean parameter.

```
PROCEDURE AddMatrices (A, B: ARRAY OF ARRAY OF INTEGER;  
  VAR C: ARRAY OF ARRAY OF INTEGER;  VAR ok: BOOLEAN);  
  
  VAR  
    aRow, aCol, bRow, bCol, cRow, cCol,  
    rowCount, colCount : CARDINAL;  
  
  BEGIN  
    aRow := HIGH (A);  
    bRow := HIGH (B);  
    cRow := HIGH (C);  
    aCol := HIGH (A[0]);  
    bCol := HIGH (B[0]);  
    cCol := HIGH (C[0]);  
  
    IF (aRow # bRow) OR (aRow # cRow) OR (aCol # bCol) OR (aCol # cCol)  
    THEN  
      ok := FALSE;  
      RETURN; (* gives up *)  
    ELSE  
      ok := TRUE;  
      FOR rowCount := 0 TO aRow  
      DO  
        FOR colCount := 0 TO aCol  
        DO
```

```

        C[rowCount, colCount] :=
            A[rowCount, colCount] + B[rowCount, colCount]
    END
END
END
END AddMatrices;

```

A similar method can be used to write a procedure *SubMatrices* to subtract two matrices. Products of matrices are more complicated. First, the product of a row by a column of the same length is defined as the matrix whose single entry is the sum of their component-wise products. That is, if

$$A = \begin{bmatrix} 1 & 4 & 0 & -7 \end{bmatrix} \text{ and } B = \begin{bmatrix} 2 \\ 3 \\ 5 \\ 1 \end{bmatrix}$$

then $AB = [1(2) + 4(3) + 0(5) + -7(1)] = [7]$

If A is a matrix with n rows each of length m (n by m) and B is a matrix with n columns each of length m (m by n), then the product AB is defined by calculating its ij th entry as the product of row i in A by column j in B in the sense shown above. Thus if

$$A = \begin{bmatrix} 2 & 4 \\ 3 & 1 \\ 2 & 9 \end{bmatrix} \text{ and } B = \begin{bmatrix} 1 & 3 & -2 \\ -3 & 0 & -5 \end{bmatrix}$$

then

$$AB = \begin{bmatrix} -10 & 6 & -22 \\ 0 & 9 & -11 \\ -25 & 6 & -49 \end{bmatrix} \text{ and } BA = \begin{bmatrix} 7 & -11 \\ -16 & -57 \end{bmatrix}$$

Note that not only is AB not equal to BA, one product may be defined when the other is not. For instance, if A has four rows and five columns, then AB is defined provided B has 5 rows. BA is defined if B has 4 columns. So, if B is five by two, AB can be computed, but not BA. With this information in hand, it is an easy matter (left for the student as an exercise) to write:

```

PROCEDURE MulMatrices (A, B:  ARRAY OF ARRAY OF INTEGER;
    VAR C:  ARRAY OF ARRAY OF INTEGER;  VAR  ok:  BOOLEAN);

```

7.10.2 Matrices and Determinants

Matrices are used in the solution of a variety of problems, particularly those requiring the solution of systems of linear equations.

Example:

The basic cost of manufacturing shirts retail is \$3.00. For every dollar over this price, the manufacturer is willing to produce another hundred shirts. (Below this, she will of course produce none.) For their part, the maximum consumer demand is for 1000 shirts, but that is only if they are given away for nothing. For each dollar increase in price, the demand is reduced by 125 shirts. Assuming an orderly and rational market, at what level will the sales and price of shirts settle?

Discussion:

This is a simple example of supply and demand economics. It ignores the effect of wholesalers, taxes, and subsidies and presents a rather simple market pattern. Nonetheless, it serves as a starting point for the examination of a variety of similar problems, and for the abstraction of solution techniques.

Solution:

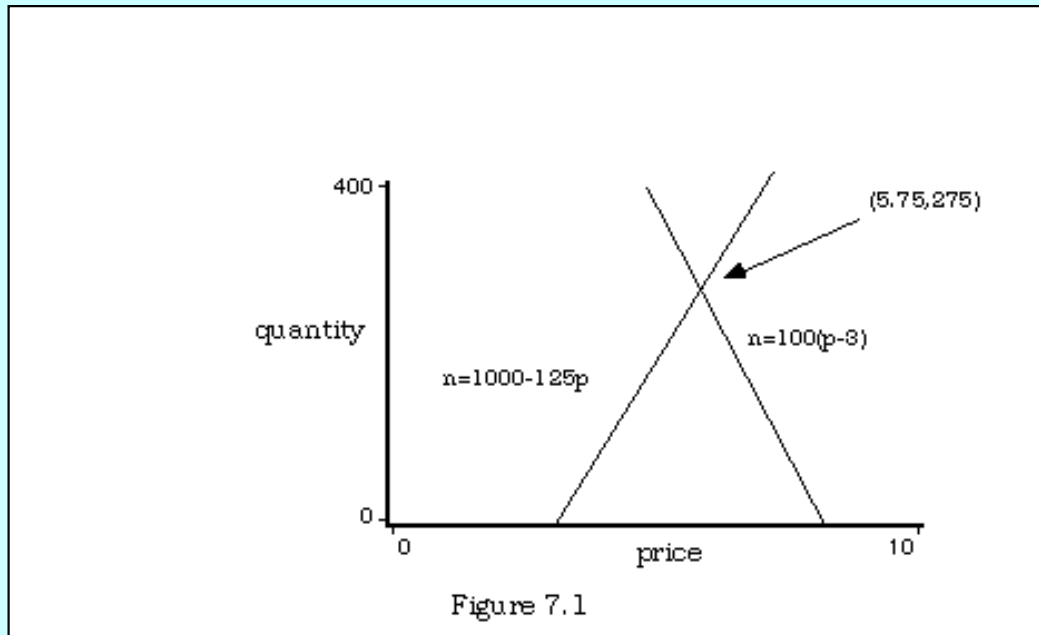
Representing the quantity of shirts by n and the price by p , the relationship in the data provided are:

$n = 100(p - 3.00)$ for the supply of shirts

$n = 1000 - 125p$ for the demand

The market should settle at a point where demand quantity and price match supply quantity and price; that is, at the point represented by the solution to these two equations.

If the two equations are graphed, straight lines are produced, and a visual inspection reveals that they cross at a price of about \$5.75 and a quantity of about 275. (Figure 7.1)



An equation in two variables that can be written in the form $ax + by = c$ (a and b not both zero) is called a linear equation. The point at which the straight lines represented by two such equations cross is called the solution to the system of simultaneous linear equations.

A graphical solution of such equations is not sufficiently accurate, however, and it is more profitable to use an algebraic analysis. The common method is called Gaussian elimination, and is based on the simple notion that equal numbers can be added to both sides of any equation without disturbing its truth value(s) (that is, its solution[s]). Given the linear system:

$$x + y = 7$$

$$x - y = 19$$

and regarding the second as an equality of numbers to be added to the first, one adds the left-hand-sides and right-hand sides to obtain

$$2x = 26 \text{ (the } y\text{-variable has been eliminated)}$$

and then proceeds to multiply both sides by one-half to obtain

$$x = 13$$

Now, substituting this value into one of the originals gives $y = -6$ and the solution $(13, -6)$ as the (x, y) pair that simultaneously satisfies both equations.

One or both of the equations may need to be multiplied on both sides by an appropriately chosen constant before adding to eliminate one or the other variable. Thus one rewrites the system

$$2x + y = 7 \text{ multiply by } 4$$

$$3x - 4y = 5$$

as the equivalent system

$$\begin{array}{rcl}
 8x + 4y & = & 28 \\
 3x - 4y & = & 5 \\
 \hline
 11x & = & 33 \\
 x & = & 3
 \end{array}$$

Substituting this value in one of the original equations produces the solution (3,1).

Likewise, the system

$$3s + 4t = 7 * 4$$

$$4s - 5t = 1 * -3$$

can be re-written as

$$\begin{array}{rcl}
 12s + 16t & = & 28 \\
 -12s + 15t & = & 3 \\
 \hline
 31t & = & 31
 \end{array}$$

or, by multiplying by 5 and 4, respectively, as

$$\begin{array}{rcl}
 15s + 20t & = & 35 \\
 16s - 20t & = & -4 \\
 \hline
 31s & = & 31 \\
 s & = & 1
 \end{array}$$

and the solution (s, t) = (1, 1)

In the case of the initial supply and demand example, the system

$$n = 100 (p - 3.00)$$

$$n = 1000 - 125p$$

is re-written first as

$$n - 100p = -300$$

$$n + 125p = 1000$$

Multiplying the first by -1 and adding yields

$$225p = 130$$

$$p = 5.76 \text{ (nearest cent)}$$

Likewise, multiplying the first by 5 and the second by 4 yields

$$\begin{array}{rcl}
 5n - 500p & = & -1500 \\
 4n + 500p & = & 4000 \\
 \hline
 9n & = & 2500 \\
 n & = & 278
 \end{array}$$

The actual solution (6.76, 278) is quite close to the visual estimate.

Gaussian elimination can be abstracted. Starting with:

$$ax + by = c$$

$$dx + ey = f$$

and using Gaussian elimination, first eliminating y by multiplying by e and -b respectively yields:

$$aex + bey = ce$$

$$-bdx - eby = -fb$$

so that, adding these two produces

$$(ae - bd) x = ce - fb$$

whence

$$x = \frac{ce - fb}{ae - bd}$$

Then, eliminating x by multiplying the original equations by d and -a respectively yields

$$y = \frac{af - cd}{ae - bd}$$

Now consider the *form* of the numerators and denominator of the two fractions in this solution. These three numbers are all of the same kind. If the various coefficients are written in the matrices below, with the first being the original coefficients, and the second and third having the column of coefficients from the right hand substituted for the x-column, and the y-column of coefficients respectively,

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix} \quad \begin{bmatrix} c & b \\ f & e \end{bmatrix} \quad \begin{bmatrix} a & c \\ d & f \end{bmatrix}$$

and in each case, if the a11 and a21 terms are multiplied and then the product of the a12 and a21 terms is subtracted from this, the results are:

$$ae - bd, ce - fb, \text{ and } af - cd.$$

In a two by two matrix A, the product $a_{11}a_{22} - a_{12}a_{21}$ is called the determinant of A.

The determinants of matrixes constructed from the coefficients of a linear equation, by substituting the constants from the right hand side of the equation for either the coefficients of x or y are denoted Dx and Dy respectively.

The solution of the system can therefore be expressed in terms of determinants as

$$x = \frac{D_x}{D} \quad y = \frac{D_y}{D}$$

where D is the determinant of the coefficient matrix.

This method is called Cramer's rule, and it extends to systems with a larger number of variables as well. It turns out that the determinant of a three-by-three matrix is computed from two-by-two sub-determinants as follows. Given:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$\det A = a_{11} \det(M_{11}) - a_{12} \det(M_{12}) + a_{13} \det(M_{13})$$

where the M_{ij} are called *minors* and are obtained by crossing out the i th row and the j th column of the original matrix and working temporarily with whatever is left. The sign of each term is computed by $(-1)^{i+j}$ and so is positive when $(i + j)$ is even, and negative when $(i + j)$ is odd.

Notice that because of the definition of determinants in terms of the determinant of successively smaller matrices, (until one reaches a two by two matrix, for which the calculation is easy), determinants are inherently recursive. In this particular case, the recursive expansion by minors has been done by targeting the first row elements and using the corresponding minors. This is usual, but it turns out that any row or column could have been used. (For a discussion of why this is so, see an appropriate text on linear algebra). Since the definition is recursive, it can be used to compute the determinant of a matrix of any size, provided that sufficient computational power is available to do all the calculations..

Cramer's rule for a system of three equations in three variables can now be expressed as

$$x = \frac{D_x}{D} \quad y = \frac{D_y}{D} \quad z = \frac{D_z}{D}$$

It is also useful to employ the notation of indicating the determinant of a matrix by using vertical bars rather than brackets around the array of numbers. This notation resembles that of absolute value as used with real numbers.

Example:

$$x - 7y + 2z = 7$$

$$3x + y - z = 5$$

$$2x - 3y + z = -10$$

has the solution calculated as above where

$$D = \begin{vmatrix} 1 & -7 & 2 \\ 3 & 1 & -1 \\ 2 & -3 & -10 \end{vmatrix} \quad D_x = \begin{vmatrix} 7 & -7 & 2 \\ 5 & 1 & -1 \\ -10 & -3 & -10 \end{vmatrix} \quad D_y = \begin{vmatrix} 1 & 7 & 2 \\ 3 & 5 & -1 \\ 2 & -10 & -10 \end{vmatrix} \quad D_z = \begin{vmatrix} 1 & -7 & 7 \\ 3 & 1 & 5 \\ 2 & -3 & -10 \end{vmatrix}$$

In order to code this, it is necessary to keep track (at each level of recursion) which rows and columns have been "crossed out". This is easy to do for the rows, as the expansion can be performed on the top row of the minor each time. The trap door for escaping from the lowest level of the recursive calls can be the fact of having arrived at the second-to-last row, for then all but two rows have been expanded upon and crossed out, and only a two-by-two minor remains to be computed. Suppose, for example, while calculating the determinant:

$$\begin{vmatrix} 1 & -7 & 3 & 7 \\ 3 & 1 & 5 & 5 \\ 2 & -3 & -2 & -4 \\ 2 & -3 & -1 & -10 \end{vmatrix}$$

one expanded on the first row, working column by column with the minors in the usual way. At, say the calculation of the third minor, one conceives of the remaining active items of the matrix for subsequent recursive steps looking like:

$$\begin{vmatrix} 3 & 1 & 5 \\ 2 & -3 & -4 \\ 2 & -3 & -10 \end{vmatrix}$$

If one now continues recursively, expanding on the first row of this, then the first minor calculation leaves the active items:

$$\begin{vmatrix} -3 & -4 \\ -3 & -10 \end{vmatrix}$$

If as the recursion has proceeded, a record has been kept as to which columns have been crossed out, it is a simple matter now that the second-to-last row has been arrived at to scan that record for the first two active columns and perform the simple computation that yields the determinant of this two-by-two minor. Keeping track of what columns have been crossed out presents some difficulty. Each cross out needs to be recorded, and this task requires data space. The storage needs for this of a general purpose recursive procedure that is supposed to operate on any two dimensional open array parameter cannot be pre-determined, so the procedure itself must find a way to reserve sufficient memory. This could be done by using a second (one-dimensional) open array parameter of the same length as the number of columns in the matrix. The simplest and most obvious way to do this might be to write:

```
PROCEDURE Determinant (theMatrix: ARRAY OF ARRAY OF REAL;  
    crossed: ARRAY OF BOOLEAN) : REAL;
```

where the second array was the same length as the number of columns (i.e. the length of a row) in the first matrix. It would be up to a client program to declare and pass an array of booleans of the correct size to work with the matrix parameter being used. This approach, however, burdens the client program with more detail than should be used in an abstract approach. It ought to be possible to invoke a procedure to do a determinant computation without requiring the user of the procedure to be concerned with such details.

A solution can be had to this difficulty if the main procedure is simply a shell, and it calls the recursive procedure with an extra parameter for the storage space. The difficulty lies in the main procedure having to compute the size of this second parameter. One might be tempted to write:

```
PROCEDURE Determinant (theMatrix: ARRAY OF ARRAY OF REAL): REAL;
```

```
VAR
```

```
    temp = ARRAY [0 .. HIGH (a)] OF BOOLEAN; (* not allowed *)
```

```
PROCEDURE Det (aMatrix: ARRAY OF ARRAY OF REAL;  
    crossed: ARRAY OF BOOLEAN): REAL;
```

```
(* body of Det with working code here *)
```

```
END Det;
```

```
BEGIN (* main *)
```

```
    RETURN Det (theMatrix, crossed);
```

```
END Determinant;
```

However, a Modula-2 compiler has to be able to calculate the storage space for all variables declared under a VAR heading, and the size of temp is not available until run time. Consequently, Modula-2 does not allow the declaration of arrays using run time calculations such as this. Therefore, another approach to obtaining a data storage of the right size for marking off columns is needed.

Recall that a value parameter makes a copy of data and allows it to be modified without affecting the original. Also recall that a two dimensional array of items (an ARRAY OF ARRAY OF item) can be thought of as a one dimensional array of rows. Thus, if one row (say, the first) of the original matrix is passed to a formal value parameter, a sufficient number of data items will be available within the procedure to flag columns. These items will be of a numeric type rather than the type BOOLEAN,

but that is of no consequence, as numeric values can be chosen to indicate the (inherently BOOLEAN) values *crossed* and *not crossed*. When the recursive procedure is first entered, it can set the values of these data items to some non-zero value (representing *not crossed*); on a subsequent call the entry can be set to zero (representing *crossed*) in the column being crossed out for that call. Since the rows employed for this purpose are to be value parameters, there is no threat posed to the values of the original matrix by making use of a copy of one of its rows for something else. Moreover, when one backs out through the levels of the recursion, the record of columns crossed out is preserved at each level, for alterations made at deeper levels are again made on a copy of the row.

This leads to the employment of two more parameters in the header of the recursive procedure, for it must always have available the number of the row being expanded upon (for the multipliers of the minors) and the column number currently being used (and that it must therefore cross out). Taking all these ideas into consideration leads to the following pseudocode:

```
function procedure determinant
  parameter: open m by n two-dimensional array of real
  returns:   the determinant of the array.
  calls the recursive procedure dMinor, passing
    the original array
    the first row of the original array
    0,0 as the row and column being expanded on.

function procedure dMinor
  parameters: open n by n two-dimensional array of real
              : open length n one-dimensional array of real (crossed)
              : the row and column being expanded on (cardinals)
  returns:   the determinant of the (row, column)th minor

  if row is 0
    call fix row to ensure all items in array crossed are non-zero
  else
    columnth element of crossed is set to 0
  end if
  if row is n-2, call scan to get the first and second column numbers to use
    return a [n-2, first Col] x a [n-1, second Col]-
           a [n-1, first Col] x a [n-2, second Col]
  else
    set sum to zero
    set col count to zero
    while col count < n
      if crossed [col count] # 0 (* use active columns only for minor *)
        add to sum sign (row, col) * a [row, col count] * DMinor (a, crossed,
                                                                    row + 1,
col count)
        alternate sign
      end if
      increment col count
    end while

procedure scan
  parameters: open one-dimensional array of real of length n
              : a cardinal indicating the last column to check
              : (out) two column numbers

  set count to zero
  set got to zero
  while count < n - 1 and got < 2
```

Translating into Modula-2 (with the independent procedures first and the dependent one last) yields the code below. Note that the procedure *FixRow* was not given in the pseudo-form, as it is quite simple.

```

        END; (* IF a [count] *)
    INC (count);
    END (* while *);
END Scan;

```

(* this is the procedure that actually does the work *)

```

PROCEDURE DMinor (VAR a : ARRAY OF ARRAY OF REAL; (* matrix *)
    crossed: ARRAY OF REAL; (* for crossed out rows *)
    row, col: CARDINAL) : REAL;

```

```

VAR
    size, colcount, firstCol, secondCol: CARDINAL;
    sum, sign: REAL;

```

```

BEGIN
    size := HIGH (a); (* number of rows *)
    IF row = 0
        THEN
            FixRow (crossed);
        ELSE
            crossed [col] := 0.0;
        END;
    IF row = size - 1
        THEN
            Scan (crossed, size, firstCol, secondCol);
            RETURN a [size - 1, firstCol] * a [size, secondCol]
                -a [size, firstCol] * a [size - 1, secondCol];

        ELSE
            sum := 0.0;
            colcount := 0;
            sign := 1.0;
            WHILE colcount <= size
                DO
                    IF crossed [colcount] # 0.0
                        THEN
                            sum := sum + (sign * a [row, colcount]
                                * DMinor (a, crossed, row + 1, colcount));
                            sign := -1.0 * sign; (* alternates *)
                        END; (*if *)
                            INC (colcount );
                        END; (* WHILE *)
                    RETURN sum;
                END; (* IF row = size - 1 *)
    END DMinor;

```

(* finally, the procedure that would be exported from a library *)

```

PROCEDURE Determinant (theMatrix: ARRAY OF ARRAY OF REAL): REAL;

BEGIN
    RETURN DMinor (theMatrix, theMatrix[1], 0, 0);
END Determinant;

```

The client using this latter procedure still has to be able to substitute columns appropriately before involving *Determinant*. To do this, it needs access not only to the coefficients on the left-hand-side of the equation, but also to the constants on the right. For this purpose, it is often useful to employ an augmented matrix of n rows by $n + 1$ columns with the last column being the constants on the right-hand-side. In such a method, the equations

$$x - 3y + 7z = 4$$

$$2x + 5y - z = 10$$

$$3x - y + 15z = -5$$

give rise to the so called *augmented* matrix:

$$\left[\begin{array}{ccc|c} 1 & -3 & 7 & 4 \\ 2 & 5 & -1 & 10 \\ 3 & -1 & 15 & -5 \end{array} \right]$$

The bar before the last column is written to emphasize the nature of the matrix as a three by three matrix of coefficients augmented by the column constants from the right hand side. The practitioner who is used to these will usually not use the bar. Also, in common use, employing x, y , and z for the variables loses generality and restricts one to the number of letters available. It is better to denote them $x_1, x_2, x_3, \dots, x_n$. The solution to the system is given by the D_i/D , employing the same notation as previously.

D can be found by passing the entire n by $n+1$ matrix to the procedure *determinant*, as the latter ignores the last column. To compute D_i , it is necessary to use an intermediate step that places column n into column i before calling *Determinant*. This is formulated as:

```
(* the next one is for use with Cramer's rule.  It does a substitution first, then
computes the determinant. *)
```

```
PROCEDURE DetJ (a: ARRAY OF ARRAY OF REAL; col: CARDINAL): REAL;
```

```
(* Pre: none
```

```
   Post: the column at position col is replaced by last column in the array passed.
*)
```

```
VAR
```

```
   count, size: CARDINAL;
```

```
BEGIN
```

```
   size := 1 + HIGH (a);
```

```
   FOR count := 0 TO HIGH (a)
```

```
   DO
```

```
       a [count, col - 1] := a [count, size];
```

```
       (* recall that column numbering inside starts at zero *)
```

```
   END;
```

```
   RETURN Determinant (a);
```

```
END DetJ;
```

One potential difficulty remains to be considered. If one of the equations has coefficients that are multiples of those belonging to another equation in the set, there will not be a unique solution to the system. In such cases, the determinant of the coefficient matrix will be zero. To avoid a run time error, this possibility must be checked before performing the division D_i/D . A client program containing an n row by $n+1$ column augmented coefficient matrix *coefMat* and a vector solution of length n could employ the following code to determine the solutions:

```
det := Determinant (theMatrix);
```

```
WriteString ("The determinant is ");
```

```
WriteFixed (det, 5, 0);
```

```
WriteLn;
```

```

IF ABS (det - 0.0) > 0.000001 THEN "The solutions by Cramer's Rule are: ";
  WriteLn;
  FOR count := 1 TO n
  DO
    WriteString ("x");
    WriteCard (count, 0);
    WriteString (" = ");
    WriteFixed (DetJ (theMatrix, count) / det, 5, 0);
    WriteLn;
  END; (* for *)
ELSE
  WriteString ("There are no unique solutions.");
  WriteLn;
END;

```

Notice that no attempt has been made to distinguish between the case where there is no solution, for instance:

$$2x + y = 7$$

$$2x + y = 13$$

where the elimination method produces $0 = 6$ (inconsistency) and the case where there are many solutions, for instance:

$$x + 2y = 5$$

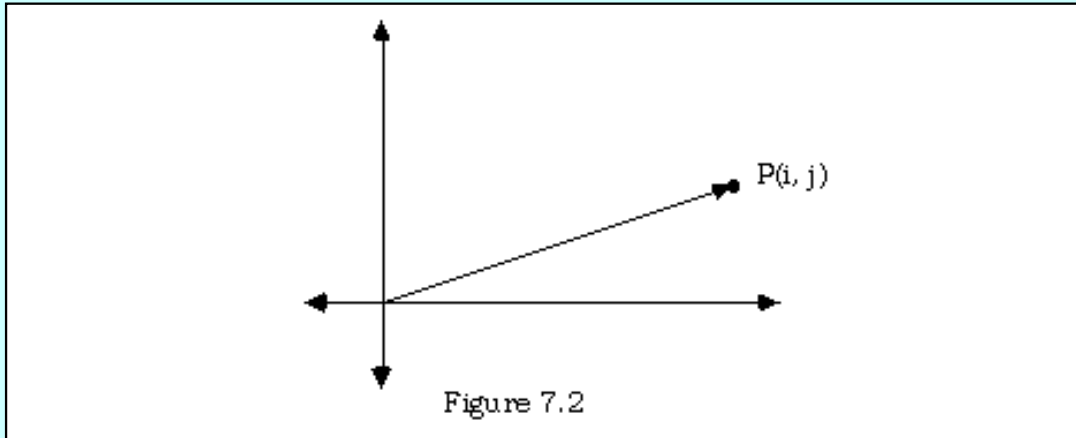
$$2x + 4y = 10$$

where the elimination method produces $0 = 0$ (the equations are dependent--i.e. essentially the same equation). Both of these yield a zero determinant for the coefficient matrix, so the two situations must be distinguished in some other way. This discrimination can be made more easily using methods other than computing the determinant, but that has been left either as an exercise or for discussion by a text more specifically dedicated to this topic.

[Contents](#)

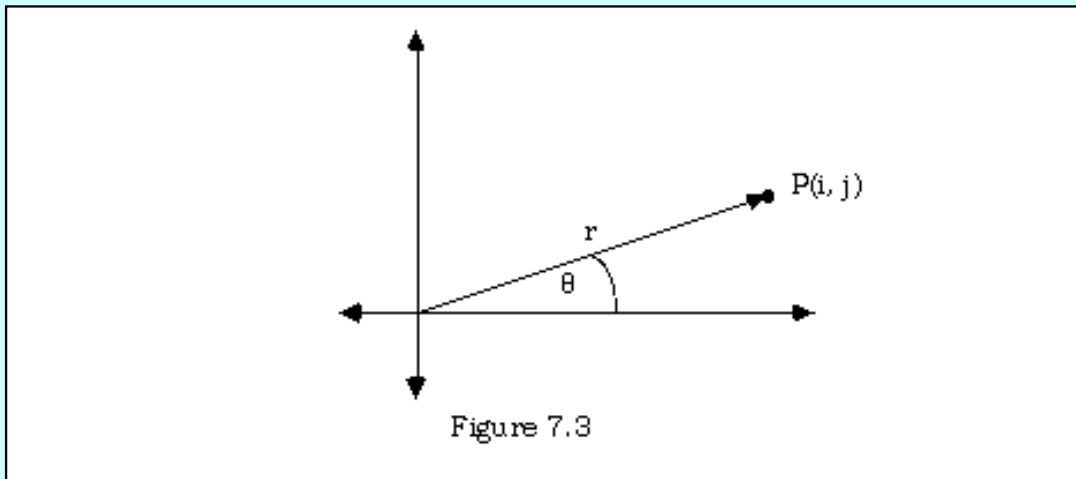
7.11 Applications from Physics

Vectors were introduced in chapter 5 as one-dimensional arrays. The simplest applications for vectors are motivated by representing them geometrically as arrows on a coordinate system.



Thus the length-2 vector $\begin{pmatrix} i \\ j \end{pmatrix}$ is represented by an arrow with its initial point at the origin and terminating at the point (i, j).

Notice that the vector can be completely determined by specifying either the *rectangular coordinates* (i, j), or the *polar coordinates* (r, θ) where r is the length of the vector, and θ is the angle it makes with the positive x-axis.



In most, if not almost all applications employing vectors, it is the latter representation that best abstracts the data.

Examples of vector quantities:

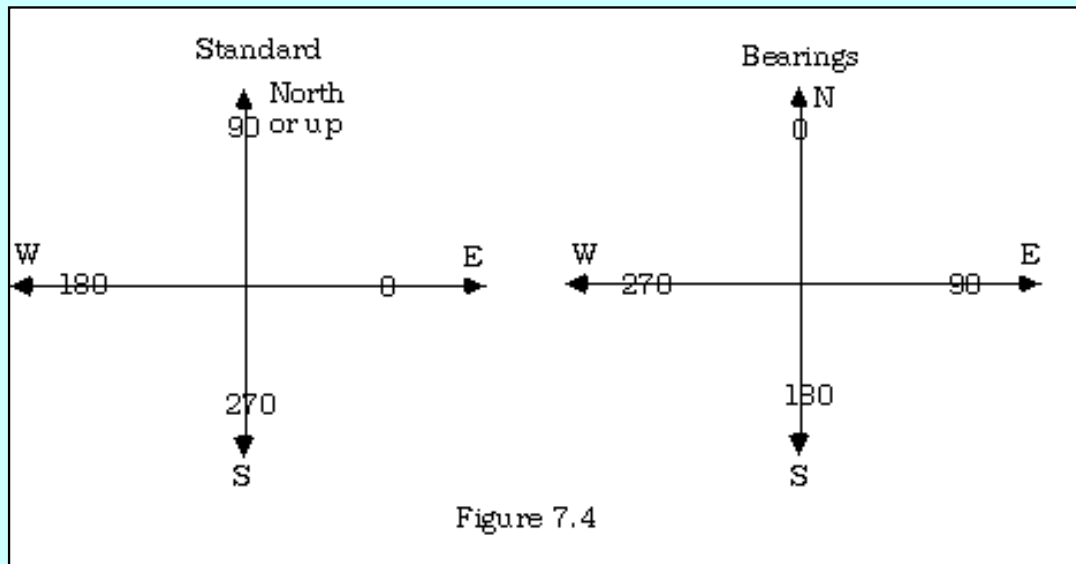
1. A current of 15 km/h from the Northwest.
2. A wind of 80 km/h from the Southeast.
3. A force of 15 Newtons at 45° to the x-axis.
4. A car going 80 km/h East.
5. A vertical velocity of 20 m/s.
6. A plane travelling 1000 km/h on a 180 heading.

A vector is any quantity measured with both a length and a direction. The length r of the vector is also

called the magnitude, modulus or absolute value, and the direction θ is the argument.

A quantity that has only a magnitude but no direction is called a scalar.

The examples of vectors given above are inconsistent, in that some have the direction they point *from* and others the direction *to*. For consistency, actual computations are always done with the direction *to* which the vector points, and this is measured counterclockwise from the positive x-axis, which is identified with East. (Note that *headings* or *bearings* are by convention measured from North at 000 clockwise to 359 and must be translated into the standard system for computations.)



Thus the above examples would be translated into the standard measurement system as: (radian measure in parentheses)

1. 15 km/h at 315° ($7\frac{1}{4}/4$)
2. 80 km/h at 135° ($3\frac{1}{4}/4$)
3. 15 Newtons at 45° ($\frac{1}{4}/4$)
4. 80 km/h at 0° (0)
5. 20 m/s at 90° ($\frac{1}{4}/2$)
6. 100 km/h at 270° ($3\frac{1}{4}/2$)

Thinking about an ADT Vector for a moment, it ought to be clear that there are similarities to the type *Point* discussed in section 6.9. Assignment of a pair of numbers to an abstract vector and the extraction of components (these are called *projections onto the axes*) are both required to maintain the abstract nature of the type. Extraction of the length (absolute value, denoted $|v|$) and angle (argument) as well as the construction of a vector from this information are also required.

Most of the translations defined on the ADT Point are not used with vectors, however. The exceptions are:

1. reflection in the origin, and this is termed *negation* as it results in a vector with the same length but the opposite direction, and
2. scaling the vector, which is termed *scalar multiplication*.

There are other operations required on vectors that are not appropriate for the type *Point*:

Addition:

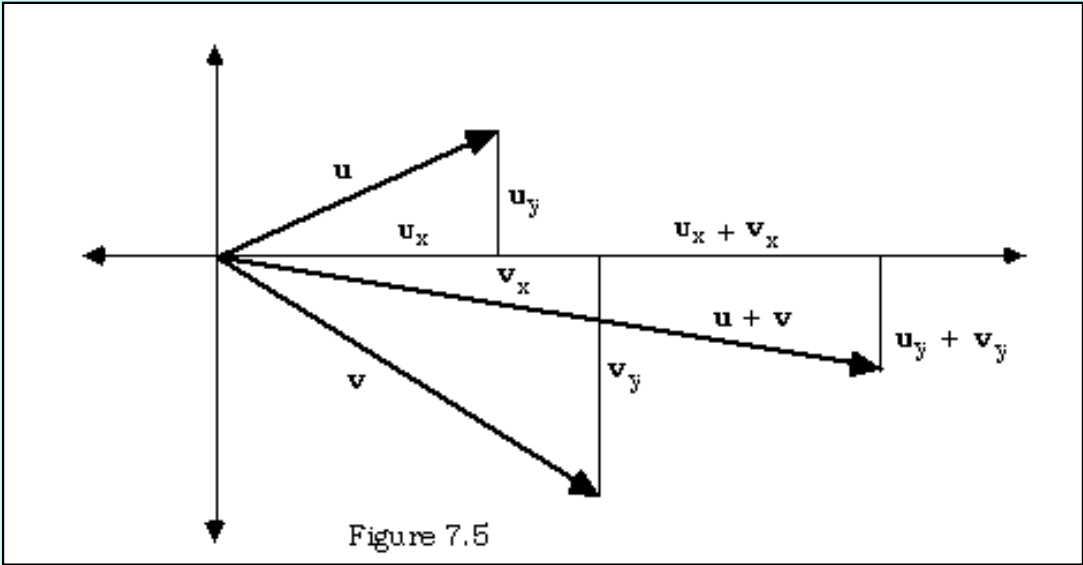
The first operation is motivated by the desire to determine the net effect of the two vectors of a similar kind acting simultaneously.

Example:

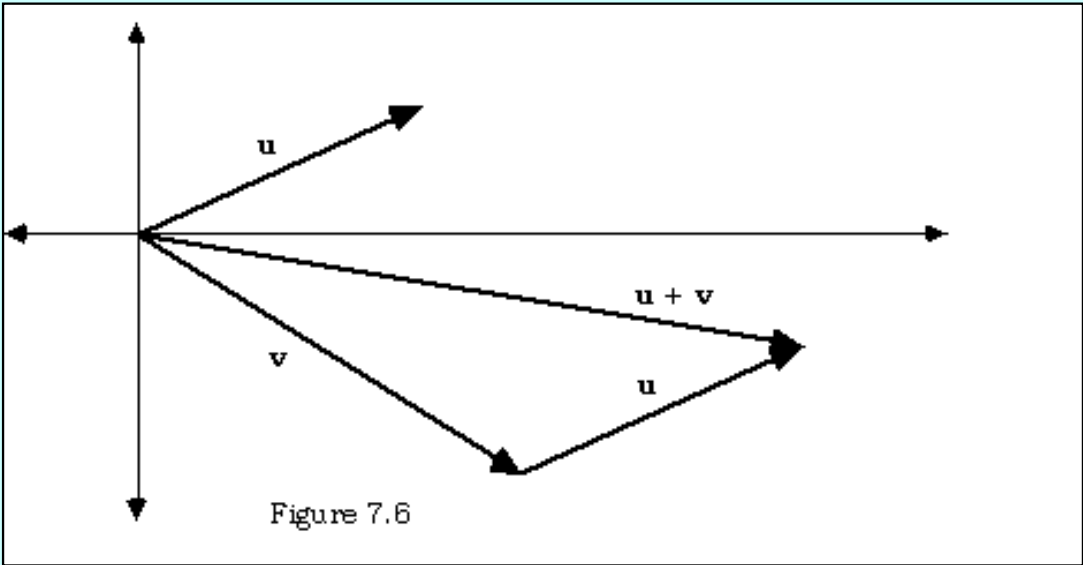
What single force is the equivalent of the two forces 15N at 10° above the positive x-axis and 25N at 40° below the positive x-axis?

Answer:

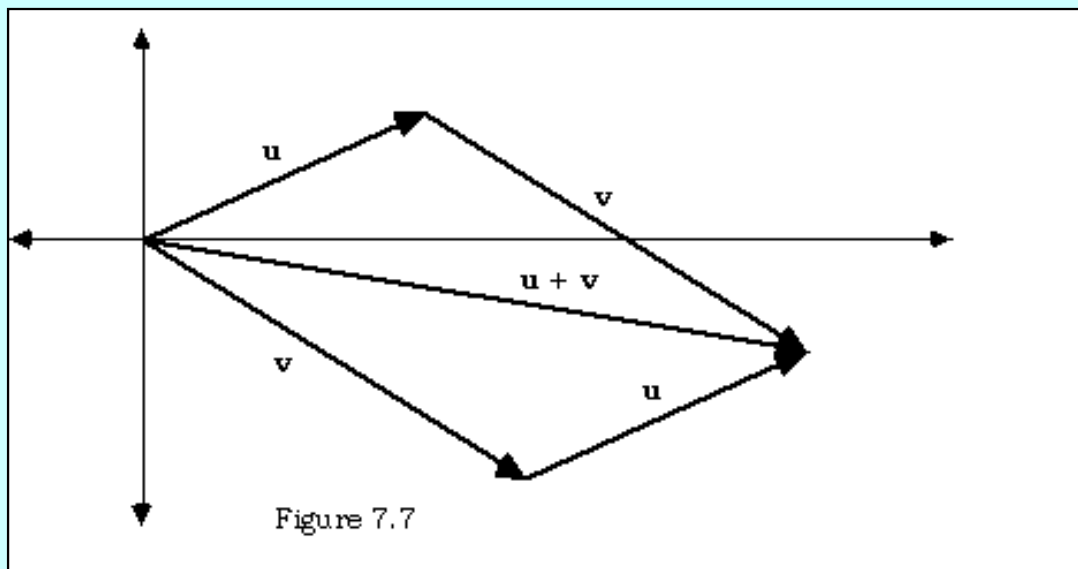
Algebraically, the net forces in the x-direction are $u_x + v_x$ and in the y-direction are $u_y + v_y$.



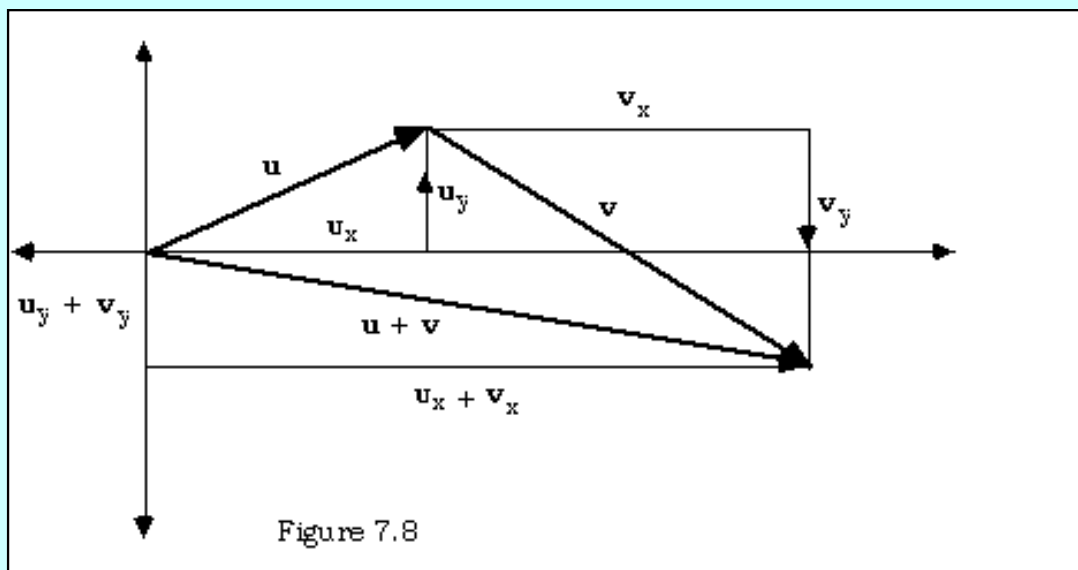
Geometrically, this can be expressed by sliding one of the vectors until its origin becomes the head of the other, and thinking of the two as being in some sense "successive". The sum of the vectors is the diagonal of the triangle formed.



The order in which this is done is irrelevant, so that $(\mathbf{u} + \mathbf{v}) = (\mathbf{v} + \mathbf{u})$. That is, this operation is commutative, and the sum can also be thought of as the diagonal of a parallelogram.



An alternate labelling of the diagram shows the correspondence between the algebraic and geometric views of addition.



The sum of two vectors u and v is the vector formed from summing the corresponding components of u and v .

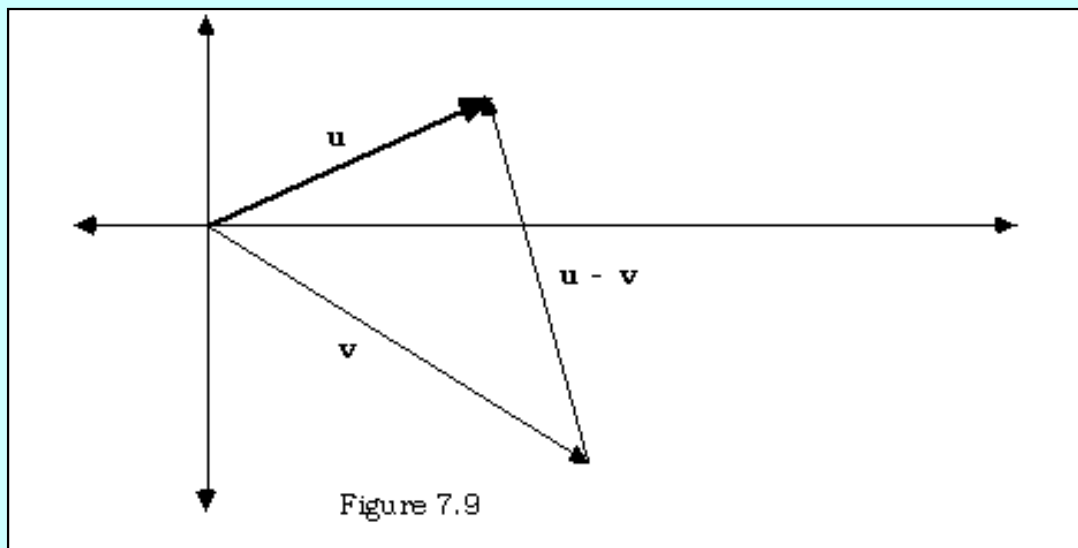
The solution to the specific question posed above is therefore the vector:

$$\mathbf{u} + \mathbf{v} = \begin{pmatrix} 15\cos 10^\circ \\ 15\sin 10^\circ \end{pmatrix} + \begin{pmatrix} 25\cos 320^\circ \\ 25\sin 320^\circ \end{pmatrix} = \begin{pmatrix} 14.77 \\ 2.61 \end{pmatrix} + \begin{pmatrix} 19.15 \\ -16.06 \end{pmatrix} = \begin{pmatrix} 33.92 \\ -13.45 \end{pmatrix}$$

If this is converted into the conventional (r, θ) notation using $r = \sqrt{x^2 + y^2}$ and $\theta = \arctan \frac{y}{x}$ the result can be expressed as a force of 36.4N at 21.6° below the positive x-axis.

Subtraction:

As in arithmetic, subtraction is a short-cut for adding the opposite, that is, $\mathbf{u} - \mathbf{v} = \mathbf{u} + (-\mathbf{v})$. Since negation has already been postulated as an operation on an ADT *Vector*, no further algebra is necessary. Geometrically, since $\mathbf{u} = \mathbf{v} + (\mathbf{u} - \mathbf{v})$, the difference $\mathbf{u} - \mathbf{v}$ between two vectors produces the arrow from the endpoint of first to the endpoint of the second. (The diagram is drawn with \mathbf{u} as the resultant.)



Multiplication:

The only multiplication operation defined on two dimensional vectors is the *dot product*, given by $u \cdot v = u_1 v_1 + u_2 v_2$. (On three dimensional vectors, there is another multiplication called *cross product*.) Here, the corresponding components are multiplied and summed. Since the result is a single number, and not a vector (it has magnitude but not direction), this result is also known as the *scalar product*.

By their very nature, vectors are more often abstracted using polar representation than using rectangular representation. Addition must be done with the rectangular coordinates, but the resultant is almost always expressed in polar form, to correspond with the form in which the addends have been given.

However, there are calculations that can be done in either rectangular or polar coordinates, as convenient, and the dot product is one of these. Refer to [figure 7.9](#) above and suppose θ is the angle between two vectors u and v . By the law of cosines;

$$\begin{aligned}
 |u - v|^2 &= |u|^2 + |v|^2 - 2|u||v|\cos \theta \\
 (u_1 - v_1)^2 + (u_2 - v_2)^2 &= (u_1^2 + u_2^2) + (v_1^2 + v_2^2) - 2|u||v|\cos \theta \\
 u_1^2 - 2u_1v_1 + v_1^2 + u_2^2 - 2u_2v_2 + v_2^2 &= u_1^2 + u_2^2 + v_1^2 + v_2^2 - 2|u||v|\cos \theta \\
 -2u_1v_1 - 2u_2v_2 &= -2|u||v|\cos \theta \\
 u_1v_1 + u_2v_2 &= |u||v|\cos \theta \\
 u \cdot v &= |u||v|\cos \theta
 \end{aligned}$$

This provides a way of computing the dot product from the lengths of the two vectors and the angle between them without

using the rectangular coordinates. Alternately, since from this last line $\cos \theta = \frac{u \cdot v}{|u||v|}$ the angle between two vectors could be calculated from their components without first obtaining the individual arguments as intermediate results.

This latter observation may be more useful in a client program than in a library module, but taking all the other ideas into consideration, the following interface can be constructed.

DEFINITION MODULE Vectors;

TYPE

Vector = **ARRAY** [0..1] **OF** **REAL**;

PROCEDURE assignR (x, y : **REAL**) : Vector;

(* returns the vector with the supplied parameters as rectangular components *)

PROCEDURE assignP (abs, arg: **REAL**) : Vector;

(* returns the vector with the supplied parameters as its length and argument *)

```

PROCEDURE projectX (v : Vector) : REAL;
    (* returns the x-coordinate of v *)

PROCEDURE projectY (v : Vector) : REAL;
    (* returns the y-coordinate of v *)

PROCEDURE abs (v : Vector) : REAL;
    (* returns the length of the vector v *)

PROCEDURE arg (v : Vector) : REAL;
    (* returns the angle between v and the x-axis *)

PROCEDURE neg (v : Vector) : Vector;
    (* returns the vector with opposite components to v *)

PROCEDURE scalarMult (v : Vector; scaleFactor: REAL ) : Vector;
    (* returns the vector that is scaleFactor times as long as v and in the same
direction *)

PROCEDURE add (u, v : Vector) : Vector;
    (* returns the vector sum of u and v *)

PROCEDURE sub (u, v : Vector) : Vector;
    (* returns the vector difference u - v *)

PROCEDURE dotProduct (u, v : Vector) : REAL;
    (* returns the scalar or dot product of the two vectors *)

END Vectors.

```

At this point, it is necessary to make a decision about the meaning of representation of the type *Vector*. (The syntax has already been chosen, but the semantics and the implementation details have not yet.) Will the ordered pair stored contain the rectangular coordinates or the polar coordinates? So long as the client program will not be looking at these numbers directly, it may not matter in one sense. However, unlike the similar type *Point* discussed in [section 6.9](#), vectors are more naturally thought of in polar terms (angle and length) than in rectangular ones (x and y-coordinates). Polar is therefore the best representation.

With this much of the work done, however, and all the important decisions made, the production of the implementation part of the module and a test harness to solve vector problems is left as one of the exercises at the end of the chapter.

[Contents](#)

7.12 Further Applications From Business

There are a variety of additional problems that can be solved using the formulas and techniques of section [4.9](#).

Example 1:

At a given compound interest applied annually, how long does it take to double an investment?

Discussion:

The annual version of the compound interest formula $A_t = A_0 (1 + r)^t$ can have $2A_0$ substituted for A_t (because the amount doubles) and then solved for t as follows:

$$2A_0 = A_0 (1 + r)^t$$

$$2 = (1 + r)^t$$

$$\ln(2) = t \ln(1+r)$$

$$t = \frac{\ln(2)}{\ln(1+r)}$$

If the interest is, say 6%, this yields 11.89, which, since the question specifies annual interest, must be rounded to 12 years.

Note that any growth factor A_t/A_0 could be substituted for the number 2 in this discussion. Modula-2 code would be:

```
time := ( ln ( Amount / Principal ) / ln ( 1 + rate ) )
```

Example 2:

What is the effective annual rate for interest that is compounded more often than annually?

Discussion:

The idea here is that compounding more than annually produces a slightly higher rate of interest than if it were annual. Since many jurisdictions require a lender to report the rate as an annual one, an adjustment must be made.

The nominal interest rate is the rate that is divided by the number of annual compounding periods to obtain the rate per period. The effective annual interest rate is the rate that would be applied annually to produce the same result as the nominal rate.

For instance \$1000 at a nominal 10% for one year compounded each six months yields $(100 * .05) = \$50$, and then $(1050 * .05) = \$52.50$ or 102.50 altogether. Clearly the effective annual rate is 10.25%.

In general, if the nominal rate is i , the number of compounding periods is n , and the effective rate is r , for the two to yield an equivalent amount in t years

$$A_0 (1 + r)^t = A_0 \left(1 + \frac{i}{n} \right)^{nt}$$

Multiplying both sides by $1/A_0$ and then taking t^{th} roots yields

$$(1 + r) = \left(1 + \frac{i}{n} \right)^n$$

or

$$r = \left(1 + \frac{i}{n} \right)^n - 1$$

Thus, if the investment (or loan) is of 10% compounded monthly, we would have

$$r = \left(1 + \frac{.1}{12}\right)^{12} - 1$$

= .1047 or 10.47% for the effective rate.

These new formulas, along with several others discussed in section [4.9](#) can be collected into a library, say, with the following definition:

DEFINITION MODULE FinanceMath;

PROCEDURE SimpleInterest (principal, rate: **REAL**; time: **CARDINAL**) : **REAL**;

(* Pre: none

Post: computes interest= principal * rate * time *)

PROCEDURE CompoundAmountA (principal, rate: **REAL**; time: **CARDINAL**) : **REAL**;

(* Pre: none

Post: computes A = principal (1 + rate)^time *)

PROCEDURE CompoundAmount (principal, rate: **REAL**; time, numberPerYear: **CARDINAL**) : **REAL**;

(* Pre: numberPerYear # 0

Post: computes A = principal (1 + rate/numberPerYear)^(time * numberPerYear) *)

PROCEDURE EffectiveRate (nominalRate: **REAL**; numberPerYear: **CARDINAL**) : **REAL**;

(* Pre: numberPerYear # 0

Post: computes r = (1 + nominalrate/numberPerYear)^numberPerYear - 1 *)

PROCEDURE SCInterestFactor (interestPerPeriod: **REAL**; numberOfPeriods: **CARDINAL**) : **REAL**;

(* Pre: interestPerPeriod # 0

Post: computes the series compound interest factor ((1 + i)^n - 1) *)

PROCEDURE FutureValue (payment, interestPerPeriod: **REAL**; numberOfPeriods: **CARDINAL**) : **REAL**;

(* Pre: interestPerPeriod # 0

Post: computes the future value of a payment by the formula payment * SCInterestFactor *)

PROCEDURE SinkingFundFactor (interestPerPeriod: **REAL**; numberOfPeriods: **CARDINAL**) : **REAL**;

(* Pre: interestPerPeriod # 0

Post: computes the sinking fund factor, which is the reciprocal of the series compound interest factor *)

PROCEDURE AnnuitySize (amount, interestPerPeriod: **REAL**; numberOfPeriods: **CARDINAL**) : **REAL**;

(* Pre: interestPerPeriod # 0

Post: computes the size of an annuity or payment for a total amount by the formula AnnuitySize = R = Amount * SinkingFundFactor *)

PROCEDURE PWorthFactor (interestPerPeriod: **REAL**; numberOfPeriods: **CARDINAL**) : **REAL**;

(* Pre: interestPerPeriod # 0

Post: Computes the series presentworthfactor as (1 - (1 + i)^-n)/i *)

PROCEDURE PresentWorth (payment, interestPerPeriod: **REAL**; numberOfPeriods: **CARDINAL**):


```

REAL;
(* Pre: interestPerPeriod # 0
   Post: Computes the present worth, which is the payment times present worth factor
   *)

PROCEDURE CRFactor (interestPerPeriod: REAL; numberOfPeriods: CARDINAL) : REAL;
(* Pre: interestPerPeriod # 0
   Post: Computes the capital recovery factor, which is the reciprocal of the series
   present worth factor *)

PROCEDURE AmortizePayment (amount, interestPerPeriod: REAL;  numberOfPeriods:
CARDINAL) : REAL;
(* Pre: interestPerPeriod # 0
   Post: Computes the payment to amortize an amount as amount * CRFactor *)

END FinanceMath.

```

Implementation and testing of this library module is left as an exercise for the student.

[Contents](#)

7.13 Chapter Summary

This chapter covered these topics:

- more technical details about Strings and how to use them
- how to design menus for text-based interfaces
- some elementary encryption techniques
- the statistical functions mean, variance and standard deviation
 - techniques for computing the latter by accumulating data
- random number generation
 - a technique for multiplying cardinals and avoiding overflow
- a little about operations on long whole numbers
- some information about matrices and operations on them
- applications of Modula-2 to matrices and vectors

It included discussion of the following Modula-2 built-ins:

Reserved Words	Standard Identifier
-----------------------	----------------------------

<none From Strings:

Length, Assign, Extract, Delete,
Insert, Replace, Append, Concat,
CanAssignAll, CanInsertAll,
CanReplaceAll, CanAppendAll,
CanConcatAll, CompareResults,
Compare, Equal, FindNext,
FindPrev, FindDiff, Capitalize.

- From SYSTEM:

CAST

- From Events (Mac specific):

TickCount

Contents

7.14 Assignments

Questions:

1. What is a Modula-2 string? Explain all terms used.
2. What is an implied abstract data type, and why is it more correct to use this terminology than to say that Modula-2 has a string type?
3. Suppose `str1`, `str2`, and `str3` are of type `String (ARRAY [0 .. 79] OF CHAR,)` and `str1 = "dogs "`, `str2 = "cats "`, `str3 = "chase"`. What will the assigned variable hold in each case, assuming it to be of the correct type?
 - a. `Concat (str1, str3, D); Concat (D, str2, E)`
 - b. `m := Length (str3)`
 - c. `FindNext ("t", str3, 0, found, position)`
 - d. `i := CompareStr ("cat" , str2)`
 - e. `Copy (str2, 0, 1, F); Copy (str1, 1, 3, G); Concat (F, G, H)`
 - f. `Insert (str2, 2, str1); Delete (str1, 6, 2)`
 - g. `FindNext ("cat", "the old cat in the hat", 0, found, position);`
 - h. `FindPrev ("cat", "the cat in the hat sat on the bat", 26, found, position)`
4. "Some string operations may silently truncate the result." What does this statement mean, and under what circumstances may this arise?
5. What is the difference between the *Length* of a string and the number returned by `HIGH` when it operates on a string?
6. When the following pairs of strings are compared, will the result be *less*, *equal*, or *greater*? Why?
 - a. "hare" and "hair"
 - b. "all" and "ALL"
 - c. "rick" and "Rick"
 - d. "%ages" and "pages"
 - e. "1" and "one"
7. For client programs of the module [MenuHandler](#) (Section [7.5](#)) why is it necessary to initialize the first unused string to the null string?
8. (Research) Look up and be able to explain how DES encryption works.
9. Define mean, variance and standard deviation (both versions of the latter.)
10. Look up and write out a definition of the *median* of a data set. What additional facilities would be needed in addition to those in the statistical modules presented to calculate the median?
11. Look up and write out a definition of the *mode* of a set of data. What additional facilities would you suggest are necessary to do this calculation?
12. What are the advantages of separating the low level statistical functions from the high level ones?
13. Why are random numbers generated by a formula not really random numbers? (What *are* they called?) What can be done to make them *more* random?

14. What is a matrix? State at least four substantially different applications of matrices.
15. What is a vector? State at least four substantially different applications of vectors in science.
16. Make a table listing the differences between the ADT *Point* and the ADT *Vector*.

Problems

17. Write a module that will read an input paragraph of text from the keyboard and reformat it and output it to the printer according to the following rules:
 - a. The first line is indented exactly five spaces.
 - b. Multiple spaces between words are reduced to single spaces.
 - c. There are exactly two spaces between sentences, regardless of how many are found in the original text.
 - d. As many complete words are printed on a line as possible and there are no words that are broken between lines. You may assume any convenient line length.
18. Pretend that you do not have the procedures *Insert* and *Delete* available from the module *Strings*. (In some cases, you won't.) Write them, and include them with the procedures *Concat* and *Length* in a module of your own. Test your module by calling on it in a program module and show the results on a printout.
19. Write a program module that will take lines of text as they are typed at the keyboard and save them into a disk file under an appropriate name. The text should go into the disk file as a continuous block with carriage returns only at the end of a paragraph. Define a paragraph as a carriage return followed by one or more spaces.
20. A Pascal string or a Macintosh string is an ARRAY [0 .. 255] OF CHAR, but the first CHAR is in position 1, with str [0] containing a CHAR whose ORDinal value is the length of the string (number of the last position). Note that such strings are limited to lengths of no more than can be accommodated in a storage location used by the type CHAR--usually this means no more than 255 characters to a string. Write a procedure StrModToPas that converts a Modula string to a Pascal string.
21. Add to #20 the complementary procedure StrPasToMod. Encapsulate both in a library module and test them with a suitable client program.
22. A word search game is played with an array of letters and a list of words that the player is expected to find in the array. Words may go in any direction (including diagonally) and a given letter may be part of more than one word.

```
A X Y T J K R E W
B L O O D F F W G
R A U D I T N W Q
A U L M O A O R U
T M I N T M O R E
E A T L Y Y E S R
M O D U L A M G Y
```

Some of the words in the puzzle above include *modula*, *mint*, *blood*, *query*, *brat*, *more*, and there are others. The letters between the words that are to be found and circled can be anything. Write a program

module that will read in the words to be hidden in the puzzle, and construct and print the letter array. The program should either decide the size of the array based on the number and length of the words, or be told this by the puzzlemaster.

23. Write and test a procedure *ConvertToDay*, along similar lines to the procedure [ConvertToMonth](#) in section [7.4](#). Write also the procedure *WhatDay*, similar to *WhatMonth* in the same section. Test both. Now, test again with a driver program that makes a large number of calls to each and see which is more efficient.
24. Implement the menu generating module and test it with one of your own programs.
25. Implement the module [Substitution](#) in section [7.6](#) (the second way) and test it with a driver program that reads the coder/decoder key from the user. Now try encrypting a file and decrypting it again.
26. Rewrite the suite of statistical modules so as to have appropriate error handling. Include

TYPE

```
StatsState = (StatsOK, ...); (* with appropriate values *)
```

VAR

```
StatsStatus : StatsState;
```

and revise the code to set the value of StatsStatus appropriately. If it is available, reduce the probability of an overflow by accumulating the sums in the type LONGREAL.

27. Section [7.8](#) included discussion of the system dependent nature of randomizers that depend on a special memory location that the system periodically updates. Either implement the module [Randoms](#) without this provision, or find such a location for your system, or some similar means of obtaining a seed from the system and rewrite the system dependent code in the suggested module.
28. Write a short program module to generate an array of at least ten thousand random cardinals. Write the array out to a disk file for further reference.
29. At the beginning of section [7.8](#) the procedure *Generator.Random* was developed to produce random numbers between zero and one. Write a new version of the later module [Randoms](#) that uses this procedure as the basis for generating random cardinals or random numbers in some cardinal subrange.
30. The procedure [RndInRange](#) as presented in section [7.8](#) will not work very well if the range is a large fraction of the whole cardinals. For instance, suppose the cardinals are in the range [0 .. 65535] and the selected range for random number generation is [0 .. 9999]. In this case too many of the random cardinals generated have a second digit of 5 or less, so more than the proper share of the numbers in the range [0 .. 9999] will be in the subrange [0 .. 5535]. Compute what are the expected and proper percentage of numbers generated in the subrange [0 .. 5535]. Run the code given in the text, generating a large number of random numbers and determine the actual percentage generated in this range to confirm this. (This will require a replacing of the question with an appropriate range and some extra steps if the range of cardinals is different on your machine.) Now rewrite the code to eliminate this bias.
31. Can you predict what ought to be the standard deviation of a population (and a sample) of randomly distributed numbers over a range? Test your prediction with a suitable program module and the file of random numbers you created earlier.
32. There are a variety of ways of testing the effectiveness of random number generators. One is simply to find the mean of the numbers and see if it lies in the middle of the range selected. You may wish to try

this and to research other methods as well. One method that is fairly easy to implement if your terminal is capable of graphics is to convert the random numbers generated into ordered pairs that represent points on the graphics screen over the complete range of possible horizontal and vertical coordinates allowed (look this up for your machine) and then plot a dot at that point. If your program is allowed to run long enough, it should eventually plot every possible dot and there will be no blank spaces at all on the screen. If, on the other hand, your random number generator is flawed, repetition will set in, and some space will remain no matter how long you allow it to run. How good is yours on a test like this? You will, of course, have to look up the module for handling graphics on your terminal.

33. Rewrite the procedure [Multiply](#) in section [7.8](#) to operate by converting both numbers to reals, performing a real multiplication, stripping the amount over *maxcard* and converting back to a cardinal.

34. Implement Addition, Subtraction, Multiplication, and Division (the latter is a challenge) on the type *BigCard*. You will have to change the error type to an enumeration, as there are more than just overflow errors to contend with now.

35. Modify the code produced in the last section so that the procedures can handle user-defined types of a type similar to but bigger than the convenience type *BigCard* that the suggested module exports. That is, they should take parameters that are ARRAY OF CARDINAL rather than *BigCard* and operate correctly regardless of how many components there are. If you have a larger range for CARDINAL on your machine, you may wish to store more than four digits in each component.

36. Modify the procedure *WriteBigCard* to take a second parameter that is the field length in which to write the *BigCard*. Your code will have to pre-scan the *BigCard* to be written to find out how many spaces it would occupy. If too little is available, it should take what space it needs. If the field length is longer than needed, the number should be padded on the left with blanks.

37. Design, define, implement, and test a module to do matrix arithmetic and operations. You must decide which operations to include, providing a rationale for your decisions, write and compile the definition module, and then implement that definition, define a test harness program, and thoroughly test your implementation.

38. Implement the determinant computation procedures given in section [7.10.2](#) with the addition in the client program of a test to distinguish between the two cases where the determinant is zero, and appropriate messages for the two cases. Also, test your program on systems of two, three, four, five, and six variables, timing each solution. You may wish to make use of files to input the data, if *OpenInput* is available in some form. What is the practical limit on your machine for solving systems of equations in this way?

39. Complete and test the module [Vectors](#) in section [7.11](#) and develop a test harness to check your work thoroughly.

40. A ship heads due East at 15 knots. It is influenced by a current of 10 Knots setting to the Northwest, and a wind of 10 Knots from the South. Ignoring the effects of friction, what will be the resultant velocity? (Solve this with a program module that is a client of the module *Vectors*.)

41. An old problem in probability has a solution that rather astounds many people. If two people are in a room, what is the probability that both were born on the same day of the year? (Answer 1/365--ignore Feb 29) Now toss in a third, and ask what is the probability that two of the three have the same birthday. This is a little harder computation, but still not very difficult. The problem is to discover how many people need to be in the room before it becomes a 50-50 proposition that at least two will have the same birthday. Get the computer to do the computations for you. The answer is not very large, is it?

Research Projects and Challenges

42. What are the most secure encryption algorithms? Implement and test one of these.
 43. What is the cross product of two vectors? What is it used for in practical applications? Design a library module abstracting a three dimensional vector type. Implement and test this, including the cross product. Build a client application to solve a practical problem using cross products and test it.
 44. Implement and completely test the module [*FinanceMath*](#) in section 7.12, ensuring that all formulas are correct by comparing the results with those obtained on a financial calculator. You may decide to add one or more procedures. If you do, be sure to provide a complete rationale and documentation.
-

[Contents](#)

Chapter 7

Solving Real World Problems in Modula-2

[7.0 Chapter Goals](#)

[7.1 Introduction to Some Applications of Modula-2](#)

Part A--Strings

[7.2 Communicating in English](#)

[7.3 Library String Functions](#)

[7.4 Comparing and Manipulating Strings](#)

[7.5 An Application for Strings--Program Menus](#)

[7.6 Message Encoding and Cryptography](#)

Part B--Other Applications

[7.7 Some Statistical Tools](#)

[7.8 Random Numbers](#)

[7.9 Longer Cardinals](#)

[7.10 Matrices](#)

[7.10.1 Matrix Operations](#)

[7.10.2 Matrices and Determinants](#)

[7.11 Applications from Physics](#)

[7.12 Further Applications From Business](#)

[7.13 Chapter Summary](#)

[7.14 Assignments](#)

[Contents](#)

8.0 Chapter Goals

The purpose of this chapter is to explain how data is stored, both within the computer itself, and on separate storage media. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

the abstract notion of storage, and in particular, both low level machine representation of data and the collection of data into files for storage on external devices

the idea of a data stream at the logical and physical level

Realized in the Modula-2 notation:

the types WORD, LOC, and ADDRESS

the type ChanId (File) and the SeqFile and StreamFile models for data streams

Data Manipulation Abstractions

General:

manipulation of low-level data types, including address arithmetic

I/O at high, medium and low levels

Realized in the Modula-2 notation:

the module SYSTEM and its contents as a grouping of low-level and system dependent entities

creating, opening, renaming, writing to and reading from sequential files and restricted streams

high level raw (binary) I/O

the use of the low-level device independent module IOChan

Programming Abstractions

No new programming abstractions are developed in this chapter.

8.1 Storage--An Introduction

To this point the main concern of the text has been issues of data and program structure--how to represent data effectively, and how to implement in the Modula-2 notation an effective means of manipulating that data. Little has been said thus far about how that data is stored, either in the machine itself or on external media. As long as the amount of data is relatively small, this approach suffices, for small amounts of data can be typed in with each program run, and without much concern for how it is stored in the machine.

However, there comes a point when attention must be given to data storage issues, for at the program level the amount of data eventually becomes too large to enter anew with each run of a program. In the real world, data collections are very large, and new programs are frequently written for the sole purpose of manipulating files of existing data collections in ways not previously thought of. Such programs must be structured with the existing stored data in view. The file manipulation facilities provided by the simple classical high level Module *InOut* or that can be attached to *STextIO* using *RedirStdIO* are very primitive and will not do for much beyond simple demonstration purposes. Thus, a Modula-2 implementation always has a suite of more elaborate modules for handling files and arranging for I/O between those files and a running program.

Likewise, there are aspects of the internal data storage that are worthwhile knowing, even though many programmers will not make extensive use of this knowledge. Since some of this has an impact on the understanding and use of file handling modules, this chapter will begin with the internal storage issues, and later consider the external ones. Depending on the depth required by a given student, it may be possible to give this first part only a light read and move into Part B. On the other hand, extensive low level work will require much more experience and knowledge than can be obtained in this chapter. However, the information given here will provide the basis for applying that knowledge to a solution not only when using Modula-2, but also for other notations.

Part A--Machine and System Level Storage Issues

8.2 An Introduction to the Lower Level

Not every aspect of the semantics of a programming language like Modula-2 can be abstractly specified at a high level in the language definition. There are always matters that must be left to the reasonable discretion of the implementor. Some of these (the code for the compiler and many of the library modules) are of little interest or practical use to the user of that implementation, and are private to the vendor. Other items are of more interest to the programmer employing the vendor's product, and these fall into three broad categories:

1. A numerical limitation on the use or complexity of data types or programming structures is called an implementation restriction.

The manufacturer will specify these restrictions in the documentation, and they may include among them some or all of:

- how deeply nested may loops, selections, parentheses or procedures be,
- limits on the size of code that may be generated
- the length of an identifier,
- the length of a string literal,
- the maximum size of an array,
- the maximum number of dimensions of an array,
- the maximum number of items in an enumeration.

The ISO standard imposes minimums on some of these items for a compiler to be able to claim conformance with the standard, but most implementations will have larger limits than those minimums. The user of the product must consult the manual to determine any such restrictions. In general, however, a program that fails because, for instance, it has too many WHILE loops nested one within another is probably not very well written, and would benefit from being re-thought. For practical reasons, programmers often work at levels much lower than those assumed by the programs presented thus far. When doing so, it may be important to know more about the way that some data items are represented or stored.

2. Information provided on the range of possible values for a data type or the size of its storage is said to be implementation defined.

Again, the manufacturer should specify this data in the documentation; it may include among them some or all of:

- the maximum and minimum values of the built in scalar types,
- the maximum available precision for real types,
- the smallest available increment between two adjacent values of a real type,
- the number of storage locations used by each data type.

At times, it is not possible to define in the standard what the exact behaviour of the compiler ought to be at certain points. For example, when passing parameters to procedures, these could be evaluated from left-to-right, or from right-to-left. The standard does not specify this one way or the other because the most convenient order may depend on the underlying hardware or operating system.

3. Behaviour that depends on the underlying machine and is potentially different on different machines is said to be implementation dependent.

Implementation dependent issues are, fortunately, few in number. Suffice it to say that any program whose meaning depends on knowing some detail of the underlying system or of its behaviour is erroneous, for that program will surely have a different meaning on another processor. For instance, the code

```
MyProc (a, function(a))
```

where *a* is a variable parameter depends for its meaning on the order of parameter evaluations and is therefore erroneous.

However, there are a number of considerations among low level issues, particularly of the implementation defined nature, that are of interest to programmers, and the following sections will detail some of these.

8.2.1 General Considerations

This text has, on a number of occasions, commented on what were called a "hierarchy" of library modules that is available in a typical Modula-2 implementation. The broadest and most general of these are referred to as *high-level*, and the most specific, hardest-to-use and machine or operating system bound as *low-level*.

Languages are themselves categorized in much the same way and for much the same reasons. As has been seen already, a compiler is necessary to translate the instructions the programmer provides to transform (using any high level notation) into a series of more basic *machine-level* commands upon which the central processor itself may act directly. This latter set of commands is referred to as the *machine language* for that particular computer, and constitutes the lowest possible level of interaction between a programmer and the machine.

Computer languages or individual language constructions are called low level if they require for their correct use a knowledge of machine language, if they name or manipulate the machine's memory locations directly, or if their correct use involves a knowledge or use of details of that machine's hardware construction or the operating system.

To the degree by which the use of a language is independent of and isolated from any of these specific details, it is said to be high level.

Thus, the terms *high level* and *low level* may be used in varying degrees to apply to the way data representation and manipulation abstractions, and to machine representation and manipulation abstractions. On the one hand, the more abstractly a data type may be treated, the higher the level of its representation and manipulation. As previously noted, this at times requires a hierarchy of modules, and the use of any one of them depends on the level of detail a programmer needs at the time. Likewise, there are varying degrees to which language notations, and particular elements within them may be referred to as *high* or *low level*, and it is usually not difficult to decide whether a feature belongs to one or the other category.

High level instructions tend to use English words or near English words for their commands, and low level languages and commands employ more cryptic abbreviations that have meaning only in the context of a particular machine, type of machine, or operating system.

In general, low level language instructions are very detailed, very specific, and accomplish only a single step with each instruction. It might take hundreds of them to output a single character to the screen, for instance. High level instructions hide all this detail with a command like *WriteCard* or *ReadString*. The very large number of machine language codes needed to carry out the high level instruction are not visible to the programmer.

It should be noted that even machine languages have standard notations within which to express their syntax, and that programs expressed in such *assembly languages* also have to be translated for the machine. Such translators are called *assemblers*. This is still at a rather low level, though it is one step removed from trying to program the machine directly in its own numeric language. Even in this style, the programmer must know what are the assembly language abbreviations for the various machine language codes, and must also know a great deal about the particular hardware and the operating system under which the finished product will run.

High level instructions must be more general so as to handle many possibilities and this necessary complexity tends therefore to make their final translations into machine code slower to execute than custom designed low level ones, though they are very much easier to program. Generally, the major portions of very large programs are written in high level compiled languages, with the speed-sensitive parts done directly in machine code with an assembler. (The student will not be needing to do this in the present course, but should be aware of this practice.)

When a language or an operating system is adapted to a particular machine, it is necessary to know something about the architecture of that machine in order to ensure that the high level language produces the correct results on it. This adaptation is easier in Modula-2 than in many languages, because only certain low-level modules that provide the interface to the hardware and operating system need to be rewritten. Since Modula-2 compilers are themselves usually written in Modula-2, any non-machine specific portions of a compiler need not be re-written when moving the software system to another

hardware platform. The same is true of programs. Only those portions of a program that make specific use of machine-dependent features need to be rewritten when adapting to another machine; the rest can remain untouched.

Generally, one does not need to rewrite the definition modules in doing this, only their implementations, for it is in these that knowledge of the particular system is expressed. In theory, client programs also do not need to be rewritten, though they will likely have to be re-compiled in order to produce the correct machine coding for their new environment.

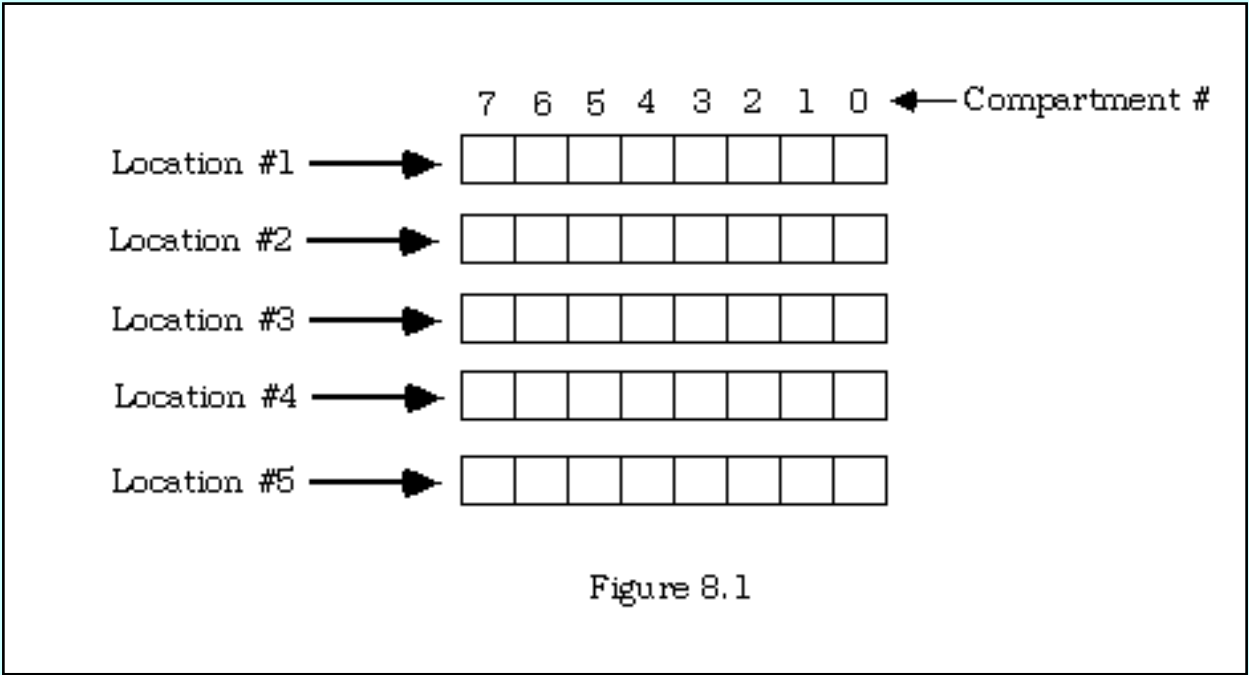
The Modula-2 language has the additional advantage over traditional languages that it can itself be used either as a high level one (as thus far in this text), or to manipulate entities directly on a low level. This makes it possible to write systems software (such as operating systems) as well as applications programs in a single language--a decided advantage to the programmer, who now may not have to learn a separate low level language.

8.2.2 Low Level Numeric Notations

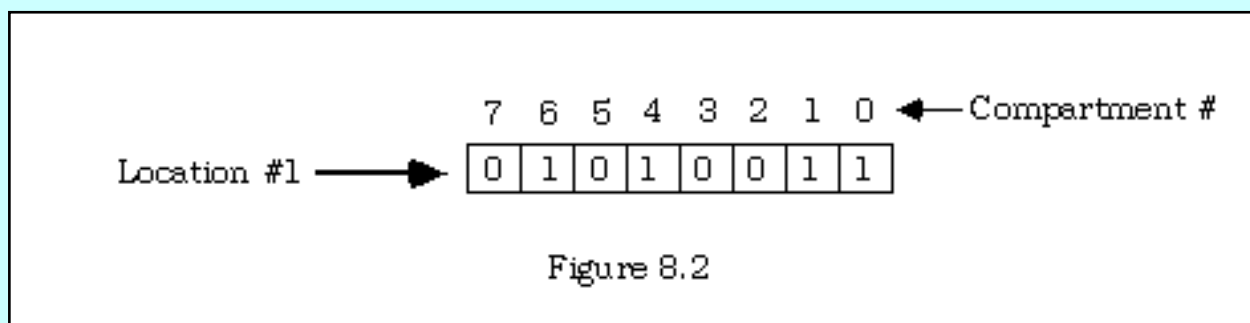
In order to understand some of these low level facilities, it is necessary to refine the model given earlier of the way in which a computer's memory is organized.

Every one of the thousands of individual locations or pigeon holes of which this memory consists of can actually be thought of as having several compartments, each of which is essentially just an on/off (actually low/high voltage) electrical switch. In practice, one can usually assume that *several* means exactly eight in this context.

The electrical state *low* or *off* is interpreted numerically as the value zero, and the electrical state *high* or *on* as the value one. Thus, each location is thought of as storing an eight digit *binary* (base two) number.



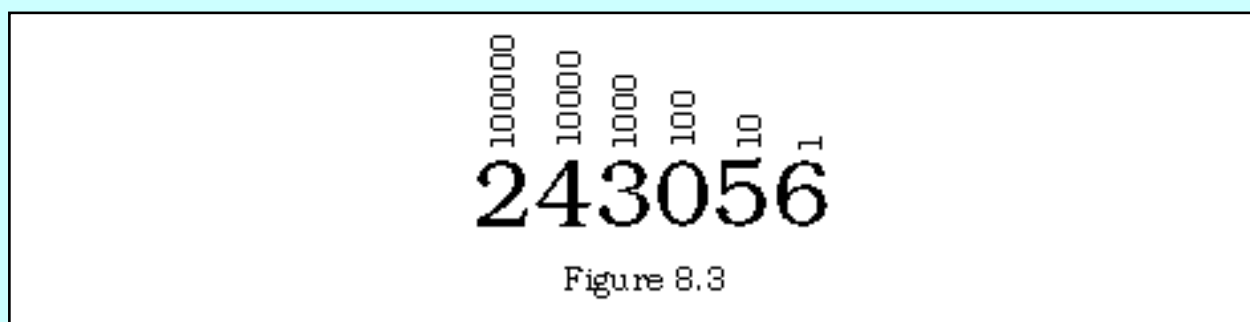
The binary numeral 01010011 is stored as in Figure 8.2



Counting in binary form provides the following translation table:

Decimal	Binary
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001
...	
15	00001111
16	00010000
...	
254	11111110
255	11111111

The binary (base two) system uses place value in the same manner as the more familiar decimal (base ten) system. Each place in the numeral has a value, and the place holder indicates the number of times that value is included in the total value represented by the numeral as a whole. For instance, a decimal numeral can be broken down into places as follows:



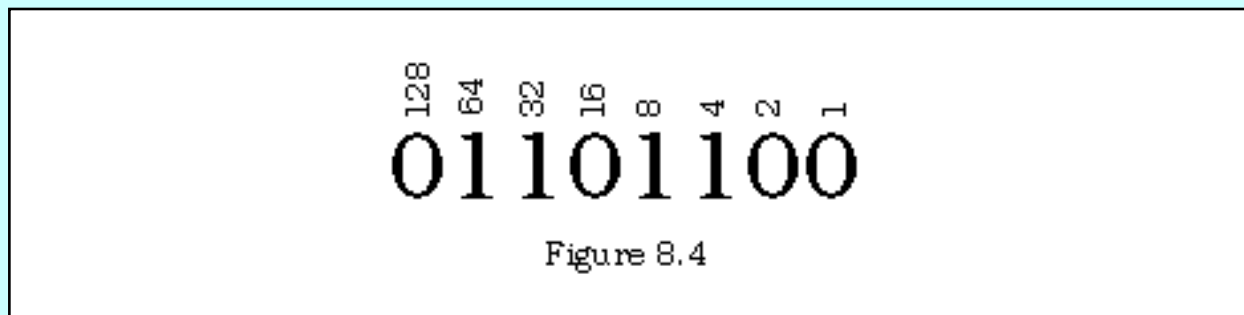
Anyone who is used to this system can abstractly, without having to think about the parts, translate this as:

$$(2 * 100000) + (4 * 10000) + (3 * 1000) + (0 * 100) + (5 * 10) + (6 * 1)$$

or, to use a more compact notation:

$$(2 * 10^5) + (4 * 10^4) + (3 * 10^3) + (0 * 10^2) + (5 * 10^1) + (6 * 10^0)$$

If, instead of ten, the base is two, the only changes that need to be made to this numeration system are to restrict the symbols to just *0* and *1*, and to re-label the columns using the new base. Begin at the rightmost column, labelling it the *ones* column as before. Proceeding from right to left by powers of two, one has, successively, the *twos*, *fours*, *eights*, *sixteens*, *thirty-twos*, *sixty-fours* and *hundred and twenty-eights* columns. This is illustrated in figure 8.4, and the similarities to figure 8.3 can readily be seen.



Translating the numeral in this figure produces the representation:

$$(0 * 128) + (1 * 64) + (1 * 32) + (0 * 16) + (1 * 8) + (1 * 4) + (0 * 2) + (0 * 1)$$

or, better:

$$(0 * 2^7) + (1 * 2^6) + (1 * 2^5) + (0 * 2^4) + (1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (0 * 2^0)$$

Observe that the exponents on the base are just the numbers of the component boxes used in figures 8.1 and 8.2. In the case of the computing machine (and any person who naturally thinks in base two), these numeric representations are sufficient and obvious abstractions for the value represented, just as is the earlier rendering of a number in a base ten numeral. If it is necessary to use base ten, the values in the non-zero columns can be individually translated into base ten and added to obtain

$$64 + 32 + 8 + 4 = 108 \text{ (base ten)}$$

It can be readily seen that the possible numbers that can be represented in a single eight component memory location range from 00000000_2 (0_{10} through 11111111_2 ($128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255_{10}$).

To change numbers from decimal to binary notation, uses successive division by two and then examine the remainders.

$$\begin{array}{r} 2 \overline{) 215} \\ 2 \overline{) 107} \text{ r1} \\ 2 \overline{) 53} \text{ r1} \\ 2 \overline{) 26} \text{ r0} \\ 2 \overline{) 13} \text{ r1} \\ 2 \overline{) 6} \text{ r0} \\ 2 \overline{) 3} \text{ r1} \\ 2 \overline{) 1} \text{ r1} \\ 0 \text{ r1} \end{array}$$

Taking the remainders as shown in reverse order, one gets the result $215_{10} = 11010111_2$. This algorithm was detailed in section 4.8, where a recursive procedure was written to accomplish the required conversion.

8.2.3 Machine Level Data Storage

To complete this general discussion, it is appropriate to provide the standard definitions of the abstractions *memory location* and *component* as they have been used here.

A single memory component that stores a one or a zero is called a binary digit or bit. A sequence of eight of these abstracted as a single memory location and that can store a number from zero through 255 is called a byte.

For the sake of completeness in this discussion of groupings of bits, the following is also useful:

Half a byte (four bits) is called a nibble. The bits of a byte numbered zero through three starting from the right are the low nibble, and the bits numbered four through seven are the high nibble.

For various technical reasons, many machines and/or operating systems employ a basic unit of data consisting of two or more (but an even number) bytes taken together. Such machines can only store and retrieve data in these larger units, not in individual bytes. That is, although the common unit of a byte has the normal meaning for such a machine, it is incapable of using data chunks that small. Moreover, a data type that can only store 256 distinct items is not as useful as one with a larger range, so multi-byte storage may be employed at the language level, even when the underlying hardware does not require this.

A grouping of two or more adjacent bytes (memory locations) collectively employed as a single data storage unit is called a word, or, if thought of as storage, as a data location.

In general, words come in even numbers of bytes, often either two, four, or eight. If two bytes are placed side-by-side, and the column naming given above is continued from the rightmost (column 0) to the (new) leftmost (column 15), the place values range from 2^0 through 2^{15} . If there is a value of one in all sixteen places, and the result is converted to base ten, one finds that a two-byte word can store numbers in the range from 0 through 65535. This has been a common range for cardinal types in high level languages. Evidently, implementors of such notations found two byte words to be convenient for data storage, whatever the underlying hardware. When the same calculation is applied to the thirty-two bits of a four byte word, the range produced is 0 through 4 294 967 296--a much more satisfactory range for many types of calculations, and now more commonly used even on microcomputers. The individual storage locations in the memory of a computer are each given a consecutive number, by which means its contents can be accessed.

The unique number identifying of a particular storage location is called its address.

A CPU accesses data by forming the bits of an address on a group of electrical lines (called the *address bus*). Data is either fetched from this memory location or stored into it depending on whether the CPU is in a *read* or a *write* state. It should be evident that the maximum amount of memory that a given type of machine can use depends on the number of different addresses its CPU can form.

While on the subjects of memory and the natural use of binary numbers to describe both what it does and

the amount of it, note that 1024 (210) bytes of memory is called 1K. Thus, the 65 536 bytes that are addressable by an older type of typical eight bit processor (which usually had 16 electrical lines on the address bus) were referred to as 64K, and a megabyte (1M) of memory or disk space is actually 1 048 576 bytes. Two smaller units of memory/storage also have names that are worth remembering, as they may come up in describing programs:

A page of memory is 256 bytes. This is also a common unit of disk storage, and there it is often called a sector. Two of these sectors taken as a unit (512 bytes) may be referred to as a block.

NOTE: If the unit of disk storage is 512 bytes, 1024 bytes, or some larger number, then this unit may be called a *sector* and the name *block* may not be used.

Disks themselves are organized into a series of concentric rings or *tracks* on which the information is magnetically recorded. The number of bytes per sector, the number of sectors per track, the number of tracks on one side of a disk, and whether these disks are single or double sided, will together determine the total capacity of one surface or *platter*. In the case of a hard disk, there are often several platters stacked vertically, each with its own read/write head attached to an arm that moves them all together. All the tracks at a single location of the moving arm may be thought of as a *cylinder*. In that case, one usually calculates capacity by the size and number of cylinders. Unfortunately, all these parameters vary from one manufacturer to another, as do the precise formats used for the recording. In the early days of the industry, it was unlikely that a disk recorded by one type of machine will be readable by another type, or even by the same machine using a different operating system.

Such days are fortunately past, for the most part. Most operating systems are now robust enough to be able to recognize a variety of recording formats, and to read from the disk itself the critical information that allows it to determine the capacity of the medium.

8.2.4 Hexadecimal and Octal Notation

The *nibble* mentioned in the last section provides a convenient size for abstracting binary data in larger "pieces." Nibbles allow one to count from zero to fifteen, and thus can be employed as the basis for implementing a base sixteen (Hexadecimal) numeric storage abstraction system.

In order to have sixteen symbols for each column of such a notation, the letters A, B, C, D, E, and F as equivalents for the decimal numerals ten through fifteen. Thus one has:

Decimal	Binary	Hexadecimal	
1	0000 0001	01	
2	0000 0010	02	
3	0000 0011	03	
10	0000 1010	0A	
15	0000 1111	0F	
80	0101 0000	50	(5 * 16)
137	1000 1001	89	(8 * 16) + 9

254	1111 1110	FE	$(15 * 16) + 14$
255	1111 1111	FF	

Notice how each nibble of the binary representation expresses a single hexadecimal digit. Thus the hexadecimal numeral

FACE

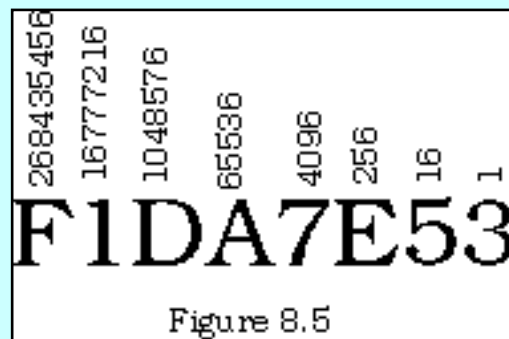
is equivalent to the binary numeral

1111 1010 1100 1110

or the decimal numeral

$$(15 * 16^3 + 10 * 16^2 + 12 * 16 + 14) = 64206$$

The easiest way to change from Binary (base two) to hexadecimal (base sixteen) is to split the binary number into nibbles, and represent each number a nibble at a time (e.g.. 1011 1100 = BC). With hexadecimal numbers of course, the columns are numbered from right to left in powers of sixteen. (1, 16, 256, 4096, ...) as illustrated in Figure 8.5.



In order to make it clear that one is using hexadecimal notation, one must use some kind of marker. The most common are a post-subscript of *16*, or a prefix of the symbol \$ or a suffix of the letter *H*. Thus, the following are equivalent:

DEAD₁₆ \$DEAD DEADH

Eight was at one time another commonly used base. These "Octal" numbers are not so often used now, but if one numbers the columns from right to left as "ones", "eights", "sixty-fours", and so on, in powers of eight one would have, for instance $45_8 = (4 * 8) + 5 = 37_{10}$.

[Contents](#)

8.3 High Level Access to Low Level Facilities in Modula-2

In common with the implementations of most other languages, all versions of Modula-2 use words as units of data and one memory location of this size is employed to store a CARDINAL. One of them is usually used to store an INTEGER as well, just by interpreting any binary number with a "1" in the highest bit place as a negative and with a "0" in that location as a positive. Thus the sixteen bit binary number

1111 1111 1111 1111

interpreted as a CARDINAL is typically 65535, but as an INTEGER is typically a representation of -1 (subtracting one from zero changes all the bits to ones).

These details are not usually important to the programmer working at the high level, but when they are, it becomes necessary to access such low level features from the high level platform Modula-2 affords. For such instances:

A Modula-2 data storage unit has the type LOC, and a Modula-2 storage location has the type ADDRESS.

Because the implementation of these two data types must always imply some knowledge of the machine on which they are implemented, neither is included in the language proper. Rather, along with certain other low-level constructs, these items are segregated from the language and placed into the module SYSTEM, from whence they must be imported into any other module. This means that the occurrence of the line

```
FROM SYSTEM IMPORT  
  LOC, ADDRESS, <anything else>;
```

immediately marks the module containing this import as low-level dependent, and therefore probably not portable to another environment.

Strictly speaking, SYSTEM is not a separate library module at all, but a segregated part of the compiler. By convention, its name and contents are all uppercase, but, SYSTEM is not a standard identifier, and neither are LOC or ADDRESS.

The Module SYSTEM, and any other low-level dependent module that is a segregated part of the compiler and not a separate library module is called a system module or a pseudo-module.

8.3.1 The Module SYSTEM

While not actually a separate library module, SYSTEM behaves as though it had the following definition module shown below.

NOTE: Some of the contents of SYSTEM have been omitted and will be discussed later in the text. The meaning of the standard identifier POINTER can be found in chapter 12, and its use will not be detailed here. Some versions of SYSTEM may have additional items necessary for the implementation at hand.

```
DEFINITION MODULE SYSTEM;
```

```
CONST
```

```
  BITSPERLOC      = <implementation-defined constant> ;  
  LOCSPERWORD     = <implementation-defined constant> ;
```

```
TYPE
```

```
  LOC;  
  ADDRESS = POINTER TO LOC;
```

```
WORD = ARRAY [0 .. LOCSPERWORD-1] OF LOC;
```

```
(* BYTE and LOCSPERBYTE are provided if appropriate for machine *)
```

CONST

```
LOCSPERBYTE = <implementation-defined constant> ;
```

TYPE

```
BYTE = ARRAY [0 .. LOCSPERBYTE-1] OF LOC;
```

```
PROCEDURE ADDADR (addr: ADDRESS; offset: CARDINAL): ADDRESS;
```

```
(* Returns address given by (addr + offset), or may raise an exception if this address is not valid. *)
```

```
PROCEDURE SUBADR (addr: ADDRESS; offset: CARDINAL);;
```

```
(* Returns address given by (addr - offset), or may raise an exception if this address is not valid. *)
```

```
PROCEDURE DIFADR (addr1, addr2: ADDRESS): INTEGER;
```

```
(* Returns the difference between addresses (addr1 - addr2), or may raise an exception if the arguments are invalid or address space is non-contiguous. *)
```

```
PROCEDURE MAKEADR (val: <some type>; ... ): ADDRESS;
```

```
(* Returns an address constructed from a list of values whose types are implementation-defined, or may raise an exception if this address is not valid. *)
```

```
PROCEDURE ADR (VAR v: <anytype>): ADDRESS;
```

```
(* Returns the address of variable v. *)
```

```
PROCEDURE CAST (<targettype>; val: <anytype>): <targettype>;
```

```
(* CAST is a type transfer function. Given the expression denoted by val, it returns a value of the type <targettype>. An invalid value for the target value or a physical address alignment problem may raise an exception. *)
```

```
END SYSTEM.
```

The procedures ADDADR, SUBADR, and DIFADR, are provided to allow arithmetic to be performed on items of the abstract data type ADDRESS. The details of MAKEADR vary from one machine to another. This procedure is intended to allow for the construction of a valid address from some other type of data. The parameters will in most implementations be one or more CARDINALs.

ADR is intended to allow a program to discover the address of one of its own variables. When a module is loaded into memory, the variable declarations are resolved into addresses, and all references to them within the actual code from that point on are to these addresses. ADR returns, in an item of the type ADDRESS the machine location of one of these variables. One might write a fragment such as:

```
FROM SYSTEM IMPORT
```

```
WORD, ADDRESS, ADR;
```

VAR

```
theWord : WORD;
```

```
theAdr : ADDRESS;
```

```
card : CARDINAL;
```

```
BEGIN
```

```
theAdr := ADR (card);
```

However, the useful purposes to which this can be put are not described in detail until Chapter 12.

CAST plays a role related to, but different from the standard procedure VAL. When it is necessary to safely convert a value of one type to a value of another type (so as to construct an expression) VAL is always preferable. When, on the other hand, it is desired to forcibly re-interpret the bit pattern of an item of one type as though it were of another type without any conversion, CAST is used instead. Clearly, it is necessary for the programmer to know how the two data types are represented at the low level (bit pattern) or the result of the CAST operation is not usable. Thus, if one has

VAR

```
int : INTEGER;  
card : CARDINAL;
```

then, both

```
int := int + VAL (INTEGER, card);
```

and

```
int := int + CAST (INTEGER, card);
```

are syntactically valid, but the latter assumes that the storage bit pattern of the INTEGER has some particular meaning when interpreted as a CARDINAL. This may or may not be true. If, for instance *int* were negative, CASTing it into a positive valued be meaningful only if the programmer also knew how many bits were used to store the INTEGER and how to interpret the sign bit after the CAST.

Type changes made using VAL are called safe conversions.

Type changes made using CAST are called unsafe conversions.

These low level facilities are detailed at this point not because the student is expected to make great use of them yet, but to illustrate that Modula-2 can be used at the low level. This makes it a language in which operating systems and other software that uses intimate knowledge of the machine can be written.

8.3.2 Variables at Fixed Addresses

Modula-2 provides the ability to declare a variable to reside at a fixed address (or, more accurately, to assign a variable name to the contents of the machine at a particular address). This is done by giving a constant in brackets after the variable at the time it is declared.

VAR

```
flag [768] : INTEGER;  
bottom [0] : CARDINAL;  
somewhere [16238] : CHAR
```

The variable can be of any type and will start at the specified address. Its space extends for the number of storage locations normally taken by an integer, cardinal, char, etc. This particular facility was not required by Wirth's definition of Modula-2, but his suggestion that it be provided was quite strong and most implementations had it. It is required to be provided in ISO standard Modula-2.

NOTES: 1. This is a machine-dependent facility, and code written to take advantage of it is not portable to another system.

2. For this reason, this syntax cannot be used in ISO standard Modula-2 unless the identifier MAKEADR has been imported from SYSTEM even if MAKEADR is not explicitly used.

3. In some operating systems, user programs are not allowed to have low level access to addresses and some of these capabilities will not be present in any notation available to the programmer.

One use of this facility for system programmers is to access single memory locations (LOCs). Some such memory locations may serve the role of hardware switches on I/O locations (among many other uses). Referencing one memory location (LOC) might control turning on, say, the high resolution graphics mode for the machine, and the one just before it might turn it off again. Usually, any reference at all to such a "switch" will cause it to act. An assignment to the variable declared to be there will do quite nicely.

Clearly, one must consult the manual of the computer before accessing the contents of specific addresses, as very undesirable side effects can easily (erasing a disk?) be caused if one acts without due care.

8.3.3 Hexadecimal and Octal Notation in Modula-2

If one finds it useful, Modula-2 allows one to declare constants in Hexadecimal, provided they begin with a number and are followed by the letter *H*. It will also allow one to declare them in Octal, by following them with the letter *B*. As indicated briefly in Chapter 7, single character constants compatible with CHAR or the implied string type can be given by the ordinal value in Octal, followed by a *C*. Here are some examples:

CONST

```
a = 0A5H;      (* can't start with letter *)
b = 651B;
c = 177777B;
EOL = 15C;     (* common end of line character *)
d = 789H ;     (* starts with number *)
```

VAR

```
somewhere [0FFFFH] : CARDINAL;
```

PROCEDURE Length (str : **ARRAY OF CHAR**) : **CARDINAL**;

VAR

```
count : CARDINAL;
```

BEGIN

```
count := 0;
WHILE (count <= HIGH (str)) AND (str [count] # 0C)
  DO
    INC (count)
  END;
RETURN count;
END Length;
```

Note the handy use of *0C* instead of *CHR(0)* in this version of the procedure *Length* for a system where it was known that the string terminator was the null character. If it were not known, the more portable character literal "" is preferable. This only matters because the one form takes six keystrokes to type, and the other only takes two.

NOTES: 1. Only the numbers 0 through 127₁₀ define standard characters according to the ISO sequence that underlies Modula-

2. Since a LOC (usually byte) is used to store such a character, codes through CHR (255) or 377C (at least) are valid.

However, what such a "character" would look like if output to the screen or printer is very much hardware dependent. On some machines it could be a regular ISO character in black/white inverse, and on others it could be a special graphics symbol, a Greek letter, accented character, or something else. There is no Modula-2 standard for characters in the range 127 .. ORD (MAX (CHAR)).

2. Many machines are now using two byte character coding called *Unicode*, in order to code languages such as Chinese that

have many thousands of characters. In such machines, when the actual language employed is based on Roman script and only 128 characters are needed, the most significant byte of the character is usually set to zero and ignored.

Hexadecimal numbers are convenient for expressing addresses as well as data. Small computers once had sixteen bits with which to express an address, these could range from zero through hexadecimal 0FFFFH (65535₁₀).

NOTE: The number of bits available to the computer to make addresses is independent of the number of data bits. While a computer with eight data bits generally had 16 address lines, a sixteen bit computer may have had sixteen, twenty, or twenty-four address lines. The maximum amount of directly addressable memory in these three cases is therefore 64K, 1M and 16M bytes respectively. A thirty-two data bit computer might have many more address lines.

And now, a little example. Here is a procedure that takes advantage of the fact that on its target machine, there is a fixed location in memory for keyboard data coming in. The keyboard location in this machine is 0C000H and the value at that location is less than 127 if no key has been pressed since the last time any reference was made to the location 0C010H.

Whenever the latter location is accessed, the most significant bit in 0C000H is set back to zero, leaving a number below 128. This procedure waits for the user to press a key before going on, hits the keyboard strobe (0C010H) to reset the key location for the next routine checking this, and then exits with the character value of the key pressed in the type CHAR.

```
PROCEDURE Keypress ( ):CHAR;
```

```
VAR
```

```
    Keyboard [0C000H] : CHAR ; (* single loc in this version *)
```

```
    KbdStrobe [0C010H] :CHAR;
```

```
BEGIN
```

```
    REPEAT
```

```
    UNTIL Keyboard > CHR (127); (* A keypress--high bit is set *)
```

```
    KbdStrobe := CHR (0);  (* reset *)
```

```
    RETURN Keyboard;
```

```
    (* High bit only is stripped to zero by strobe reset, so in correct ISO range *)
```

```
END Keypress;
```

As can be seen, writing a *Keypress* function procedure on this level always involves an intimate knowledge of the specific workings of the target hardware device. However, with such knowledge, suitable low-level procedures can be coded directly in Modula-2, without resorting to separate machine language routines. It should, in fact, be possible to modify this particular example for specific hard-wired memory locations on many computers, provided their function is well-documented. Clearly, such code cannot be ported to another system than the original target.

[Contents](#)

8.4 Extended Low Level Examples

8.4.1 Keyboard Reading--Operating System Level

At a slightly higher level than that employed in the last section, it may be possible to take advantage of one's knowledge of low level procedures that have been provided in the operating system interface to read the keyboard. This technique is illustrated by the following library module:

DEFINITION MODULE Keyboard;

```
(* =====
  Definition and Implementation © 1993 by R. Sutcliffe
    Trinity Western University
  7600 Glover Rd., Langley, BC Canada V3A 6H4
    e-mail: rsutc@twu.ca
    Last modification date 1997 10 06
===== *)
(* Note that this module functions independently of the Terminal, *)
```

```
PROCEDURE BusyRead (VAR ch: CHAR);
  (* return character if one there, otherwise return 0C *)
```

```
PROCEDURE Read (VAR ch: CHAR);
  (* return character *)
```

END Keyboard.

IMPLEMENTATION MODULE Keyboard;

```
(* =====
  Definition and Implementation © 1993 by R. Sutcliffe
    Last modification date 1997 10 06
===== *)

(* first implementation 1993 10 20 revised for pl 1994 05 18
  modifierSet becomes modifiers characterCode becomes charCode
  Desk changes to Events 1995 09 14
  1996 08 08 changed to WaitNextEvent as problems with translating GetNextEvent to C *)
```

```
FROM Events IMPORT
  EventRecord, GetCaretTime, WaitNextEvent, keyDown, cmdKey, everyEvent;
FROM SYSTEM IMPORT
  CAST;
```

```
VAR
  theEvent : EventRecord;
  gSleep   : INTEGER;
```

```
PROCEDURE BusyRead (VAR ch: CHAR);
  (* return character if one is there, otherwise return 0C *)
```

```

BEGIN
  IF WaitNextEvent (everyEvent, theEvent, gSleep, NIL)
    AND (theEvent.what = keyDown) AND NOT (cmdKey IN theEvent.modifiers)
  THEN
    ch := theEvent.charCode;
  ELSE;
    ch := 0C
  END; (* if WaitNextEvent *)
END BusyRead;

PROCEDURE ReadChar (VAR ch: CHAR);

BEGIN
  REPEAT
    BusyRead (ch);
  UNTIL ch # 0C
END ReadChar;

BEGIN (* main *)
  gSleep := GetCaretTime (); (* set idle time used by WaitNextEvent *)
END Keyboard.

```

A program could now implement a procedure to wait for a keypress before continuing processing by importing *BusyRead* and then proceeding much as does *Read* in the above, except not returning anything to the caller.

```

PROCEDURE Keypress;
BEGIN
  REPEAT
    Keyboard.BusyRead (ch);
  UNTIL ch # 0C
END Keypress;

```

While the code shown here to achieve this is necessarily implementation specific to the Macintosh, something similar should be possible under any operating system.

8.4.2 Generic Swap

It is often necessary to swap the values of two variables, and on a number of occasions such procedures have been used in this text. Each time one writes something like:

```

PROCEDURE Swap (VAR x, y : REAL);

```

and each time it is necessary to swap two items of a type not previously used, the procedure must be rewritten, if ever so slightly. Using the low level facilities of Modula-2 , it is possible to write a swap procedure that can swap two items of any type presented to it.

A procedure that is capable of acting on any data regardless of type is said to be generic.

Generic procedures are written using the ARRAY OF LOC as parameter. Since this has an equivalent high level meaning on every machine, the result code ought to be portable, even if the low level nature of LOC is different. Of course, the items ought both to be of the same type or the results will be meaningless. Indeed, if the items are not at least of the same size, attempting

to swap on a memory location by memory location basis could prove disastrous unless the swap stops at the end of the shorter item, and even then the results of such a swap are not likely to be meaningful. The library module below encapsulates a generic swap. It also employs `HIGH` to compute the number of LOCs occupied by the variable passed to it as a parameter.

```
DEFINITION MODULE Swaps;
```

```
FROM SYSTEM IMPORT
```

```
    LOC;
```

```
PROCEDURE CanSwap (a, b : ARRAY OF LOC) : BOOLEAN;
```

```
(* Pre : None
```

```
Post: if a and b are of the same size, returns true, else returns false *)
```

```
PROCEDURE Swap (VAR a, b : ARRAY OF LOC);
```

```
(* Pre : None, but does no size checking, so if they are not the same size, the  
result may be meaningless
```

```
Post: Swaps the number of bytes of the smaller of the two arrays. *)
```

```
END Swaps.
```

The Procedure *CanSwap* is only able to check on the size of the items, not whether they really are the same type. For its part, the procedure *Swap* does no checking; it just swaps as many bytes as it can. Here is the code:

```
IMPLEMENTATION MODULE Swaps;
```

```
FROM SYSTEM IMPORT
```

```
    LOC;
```

```
PROCEDURE CanSwap (a, b : ARRAY OF LOC) : BOOLEAN;
```

```
(* Pre : None
```

```
Post: if a and b are of the same size, returns true, else returns false *)
```

```
BEGIN
```

```
    RETURN (HIGH (a) = HIGH (b));
```

```
END CanSwap;
```

```
PROCEDURE Swap (VAR a, b : ARRAY OF LOC);
```

```
(* Pre : None, but does no size checking, so if they are not the same size, the  
result may be meaningless
```

```
Post: Swaps the number of bytes of the smaller of the two arrays. *)
```

```
VAR
```

```
    temp : LOC;
```

```
    max, count : CARDINAL;
```

```
BEGIN
```

```
    max := HIGH (a);
```

```
    IF max > HIGH (b)
```

```
        THEN (* use lesser of two for # to swap *)
```

```
            max := HIGH (b);
```

```
        END;
```

```
    FOR count := 0 TO max
```

```
        DO
```

```
            temp := a [count];
```

```
            a [count] := b [count];
```

```
        b [count] := temp;  
    END;  
END Swap;  
  
END Swaps.
```

[Contents](#)

Part B--Input, Output, and Files

8.5 Files--Introduction and Terminology

Programs depend on data input in order to properly express the solution to a problem. In the examples seen thus far in this text, programs have taken their data from:

- within themselves, perhaps as constants (batch style,) or
- dialogue with the user (interactive style)

These two styles, used alone, have in common that both the data employed and the results produces do not survive the run of the program. When the program is next run, the data must be obtained again (even if it is the same data, or only slightly changed), and the results cannot be fed forward to become the input for some other program. In order to work around these obstacles, a means of storing data outside programs is required. Files serve not only this purpose, but also provide a way of storing very large data collections, for which individual entry to every program would be impractical. Indeed, it is often the case in such instances that the data is the central theme to an entire symphony of programs operating on it, and that no one of the programs in the collection is nearly as important as the data itself.

A file resembles a book. Its structure (plot) is created in the mind of an author and it must be written (encoded) on some medium. Once this has been done, others can read it. However, in order to read it intelligibly, they must follow the structure created by the author.

A file is a source or a sink for a collection of data.

Just as data must be structured or arranged in such a way as to represent some real life problem, so also files must be structured (the plot, again) so as to represent the data they are intended to store. There are as many ways to do this as there are programmers, computers, operating systems, programs, and problems. The definition of a *file* has been expressed in a broad and general form for this very reason--the meaning must cover a lot of ground. In fact, by this definition, the batched data within a program and the data input interactively at a keyboard by the user of a program are both files--at least conceptually.

At the highest and most abstract level, a book can be thought of in terms of its plot, character and events. At a middle level, the book is perhaps structured by named chapters. On a more detailed level, the book is a collection of words on a page. That is, there are degrees of abstraction to the concept *book* as there are for many other commonly used ideas. This observation leads to the first classification of files, by the degree of their abstraction:

A logical file is an abstract structuring of data storage as viewed by the programmer and/or user of the program. This is the high level, or conceptual view of a file.

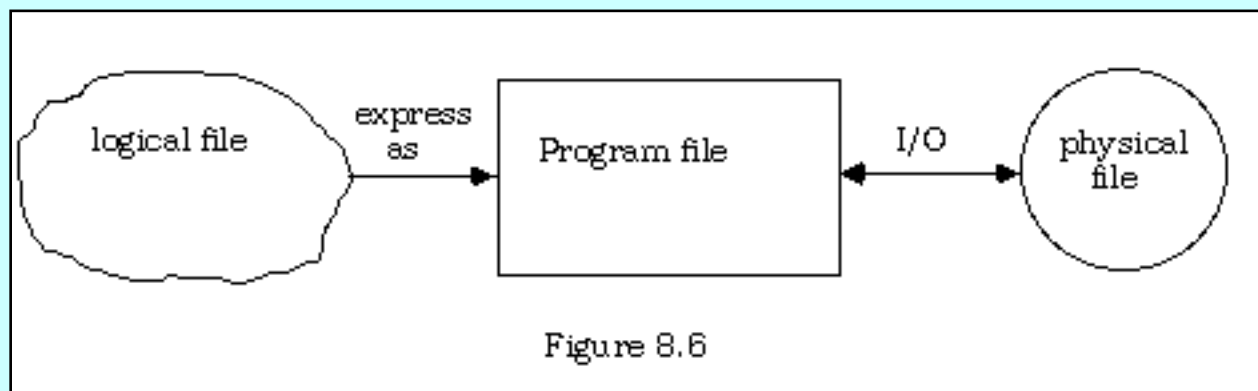
A program file is a specific data collection as seen and manipulated by a program. It is often (but not always) represented by a variable, perhaps of type "file." At this middle level of abstraction a file can be regarded as residing within the machine's memory, as existing only as long as the program that employs it is active.

A physical file is a recording of a logical file. This usually takes the form of a magnetic image on a disk or tape surface, in which form it exists independently of any particular program. The details of this recording provide the lowest level view of a file.

The distinction between the latter two levels of abstraction is not always maintained, and the term *physical file* is used by some for both the memory image and the external storage.

In order to write a useful program that employs files, some attention needs to be paid to all three levels of abstraction. The contents of the file must be decided on abstractly and conceptually; a program must be written to render the abstractions into a solution; and the resulting data must actually be recorded externally to the program so that it survives when the program terminates. Alternatively, an existing file may have to be read by a program, and this operation in turn can only be expressed if the original logical structure of the physical file is known.

The relationships among the three views of a file are expressed in figure 8.6 below. Observe that if the broadest possible view of a file is used, *all* input to and output from a program uses files.



Fortunately for programmers, the troublesome details of how files are to be physically stored can be left to the operating system, whose function it is to make such recordings, name them, keep track of them, ensure their integrity, and deliver them back to a program upon demand. This observation might seem to indicate that a simple and universal file handling module could serve as the interface between programs and operating systems.

Unfortunately there are many different operating systems, and widely differing views of what constitutes a simple program interface with any one of them. Thus, there have been many attempts to provide universal file handling interfaces, and these differ widely. Indeed, perhaps the most troublesome area for both the designers of a computing notation and for those who program using it is how to deal with input and output, as they are on the one hand essential to any substantial programming activity, and on the other closely tied to a particular system.

The problem for a language designer is the necessity to find common ground ahead of time for all possible external devices and operating systems so that I/O routines can be universally applicable. In Modula-2, this problem has been partially avoided by placing such matters outside the purvey of the language proper, and by assuming instead that any device needing communications links with a program will have these facilities supplied in a particular implementation by appropriate library modules. This results in the Modula-2 notation itself being small and versatile, but causes somewhat more work for the programmer, who often had a large number of library routines to keep straight--especially if using more than one version--for then there was no guarantee that such libraries would correspond. In spite of this deliberate lack of pre-specification by Wirth (he required no particular I/O routines or modules, and only made a few suggestions of modules he had found generally useful), much can still be said about such functions. Although operating systems differ widely, there are many things that they do have in common. Moreover, there are not many kinds of logical file in common use, even though the recordings of such files may be very different. As a result, many vendors of Modula-2 products produced very similar libraries for I/O and to some extent, this tended to create a *de facto* or marketplace standard. One section of this chapter is devoted to outlining the most common I/O and file handling routines used by commercial vendors in the years when no official standards existed. Even the ubiquitous classical high level modules *InOut* and *RealInOut* have many variations however, and lower level modules often have more differences than similarities between implementations. This was one of the major reasons for convening a working group (WG13) of the International Standards Organization (ISO) in April of 1987 to produce a standard definition of both Modula-2 and its libraries. This standard will be the focus for most of the rest of this chapter and will be used subsequently when a sample solution happens to call for the use of files. Before looking at the specifics of handling the program/logical file communication, however, some additional attention needs to be paid to the logical view of data storage.

8.5.1 Sequences, Streams, and Channels

Sequences have been discussed before in sections 3.8, (arithmetic sequences) and section 4.9 (geometric sequences.) The concept of a *statement sequence* is also important in Modula-2. In general, a sequence is any collection of entities that can be numbered in some well-defined way using the cardinal numbers 1, 2, 3, 4, 5,

More Mathematically,

A sequence is a function from the positive whole numbers into some other collection of objects.

Here are some examples of sequences with commas between the elements for clarity:

I, n, ,t, h, e, , b, e, g, i, n, n, i, n, g, , G, o, d, ...

1, 3, 5, 7, 9, ...

1, 4, 9, 16, 25, 36, ...

1, 1, 2, 3, 5, 8, 13, ...

1, 1.1, 2, 2.2, 3, 3.3, 4, 4.4, ...

From the point of view of computer input and output, the definition above is specialized as follows:

A stream is a sequence of data items of the same type.

All but the last of the sequences above could be streams if the items in them are considered as data being manipulated by a program.

In general, streams all have some properties in common. These are:

1. The number of elements in the stream is not known ahead of time (i.e. by the writer of the stream handling module before the data type is ever used in a program). In an actual instance of a particular stream, there is (eventually) a finite length associated with it that may be possible to compute.
2. A stream has an origin and a destination, which are also not necessarily known ahead of time (again, when the stream handling module is designed), but which may default to some standard place or device. It is usually possible to change these *default* connections to something else.
3. It is possible to write only at the end of a stream (i.e. writing is always appending), and the only other way to modify a stream is to delete it entirely.
4. Any element of a stream can be read, provided that reading starts at the beginning of the stream and proceeds through the elements one at a time in order until the desired one is reached (sequential reading).
5. For some specialized streams, it may be possible to maintain a *window* or a *pointer* to the last element that was read so that the next one can be read at a later time without starting from the beginning again. If this is done, the position pointer is usually advanced one place by the very act of performing the read operation.
6. A stream has a *mode*. It is either being read from or written to at any given time. Some streams may have only one possible mode, and others may allow procedures to operate on their status to alter the mode.

As can readily be seen, the high level modules *STextIO*, *SWholeIO*, *SRealIO*, and *SLongIO* operate on streams. They employ a variety of *Read* and *Write* operations for the sake of convenience, but the actual streams being manipulated in this case consist of text characters, regardless of the formal type of data being read or written.

That is, procedures like *WriteString* and *WriteCard* both write characters to the output device, and both *ReadChar* and *ReadInt* read characters. It is the further duty of *SWholeIO* and *SRealIO* to then arrange for another module to convert the character strings to and from the correct type of program data (perhaps an integer), but that is a separate consideration.

A stream whose items are of the character type is called a text stream (or a legible stream.)

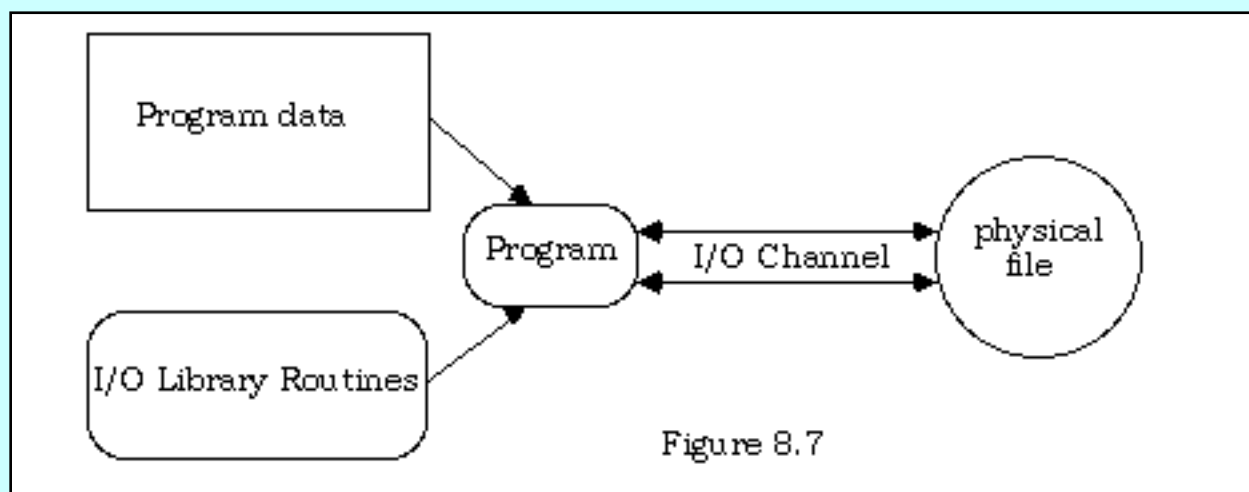
On the other hand, an input or output stream could be thought of on a lower level of abstraction as a sequence of binary items, rather than as a sequence of text characters. Indeed, not all streams are best abstracted as text; a binary abstraction might be the only practical way to think of a some streams.

A stream whose elements are thought of as binary digits is called a binary stream (or a raw stream.)

Implicitly, there are exactly two default streams used by *STextIO*, *SWholeIO*, *SRealIO*, and *SLongIO*--one for input, and one for output. However, it is also possible to have a program to maintain several (other) streams too, with various origins, destinations, modes, lengths, and position pointers. The analogy of observing flowing water (a real stream) is quite apt. It is helpful to think of data streams as being contained by their banks and as consisting of a flow passing by some program aperture, and into which new data can be placed, or from which previously placed items can be plucked--all under program control via routines imported from stream management modules. The ISO committee found it useful to assume that file management was a low level function that could be different in every implementation, and to standardize instead the containment (stream bed) abstraction at a somewhat higher level than that of file management. This abstraction could also be thought of as the connection mechanism joining logical files to physical files.

A channel is an abstract medium through which a stream flows.

The role of a stream managing module is analogous to that of a plumber, whose task it is to connect and maintain the pipes or channels to the various sources and sinks in the system. Using such a facility, a program may be connected to one or more of a variety of input streams and one or more of a variety of output streams at a given time. Figure 8.7 illustrates:



8.5.2 Sequential and Random Access Files

Having examined flows of data, consider now where the data flows lead to and from. On one end of a data stream is the program. It is tempting to say that the other end is some physical device such as a disk drive, or a printer, but it is more accurate to view the destination as a physical file, and not as a location. The location of the physical file could be a recipe box, a disk drive, or the main memory of the computing device. The program deals with the file only on the logical level, and in practice, may do so just by writing to the stream in an appropriately connected channel. The logical file and the stream are not identical, though the two do share in common that they only exist as the creatures of some running program, and are therefore both more abstract than the physical file. In fact, a file *may* be organized by a program as the recording of a stream, or it may not be--depending on the way that information is

accessed in the file. To distinguish, a new classification scheme for files is necessary:

A file that is organized as a stream, with writing allowed only at the end, is called a sequential file. If the underlying type of data in the stream is character, it is further known as a text file.

On the other hand, a file might be logically organized more like an array. The elements of an array are numbered consecutively, and *may* be read consecutively. However, and one element can be accessed by its own number in the sequence at any time (e.g. *element [1243]*) without reading all the ones that came before it first.

A file that may have any of the data elements read or written directly by some indexing scheme without having to start at the beginning is called a random-access file.

If the elements making up a random access file are, say, integers, then they must *not* be stored as text, for in that form different integers take a different amount of storage. For instance, the integer 123 would be expressed as the character stream "1", "2", "3" and take three storage positions in a text file, whereas the integer -46553 would require six storage positions for its characters. In the memory of a computer, every integer takes the same amount of room, as a given integer storage location must be able to hold any valid integer value. This is why the read/write memory is called RAM or *random access memory*. The same storage considerations hold for a random access file. Since there can only be a certain amount of room in the physical file for a given element, it follows that each numbered element of a random access file must be the same size as every other element.

Thus, a random-access file-managing module may be similar to that for sequential files, but it must also have a means of calculating the correct position in the file at which to read or write a given data item. What one would expect to find in a well-developed library suite is modules that impose one of these models on I/O, perhaps overlaid on top of some lower level ones that are in turn manage the connections between programs and the physical files through the operating system. The specifics of handling files using the random access model will be covered in chapter 9; sequential file models are of more immediate interest.

8.5.3 Planning to Use Streams with Files

Consider first the stream model for a file. Until now, this text has used only the high level modules *STextIO*, *SWholeIO*, *SRealIO*, and *SLongIO* or *InOut* and *RealInOut* to connect to input and output streams. These connections could be diverted from their default source and sink via the non-standard procedures *OpenInput* and *OpenOutput* (where available). While there is a certain convenience to using these, there are times when one would like the redirection of I/O to be handled automatically by the program, without the intervention of the operator at the console to answer the prompts (or fill in dialogue boxes) as required by these two high level procedures. Moreover, these procedures allow only sequential files, and they do not allow one to append to a file. Furthermore, they are not a part of the ISO standard, and therefore cannot be counted on to be present at all.

Suppose, for instance, that one wanted to collect data from the keyboard and store it into a disk file. Then later, (in this, or another program) one wanted to examine that disk file and manipulate the stored information further. All this is to be done without any prompts being supplied or any file names being requested from the keyboard; those are to be contained in the program itself. To achieve this, one must pursue library routines at a lower level than the simple I/O modules used thus far.

In general, one performs operations on sequential files with these steps (assuming that appropriate imports have been made):

1. Declare a file variable to logically identify the file within the program
or declare a channel variable to identify the logical/physical connection
2. Use the file's actual name (a string) to look it up (or create it) on the external device--say, a disk
3. Identify the logical file variable with the actual disk file (open the file)
4. Connect a program I/O stream to the previously opened file,
5. Read from or write to the stream, and hence the logical, and so the physical file
6. When finished, disconnect the stream from the program and
7. Close the file, ensuring that the data is secure on the disk.

Depending on the version of Modula-2 involved, there may be from one to five modules in the library that contain the functions necessary to implement this model. In classical versions, one was usually called *Files* (or *FileSystem*) and contained the data type *FILE* (or *File*), together with the functions to create new files on a disk (or other device), open them for I/O by identifying them with a logical file name, close them when finished, rename, or delete existing files, and handle errors. Such a module is concerned with the manipulation of the physical data on the disk or other storage medium. It usually has only rudimentary read/write procedures for text material and its functions may be specialized into two or more smaller modules each of which handles a different aspect of these tasks.

On the other hand, the functions for establishing a stream of text characters, connecting this to the already opened file, and then performing the actual read/write operations might also be in *Files*, (in older versions) or they might be situated in a separate module, variously called *Texts*, *TextIO* or *Streams* (Wirth suggested the last of these, but did not require it). This second module, whatever its name, manipulates streams through the logical stream name within the program, and its activities affect the contents of the physical (disk) files only indirectly through the connections that the program has established between the stream and the logical file, and between this and the physical file. Note that in the fullest elaboration of this scheme, the logical stream is an entirely different entity from the logical file, which in turn is a different entity from the physical file.

Some additional associated modules may be provided to ease the entering (and checking for validity) of valid file names, may specialize the directory lookup step, or may provide for separate encapsulation of the procedures for numeric I/O to files, as opposed to text I/O. One must consult the documentation for the specific implementation on this point, as the details vary widely.

Some early implementations actually used all the ideas presented so far and employed two data types to do so--*FILE* and *TEXT* (or *STREAM*). Many other classical versions of Modula-2 took a higher-level view of the matter, and employed only a single data type *File* that did double duty as both stream and file, for opening a file and for making the necessary connections between the physical file and the stream, or otherwise combined some of the distinct tasks on the list above into single steps.

ISO standard Modula-2 also takes a high level view, but hides the concept of a logical file altogether, and uses the abstraction of a *Channel* as the means to connect program data with physical files. Thus, in the

ISO model, one has:

- low level models to define and manipulate general channels
- middle level modules to define and manipulate specific kinds of channels
- modules to do the actual I/O once a channel has been opened

There might also be a low level housekeeping module called *Files* or *Filer* or *FileSystem* on top of which the specific models for different kinds of channel are built. This module and its contents are deliberately not standardized, as it is here that the differences between operating systems must be expressed. Thus, portable programs should avoid calling on routines from such a module.

[Contents](#)

8.6 Text I/O in ISO Standard Modula-2

This section assumes that the files being used are *text* files--that is, that they are sequential recordings of text streams. There are (at least) two ways of viewing a text stream.

A *Restricted Stream*: Reading is only from the beginning and writing is only to the current position. If an existing file is opened for writing, it is written over, not appended to, unless it is first read all the way to the end.

A *Rewindable Sequential Stream*: As above, except that there is the capability of rewinding the read or write position back to the start of the file. This permits it, for instance, to be written to and then read over, or vice versa, without having to close it and then re-open it first.

In the two subsections that follow, the ISO modules that elaborate each of these models will be discussed.

8.6.1 The Restricted Stream Model

Sequential files are connected to using the facilities of the module *StreamFile* that contains procedures for defining, opening, and closing such files. Writing to and reading from such files is done using *TextIO*, *WholeIO*, *RealIO*, and *LongIO*. These are the analogues of the corresponding S-modules, except that each procedure has one more parameter--the identifier of the file channel being employed. For instance, to write a cardinal to the standard output, one uses

```
SWholeIO.WriteCard (theCard, n);
```

whereas, to write to some previously opened file channel called *outChan*, one employs

```
WholeIO.WriteCard (outChan, theCard, n);
```

To see how the various components of *StreamFile* and the *xxIO* modules work together, consider the following module, which gathers a sequence of numbers from the keyboard and records them in a sequential file.

```
MODULE GetNStash;
```

```
(* Written by R.J. Sutcliffe *)
(* to illustrate the use of StreamFile *)
(* using ISO standard Modula-2 *)
(* last revision 1994 02 23 *)
```

```
(* This module reads a series of Integers from the keyboard and places them into a
disk file called "numbers". *)
```

```
FROM StreamFile IMPORT
  ChanId, Open, write, Close, OpenResults;
FROM SWholeIO IMPORT
  ReadInt;
FROM STextIO IMPORT
  WriteLn, WriteString, WriteChar, SkipLine;
FROM SIOResult IMPORT
  ReadResult, ReadResults;
IMPORT WholeIO; (* done qualified to distinguish *)
IMPORT TextIO;  (* from the S-modules above *)
```



```

VAR
  outfile : ChanId; (* identifier of the channel to use *)
  number  : INTEGER;
  res : OpenResults; (* to store the result of an open attempt *)

BEGIN
  (* Establish output channel and attach to disk file *)
  Open (outfile, "numbers", write, res);
  IF res = opened
    THEN (* Collect the numbers from the keyboard *)
      WriteString ( "Type in the integers.  Separate them");
      WriteLn;
      WriteString (" by carriage returns, and end  ");
      WriteLn;
      WriteString ("by typing a non-integer");
      WriteLn;
      REPEAT
        WriteChar (""); (* prompt *)
        ReadInt (number);
        IF ReadResult() = allRight
          THEN
            SkipLine;
            WholeIO.WriteInt (outfile, number, 0);
            TextIO.WriteLn (outfile); (* separate the numbers *)
          END; (* if *)
        UNTIL ReadResult() # allRight;
      Close (outfile); (* Now close physical file. *)
    END (* if res = opened *)

END GetNStash.

```

When this program was run the file *Numbers* was created and as numbers were typed, its contents became:

```

12
15
54
-100
0

```

It is important to understand that the procedures in such modules as *TextIO*, *WholeIO*, *RealIO*, and *LongIO* are identical to those of *STextIO*, *SWholeIO*, *SRealIO*, and *SLongIO* except for requiring the additional parameter specifying the channel. Notice the import of procedures from both *STextIO* and *TextIO*, as some writing was done to the standard output and some was done to the file. If it is considered desirable, all output could be done with *TextIO* provided that output to the standard channel is identified as such. This requires:

```

FROM StdChans IMPORT
  OutChan;

```

and then in the code, one may have both of:

```

WriteInt (outfile, number, 0);

```

```
WriteString (OutChan (), "by typing a non-integer");
```

by importing only from *TextIO* and *WholeIO*, and not bothering with *STextIO*. The same considerations apply to input, so that *SWholeIO* could be dispensed with altogether if one wrote:

```
FROM StdChans IMPORT
    StdInChan;
```

and then in the code:

```
ReadInt (StdInChan(), number);
```

The function procedures *StdChan.OutChan* and *StdChan.InChan* return the currently selected output (input) channels for standard output (input), even if it has been redirected by *RedirStdIO.OpenOutput* or *RedirStdIO.OpenInput*. If it is important to guarantee that the output goes to the default standard output stream (and is not subject to redirection) then import and use *StdChan.StdOutChan* or *StdChan.StdInChan* in the same way.

WARNING: The enclosing compiler or outer operating system cannot necessarily be assumed to know anything about the files created by user written program modules. Consequently, the program is responsible for closing all its own files. If this is not done, there is no guarantee that data will not be lost from the file. In fact, one can almost guarantee that data will be lost. Some implementations provide protection by having any program termination automatically close all open files, but this behaviour cannot be counted upon.

In the example that follows, some of the variations mentioned above have been adopted for another module that reads all those numbers back from the disk file, sums them and prints out the sum.

```
MODULE ReadNAdd;
```

```
(* Written by R.J. Sutcliffe *)
(* to illustrate the use of StreamFile *)
(* using ISO standard Modula-2 *)
(* last revision 1994 02 23 *)
```

```
(* This module reads a series of Integers from the disk file called "numbers". It
sums them and prints out the sum. *)
```

```
FROM StreamFile IMPORT
    ChanId, Open, read, Close, OpenResults;
FROM TextIO IMPORT
    WriteLn, WriteString, SkipLine, ReadChar;
FROM IOResult IMPORT
    ReadResult, ReadResults;
FROM WholeIO IMPORT
    ReadInt, WriteInt;
FROM StdChans IMPORT
    StdInChan, StdOutChan;
```

```
VAR
    infile, stdOut, stdIn : ChanId;
    number, sum : INTEGER;
    res : OpenResults;
    ch : CHAR;
```

```
BEGIN
```



```

stdOut := StdOutChan(); (* to force screen output *)
stdIn := StdInChan(); (* to force keyboard input *)

Open (infile, "numbers", read, res);
IF res = opened
  THEN
    sum := 0; (* initialize sum *)
    REPEAT (* Collect the numbers from the file *)
      ReadInt (infile, number);
      IF ReadResult (infile) = allRight
        THEN
          SkipLine (infile);
          INC (sum, number);      (* OK, so add to sum *)
        END; (* if *)
    UNTIL ReadResult (infile) # allRight;

    Close (infile);
    WriteString (stdOut, "The sum of the numbers is ");
    WriteInt (stdOut, sum, 6);
    WriteLn (stdOut);
  ELSE
    WriteString (stdOut, "Sorry, couldn't open the file");
    WriteLn (stdOut);
  END; (* if *)
WriteString (stdOut, "type a character to continue==");
SkipLine (stdIn);

END ReadNAdd.

```

When this program was run and allowed to work on the file generated above, the output was:

```

The sum of the numbers is      -19
type a character to continue=="numbers", read+write, res);

```

Of course, since in this model streams cannot be rewound, writing can only be done at the point last read. If it is required that the program be permitted to write over (erasing) the contents of an existing file, then the command is given in the form:

```

Open (thefile, "numbers", write+old, res);

```

8.6.2 The Rewindable Sequential Stream Model

The rewindable sequential file model is supported by the ISO library suite using the facilities of the module *SeqFile* that contains procedures for defining, opening, and closing such files. While a file is open, and has been read in part or all, the reading position can be rewound to the beginning. This is done using

```

PROCEDURE Reread (cid: ChanId);

```

If the purpose is to erase the current contents of the file and write over the top of them from the beginning, use

```

PROCEDURE Rewrite (cid: ChanId);

```

In order to open a file initially for reading, one uses *OpenRead*. If it is desired to open the file for the purpose of writing to the end rather than to the beginning as was done in the *StreamFile* example above, it should be opened with *OpenAppend*, otherwise with *OpenWrite*. However, all three open procedures take the *read*, *write*, and/or *old* parameters so as to allow additional combinations. Whatever parameter is passed in the third position:

OpenRead implies *read* mode and unless *write* is passed, it implies *old*

OpenWrite implies *write*

OpenAppend implies both *write* and *old*

Each of the three Open procedures initializes a *mode*; a sequential file is either available for being read or written to at any given time, but not both. The third parameter in these routines is actually a set of permission flags relating to these modes. For the present purposes, these flags can be any combination of *read*, *write*, and *old*. If more than one is given, use the + operator between them. (The syntax of working with such sets will be covered in more detail later in the text.) Thus, if a file is to be permitted to be open for either reading and writing, one can use either *OpenRead* or *OpenWrite* depending on what is to be done first, but the flag parameter should have *write* in the first case, and *read* in the second. This makes it possible to change the mode of the file at a later time.

If the flag *old* is used with *OpenWrite* then an existing file can be written over; otherwise, it is an error to attempt to open an existing file for writing.

One might begin with *OpenWrite* or *OpenAppend* and also supply the *read* parameter if planning to issue a *Reread* later on. Indeed, if such was the plan, the read would have to be supplied, or *Reread* would be unable to select a mode and after it tried to execute neither reading nor writing could then take place. Likewise, if the plan were to begin with *OpenRead* and later to write over or append to the file (the latter after reading to the end) by issuing a *Rewrite* then the *write* parameter would have to be supplied initially.

However, at any *given* time, the file is in either *input* mode or *output* mode. The mode is chosen initially according to the routine that opens it. The mode can then be changed by *Rewrite* or *Reread*, provided that the appropriate *write* or *read* flags were provided in the first place. Attempts to write when not in *output* mode or to read when not in *input* mode both produce run time errors.

Notice that the same physical file may be opened with either *StreamFile* or with *SeqFile* depending on the logical needs at the time. Of course, few operating systems would allow the file to be open in two ways simultaneously. One might make a devious attempt:

```
IMPORT  SeqFile;
IMPORT StreamFile;
VAR
    seq : SeqFile.ChanId;
    stream : StreamFile.ChanId;

BEGIN (* pathological code *)
    StreamFile.Open (stream, "Myfile", StreamFile.read+StreamFile.write);
    seq := stream; (* they are assignment compatible *)
    SeqFile.Rewrite (seq); (* exception occurs *)
```

This would fail, because the procedures of *SeqFile* check first to ensure that the channel operated on by them has in fact been opened by *SeqFile* and not by some other module (in this case *StreamFile*).

What follows is a module to illustrate some of the points made thus far. The same file employed above is opened once again, some more numbers are typed for it; the *Reread* is issued, and then the contents typed to the screen. Note the use of both user-connected and standard channels, and the need, for instance, to use a channel parameter when issuing *SkipLine* commands.

```
MODULE StashMoreAndType;

(* Written by R.J. Sutcliffe *)
(* to illustrate the use of SeqFile *)
(* using ISO standard Modula-2 *)
(* last revision 1994 02 23 *)
```

```
(* This module reads a series of Integers from the keyboard and appends them to a
disk file called "numbers". *)
```

```
FROM SeqFile IMPORT
```

```
    ChanId, OpenAppend, write, read, Close, Reread, OpenResults;
```

```
FROM TextIO IMPORT
```

```
    WriteLn, WriteString, SkipLine, ReadChar, WriteChar;
```

```
FROM IOResult IMPORT
```

```
    ReadResult, ReadResults;
```

```
FROM WholeIO IMPORT
```

```
    ReadInt, WriteInt;
```

```
FROM StdChans IMPORT
```

```
    StdInChan, StdOutChan;
```

```
VAR
```

```
    thefile, stdOut, stdIn : ChanId;
```

```
    number : INTEGER;
```

```
    res : OpenResults;
```

```
    ch : CHAR;
```

```
BEGIN
```

```
    stdOut := StdOutChan(); (* to force screen output *)
```

```
    stdIn := StdInChan(); (* to force keyboard input *)
```

```
    (* Establish channel and attach to disk file *)
```

```
    OpenAppend (thefile, "numbers", write+read, res);
```

```
    IF res = opened
```

```
        THEN (* Collect the numbers from the keyboard *)
```

```
            WriteString (stdOut, "Type in the integers. Separate ");
```

```
            WriteLn (stdOut);
```

```
            WriteString (stdOut, " by carriage returns, and end ");
```

```
            WriteLn (stdOut);
```

```
            WriteString (stdOut, "by typing a non-integer");
```

```
            WriteLn (stdOut);
```

```
            REPEAT
```

```
                WriteChar (stdOut, ""); (* prompt *)
```

```
                ReadInt (stdIn ,number);
```

```
                IF ReadResult(stdIn) = allRight
```

```
                    THEN
```

```
                        SkipLine (stdIn);
```

```
                        WriteInt (thefile, number, 0);
```

```
                        WriteLn (thefile); (* separate the numbers *)
```

```
                    END; (* if *)
```

```
            UNTIL ReadResult(stdIn) # allRight;
```

```
            Reread (thefile); (* go back to the start *)
```

```
            REPEAT (* Collect the numbers from the file *)
```

```
                ReadInt (thefile, number);
```

```
                IF ReadResult (thefile) = allRight
```

```
                    THEN (* ok, so skip to next and print on screen *)
```

```
                        SkipLine (thefile);
```

```
                        WriteInt (stdOut, number, 10);
```

```
                    END; (* if *)
```

```

UNTIL ReadResult (thefile) # allRight;

Close (thefile); (* Now close physical file. *)
WriteLn (stdOut);
WriteString (stdOut, "Contents typed to standard output.");
WriteLn (stdOut);
WriteString (stdOut, "type a character to continue ==");
ReadChar (stdIn, ch);

END (* if res = opened *)

END StashMoreAndType.

```

When this program was executed and a few more numbers typed, the screen looked like this:

```

Type in the integers. Separate
by carriage returns, and end
by typing a non-integer
>12
>15
>-3
>done
      12      15      54      -100      0
12      15      -3
Contents typed to standard output.
type a character to continue ==
• those for handling file tasks (open, close, etc.)

```

Library module: *Files/Filer/FileSystem*

- those for doing the actual stream I/O

Library module: included above, or a comprehensive *TextIO*

The non-standard comprehensive *TextIO* referred to here usually includes routines for all the *Readxx* and *Writexx* tasks that in the ISO libraries are split among *WholeIO*, *RealIO*, *LongIO*, and *TextIO*. The file housekeeping module (whatever its name) defines the data type *File* (or *FILE*) and handles all the opening, creating, renaming, and other housekeeping tasks, and usually has some rudimentary text I/O such as *ReadChar* and *WriteChar* upon which the routines in *TextIO* are usually built.

```

DEFINITION MODULE TextIO; (* non-standard *)

```

```

FROM Files IMPORT
    File;

```

```

CONST

```

```
eol = 15C; (* implementation dependent character *)
```

VAR

```
    consoleOK, txDone : BOOLEAN;  
(* The first of these is for examining the success of the last operation on the  
console, and the second is for all other streams. *)  
    termCH : CHAR;  
    console : FILE;
```

```
PROCEDURE WriteChar (f : FILE; ch : CHAR);  
PROCEDURE WriteString (f : FILE; s : ARRAY OF CHAR);  
PROCEDURE WriteLn (f : FILE);  
PROCEDURE WriteInt (f : FILE; i : INTEGER; flen : CARDINAL);  
PROCEDURE WriteCard (f : FILE; c : CARDINAL; flen : CARDINAL);  
PROCEDURE WriteReal (f : FILE; r : REAL; flen : CARDINAL;  
                    digits: INTEGER);  
PROCEDURE ReadChar (f : FILE; VAR ch : CHAR);  
PROCEDURE ReadString (f : FILE; VAR s : ARRAY OF CHAR);  
PROCEDURE ReadInt (f : FILE; VAR i : INTEGER);  
PROCEDURE ReadCard (f : FILE; VAR c : CARDINAL);  
PROCEDURE ReadReal (f : FILE; VAR r : REAL);  
END TextIO.
```

There is much more variation in modules for file handling. Not only are the names different, so is the syntax. Only a careful perusal of the manuals can enlighten the user. Indeed, since such a module is somewhat low level, even an ISO standard library suite probably imposes the two sequential file models on top of some implementation specific file handling module that is called to do all the actual work. An example of such a module can be found in the appendix. In any event, it is not too difficult to adapt the programs of the first two sections to work with these non-standard versions with a little re-writing.

To illustrate the use of such a module, here follows a program that will copy the file of integers above to another file with a different name. This copy will not employ *TextIO*, but instead will import only from a module called *Filer*.

The student should make careful note of the differences in syntax and spelling that may be necessary.

MODULE FileCopy;

```
(* Written by R.J. Sutcliffe *)  
(* to illustrate the use of nonstandard/low-level file handlers *)  
(* last revision 1994 02 23 *)
```

```
(* This module reads a file called "numbers" and copies it into a file called  
"numbers.bak" with type "TEXT" and creator "NISI". *)
```

FROM Filer **IMPORT**

```
File, (* the type of the ADT *)  
Open, Close, Create, (* typical housekeeping stuff *)  
FileErr, (* the type of the result variables *)
```

```
FileDone, (* a global of the above type, set by all operations *)  
ReadChar, WriteChar; (* simple I/O within the filing module *)
```

```
FROM InOut IMPORT (* let's go entirely classical on this one *)  
    WriteLn, WriteString;
```

```
VAR  
    infile, outfile : File;  
    nameIn, nameOut : ARRAY [0 .. 30] OF CHAR;  
    ch : CHAR;
```

```
BEGIN  
    nameIn := "numbers";  
    Open (infile, nameIn);  
    nameOut:= "numbers.bak";  
    Create (nameOut, "NISI", "TEXT"); (* Macintosh specific *)  
    Open (outfile, nameOut);
```

```
REPEAT  
    ReadChar (infile, ch);  
    IF FileDone = FileOK  
        THEN  
            WriteChar (outfile, ch);  
        END;  
UNTIL FileDone # FileOK;
```

```
Close (infile);  
Close (outfile);  
WriteString ("Copy complete.  ");  
WriteString (nameIn);  
WriteString ("==");  
WriteString (nameOut);
```

```
END FileCopy.
```

[Contents](#)

8.7 Binary I/O

The text files used thus far are very versatile, for they are stored in much the same form by all applications in all environments. Thus it is possible for the text editor to be used to create a data file for program use. Likewise, the text output files generated by programs can easily be read by any other program with text capability.

There are times, however, when it is more desirable to input or output data in a binary form. The channels employed are still streams, but not of text. Rather, they are streams of raw data copied directly from the computer's memory.

Typical facilities for raw I/O may be found in non-standard versions residing directly in the Filer/Files/FileSystem module, usually in the form of the following procedures:

```
(* typical non-ISO procedures *)
PROCEDURE ReadWord (file : File; VAR input : WORD);
  (* Reads the next word from the file 'file' and stores it in 'input'. *)

PROCEDURE ReadBytes (file : File; buffer : ADDRESS; VAR length : CARDINAL);
  (* Reads 'length' bytes from the file 'file' and stores them at 'buffer'. Actual
  number of bytes read returned in length. If the buffer is too small, data will be
  overwritten. This is a low level procedure -- don't use unless you know what you are
  doing *)

PROCEDURE WriteByte (file : File; output : BYTE);
  (* Writes the byte in 'output' to the file 'file'. *)

PROCEDURE WriteWord (file : File; output : WORD);
  (* Writes the word in 'output' to the file 'file'. *)

PROCEDURE WriteBytes (file : File; buffer : ADDRESS; VAR length : CARDINAL);
  (* Writes the 'length' bytes starting at 'buffer' to the file 'file'. Actual number
  of bytes read returned in length. If the buffer is too small, undefined bytes will be
  written. So don't use this unless you know what you are doing *)
```

As it will be used below, it is useful to define the term *buffer*.

A buffer is a temporary storage area that is used to store information being transmitted to or from an external location (including a physical file).

In the ISO suite of I/O library modules, on the other hand, channels can be opened in the same way as previously described using *SeqFile* or *StreamFile* and then I/O operations handled using facilities of *RawIO*. There is also an *SRawIO*, but unless the user is aware of the meaning of sending raw binary data to the standard output, there is little point in employing its routines. Here is the definition of *RawIO*.

```
DEFINITION MODULE RawIO; (* from ISO suite *)

  (* Reading and writing data over specified channels using raw operations, that is,
  with no conversion or interpretation. The read result is of the type
  IOConsts.ReadResults. *)

IMPORT IOChan, SYSTEM;
```

```

PROCEDURE Read (cid: IOChan.ChanId; VAR to: ARRAY OF SYSTEM.LOC);
  (* Reads storage units from cid, and assigns them to successive components of to.
  The read result is set to the value allRight, wrongFormat, or endOfInput. *)

PROCEDURE Write (cid: IOChan.ChanId; from: ARRAY OF SYSTEM.LOC);
  (* Writes storage units to cid from successive components of from. *)

END RawIO.

```

For instance, the integers collected in the first example *GetNStash* of section 8.5.1 could have been stored to a file in raw form rather than in text form by replacing the line

```
WholeIO.WriteInt (outfile, number, 0);
```

with the line

```
RawIO.Write (outfile, number);
```

as the variable *number* of the type `INTEGER` (and all other variables, of whatever type) is compatible with the parameter `ARRAY OF SYSTEM.LOC`.

Then, in the module *ReadNAdd* that followed the line

```
ReadInt (infile, number);
```

would be replaced with the line

```
Read (infile, number);
```

where *Read* had been imported from *RawIO*.

In both cases, however, the file should be opened with the parameter *raw* as shown below:

```
Open (outfile, "numbers", write+raw, res);
```

and

```
Open (infile, "numbers", read+raw, res);
```

When using either *StreamFile* or *SeqFile* without the *raw* flag, a flag of *text* is implied. Both *raw* and *text* forms of I/O could be done with a channel if both flags were given, but that combination is rather unlikely. The disadvantage to using raw I/O is that the resulting files would not be text files, but images of the computer's memory. Thus, they could not be read by a text editor. On the other hand, the memory storage of an integer is likely to occupy only two or four locations, whereas one location is required for each character when it is written as text. The raw file is much smaller in size, and can be written to and read from much more quickly than the corresponding text file. As is often the case, speed is gained by taking a low level view, but the benefit is at the expense of convenience.

To illustrate some of these ideas with a fuller example, the *FileCopy* module is here rewritten to employ the ISO modules rather than the low-level *Filer* as in [section 8.6.3](#). Observe that although the file being copied happens to be in fact a text file, this is not used by the program, and its code could copy any file. In addition, this program makes use of its own internal buffer to store the file.

```
MODULE FileCopyRaw;
```



```

(* Written by R.J. Sutcliffe *)
(* to illustrate the use of ISO module RawIO *)
(* last revision 1994 02 10 *)
(* This module reads a file called "numbers" and copies it to "numbers.bak". *)

FROM StreamFile IMPORT
    ChanId, Open, read, write, old, raw, Close, OpenResults;
FROM IOResult IMPORT
    ReadResult, ReadResults;
IMPORT RawIO;
FROM SYSTEM IMPORT
    LOC;

CONST
    bMaxIndex = 20;
TYPE
    Buffer = ARRAY [0 .. bMaxIndex] OF LOC;
VAR
    infile, outfile : ChanId;
    ch : CHAR;
    buffer: Buffer;
    res : OpenResults;
    countR, countW : CARDINAL;

BEGIN
    Open (infile, "numbers", read+raw, res);
    IF res = opened
    THEN
        Open (outfile, "numbers.bak", write+raw+old, res);
        (* allow overwrite *)
        IF res = opened
        THEN
            REPEAT
                countR := 0;
                REPEAT (* fill the buffer as much as possible *)
                    RawIO.Read (infile, buffer [countR]);
                    INC (countR);
                UNTIL (ReadResult (infile) = endOfInput) OR (countR > 0 (* write stuff
read in last loop *))
                DO
                    RawIO.Write (outfile, buffer [countW]);
                    INC (countW);
                    DEC (countR);
                END;
            UNTIL (ReadResult (infile) = endOfInput)
            END;
        END;
    END FileCopyRaw.

```

Contents

8.8 Notes on File I/O

There are a number of observations that should be made in connection with the module(s) that handle I/O at this particular level (stream and/or file) in Modula-2.

1) As already noted, apart from ISO standard Modula-2, these I/O module names may vary from one implementation to another. Naturally, the names of the individual objects in them will vary also, but the ones given here should be fairly typical.

2) In creating and using names for the physical files situated on the disk, there is almost always a strict limitation on the number of characters that may appear in a name. Exclusive of a three or four letter suffix such as .TXT or .OBJ, a typical limit may be as low as eight to ten characters in some primitive operating systems. If more than this number of characters are specified, the system may generate an error, or it may truncate the identifier down to the limit and then try to open the file, perhaps without telling the user that it has done so. In most cases, this will probably be satisfactory, but care must be taken not to end up with the same first characters for what were supposed to be two different file names. In that case, when the second file is written to, all the data from the first one will be lost. This system limitation, whatever it is, will probably apply to the names of all separately compiled modules, whether they are collected into a library or not.

3) There may well be several disk devices on line at the same time. These are usually distinguished by either a "device number" or by having a "volume name" of their own. In these cases, the file name could also have a "prefix" to identify the device. Here are a few examples from various systems:

"MYDISK:MYFILES.TEXT" denotes the text file named "MYFILES" on the disk whose name is "MYDISK".

"#5:NEWFILES" denotes the file "NEWFILES" on the device numbered 5.

"C:DATAFILE.TXT" denotes the device name "C:" (usually the hard disk) and the text file "DATAFILE". This particular designation assumes that any directory and subdirectory names that might be required have previously been set so that they are not needed in the file name.

"MYDISK/MYFILES/STUFF" (or with backslashes in some systems) denotes the main directory "MYDISK", the subdirectory "MYFILES" and the file "STUFF" within that subdirectory. In operating systems employing such designations, these are called "Pathnames"

Although these are from different operating systems, and no one of them would allow all these ways of locating a file, one can make a few general comments. Where volume names are employed, the system should be able to find the file regardless of which physical device contains the disk. If a number or letter to designate a device is employed, that device will be used, regardless of the volume name found there. If no prefix is supplied by the program, the system itself will look for the volume name or path name set as the prefix either by the system filer or when the disks were first started up.

Several other approaches to the use of volumes may be used, and in some systems there will be a module containing a collection of procedures for mounting, dismounting, and otherwise manipulating volumes as distinct from files.

4) Notice that in using the version's higher level procedures *OpenInput* and *OpenOutput* from the classical procedure *InOut*, one has to provide the default suffix ".TEXT" as the parameter. In that case, if the file could not be found, the system would add the suffix provided and try again. In using the lower level modules in this section one must specify the exact filename, including any necessary suffix. This is also the case in those systems that vary the use of *OpenInput* and *OpenOutput* to take filenames instead of default extensions to filenames.

5) Note too that on many systems physical file names are case-insensitive. That is, "MyProgram" (as a file name) would be equivalent to "MYPROGRAM" as far as the operating system is concerned.

6) When reading a "stream" or "text" in those systems that have this type, the fact of EOS (or EOT) becoming true does not necessarily imply that the end of the connected physical file was reached. This function can also return TRUE if the EOF *character* (where one is defined) was read (usually Control-C or Control-Z). Hopefully, this also corresponds to the physical end of the file, and it will in most cases, but a stray control character could prematurely terminate a file and cut off access to part of it.

8.8.1 Special Notes on the Macintosh Operating System

In addition to some of the other considerations, the Macintosh operating system has as an integral part of its operation the

concept of a *Creator* and a *Type* as explicit file information. Applications store creator information with their files, so that when these files are opened from the finder, the creating application will be opened as well. In addition, each application may be capable of creating and opening one or more types of files, including ones normally used by other applications. *Creator* and *Type* are both defined as ARRAY [0..3] OF CHAR, and must be provided whenever a file is created. Thus, the ISO modules *SeqFile*, *StreamFile*, and *RndFile* must pass this information to the operating system whenever they create a file. Since this facility is not included in the ISO modules themselves, some default type and creator must be maintained at a lower level. There should also be a means of changing this information. In the p1 compiler this is handled by importing from MacMisc using:

```
PROCEDURE SetTypeAndCreator (cid: ChanId; type, creator: OSType);
(* if "cid" identifies an open channel to a disk file, type and creator are set
according to the parameters. Otherwise the exception "notAChannel" or "notAvailable"
is raised.
*)
```

Note that here, the type and creator have to be set on a channel once it is opened. In the author's own implementation, this is done a little differently, using:

```
DEFINITION MODULE   MacFileTypes;
```

```
(* =====
  Definition and Implementation © 1993
                        by R. Sutcliffe
        Trinity Western University
6600 Glover Rd., Langley, BC Canada V3A 6H4
        e-mail: rsutc@twu.ca
        Last modification date 1993 10 20
===== *)
```

```
(* The purpose of this module is to allow for standard creators and file types to be
communicated from one module to another -- usually from a client program to some
library module.  In this implementation, the intention is that the ISO device drivers
will call the procedures here to determine what type and creator to use when
connecting channels to files. *)
```

```
FROM Filer IMPORT
```

```
  NameType; (* an ARRAY 0..3 OF CHAR *)
```

```
(* Set and get file types *)
```

```
PROCEDURE SetSeqType (t : NameType);
```

```
PROCEDURE SeqType (): NameType;
(* the default is "TEXT" *)
```

```
PROCEDURE SetStreamType (t : NameType);
```

```
PROCEDURE StreamType (): NameType;
(* the default is "TEXT" *)
```

```
PROCEDURE SetRndType (t : NameType);
```

```
PROCEDURE RndType (): NameType;
(* the default is "???" *)
```

```
PROCEDURE SetGenType (t : NameType);
```

```

PROCEDURE GenType (): NameType;
    (* the default is "?????" This one is for user programs or device drivers *)

(* Set and get creator types *)

PROCEDURE SetSeqCreator (t : NameType);
PROCEDURE SeqCreator (): NameType;
    (* the default is "?????" *)

PROCEDURE SetStreamCreator (t : NameType);
PROCEDURE StreamCreator (): NameType;
    (* the default is "?????" *)

PROCEDURE SetRndCreator (t : NameType);
PROCEDURE RndCreator (): NameType;
    (* the default is "?????" *)

PROCEDURE SetGenCreator (t : NameType);
PROCEDURE GenCreator (): NameType;
    (* the default is "?????" This one is for user programs or device drivers *)

END MacFileTypes.

```

Here, there is a default type that is used all the time by each of the standard modules, but this default may be changed by the user for subsequently created files. However, to change the type or creator of an existing file requires lower level procedures. Needless to say, this information is useless unless the programmer knows what strings to use. "MPS " (note the space after the 'S') is the creator of Macintosh Programming Workshop files, and "NISI" is the creator of Nisus word processing files. "TEXT" is the type of text files, and is already the default for both stream models. "?????" is a generic creator/type marker.

[Contents](#)

8.9 Standard Channel I/O in ISO Modula-2

As has already been observed, in ISO standard Modula-2, the modules *SWholeIO*, *SRealIO*, *SLongIO*, and *STextIO* operate on the standard channels. These are defined in the Module *StdChans*:

```
DEFINITION MODULE StdChans;
```

```
IMPORT IOChan;
```

```
TYPE
```

```
  ChanId = IOChan.ChanId;  
  (* Values of this type are used to identify channels *)  
  
  (* The following functions return the standard channel values.  
     These channels cannot be closed. *)
```

```
PROCEDURE StdInChan (): ChanId;
```

```
  (* Returns the identity of the implementation-defined standard source for program  
  input. *)
```

```
PROCEDURE StdOutChan (): ChanId;
```

```
  (* Returns the identity of the implementation-defined standard source for program  
  output. *)
```

```
PROCEDURE StdErrChan (): ChanId;
```

```
  (* Returns the identity of the implementation-defined standard destination for  
  program error messages. *)
```

```
PROCEDURE NullChan (): ChanId;
```

```
  (* Returns the identity of a channel open to the null device. *)  
  
  (* The following functions return the default channel values *)
```

```
PROCEDURE InChan (): ChanId;
```

```
  (* Returns the identity of the current default input channel. *)
```

```
PROCEDURE OutChan (): ChanId;
```

```
  (* Returns the identity of the current default output channel. *)
```

```
PROCEDURE ErrChan (): ChanId;
```

```
  (* Returns the identity of the current default error message channel. *)
```

```
  (* The following procedures allow for redirection of default channels *)
```

```
PROCEDURE SetInChan (cid: ChanId);
```

```
  (* Sets the current default input channel to that identified by cid. *)
```

```
PROCEDURE SetOutChan (cid: ChanId);
```

```
  (* Sets the current default output channel to that identified by cid. *)
```

```
PROCEDURE SetErrChan (cid: ChanId);
```

```
( * Sets the current default error channel to that identified by cid. *)
```

```
END StdChans.
```

Observe that there is a third channel besides that for input and output. The error channel may be the same as the standard output, or it may be a file that logs errors. This channel is available to the programmer as well:

```
IF Error
  THEN
    TextIO.WriteString (StdChans.ErrChan(), theMessage)
  END;
```

It is also easy to see how the non-standard module *RedirStdIO* that was employed earlier in the text can be written on top of this standard one. It is only necessary to open new channels through *StreamFile* and then employ the procedures of *StdChans* to redirect the current default channel. The definition, and an implementation for the Macintosh operating system follow. Of course, the specific details that relate to the manner in which the file name is obtained before being passed to be opened will differ on other operating systems.

```
DEFINITION MODULE RedirStdIO;

( * =====
  Definition and Implementation © 1993-1997
    by R. Sutcliffe
    Trinity Western University
    7600 Glover Rd., Langley, BC Canada V3A 6H4
    e-mail: rsutc@twu.ca
    Last modification date 1997 07 02
  ===== *)

IMPORT ChanConsts;

TYPE
  OpenResults = ChanConsts.OpenResults;

PROCEDURE OpenResult () : OpenResults;
(* returns the result of the last attempt to open a file for redirection *)

PROCEDURE OpenOutput;
(* engages the user in a dialog to obtain a file for redirection of standard Output
and attempts to open the file so obtained *)

PROCEDURE OpenOutputFile (VAR fileName: ARRAY OF CHAR);
(* opens the file specified by fileName for redirection of output.  If the name
supplied is the empty string or the file could not be opened, control passes to
OpenOutput and the filename eventually used is returned in the parameter. *)

PROCEDURE CloseOutput;
(* returns the standard output channel to the default value *)

PROCEDURE OpenInput;
(* engages the user in a dialog to obtain a file for redirection of standard Input
and attempts to open the file so obtained *)
```

```

PROCEDURE OpenInputFile (VAR fileName: ARRAY OF CHAR);
(* Opens the file specified by fileName for redirection of input.  If the name
supplied is the empty string or is not found, control passes to OpenInput and the
filename eventually used is returned in the parameter. *)

PROCEDURE CloseInput;
(* returns the standard input channel to the default value *)

END RedirStdIO.

IMPLEMENTATION MODULE RedirStdIO;

(* =====
Definition and Implementation © 1993-1997
by R. Sutcliffe
Trinity Western University
7600 Glover Rd., Langley, BC Canada V3A 6H4
e-mail: rsutc@twu.ca
Last modification date 1999 07 22
===== *)

(* First cut was October 1993
1994 05 21 modified to p1 version
change some type imports
use put get File rather than File0
change string handling accordingly
change to proper name "Strings"
1996 08 08 removed extraneous local var from OpenOutput
1997 07 08 added OpenOutputFile and OpenInputFile *)

(* Mac Specific *)
FROM StandardFile IMPORT
    SFReply, SFTypeList, SFPutFile, SFGetFile;
FROM MacTypes IMPORT
    Str255, Str63, Point;
FROM Strings IMPORT
    Assign;
FROM StdChans IMPORT
    SetOutChan, StdOutChan, SetInChan, StdInChan;
IMPORT ChanConsts;
FROM ChanConsts IMPORT
    read, write, old;
FROM StreamFile IMPORT
    ChanId, Open, Close;
FROM SYSTEM IMPORT
    TOSTR255, FROMSTR255;
FROM MacString IMPORT
    MacToModString;
TYPE
    ModStr255 = ARRAY [0..255] OF CHAR;

VAR
    out, in : ChanId ;
    count : CARDINAL;

```



```

lastOpenResult : OpenResults;
inRedir, outRedir : BOOLEAN;

(*-----*)

PROCEDURE InternalOpenOutput (VAR fileName : ARRAY OF CHAR);
(* engages the user in a dialog to obtain a file for redirection of standard Output
and attempts to open the file so obtained *)

VAR
    result : OpenResults;
    defFileName, thePrompt : Str255;
    p : Point;
    sfreply : SFReply;
    mTempStr : ModStr255;
    gottaAsk : BOOLEAN;

BEGIN
    gottaAsk:=TRUE;
    IF fileName[0] <> "" (* file name non-empty? *)
        THEN (* go try it *)
            Open (out, fileName, write+old, result);
            gottaAsk:= NOT (result = ChanConsts.opened)
        END; (* if *)
    IF gottaAsk (* true if filename was empty or not found *)
        THEN
            thePrompt := TOSTR255 ("Output File?"); (* get dialog box prompt ready *)
            defFileName := TOSTR255 ("output"); (* and default file name *)
            SFPutFile (p, thePrompt, defFileName, NIL,sfreply); (* and go try with dialog
box *)
            IF sfreply.good (* got it OK *)
                THEN (* convert filename returned to Modula string *)
                    MacToModString (sfreply.fName, mTempStr);
                    Assign (mTempStr,fileName);
                    Open (out, mTempStr, write+old, result);
                END;
            END;
    lastOpenResult := result;
    IF result = ChanConsts.opened
        THEN
            SetOutChan (out);
            outRedir := TRUE;
        ELSE
            fileName[0]:= "";
        END; (* if didn't work, nothing is done *)
END InternalOpenOutput;

(*-----*)

PROCEDURE OpenResult () : OpenResults;
(* returns the result of the last attempt to open a file for redirection *)

BEGIN
    RETURN lastOpenResult;

```

```

END OpenResult;

(*-----*)

PROCEDURE OpenOutput;
(* engages the user in a dialog to obtain a file for redirection of standard Output
and attempts to open the file so obtained *)

VAR
    DummyFile : ARRAY [0..1] OF CHAR;

BEGIN
    DummyFile[0] := "";
    InternalOpenOutput (DummyFile);
END OpenOutput;

(*-----*)

PROCEDURE OpenOutputFile (VAR fileName : ARRAY OF CHAR);
(* opens the file specified by FileName for redirection of output.  If the name
supplied is the empty string or the file could not be opened, control passes to
OpenOutput and the filename eventually used is returned in the parameter. *)

BEGIN
    InternalOpenOutput (fileName);
END OpenOutputFile;

(*-----*)

PROCEDURE CloseOutput;
(* returns the standard output channel to the default value *)

BEGIN
    IF outRedir
    THEN
        Close (out);
        SetOutChan (StdOutChan ( ) );
        outRedir := FALSE;
    END;
END CloseOutput;

(*-----*)

PROCEDURE InternalOpenInput (VAR fileName : ARRAY OF CHAR);
(* This procedure is where all the work gets done.  Engages the user in a dialog to
obtain a file for redirection of standard Output and attempts to open the file so
obtained *)

VAR
    result : OpenResults;
    p : Point;
    sfreply : SFReply;
    mTempStr : ModStr255;
    gottaAsk : BOOLEAN;
    SFt: SFTypelist;

```

```

BEGIN
  gottaAsk:=TRUE;
  IF fileName[0] <> "" (* file name non-empty? *)
    THEN (* go try it *)
      Open (in, fileName, old+read, result);
      gottaAsk:= NOT (result = ChanConsts.opened)
    END; (* if *)
  IF gottaAsk (* true if filename was empty or not found *)
    THEN
      SFGetFile (p, "", NIL, -1, Sft, NIL, sfreply); (* go try with dialog box *)
      IF sfreply.good (* got it OK *)
        THEN (* convert filename returned to Modula string *)
          MacToModString (sfreply.fName, mTempStr);
          Assign (mTempStr,fileName);
          Open (in, mTempStr, old+read, result);
        END;
      END;
  lastOpenResult := result; (* from one or the other *)
  IF result = ChanConsts.opened
    THEN
      SetInChan(in);
      inRedir := TRUE;
    ELSE
      fileName[0]:= "";
    END; (* if didn't work, nothing is done *)
END InternalOpenInput;

(*-----*)

PROCEDURE OpenInput;
(* engages the user in a dialog to obtain a file for redirection of standard Input
and attempts to open the file so obtained *)

VAR DummyFile : ARRAY [0..1] OF CHAR;

BEGIN
  DummyFile[0]:= "";
  InternalOpenInput (DummyFile);
END OpenInput;

(*-----*)

PROCEDURE OpenInputFile (VAR fileName : ARRAY OF CHAR);
(* This is the procedure that does all the work. Opens the file specified by fileName
for redirection of input. If the name supplied is the empty string or is not found,
control passes to OpenInput and the filename eventually used is returned in the
parameter. *)

BEGIN
  InternalOpenInput (fileName);
END OpenInputFile;

(*-----*)

```

```
PROCEDURE CloseInput;
(* returns the standard input channel to the default value *)

BEGIN
  IF inRedir
    THEN
      Close (in);
      SetInChan (StdInChan ( ) );
      inRedir := FALSE;
    END;
END CloseInput;

BEGIN (* main program *)
  lastOpenResult := ChanConsts.otherProblem;
  inRedir := FALSE;
  outRedir := FALSE;
END RedirStdIO.
```

In this system, the dialog boxes that are standard to the Macintosh operating system are presented to the user so that folders can be navigated in the usual way to obtain the file name. In a text based operating system, these sections would have to be replaced with a dialog to the interactive terminal where the user types in a file name.

[Contents](#)

8.10 Lower Level I/O in ISO Modula-2

The modules *StreamFile* and *SeqFile*, when used in conjunction with the procedures in *TextIO*, *WholeIO*, *RealIO*, and *LongIO* impose a model for I/O for channels opened by them based on the notion of the restricted stream or the rewindable stream, respectively.

The modules StreamFile, SeqFile, and others of a similar kind are examples of device drivers, that is, of modules that enforce a high level model of I/O appropriate for a particular abstract (logical) device.

The *device* referred to in the above definition is *abstract* because the physical device (say, disk media) may be capable of handling channels connected to any one of several different logical devices. Thus, from a logical point of view, the ISO standard sees sequential devices as distinct from stream devices or other devices, regardless of the actual hardware being employed. Operations (such as *Close* or *Rewind*) performed on channels in this way depend on the logical model; that is, a check is made to ensure that the channel was in fact opened by the correct device driver.

In some cases, it may be necessary to perform operations at a lower level than that available directly from the device driver facilities. This may be because it is necessary to perform just a few operations that fall outside the strict model, or because the application program needs a rather different model than those provided by the vendor. In the extreme case, the user may wish to write her own device driver to implement an appropriate abstract model.

In order to meet such needs, ISO Modula-2 provides a module at a lower level that allows I/O to be performed without reference to any device driver.

Input and output performed directly on channels without the benefit of a high level device driver is called device independent.

In non-standard Modula-2, all the I/O facilities were likely to be contained in the two modules *TextIO* and *Files/FileSystem/Filer*. It is up to the programmer to impose the logical model through the appropriate use of such routines. In ISO Modula-2, there is a separate module called *IOChan* that enables device independent operations on a channel by channel basis. Operations in *IOChan* include:

```
PROCEDURE Look (cid: ChanId; VAR ch: CHAR; VAR res: IOConsts.ReadResults);
  (* If there is a character as the next item in the input stream cid, assigns its
  value to ch without removing it from the stream; otherwise the value of ch is not
  defined.  res (and the stored read result) are set to the value allRight, endOfLine,
  or endOfInput. *)

PROCEDURE Skip (cid: ChanId);
  (* If the input stream cid has ended, the exception skipAtEnd is raised; otherwise
  the next character or line mark in cid is removed, and the stored read result is set
  to the value allRight. *)

PROCEDURE SkipLook (cid: ChanId; VAR ch: CHAR; VAR res: IOConsts.ReadResults);
  (* If the input stream cid has ended, the exception skipAtEnd is raised; otherwise
  the next character or line mark in cid is removed.  If there is a character as the
  next item in cid stream, assigns its value to ch without removing it from the stream.
  Otherwise, the value of ch is not defined.  res (and the stored read result) are set
  to the value allRight, endOfLine, or endOfInput. *)

PROCEDURE WriteLn (cid: ChanId);
```

```
(* Writes a line mark over the channel cid. *)
```

```
PROCEDURE TextRead (cid: ChanId; to: SYSTEM.ADDRESS; maxChars: CARDINAL; VAR  
charsRead: CARDINAL);
```

```
(* Reads at most maxChars characters from the current line in cid, and assigns  
corresponding values to successive components of an ARRAY OF CHAR variable for which  
the address of the first component is to. The number of characters read is assigned  
to charsRead. The stored read result is set to allRight, endOfLine, or endOfInput. *)
```

```
PROCEDURE TextWrite (cid: ChanId; from: SYSTEM.ADDRESS; charsToWrite: CARDINAL);
```

```
(* Writes a number of characters given by the value of charsToWrite, from  
successive components of an ARRAY OF CHAR variable for which the address of the first  
component is from, to the channel cid. *)
```

```
PROCEDURE RawRead (cid: ChanId; to: SYSTEM.ADDRESS; maxLocs: CARDINAL; VAR locsRead:  
CARDINAL);
```

```
(* Reads at most maxLocs items from cid, and assigns corresponding values to  
successive components of an ARRAY OF LOC variable for which the address of the first  
component is to. The number of characters read is assigned to locsRead. The stored  
read result is set to the value allRight, or endOfInput. *)
```

```
PROCEDURE RawWrite (cid: ChanId; from: SYSTEM.ADDRESS; locsToWrite: CARDINAL);
```

```
(* Writes a number of items given by the value of locsToWrite, from successive  
components of an ARRAY OF LOC variable for which the address of the first component  
is from, to the channel cid. *)
```

Several other operations are also available; these are detailed in a full listing of this module in [Appendix 5](#). When a channel is opened by a device driver, procedures to do all the above operations on that channel must be provided by the device driver, even though the specific I/O model does not necessarily use them all. When I/O is done at the level of *IOChan*, however, any of the channel's low level procedures may be employed through calls to the above. Here is yet another version of the file copying program to illustrate some use of *IOChan*.

```
MODULE FileCopyLow;
```

```
(* Written by R.J. Sutcliffe *)  
(* to illustrate the use of ISO module IOChan *)  
(* last revision 1994 02 23 *)
```

```
(* Copies a text file character by character using the low level procedures  
IOChan.Look and IOChan.Skip. *)
```

```
FROM STextIO IMPORT
```

```
WriteString, WriteChar, WriteLn, ReadString, SkipLine;
```

```
FROM SeqFile IMPORT
```

```
text, old, ChanId, OpenResults, OpenRead, OpenWrite, Close;
```

```
FROM IOConsts IMPORT
```

```
ReadResults;
```

```
FROM IOChan IMPORT
```

```
Look, Skip;
```

```
IMPORT TextIO;
```

```
VAR
```

```
inFile, outFile: ChanId;
```

```

inFileName, outFileName: ARRAY [0 .. 79] OF CHAR;
ch: CHAR;
resOpen: OpenResults;
resRead: ReadResults;

```

BEGIN

```

WriteString ('Name of input file ? ==>');
ReadString (outFileName);
SkipLine;
OpenWrite (outFile, outFileName, text, resOpen);

IF resOpen = opened (* for write *)
  THEN
    REPEAT
      Look (inFile, ch, resRead);
      (* see if character available -- uses low level IOChan *)
      IF resRead = allRight
        THEN (* yes, so *)
          TextIO. WriteChar (outFile, ch); (* output it *)
          Skip (inFile); (* skip to next one in input *)
        ELSIF resRead = endOfLine THEN
          TextIO. WriteLn (outFile);
          (* place end of line in output *)
          Skip (inFile);
          (* remove input end of line mark/state *)
        END (* if resRead *);
      UNTIL resRead = endOfInput; (* only possibility left *)
      WriteString ("Copy complete. ");
      WriteString (inFileName);
      WriteString (" copied to ");
      WriteString (outFileName);
      WriteLn;
      Close (outFile);
    END; (* if resOpen = opened (* for write *) *)
  Close (inFile);
END; (* if resOpen = opened (* for read *) *)
END FileCopyLow.

```

When this program was run, one session looked like:

```

Name of input file ? ==>FileCopy.MOD.BAK
Copy complete. FileCopy.MOD copied to outFileName

```

An examination of the output file showed that its contents were indeed correct.

The procedures of *IOChan* also provide some idea how some of those in *TextIO* (which is not device dependent) might be implemented by calling device independent routines. Here are a few from the author's own implementation:

```

PROCEDURE ReadChar (cid: IOChan.ChanId; VAR ch: CHAR);
  (* If possible, removes a character from the input stream cid and assigns the
  corresponding value to ch. The read result is set to the value allRight, endOfLine,
  or endOfInput. *)

```

```

VAR
    howMany : CARDINAL;

BEGIN
    IOChan.TextRead (cid, SYSTEM.ADR(ch),1, howMany)
END ReadChar;

PROCEDURE ReadRestLine (cid: IOChan.ChanId; VAR s: ARRAY OF CHAR);
    (* Removes any remaining characters from the input stream cid before the next line
    mark, copying to s as many as can be accommodated as a string value. The read result
    is set to the value allRight, outOfRange, endOfLine, or endOfInput. *)
VAR
    res : IOConsts.ReadResults;
    ch : CHAR;

BEGIN
    ReadString (cid,s);
    IF IOChan.ReadResult (cid) = IOConsts.allRight (* more there *)
    THEN
        res := IOConsts.outOfRange;
        REPEAT
            ReadChar (cid, ch);
        UNTIL IOChan.ReadResult (cid) # IOConsts.allRight;
        IOChan.SetReadResult (cid, res);
    END;

END ReadRestLine;

PROCEDURE ReadString (cid: IOChan.ChanId; VAR s: ARRAY OF CHAR);
    (* Removes only those characters from the input stream cid before the next line
    mark that can be accommodated in s as a string value, and copies them to s. The read
    result is set to the value allRight, endOfLine, or endOfInput. *)
VAR
    numToRead, numRead : CARDINAL;

BEGIN
    numToRead := HIGH(s) + 1;
    IOChan.TextRead (cid, SYSTEM.ADR (s), numToRead, numRead);
    IF numRead < numToRead
    THEN
        s[numRead] := 0C
    END;
END ReadString;

    (* The following procedure reads past the next line mark *)

PROCEDURE SkipLine (cid: IOChan.ChanId);
    (* Removes successive items from the input stream cid up to and including the next
    line mark, or until the end of input is reached. The read result is set to the
    value allRight, or endOfInput. *)
VAR
    res : IOConsts.ReadResults;
    ch : CHAR;

```



```
BEGIN
  IOChan.Look (cid, ch, res);
WHILE res = IOConsts.allRight
  DO
    IOChan.SkipLook (cid, ch, res);
  END;
IF res = IOConsts.endOfLine
  THEN
    IOChan.Skip (cid);
  END;
END SkipLine;
```

Although these procedures are all provided in a standard conforming implementation, the reader may well require the use of the ideas they contain in writing applications programs that need to deal with channel I/O on a device independent basis.

[Contents](#)

8.11 An Extended Low Level I/O Example--TermFile

As remarked in a previous chapter, I/O using the terminal device has some special characteristics. One is that characters typed at the console keyboard are usually placed on the screen as a visual aid to the user of what has been typed.

Copying input characters to the output device is called echoing.

In most implementations, of ISO Module-2 the Module *StdChans* probably sets up its default I/O channels through the facilities of *TermFile* (yet another device driver) though this dependence is not actually required by the standard itself. Indeed, there is no requirement that an implementation even support a terminal, or even programs that do I/O at all. Assuming it does, it is somewhat unlikely that many users would employ *TermFile* directly in a program. However, *TermFile* does permit the opening of channels to the terminal (assuming one exists) in such a way that the user can determine at what point the items typed at the keyboard are to be echoed on the screen.

*If a TermFile channel is opened without the flag echo, the channel is in line mode and the echoing to the screen is done when the character is typed. If the echo flag is set for an input, then the channel is said to be in single character mode and characters are echoed only when they are read (presumably from an internal buffer) by a text read operation such as *ReadChar*, or *ReadString* and not at the time they are typed.*

This may seem like a fine distinction; after all, it appears at first glance that the characters are all echoed to the screen sooner or later. In most cases, they are, but in single character mode (echo flag) if the read operation is *not* a *text read*, then the characters will never be echoed. This can be useful if an application program requires the user is to type a password. To prevent a snooper looking over her shoulder from reading that password from the screen, echoing should be disabled for that read. This is easily done, because any number of channels may be opened to the terminal through *TermFile*, some with echoing, and some without. Indeed, the rule applied by *TermFile* is that if *any* channels are open to the terminal in single character mode, then echoing must be postponed until characters are read. Therefore, in such a case, if some characters are read with a *text read* operation such as *TextIO.ReadString*, they will be echoed. If others are read by the use of *IOChan.Look* and *IOChan.Skip*, (defined in the standard as not being text reads for this purpose) they will not be echoed. Here is a module to determine whether this behaviour is properly exhibited by an ISO compliant *TermFile* device driver.

```
MODULE TestTerminal;
```

```
(* Written by R.J. Sutcliffe *)
(* to illustrate aspects of the use of ISO module TermFile *)
(* last revision 1994 02 21 *)
```

```
FROM TermFile IMPORT
  ChanId, Open, OpenResults, read, echo, Close;
FROM STextIO IMPORT
  WriteString, ReadString, SkipLine, WriteLn;
FROM IOChan IMPORT
  SkipLook, Look, SetReadResult;
FROM SIOResult IMPORT
  ReadResults, ReadResult;
```

```
PROCEDURE ReadToken (cid: ChanId; VAR s: ARRAY OF CHAR);
```

```
(* Skips leading spaces, and then removes characters from the input stream cid
before the next space or line mark, copying to s as many as can be accommodated as a
```

string value. The read result is set to the value allRight, outOfRange, endOfLine, or endOfInput. *)

VAR

```
    lastToRead, count : CARDINAL;  
    ch : CHAR;  
    resRead : ReadResults;
```

BEGIN

```
    count := 0;  
    lastToRead := HIGH(s);  
    Look (cid, ch, resRead);  
    WHILE (ch = " ") AND (resRead = allRight)  
        DO  
        SkipLook (cid, ch, resRead);  
    END;  
    WHILE (count <= lastToRead) AND (ch # " ")  
        AND (resRead = allRight)  
        DO  
        s[count] := ch;  
        INC (count);  
        SkipLook (cid, ch, resRead);  
    END;  
    (* if room left in string, terminate it *)  
    IF (count <= lastToRead) AND ((ch = " ")  
        OR (resRead # allRight))  
    THEN  
        s[count] := 0C  
    ELSIF count = lastToRead AND (resRead = allRight)  
        DO (* and knock the rest out *)  
        INC (count);  
        SkipLook (cid, ch, resRead);  
    END;  
    END;  
END ReadToken;
```

VAR

```
    term : ChanId;  
    resOpen : OpenResults;  
    password : ARRAY [0..63] OF CHAR;  
    count : CARDINAL;
```

BEGIN

```
    (* first, do things in the usual way *)  
    WriteString ("First test. All terminal channels in line mode");  
    WriteLn;  
    WriteString ("type password ==");  
    ReadString (password);  
    SkipLine;  
    WriteLn ;  
    WriteString ("password typed was: ");  
    WriteString (password);  
    WriteLn ;
```

```

(* Open new channel to the terminal in single character mode *)
Open (term, read+echo, resOpen);
IF resOpen = opened
  THEN
    WriteLn;
    WriteString ("Second test. One channel in char mode");
    WriteLn;
    WriteString ("type password ==");
    ReadToken (term, password);
    SkipLine;
    WriteLn ;
    WriteString ("password was:  ");
    WriteString (password);
    Close (term);
  END;
END TestTerminal.

```

When this program was executed, the output from one run looked like this:

```

First test. All terminal channels in line mode
type password ==>
password was:  ui56bn43

```

Observe that, as expected, the password was echoed when typed in the normal (line) mode, and was not when there was a channel to the terminal open in character mode, even though that channel was not explicitly used for the input. Note also that the procedure *ReadToken* contained in the above test program is also taken from the author's implementation of *TextIO* and behaves in that way when imported. That is, because it is implemented in terms of *Look* and *Skip*, it is not a *text read* for the purposes of this echoing rule. Although the author has tested other implementations and found them to have the same interpretation, it is possible that some may not, and therefore the user may need the *ReadToken* shown here.

[Contents](#)

8.12 Chapter Summary

This chapter covered these topics:

- the difference between high and low-level considerations
- implementation dependent and implementation defined items
- about binary, octal, and hexadecimal number systems
- about units of computer storage
- about some high level access to low-level facilities in Modula-2
- about the module SYSTEM
- about type transfer functions CAST (unsafe) and VAL (safe)
- about generic procedures
- what is a logical file, a program file and a physical file
- about sequences, streams, and channels
- about sequential and random access files
- how to manipulate files with sequential and restricted stream modules
- about raw input and output
- some of the inner workings of lower level ISO modules
- one method of using the device driver TermFile

It included discussion of the following Modula-2 items:

Reserved Words

none

Standard Identifiers

none

Symbols:

H (Hexadecimal numeral indicator)

B (Octal numeral indicator)

Imports:

Standard:

- From SYSTEM:
WORD, ADDRESS, BYTE ADR, CAST BITSPERLOC, LOCSPERWORD,
LOCSPERBYTE ADDADR, SUBADR, DIFADR, MAKEADR
- From SeqFile:
ChanId, OpenRead, OpenWrite, OpenAppend, Close, Reread, Rewrite
- From StreamFile:
Open, Close
- From RawIO:
Read, Write

- From TermFile:
 - ChanId, Open, OpenResults, read, echo, Close
- From one or more modules:
 - the flags:
 - read, write, text, raw, old, echo
 - the procedure:
 - OpenResults
 - the type:
 - ChanId
- From IOResult
 - ReadResult, ReadResults;
- From IOChan:
 - Look, Skip, SkipLook, TextRead, RawRead, WriteLn, TextWrite, RawWrite
- From StdChans:
 - StdInChan, StdOutChan, StdErrChan, NullChan, InChan, OutChan, ErrChan, SetInChan, SetOutChan, SetErrChan;

Macintosh Specific:

- From Events:
 - EventRecord, WaitNextEvent, GetCaretTime, keyDown, cmdKey, everyEvent, keyDownMask;
- Non-Standard:
 - From Filer/FileSystem/Files:
 - File, Open, Create, Close, FileErr, FileDone, ReadChar, WriteChar, ReadWord, ReadBytes, WriteByte, WriteWord, WriteBytes
 - From TextIO (non-standard):
 - WriteChar, WriteString, WriteLn, WriteInt, WriteCard, WriteReal ReadChar, ReadString, ReadCard, ReadInt, ReadReal,
 - From Keyboard:
 - Read, BusyRead;

[Contents](#)

8.13 Assignments

Questions:

1. What is an implementation restriction? Give an example.
2. What is meant by implementation defined? Give a specific example. What is meant by implementation dependent? Give a specific example.
3. What is the difference between a high level programming language and a low level language? Give an example of each and state which category Modula-2 falls into.
4. What is the difference between a high level language construct and a low level language construct? Give an example of each from Modula-2.
5. Represent the following decimal numerals in binary, octal, and hexadecimal: a) 123 b) 255 c) 1024 d) 4096 e) 1500 f) 4179
6. Represent the following binary numerals in decimal, hexadecimal, and octal: a) 10110101 b) 01110111 c) 11001011 d) 11110000 e) 10101010 f) 01101101
7. Represent the following hex or octal numerals in binary and decimal: a) 0A081H b) 1734B c) 0C000H d) 1000H e) 0300H f) 7617B g) 0D000H h) 0F800H
8. Represent the following decimal numerals in binary and hexadecimal: a) 10 b) 100 c) 1000 d) 10000 e) 256 f) 1024 g) 65535 h) 5234 i) 10253
9. Define the following terms: bit, byte, nibble, word, "K"
10. What is the length of a word in your system? Make a table of the storage length in words and bytes of all the standard data types in your system.
11. Define the following terms for your system: page, sector, track, block.
12. What is a data location, and how is it abstracted in Modula-2?
13. What is an address? Name some procedures in ISO standard Modula-2 that can be used to manipulate addresses.
14. What procedure is used to find the address of a Modula-2 item?
15. Show how to declare a `CARDINAL` variable to have the fixed address B7ED in Modula-2.
16. What immediately marks a Modula-2 program as system dependent or non-portable?
17. What is the difference between safe and unsafe type transfers, and how is each realized in Modula-2?
18. How is each of: a) octal character numbers b) octal numeric literals and c) hexadecimal numeric literals indicated in Modula-2? Give examples.
19. What is a file?
20. Distinguish among the terms logical file, program file, and physical file.
21. Distinguish between the terms random access file and sequential file.
22. Name the high level modules for file I/O in ISO standard Modula-2.
23. Name the device drivers in ISO standard Modula-2 that were discussed in this chapter. Are these high, medium, or low level modules?
24. What modules are used in the ISO suite to do input and output of binary data? Are these high,

medium, or low level modules?

25. Distinguish between the terms *sequence* and *stream*.

26. What are the properties of a stream?

27. Why is the sequence 1, 1.1, 2, 2.2, 3, 3.3, 4, 4.4, ... not a stream?

28. What is the difference between a text stream and a raw stream?

29. What is a channel?

30. Distinguish between the terms *sequential file* and *text file*.

31. Detail ALL the steps that are required (implicitly or explicitly) to do input/output with a physical file.

Which ones are explicitly required on an ISO system? Which ones are explicitly required on the system you use?

32. Distinguish between the terms restricted stream and rewindable sequential stream. What ISO Modula-2 libraries implement each model?

33. Why is it important that any application program that opens files also closes them?

34. What is a buffer?

35. Detail any implementation restrictions/dependencies on your system that relate to the specification and use of file names.

36. What is the difference between the *StdChan* procedures *InChan* and *OutChan* on the one hand, and *StdInChan* and *StdOutChan* on the other.

37. Look up in the documentation and explain a possible use for the ISO channel identifiers *StdChan.NullChan*, and *IOChan.InvalidChan*.

38. What ISO module is used for device independent I/O?

39. Define the term echo.

40. In *TermFile* driven channels, what is the difference between character mode input and line mode input, and how is each set?

41. How many *TermFile* channels have to be in character mode in postpone all echoing?

42. What is a generic procedure?

Problems

43. Write a program module that will change a user supplied CARDINAL into a binary number and print out the result. Do not use recursion as was done in the example in chapter 4.

44. Write a program module that will write out a user-supplied REAL byte by byte in hexadecimal format.

45. Write a program module that will write out a user-supplied REAL byte by byte in binary format.

46. Write a module to determine the size (in LOCs) of all the built in data types in Modula-2.

47. Write a module that will determine what happens when you try to do output to the channel returned by *StdChans.InChan*.

48. Write a module that will determine what happens when you try to do input or output to the channel returned by *IOChan.InvalidChan*.

49. Write a module that will allow you to type a paragraph or two of text from the keyboard and store it on the disk as a text file, with each character being stored as it is read from the keyboard.

50. Write a module that will allow you to type a paragraph or two of text from the keyboard and store it on the disk as a text file, with each line being stored as it is read from the keyboard.

51. Now rewrite the module in Chapter 7, Problem #16 to read the source text from the disk file created in #50 above, reformat it according to the rules followed in the Chapter 7 problem, and both output it to the printer and also re-write it to another text file under a different name.
52. Write a module that can read a text file from the disk and analyze the frequency of the characters in the file, and then save a chart of this information to another file.
53. Write a module that can read a text file from the disk and analyze the frequency of the words in the file, and then save a chart of this information to another file.
54. Write a program module that will collect class grading information from the keyboard, and write it out in a series of lines in a text file following the format below, where the first item is a student name, the second is a lab mark out of 25, the third and fourth are midterms out of 40, and the last a final exam mark out of 100.
- John Jacobs, 23.5, 32.7, 39.3, 74.5
- Chriss Waltermore, 13.5, 22.7, 40, 64.5
- and so on. Commas are to separate the items on each line, and end of line markers the student line items.
55. Write a program module that will read the data in the file created in #54 and print out a nicely formatted report, together with a final percentage based on weighing the lab mark as 20%, the midterms together as 35% and the final as 45%. This program should also calculate and report on class averages for each marking category and for the overall class average.
56. Write a program module that will read the data in the file created in #54 and ask the user for a student number for each student, then write a new file one above, but with the student number following the name.
57. Write a program module that will read a file, ask the user for a password, and then encrypt the file using the password and then write it to a new file.
58. Write the complement to the program in #57 to decrypt the file using the password. Compare to the original.
59. Use a previously created file of prime numbers (or create one if you have not already) to solve the problem of finding the prime factorization of cardinals typed from the keyboard. For instance an input of 150 should produce an output of: 150 has 1 factor of 2, 1 factor of 3, and 2 factors of 5. Of course, the program should be able to do any number of factorings before being exited, so the file of primes will need to be rewound to the beginning for each.
60. Use a previously created file of random numbers (or create one if you have not already) to determine the fastest way of reading and writing an array of cardinals. Should it be done one number at a time using *ReadCard*, or *ReadRaw*, (*WriteCard/WriteRaw*) or in some larger data chunk?
61. Suppose that you do not have the procedures *ReadCard* and *ReadInt* available. Write both using the facilities of *IOChan*. Test what you have written. Note that you will also have to look up and employ the facilities of appropriate conversion routines in other library modules.
62. Consider the version of [RedirStdIO](#) presented in [section 8.9](#). Rewrite this for a text based system so that a simple prompt will be presented to the user in text form, say, for *OpenInput*:
- Input File >
- The user responds by typing a file name. If the file is not found, the prompt is repeated. If the user types a control-C character (or another suitable one of your choice) as the beginning of the string, the dialog is escaped and the situation shall be as if no *OpenInput* had been encountered in the code.
63. Combine the facilities of the program modules *GetNStash*, *ReadNAdd*, *StashMoreAndType* from

[section 8.6](#) into a single menu-driven module that allows the user to create a file for the numbers, put more numbers into the file, read them and determine maximum, minimum, sum, and average at any point, clear an existing file, or and close the open file, exiting the program. Some flag will have to mark the end of a series of inputs for a return to the main menu.

Ambitious Projects

64. Implement and test *StreamFile* on top of whatever lower level file handling modules may be available. Warning: You must first fully understand the use of *IOLink* for writing device drivers.
65. Implement and test a module that opens a logging channel to a disk file and logs all input and output going through the standard channels so that it is echoed in the file as well as on the screen. Test your code.

[Contents](#)

Chapter 8

Data Storage Issues

[8.0 Chapter Goals](#)

[8.1 Storage--An Introduction](#)

Part A--Machine and System Level Storage Issues

[8.2 An Introduction to the Lower Level](#)

[8.2.1 General Considerations](#)

[8.2.2 Low Level Numeric Notations](#)

[8.2.3 Machine Level Data Storage](#)

[8.2.4 Hexadecimal and Octal Notation](#)

[8.3 High Level Access to Low Level Facilities in Modula-2](#)

[8.3.1 The Module SYSTEM](#)

[8.3.2 Variables at Fixed Addresses](#)

[8.3.3 Hexadecimal and Octal Notation in Modula-2](#)

[8.4 Extended Low Level Examples](#)

[8.4.1 Keyboard Reading--Operating System Level](#)

[8.4.2 Generic Swap](#)

Part B--Input, Output, and Files

[8.5 Files--Introduction and Terminology](#)

[8.5.1 Sequences, Streams, and Channels](#)

[8.5.2 Sequential and Random Access Files](#)

[8.5.3 Planning to Use Streams with Files](#)

[8.6 Text I/O in ISO Standard Modula-2](#)

[8.6.1 The Restricted Stream Model](#)

[8.6.2 The Rewindable Sequential Stream Model](#)

[8.6.3 File Text I/O in non-Standard Modula-2](#)

[8.7 Binary I/O](#)

[8.8 Notes on File I/O](#)

[8.8.1 Special Notes on the Macintosh Operating System](#)

[8.9 Standard Channel I/O in ISO Modula-2](#)

[8.10 Lower Level I/O in ISO Modula-2](#)

[8.11 An Extended Low Level I/O Example--TermFile](#)

[8.12 Chapter Summary](#)

[8.13 Assignments](#)

Contents

9.0 Chapter Goals

The purpose of this chapter is to explore in some detail the concept of *structured* data. The array type was considered in earlier chapters; here the set and record types are introduced. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

structured and unstructured data, sets, set elements, records, and record fields

Realized in the Modula-2 notation:

the Modula-2 set and record types

random Access Files

Data Manipulation Abstractions

General:

set operations

Realized in the Modula-2 notation:

set operations, and the manipulation of record fields

reading and writing random access files

Programming Abstractions

General:

unqualifying record field identifiers

Realized in the Modula-2 notation:

the WITH statement

9.1 Structured Data Revisited

Recall the discussion of section 1.6.3 on the subject of whether data was of a kind that could be decomposed into simpler parts.

Data items that are normally handled as indivisible whole items are called atomic or unstructured.

Examples of Modula-2 types that model unstructured data include INTEGER, CARDINAL, CHAR, BOOLEAN, all enumerated types, all subranges of any of these, and the types REAL and LONGREAL. While it would be possible to decompose, say, a real number into significant figures on the one hand and an exponent on the other, or even into individual digits, neither of these decompositions would serve much useful purpose in most problems. It is more practical to think of items of these types as entire entities, and to manipulate them as such. To this point in the text, almost all data has been atomic in nature. In real life, on the other hand, things may be much more complex.

A data type is called structured if its elements are not atomic, but have component parts.

There are three common types of structured data: arrays, sets, and records.

An array is a collection of data of the same type, organized in an ordered fashion by an indexing scheme. For instance, in an array A of three real numbers numbered a1, a2, and a3, the structured entity is the entire array A, and the individual real numbers are its atomic components. For more details on arrays and their representation in the Modula-2 notation, see chapter 5.

A set is a structured, but unorganized collection of data without any indexing scheme at all. In the set {1, 2, 3, 4, 5} the structured entity is the entire set, and the individual items in the set are the atoms. An item is either an element of a given set or it is not. The first part of this chapter will examine sets in detail, and show how they are represented and manipulated in Modula-2.

Next, consider the problem of a college that must keep student and personnel records, or that of a business that must store its customer accounts data. In both cases, a real-life person could be abstracted or modeled for computational purposes by a *record* consisting of a name, an address, phone number, student or account number, marks or balance owing, and so on.

One needs to be able to collect this information into an identifiable personal record, but cannot use an array for the enclosing structure because the component data is of different types--some parts are numbers, some are strings, and some such as marital status or sex could perhaps be represented as booleans.

At one time, one might have done this by making a small file card for each person or account, and kept the whole lot in shoe boxes on the back shelf of an office. Each card contains a structured representation of one personal record. The atoms of a record are the values on each line of the card. The second part of this chapter will examine the record type in more detail, and show how to represent and manipulate them

Contents

Part A--Sets

9.2 Representation and Membership

Abstractly, a set is a collection of items of any type. Thus, it makes sense to write:

$A = \{1, 2, 3.525, \text{Monday}, \text{true}\}$

Not many computer languages have sets as a possible data type. Those that do usually restrict the abstract concept so as to require that all the elements of a given set be of the same basic type. Modula-2 inherits much of the syntax and semantics of its set type from Pascal, and does have this restriction, along with some others.

A Modula-2 set is a collection of items of the same scalar non-real type without regard to order.

When a Modula-2 set type is declared, the declaration includes the specification of what elements set variables of the new type may contain.

Examples:

TYPE

```
WeekDaySets = SET OF [Monday .. Friday];  
DigitSet = SET OF [0 .. 9];
```

WARNING: Many older versions of Modula-2 follow the suggestion (not requirement) Wirth made in presenting Modula-2 that the maximum size of a set may be severely limited (usually to 16 or 32 elements.) Some, moreover limit set declarations to cardinal subranges starting at zero. Following this suggestion makes impossible such declarations as:

TYPE

```
CharSet : SET OF CHAR;  
HunSet : SET OF [100 .. 110];
```

in the former case, because there would be too many potential elements, and in the latter because the possible CARDINAL values used in this fashion are not in the range the range $[0 .. \text{MaxSetSize} - 1]$. Such implementations sometimes provide a separate module *LongSets* (or something similar) with procedures to perform the appropriate operations on large sets.

The restriction above effectively eliminates most of the practical applications for sets, and versions of Modula-2 that followed this suggestion were regarded by most programmers as badly crippled. For this reason, the ISO standard for the language has a minimum requirement that whatever limitation an implementation may place on set size, SET OF CHAR must at the very least be permitted in a declaration.

For abstract sets, membership of an element in a set is indicated by the symbol " \in " so that the phrase "a is an element of A" is written " $a \in A$ " and the phrase "b is not an element of A" is written " $b \notin A$."

Modula-2 set membership is indicated with the reserved word IN.

In Modula-2, therefore, the two phrases in the paragraph above are written:

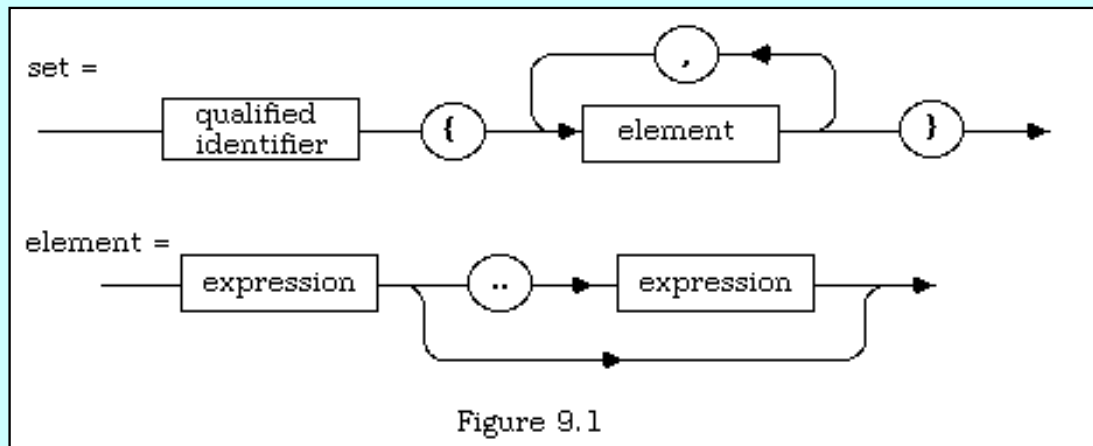
a **IN** A

and

NOT (b **IN** A)

respectively.

One other difference between standard mathematical notation and that followed by Modula-2 is that when a literal set is written out, the type of the set must appear before the opening brace. This is indicated in the syntax diagram for a set, given in figure 9.1.



The following short module illustrates some of these ideas. It counts the vowels in a line of typing supplied by the user.

```
MODULE Aeiou  'Ñ  $Úritten by R.J. Sutcliffe *)
(*  using ISO standard Modula-2  *)
(* 1  'Ñ  $Úsion 1994 02 23  *)

FROM STextIO IMPORT
  ReadChar, WriteString, WriteLn, SkipLine;
FROM SWholeIO IMPORT
  WriteCard;
FROM SIOResult IMPORT
  ReadResult, ReadResults;

TYPE
  CharSet = SET OF CHAR;

VAR
  ch : CHAR;
  'Ñ  $ÚCARDINAL;
  vowelSet : CharSet;

BEGIN
  vowelSet := C  'Ñ  $Ú'A', 'E', 'I', 'O', 'U'; (* construct the set *)
  WriteString ("Type in a line and I'll count vowels. ");
  WriteLn;
```

```

count := 0;      (* initialize *)

REPEAT
  ReadChar (ch); (* examine each character as typed *)
  IF (ReadResult () = allRight) AND (CAP (ch) IN vowelSet)
    THEN
      INC (count)
    END;      (* if *)
UNTIL ReadResult () # allRight;
SkipLine;

WriteLn;
WriteString ("The number of vowels in the line was ");
WriteCard (count, 5);
WriteLn;
WriteString ("Type a character to continue");
ReadChar (ch);
SkipLine;
END Aeiou.

```

The following is a picture of the screen from a run of this program:

```

Type in a line and I'll count vowels.
Now is the time for all good women to come to the aid of the party.

The number of vowels in the line was    21
Type a character to continue

```

- NOTES:** 1. The set type must precede the braces that enclose a set constant. Some exceptions to this rule were allowed in earlier versions of Modula-2, but not in the ISO standard.
2. Subranges are allowed in sets. Assignments involving subranges are written in the following way:

```

TYPE
  CharSet = SET OF CHAR;
  DigitType = SET OF [0 .. 9];
VAR
  charSet1 : CharSet;
  digits : DigitType;
BEGIN
  charSet1 := CharSet {'A' .. 'Z'};
  digits := DigitType {0 .. 2};
  (* A range used in this context does not need brackets. *)

```

It is important to keep in mind that while the ranges can be used to assist in constructing the set, there is no particular order to these, or any other set elements.

- For keyboards that cannot type the brace characters, use "(:" and "(:)" for "{" and "}" instead.
- In older versions of Modula-2, if the set type was not given, then it was by default a SET OF [0 .. 15]. (Or, it could be 0 .. 31.) The upper limit depended on the machine).
- Some versions of Modula-2 also have other restrictions on set membership.

The Modula-2 notation Type {list} is said to be a constructor.

Set Operations

Before working through any more examples using sets, first consider the various operations that are possible using abstract set notation, and then how those ideas are translated into Modula-2. The abstract concepts presented here review standard mathematical ideas.

9.2.1 Set Union

The union of two sets is the new set that contains all the elements present in either of the two original sets. If "A" and "B" are sets, then "A union B" is written $A \cup B$.

The Modula-2 symbol for set union is +.

Examples:

Suppose that in the rest of this section,

TYPE

```
SmallSets = SET OF [0 .. 9];  
CharSet = SET OF CHAR;
```

VAR

```
A, B : SmallSets;  
C : CharSet;
```

Then,

```
A := SmallSets{1, 2, 5} + SmallSets{3, 7};
```

leaves A holding {1, 2, 3, 5, 7}

```
B := SmallSets{3, 7, 4} + SmallSets{4, 3, 2, 1};
```

leaves B holding {1, 2, 3, 4, 7}

The elements 3 and 4 are not repeated. The only thing that matters is whether an entity is in the set. It cannot be a member more than once, nor is the order in which the elements are written important.

9.2.2 Set Intersection

The intersection of two sets is the new set that contains only those elements common to the original pair of sets. If "A" and "B" are sets, then "A intersect B" is written $A \cap B$.

*The Modula-2 symbol for set intersection is *.*

Examples:

```
A := SmallSets{1, 3, 7} * SmallSets{1, 3, 9, 16};
```

leaves A holding {1, 3}

```
B := SmallSets{2, 4, 6, 7} * SmallSets{1, 9, 10};
```

leaves B holding { }

This latter set is called the *empty set*. It has no elements. In mathematics, the preferred notation for the empty set is the symbol \emptyset , but this is not a standard Modula-2 symbol.

9.2.3 Set Difference

The set difference $A - B$ of two sets of the same type is the new set consisting of all elements of A that are not in B.

The Modula-2 symbol for set difference is also -.

Examples:

```
A := SmallSets{1, 2, 3, 7, 9} - SmallSets{2, 7};
```

leaves A holding {1, 3, 9}

```
B := SmallSets{2, 4, 6, 7} - SmallSets{1, 3, 4, 9};
```

leaves B holding {2, 6, 7}

As the last example illustrates, any superfluous elements in the second set that cannot be removed from the first one are simply ignored.

9.2.4 Symmetric Set Difference

The symmetric set difference of two sets is defined as the new set whose elements are in either of the original sets, but not in both. If A and B are sets, this can be written as indicated as $A \cup B - A \cap B$.

The Modula-2 symbol for symmetric set difference is /.

Examples:

```
C := CharSet {'a', 'b', 'c'} / CharSet {'c', 'd'};
```

leaves C holding {'a', 'b', 'd'}

```
B := SmallSets{1, 4, 6, 7} / SmallSets{2, 4, 5, 7, 9};
```

leaves B holding {1, 2, 5, 6, 9}

The symmetric set difference A / B is also called an *exclusive or* operation (sometimes written XOR).

As also illustrated by these examples, expressions involving the various set operators are also allowed. The normal arithmetic rules of precedence for the symbols +, -, *, and / are followed, so if this is not the desired meaning of a set expression, the programmer must use parentheses to change the order.

One common syntax error in writing out set expressions is forgetting to place the braces around a set, particularly if it

has only one element. Also, one must not write, say

```
A := DaySet {Mon, Tue, Wed} + Thur
```

when what is meant is

```
A := DaySet {Mon, Tue, Wed} + DaySet {Thur}.
```

9.2.5 One Element at a Time

Modula-2 also provides two built-in procedures allowing single elements to be quickly inserted into, or removed from, a set. Of course, when inserting, it is important that the proposed new element actually be of the underlying set type; what one is essentially doing is a union with a one-element subset.

INCL (theSet, element) produces the same result as $theSet := theSet + setType\{element\}$ and EXCL (theSet, element) produces the same result as $theSet := theSet - setType\{element\}$ where theSet is a set and element is expression compatible with the base type of theSet.

Example:

TYPE

```
DigitSets = SET OF [0 .. 9];
```

VAR

```
digits : DigitSets;  
number : CARDINAL;
```

CONST

```
zilch = DigitSets {1-1};
```

BEGIN

```
number := 7;  
digits := DigitSets {0 .. 3}; (* Now, digits holds {0,1,2,3} *)  
INCL (digits, 5); (* Now, digits holds {0,1,2,3,5} *)  
EXCL (digits, number - 4); (* Now, digits holds {0,1,2,5} *)  
digits := digits - zilch; (* Now, digits holds {1,2,5} *)
```

The CONST declaration illustrates that set constants can be created and that the declaration can contain expressions, as with other CONST declarations. This ability may not be particularly useful in an example like this one, but it is present.

NOTE: Here, set identifiers begin with an upper case letter and set elements with lower case letters, in accordance with standard mathematical practice, but somewhat contrary to the usual conventions in this book.

Example:

Write a procedure to check an input string to see if it is a legal Modula-2 identifier.

Refinement:

Check the first character to see if it is a letter.
Check each subsequent character to see if it is a letter or a number.

Pseudocode:

```
Set a count to zero.
Set lastCharNum to the length of the string.
If the length is zero or stringcount is not a letter then return false.
While the count is less than or equal to lastCharNum
    If stringcount is a letter or a digit then
        increment count
    else return false.
return true
```

Code:

```
MODULE TestForIdentifier;

(* Written by R.J. Sutcliffe *)
(* using ISO Modula-2 *)
(* last revision 1994 02 23 *)

FROM STextIO IMPORT
    ReadString, WriteString, WriteLn, ReadChar, SkipLine;

TYPE
    String  = ARRAY [0 .. 255] OF CHAR; (* local string type *)
    CharSet = SET OF CHAR;

VAR
    theIdent, replyStr : String;
    caps, lcase, letters, digits, okChars, yesChars, noChars, validChars : CharSet;
    validReply, again : BOOLEAN;

PROCEDURE LegalIdent (str : ARRAY OF CHAR) : BOOLEAN;

VAR
    count, strLen : CARDINAL;

BEGIN
    strLen := LENGTH (str);
    count := 0;
    (* check for zero length & ensure letter for first character *)
    IF (strLen = 0) OR NOT (str [count] IN letters)
    THEN
        RETURN FALSE;
    END;
    INC (count);
    (* now check subsequent characters *)
    WHILE count < strLen
```

```

    (* when count = length we're past last character *)
DO
    IF str [count] IN okChars
    THEN
        INC (count)
    ELSE
        RETURN FALSE
    END;
END; (* while *)
RETURN TRUE; (* string was ok if code got to here *)

END LegalIdent;

BEGIN (* main *)
    (* sets for identifiers *)
    caps := CharSet {"A" .. "Z"};
    lcase := CharSet {"a" .. "z"};
    letters := caps + lcase + CharSet {"_"};
    (* remove last item if not ISO standard *)
    digits := CharSet {"0" .. "9"};
    okChars := digits + letters;

    (* sets for keyboard answers *)
    yesChars := CharSet {"Y", "y"};
    noChars := CharSet {"N", "n"};
    validChars := yesChars + noChars;

    WriteString ("Test harness for procedure to check for legal identifiers.");
    WriteLn;
    WriteString ("by R. Sutcliffe");
    WriteLn;

    REPEAT (* main loop for testing several *)
        WriteLn;
        WriteString ("Type the identifier here ==");
        ReadString (theIdent);
        SkipLine;
        WriteString ("The string '");
        WriteString (theIdent);
        WriteString ("' is");
        IF NOT LegalIdent (theIdent)
        THEN
            WriteString (" not");
        END;
        WriteString (" a legal Modula-2 identifier.");
        WriteLn;

        REPEAT (* see if another *)
            WriteString ("Do you wish to do another? (Y/N) ==");
            ReadString (replyStr);
            SkipLine;

```

```

validReply := replyStr [0] IN validChars;
IF NOT validReply
    THEN
        WriteString ("That was not a valid reply.  Please try again.");
        WriteLn;
    END;
UNTIL validReply;
again := replyStr [0] IN yesChars;
UNTIL NOT again;

END TestForIdentifier.

```

Sample Run: (Picture of the screen)

```

Test harness for procedure to check for legal identifiers.
by R. Sutcliffe

Type the identifier here ==> oneTwoThree
The string 'oneTwoThree' is a legal Modula-2 identifier.
Do you wish to do another? (Y/N) ==> Y

Type the identifier here ==> %ages
The string '%ages' is not a legal Modula-2 identifier.
Do you wish to do another? (Y/N) ==> y

Type the identifier here ==> Welcome2Modula2
The string 'Welcome2Modula2' is a legal Modula-2 identifier.
Do you wish to do another? (Y/N) ==> y

Type the identifier here ==> The_Body
The string 'The_Body' is a legal Modula-2 identifier.
Do you wish to do another? (Y/N) ==> Spaces In Here
That was not a valid reply.  Please try again.
Do you wish to do another? (Y/N) ==> Y

Type the identifier here ==> Spaces In Here
The string 'Spaces In Here' is not a legal Modula-2 identifier.
Do you wish to do another? (Y/N) ==> n

```

[Contents](#)

9.3 Set Comparisons

9.3.1 Set Equality and Inequality

Two sets are equal when their list of elements is identical. In mathematics, one simply writes $A = B$ to express equality, and $A \neq B$ to express set inequality. Both may also be used in the sense of Boolean expressions.

The Modula-2 notation for the boolean expression "A is equal to B" is written $A = B$, and the Modula-2 notation for the boolean expression "A is not equal to B" is written $A \neq B$ and " \neq " (or $\#$) symbols have been used for equality/inequality tests of both individual numerical equality and for equality of sets. The symbol " $=$ " used to compare two sets has a different meaning than the symbol " $=$ " used to compare two numbers. The compiler must sort out which meaning is intended by the programmer according to the correct context in which the symbol is used.

Whenever a symbol or operator is used with two or more different meanings in different contexts, it is said to be overloaded.

Sometimes overloading is less obvious. Strictly speaking, the "+" operator is overloaded just because it can be used with all three of the cardinal, integer, and real types, even though the mathematical meaning of the symbol is essentially the same for all three. However, the overloading of operators like "+" is more obvious when it is also used to represent set union. As the following sections illustrate, the " $=$ " and " \neq " are not the only relational operators to be overloaded.

9.3.2 Subset

If all the elements of a set A are contained in a set B one says that A is a subset of B. The mathematical notation is $A \subseteq B$, and the relationship is illustrated in figure 9.2.

The Modula-2 notation for the boolean expression "A is a subset of B" is written $A \subseteq B$.

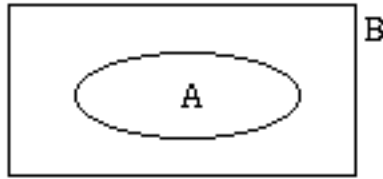


Figure 9.2

For example, $\{1,2,4\} \subseteq \{1,2,4,7\}$ and $\{a,b\} \subseteq \{a,b,c,d\}$, or, to put it in Modula-2 notation, if result is a boolean variable, then

```
result := DigitSet{1,2,4} <= DigitSet{1,2,4,7};
```

or

```
result := DigitSet{1,2,4} <= DigitSet{1,2,4};
```

leaves result holding TRUE, and

```
result := DigitSet{1,2,4,5} <= DigitSet{1,2,4,7};
```

or

```
result := NOT (Charset{"a","b"} <= Charset{"a","b","c","d"});
```

leaves result holding FALSE.

If, in addition to A being a subset of B, B and A are known not to be equal, then A is a proper subset of B. The mathematical notation is $A \subset B$. Although one might expect the Modula-2 notation for the boolean expression "A is a proper subset of B" to be written $A < B$, this notation is not in fact used, and it is necessary to employ $(A < B)$ AND $(A \neq B)$ for proper subset.

9.3.3 Superset

Sometimes the set containment symbols are written the other way around. One might have occasion to write $B \supseteq A$ rather than $A \subseteq B$, for instance. Rather than read this from right to left, it is quite in order to say "B is a superset of A."

The Modula-2 notation for the boolean expression "A is a superset of B" is written $A \supseteq B$ rather than $A \subset B$. This could be read "B is a proper superset of A." There is no single Modula-2 notation for proper superset; it must be written as a compound Boolean expression as above.

9.4 Sets and the I/O Library

It was remarked in passing that the I/O library flags *read*, *write*, *raw*, *text*, *old*, and *echo* are all in fact sets. It is instructive to see how this is done. Part of the module *ChanConsts* first defines the enumerated type *ChanFlags* and the set type *FlagSet* as follows:

TYPE

```
ChanFlags =      (* Request flags possibly given when a channel is opened *)
( readFlag,      (* input operations are requested/available *)
  writeFlag,      (* output operations are requested/available
  oldFlag,        (* a file may/must/did exist before the channel is opened *)
  textFlag,       (* text operations are requested/available *)
  rawFlag,        (* raw operations are requested/available *)
  interactiveFlag;(* interactive use is requested/applies *)
  echoFlag        (* echoing by interactive device on removal of characters from
input stream requested/applies *)
);

FlagSet = SET OF ChanFlags;
```

Observe the appropriate prettyprinting when the meaning of each value in the enumerated type is documented. Because values of type *FlagSet* are to be passed to a variety of I/O operations, and because each flag often has to be referred to as a singleton, *ChanConsts* then defines the following convenience singleton sets:

CONST

```
read = FlagSet{readFlag};
write = FlagSet{writeFlag};
old = FlagSet{oldFlag};
text = FlagSet{textFlag};
raw = FlagSet{rawFlag};
interactive = FlagSet{interactiveFlag};
echo = FlagSet{echoFlag};
```

Of these, the only one not already used in the last chapter is *interactive*. It is available for vendors to establish device drivers employing the concept of an interactive terminal in some implementation defined fashion, but is not actually employed by any device driver in the standard library itself.

For further convenience, device driver definition modules such as *StreamFile* import the constants and then redefine them as their own:

```
DEFINITION MODULE StreamFile;
```

```
IMPORT IOChan, ChanConsts;
```

TYPE

```
ChanId = IOChan.ChanId;
FlagSet = ChanConsts.FlagSet;
```

```
OpenResults = ChanConsts.OpenResults;

(* Accepted singleton values of FlagSet *)
```

```
CONST
  read = FlagSet{ChanConsts.readFlag};
  (* input operations are requested/available *)
  write = FlagSet{ChanConsts.writeFlag};
  (* output operations are requested/available *)
  old = FlagSet{ChanConsts.oldFlag};
  (* a file may/must/did exist before the channel is opened *)
  text = FlagSet{ChanConsts.textFlag};
  (* text operations are requested/available *)
  raw = FlagSet{ChanConsts.rawFlag};
  (* raw operations are requested/available *)
```

In this manner, only the flags that are useful for that particular device driver are given a specific meaning by for it and are made available by it. When *StreamFile* then defines items using the type *FlagSet* (that is, its own version of the type), such as:

```
PROCEDURE Open (VAR cid: ChanId;
                 name: ARRAY OF CHAR;
                 flags: FlagSet; VAR res: OpenResults);
```

it becomes possible to call these in the convenient fashion already used in the last chapter, and employing the set union operator to indicate more than one flag:

```
Open (theChan, theName, read+old, res);
```

Observe that *StreamFile*, in common with other device drivers, also imports and redefines as its own the type *ChanId*. Thus, a person writing software employing such a device driver need not import directly from the lower level where reside *ChanConsts* and *IOChan*.

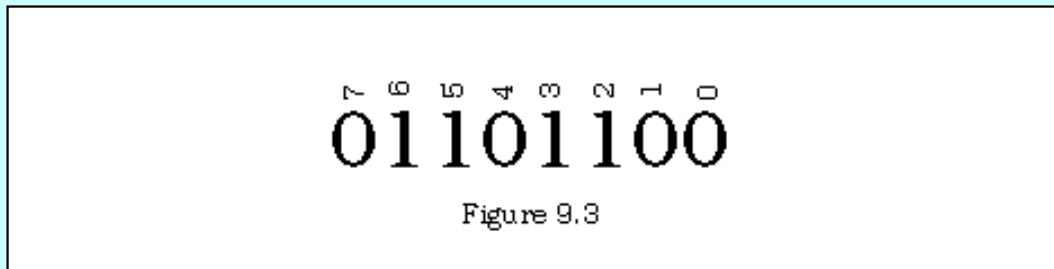
[Contents](#)

9.5 Sets at the Low Level

There are some low-level sets built in that allow bit-by-bit manipulation of memory storage. Of course, to make any use of these, something must be known about the bit and byte level numeric representations on the underlying machine.

9.5.1 Bitsets

A number represented in binary form could be thought of as a set of bit positions enumerated from least significant (lowest number) to most significant (highest number) as shown in figure 9.3 for an eight bit number..



In this case, if the number is thought of as a SET OF [0 .. 7] the elements 6, 5, 3, and 2 are in the set, and the elements 7, 4, 1, and 0 are not in the set. This idea of interpreting storage as a set of bit positions is encapsulated in the following ADT:

The Modula-2 type BITSET has the implicit definition

TYPE

BITSET = SET OF [0 .. BitsPerBitset-1]

where the constant BitsPerBitset is implementation defined and the setting of each bit position determines whether the number of that position is an element of the set.

The constant *BitsPerBitset* is usually the same as the number of bits in a WORD, (or some integral multiple of this); often it is 16 or 32 and can be computed as:

```
BitsPerBitset := SIZE (BITSET) * SYSTEM.BITSPERLOC
```

If, for instance, as is often the case, the types CARDINAL, WORD, and BITSET all have the same size (in LOCs) the following program provides an alternative to the pair of mutually recursive procedures in Chapter 4 to print out a CARDINAL in binary form. It forcibly reinterprets a CARDINAL as a BITSET, and then prints out a one or a zero depending on whether that bit is represented in the set or not.

```
MODULE BitsetDemo;
```

```
(* Written by R.J. Sutcliffe *)
(* to demonstrate the use of bitset *)
(* using ISO standard Modula-2 *)
(* last revision 1994 03 01 *)
```

```

FROM STextIO IMPORT
    WriteString, WriteLn, SkipLine, ReadChar, WriteChar;
FROM SWholeIO IMPORT
    WriteCard, ReadCard;
FROM SYSTEM IMPORT
    CAST, WORD, BITSPERLOC;

CONST
    maxBitNum = BITSPERLOC * SIZE(BITSET) - 1;

PROCEDURE WriteWordBin (b: WORD);
(* Pre : WORD and BITSET have the same size *)

VAR
    bs : BITSET;
    count : CARDINAL;

BEGIN
    bs := CAST (BITSET, b);
    FOR count := maxBitNum TO 0 BY -1
        DO
            IF count IN bs
                THEN
                    WriteCard (1,1)
                ELSE
                    WriteCard (0,1)
                END;
            IF count MOD 8 = 0 (* break into groups of 8 bits *)
                THEN
                    WriteChar (" ");
                END;
            END;
    END WriteWordBin;

VAR
    theNumber : CARDINAL;
    answer : CHAR;
    again : BOOLEAN;

BEGIN
    WriteString ("This program tests a procedure to print ");
    WriteString ("cardinals in binary form");
    WriteLn;

REPEAT
    WriteString ("Enter the number to be changed ");
    WriteString ("to binary form ==");
    ReadCard (theNumber);
    SkipLine;
    WriteLn;
    WriteString ("The cardinal ");
    WriteCard (theNumber, 0);
    WriteString (" converted to binary form is: ");

```

```

WriteLn;

WriteWordBin (theNumber); (* one must know sizes are equal *)

WriteLn;
WriteString ("Do another? Y/N ");
ReadChar (answer);
SkipLine;
again := (CAP (answer) = "Y");
UNTIL NOT again;

END BitsetDemo.

```

Observe that even the bit positioning from left (most significant) to right (least significant) is implementation defined; it is not necessarily the case in the underlying hardware of all implementations. However, the numbering from *zero* (least significant) to *bitsperbitset* (most significant) is defined to be the way that ISO standard Modula-2 must produce the results to the program, regardless of the hardware. Here is a sample run from this program module:

```

This program tests a procedure to print cardinals in binary form
Enter the number to be changed to binary form ==> 10

```

```

The cardinal  10 converted to binary form is:
00000000 00000000 00000000 00001010
Do another? Y/N y
Enter the number to be changed to binary form ==> 1000000
The cardinal  1000000 converted to binary form is:
00000000 00001111 01000010 01000000
Do another? Y/N n

```

9.5.2 Packed Sets

For the purpose of working with other types at the bit level, and provided their representation details are known, it is also possible to declare a more general kind of bitset, this time with a user-specified number of bits, rather than the fixed number allowed by the type BITSET. In addition, the underlying type in the more general case need not be CARDINAL [0..n], but may be any ordinal type (though CARDINAL is the most useful). Of course, if these sets are to be mapped to storage locations, the number of bit positions is most useful as a multiple of BITS PER LOC. Thus, in the above example, one could have declared:

TYPE

```
CardSet = PACKEDSET OF [0..SIZE (CARDINAL) * BITS PER LOC - 1]
```

and then cast to an item of this type. Indeed, the type BITSET is just a specific built in subtype of the more general type PACKEDSET. By this means, the binary representation of other types, such as REALs could also be investigated. In addition, there are some operations in SYSTEM that are intended to manipulate PACKEDSETS (including BITSETS) on the bit level. These include:

```
PROCEDURE SHIFT (val: <packset type><same packset type><packset type><same packset type>
```

This capability is most useful when programming at the low system level where individual bits must be manipulated. Note that the effect of a right shift is to divide by 2. This is illustrated in the example below. Note that it does not make use of any special knowledge about the sizes of the types in bits, but computes them

from information available from SYSTEM.

MODULE ShiftDemo;

(* Written by R.J. Sutcliffe *)
(* to demonstrate the use of packedset *)
(* using ISO standard Modula-2 *)
(* last revision 1994 03 02 *)

FROM STextIO **IMPORT**

WriteString, WriteLn, SkipLine, ReadChar, WriteChar;

FROM SWholeIO **IMPORT**

WriteCard, ReadCard;

FROM SYSTEM **IMPORT**

BITSPERLOC, SHIFT, ROTATE, CAST;

CONST

maxBitNum = BITSPERLOC * **SIZE**(**CARDINAL**) - 1;

TYPE

CardSet = **PACKEDSET OF** [0..maxBitNum];

PROCEDURE WriteCardBin (num: **CARDINAL**);

VAR

count : **CARDINAL**;

numSet : CardSet;

BEGIN

numSet := **CAST** (CardSet, num);

FOR count := maxBitNum **TO** 0 **BY** -1

DO

IF count **IN** numSet

THEN

WriteCard (1,1)

ELSE

WriteCard (0,1)

END;

IF count **MOD** 8 = 0 (* break into groups of 8 bits *)

THEN

WriteChar (" ");

END;

END;

END WriteCardBin;

VAR

num : **CARDINAL**;

answer : **CHAR**;

again : **BOOLEAN**;

BEGIN

WriteString ("This program illustrates bit shifting ");


```
WriteLn;
```

REPEAT

```
WriteString ("Enter the number to be shifted ");
ReadCard (num);
SkipLine;
WriteLn;
WriteString ("The cardinal ");
WriteCard (num, 0);
WriteString (" binary: ");
WriteCardBin (num);
WriteLn;
WriteString ("Shifted one position right yields ");
num := CAST (CARDINAL, SHIFT (CAST (CardSet, num), -1));
WriteCard (num, 0);
WriteString (" binary: ");
WriteCardBin (num);
WriteLn;
WriteString ("and, then rotated one position right yields ");
num := CAST (CARDINAL, ROTATE (CAST (CardSet, num), -1));
WriteCard (num, 0);
WriteString (" binary: ");
WriteCardBin (num);
WriteLn;

WriteString ("Do another? Y/N ");
ReadChar (answer);
SkipLine;
again := (CAP (answer) = "Y");
UNTIL NOT again;

END ShiftDemo.
```

Here is a brief run to illustrate:

This program illustrates bit shifting

Enter the number to be shifted **1**

The cardinal 1 binary: 00000000 00000000 00000000 00000001

Shifted one position right yields 0 binary: 00000000 00000000 00000000 00000000

and, then rotated one position right yields 0 binary: 00000000 00000000 00000000 00000000

Do another? Y/N **y**

Enter the number to be shifted **5**

The cardinal 5 binary: 00000000 00000000 00000000 00000101

Shifted one position right yields 2 binary: 00000000 00000000 00000000 00000010

and, then rotated one position right yields 1 binary: 00000000 00000000 00000000 00000001

Do another? Y/N **y**

Enter the number to be shifted **65535**

The cardinal 65535 binary: 00000000 00000000 11111111 11111111
Shifted one position right yields 32767 binary: 00000000 00000000 01111111 11111111
and, then rotated one position right yields 214750031 binary: 10000000 00000000
00111111 11111111
Do another? Y/N **n**

Observe the difference between the rotate and the shift when there is a one in the
rightmost position.

[Contents](#)

9.6 An Extended Example

Problem:

Write a program module that takes an input line of text and computes and prints the average number of consonants per syllable and the average number of syllables per word. Count every occurrence of one or more consecutive vowels as one syllable. (Note that this was posed as a problem in the exercise set of chapter 5, but could not be done there using sets.)

Problem suitability:

This can be viewed as an elaboration of the vowel counter problem in [section 9.1](#). Provided that an appropriate rule for counting syllables is used, text of any size and from any source can be automatically examined.

Problem restatement:

Given
a line of text
To Compute
the number of consonants
the number of syllables
the number of words
Output Desired
the number of consonants per syllable
the number of syllables per word

Refinement:

```
Print an informative header.  
Initialize counting variables to track consonants, syllables, and words.  
Ask for a line of text to analyze.
```

```
Input the text line.  
Examine each character:  
    check for end of string  
        if so, add one to word count and we're done  
    check for blank
```

```
    if so, and previous one not a blank, add one to word count
check for consonant
    if so, add one to consonant count
check for vowel
    if so, and previous one not a vowel, add one to syllable count
    otherwise, go on to next character
```

Compute consonants per syllable and output it.
Compute syllables per word and output that, too.

Variables Needed:

To hold input characters - next, last : CHAR

As counters - cons, syll, word : CARDINAL

For output - consPerSyll, syllPerWord

Sets - Letters, Vowels, Consonants

Imports Needed: WriteString, WriteLn, SkipLine, WriteReal

NOTE: As in most extended examples by this point in the book, the user documentation has been omitted in order to save space. This cannot be done in student programs or in commercial ones.

Code:

```
MODULE TextLineStats;

(* Written by R.J. Sutcliffe *)
(* using ISO Modula-2  *)
(* last revision 1994 03 02 *)

FROM SRealIO IMPORT
    WriteReal;
FROM SWholeIO IMPORT
    WriteCard;
FROM STextIO IMPORT
    ReadString, WriteString, WriteLn, SkipLine;

TYPE
    CharSet = SET OF CHAR;
    String = ARRAY [0 .. 255] OF CHAR;

CONST
    blank = " ";

VAR
    next, last : CHAR;
```

```
replyStr, theString : String;
count, numChars, numOfCons, numOfSyll, numOfWords : CARDINAL;
consPerSyll, syllPerWord : REAL;
yesChars, Letters, Vowels, Consonants : CharSet;
again : BOOLEAN;
```

BEGIN

```
(* First print out header information. *)
WriteString ("This program analyzes a line of text for");
WriteString (" statistical information.");
WriteLn;

(* Next initialize the sets and other variables *)
yesChars := CharSet {"Y", "y"};
Letters := CharSet {'A' .. 'Z', 'a' .. 'z'};
Vowels := CharSet
    {'A', 'E', 'I', 'O', 'U', 'Y', 'a', 'e', 'i', 'o', 'u', 'y'};
Consonants := Letters - Vowels;
```

REPEAT

```
(* Now tell user to type the input. *)
```

REPEAT

```
    WriteString ("Please provide line of text for analysis. ");
    WriteLn;
    ReadString (theString);
    SkipLine; (* gobble end of line state *)
    numChars := LENGTH (theString);
    IF numChars = 0
        THEN
            WriteString ("Type a proper string. Try again.");
            WriteLn;
```

END;

```
UNTIL numChars > numChars;      (* actual check for end is here *)
```

```
    (* Now do processing for output *)
```

```
INC (numOfWords); (* count last word; not followed by blank *)
IF numOfSyll "Number of consonants: ";
WriteCard (numOfCons, 6);
WriteLn;
WriteString ("Number of syllables: ");
WriteCard (numOfSyll, 6);
WriteLn;
WriteString ("Number of words: ");
WriteCard (numOfWords, 6);
```

```

WriteLn;
WriteString ("Number of consonants per syllable: ");
IF numOfSyll "undefined.";
    END;
WriteLn;
WriteString ("Average number of syllables per word: ");
WriteReal (syllPerWord, 6);
WriteLn;
    (* get ready to loop around again or quit *)
WriteString ("Do you wish to do another? (Y/N) ==");
ReadString (replyStr);
SkipLine;
again := replyStr [0] IN yesChars; (* reply starts with 'y' *)
UNTIL NOT again;

END TextLineStats.

```

Sample output:

This program analyzes a line of text for statistical information.
Please provide line of text for analysis.

Now is the time for all good parties to come to the aid of women.

```

Number of consonants:      28
Number of syllables:      19
Number of words:          15
Number of consonants per syllable: 1.4737
Average number of syllables per word: 1.2667
Do you wish to do another? (Y/N) ==> n

```

[Contents](#)

Part B--Records

9.7 Declaring and Assigning to Records

The discussions of [sections 1.6.3](#) and [9.1](#) explored the idea of modeling data that by its very nature consists of items of different types aggregated together into a single structure. For instance, one might wish to define an aggregate, or record:

```
traveller =  
  name  
  home address  
  airline  
  flight number  
  arrival  
  departure  
  luggage
```

A Modula-2 record is a data abstraction designed to allow for aggregates of various types of related data named by a single identifier.

Records are examples of what mathematicians call *cross products*. A cross product of two sets is the set of ordered pairs with first coordinate from the first set, second from the second set, thus:

$$A * B = \{ (a, b) \mid a \in A \text{ and } b \in B \}$$

For instance,

$$\{2, 4\} * \{1, 6\} = \{(2, 1), (2, 6), (4, 1), (4, 6)\}$$

In the case of records, the cross product is done with two or more (usually different) data types. That is, one expresses a structure *struct* that is

REAL * CARDINAL * BOOLEAN

as, say

TYPE

```
struct =  
  RECORD  
    salary : REAL;  
    days : CARDINAL;  
    male : BOOLEAN;  
  END;
```

where both the overall kind of structure (type) and the individual parts have names.

Consider, for instance, the representation of a *person* as a record where the abstraction (the program data as opposed to the actual person) consists of a last name, first name, birthdate, sex, identity number, marital status, and a statement about their role on a university campus. Of these, one could also in turn structure a record for *dates* that consists of a year, a month, and a day. Here are some declarations to illustrate the Modula-2 syntax:

TYPE

```
MonthType = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept, Oct, Nov, Dec);
```

```

Name = ARRAY [0 .. 20] OF CHAR;
Role = (freshman, sophomore, junior, senior, professor, administrator, maintenance,
associate, assistant, secretary);

Date =
  RECORD
    year : CARDINAL;
    month : MonthType;
    day : [1 .. 31];    (* semicolon optional *)
  END;    (* of the record Date *)

Person =
  RECORD
    lastname, firstname : Name;
    birth : Date;
    male : BOOLEAN;
    idnumber : CARDINAL;
    married : BOOLEAN;
    status : Role;
  END;    (* of the record Person *)

VAR
  student, faculty : Person;

```

NOTES: 1. RECORD is a reserved word, and each RECORD declaration must have an associated END.

2. The individual parts of a record are called *fields*.

3. A record field can be of any type, including an array, a user-defined type, or another record.

4. In the original definition of Modula-2, a function procedure could not return a RECORD as its result. Newer versions (including ISO standard ones) allow this. If this restriction does exist, it will be noted later that there may be a way around it.

5. There will probably be some implementation dependent maximum record size, though this is not likely to impact on the programs of a beginner.

The individual fields in each record each have their own identifier, and each is referred to by a qualified identifier that includes the record name itself. The following statements assign the fields of the above record:

```

student.lastname := "Hacker";
student.firstname := "Nellie";
student.birth.year := 1965;
student.birth.month := Feb;
student.birth.day := 12;
student.male := FALSE;
student.idnumber := 46725;
student.married := FALSE;
student.status := freshman;

```

In a similar way, one could assign the fields of a record representing a faculty member, except that the last one could read, say:

```

faculty.status := associate;

```

One could also define the structure of *name* and *birth* within the declaration of *person*, writing:

```

TYPE
  Person =

```


RECORD

```
lastname, firstname : Name;
```

```
birth =
```

RECORD

```
year : CARDINAL;
```

```
month : MonthType;
```

```
day : [1 .. 31];
```

```
END;      (* of the sub-record Date *)
```

```
male : BOOLEAN;
```

```
idnumber : CARDINAL;
```

```
married : BOOLEAN;
```

```
status : Role;
```

```
END;      (* of the record Person *)
```

However, it is better programming practice to employ separate type declarations for each structure, rather than bury structures without type names inside another one (anonymous types). Once *Date* is properly declared as a type, it can be used in a variety of other declarations, rather than having to write out its structure every time. Such natural abstractions should always be made explicit in the code.

[Contents](#)

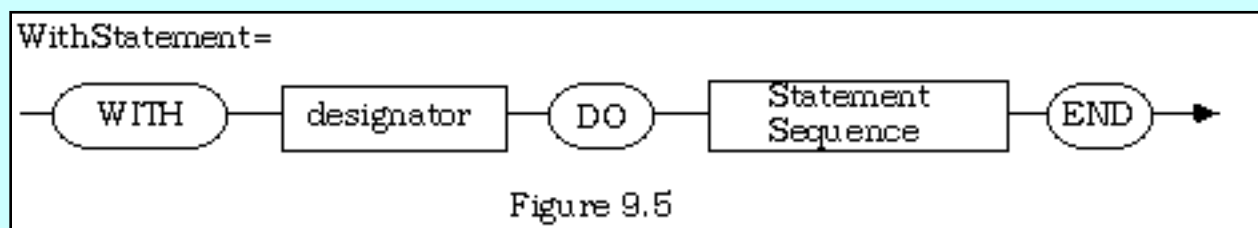
9.8 Using Records

Most people come to the conclusion while working through reading the last section that writing "*student*." in front of every record field identifier when doing assignments is a pain in the neck. The obvious thing to wonder, in view of previous experience, is whether the qualified identifiers of record fields can be unqualified in some fashion similar to the unqualifying of identifiers from module imports (in a FROM statement). They can, and this unqualifying is done as in the construction below:

```
WITH student
DO
    lastname := "Hacker";
    firstname := "Nellie";
    birth.year := 1965;
    birth.month := Feb;
    ...

END;
```

Within the boundaries of the WITH statement, a new scope is created in which the record fields of the record named in the WITH can be used unqualified. Here is the diagram of the WITH statement:



In a sense, the WITH statement serves a similar role in selecting the fields of a record (for assignment or otherwise) as does the FOR to select the indices of an array. In another sense, its effect on the identifiers within its scope is to unqualify them in the same manner as does the FROM statement. Naturally, one can nest these and further compact the above in this manner:

```
WITH student
DO
    lastname := "Hacker";
    firstname := "Nellie";

    WITH birth
    DO
        year := 1965;
```

```
    month := Feb;
    day := 12;
END;
```

```
male := FALSE;
idnumber := 46725;
married := FALSE;
status := freshman;
END;
```

NOTES: 1. Only one identifier can be named in each **WITH** statement.
2. Within a **WITH** statement, no assignment can be made to the identifier it names. It will not work to try to say
student := otherStudent
within this **WITH** statement, for the variable unqualified is evaluated only once, when the scope of **WITH** is entered, and this value is the one referenced throughout that scope. This is similar (not the same as) to the rule that one cannot re-assign the index variable of a **FOR** within its loop.
3. Each **WITH** has its own **END**.
4. Any fields not assigned are indeterminate and could contain any random data.
Naturally, one can read data from the keyboard and assign it directly to the variable parameters of the read statements from an appropriate library module:

```
WITH student
DO
    WriteString ("Enter the last name please. ");
    ReadString (lastname);    (* assigns student.lastname *)
    ...
END;
```

Once data has been assigned to a field in a record, it can subsequently be manipulated in the same way as any other data. Individual fields of a record can be used as part of numerical expressions in assignments (where appropriate) and whole records with all their fields can be assigned (provided they are of the same type). Starting with:

```
TYPE Employee =
RECORD
    lastname, firstname : Name;
    salary : REAL;
END;
```

```
VAR
    clerk, secretary : Employee;
```

and assuming the appropriate assignments have been made to the fields and that the other identifiers used below were also defined, one could write assignments like:

```
secretary := clerk;    (* someone was transferred *)
```

but, one *cannot* write statements like:

```
IF secretary = clerk
  THEN    (* comparisons not allowed *)
    pay := hours * clerk.salary
  END;
```

NOTE: If one of the fields were unassigned, or contained unassigned elements, such as the unused trailing portions of a string (or other array components), a comparison employing = or # would not give meaningful results, even if the data in the used fields and components is all equal. For this reason, such comparisons are not allowed.

Two points are worth considering when deciding whether or not to employ the WITH statement. On the one hand, some people consider it to be more trouble than it is worth to unqualify using WITH for the purpose of making a single assignment. On the other, there are those, including some teachers and language experts, who believe that WITH should *never* be used, on the grounds that a program is easier to read when all identifiers are employed in their qualified form. (For consistency, such purists will also only import qualified identifiers.)

Second, if one imports a record into a module, one automatically imports qualified all its field names along with it. This means that the field identifiers need not be specifically mentioned in the import statement, only the type name or variable name under whose umbrella they are declared. This is a similar rule to the one that was earlier applied to the import of enumerated types, which implied an automatic import of all the identifiers in the enumeration. The difference is that field type names are only available in qualified form unless WITH is employed to open a new scope, whereas enumeration item names are available unqualified.

Example:

Supposing that in the module *Files* there were a variable *FilePos* defined whose structure was:

```
TYPE
  FilePos =
    RECORD;
    hi, lo : CARDINAL
  END
```

and then, in some program module there was the code:

```
FROM Files IMPORT  
    FilePos;
```

then the following code would be correct:

```
PROCEDURE WritePos;  
BEGIN  
    WriteCard (FilePos.hi, 6);  
    WriteCard (FilePos.lo, 6);  
END WritePos;
```

Because the record name has been imported, its field names are also visible in the importing scope as qualified identifiers, and can also be unqualified with an appropriate **WITH** statement.

[Contents](#)

9.9 Records and Arrays--Which, or Both?

Each kind of structured data has its own advantages and disadvantages for specific purposes. Referencing individual items in an array is fairly simple, and a FOR loop is compact, easy to write, and easy to understand. However, the data entered in a single array must all be of the same type, and that makes grouping of different data types into related categories rather difficult.

On the other hand, records can do this kind of grouping of diverse data types, and have great flexibility, but that data cannot be referred to by a numerical position; rather a field name must be employed.

Records are (potentially) more elaborate structures than arrays, and the mechanism for using them requires more work to set up, but may be worth it.

In many cases, it is actually possible to use either kind of structure.

Suppose one wanted to design a data structure to hold these class records, say student names, identity numbers, four percentages, and a final letter grade.

One could arrange several arrays in a "side-by-side" fashion as follows:

```
MODULE Markbook;  
FROM STextIO IMPORT  
    WriteString, ReadString, SkipLine, WriteLn, WriteChar;  
FROM SWholeIO IMPORT  
    WriteCard, ReadCard;  
  
PROCEDURE GetNum ( ) : REAL;  
BEGIN  
    (* usual code here *)  
    SkipLine;  
    RETURN 0.0 (* dummy *)  
END GetNum;  
  
TYPE  
    Name = ARRAY [0 .. 20] OF CHAR;  
    ClassList = ARRAY [1 .. 30] OF Name;  
    Idnumber = ARRAY [1 .. 30] OF CARDINAL;  
    Marks = ARRAY [1 .. 30], [1 .. 4] OF REAL;  
    Grades = ARRAY [1 .. 30] OF CHAR;  
  
VAR  
    cmpt141 : ClassList;  
    num : Idnumber;  
    scores : Marks;  
    final : Grades;  
    classCount, markCount : CARDINAL;
```

```

BEGIN
  FOR classCount := 1 TO 30
    DO
      WriteCard (classCount,1);
      WriteChar ("");
      WriteString (" What is the student's name? ");
      ReadString (cmpt141 [classCount]);
      SkipLine;
      WriteLn;
      WriteString ("and the i.d. number? ");
      ReadCard (num [classCount]);
      SkipLine;
      WriteLn;

      FOR markCount := 1 TO 4
        DO
          WriteString ("Give me the mark, with decimal, ");
          WriteString ("for term number ");
          WriteCard (markCount, 10);
          WriteString (" please. ");
          scores [classCount, markCount] := GetNum ();
          (* don't forget to include this procedure *)
          WriteLn;
        END;  (* for markCount *)
      END;  (* for classCount *)

      (* process marks and assign letter grade here *)
    END Markbook.

```

In this approach, the index number *classCount* within the class effectively groups the data from the different types of arrays. The class is a collection of arrays that are not formally gathered into a single structure, and the data pertinent to a single student must be extracted by knowing the number for that student and using it consistently through all the arrays.

While this approach would work, it hides the real structure of the class, that is, a collection of students, each of whom has various associated data. Consider the following (better) approach:

```

MODULE Mark2Book;

FROM STextIO IMPORT
  WriteString, ReadString, SkipLine, WriteLn;
FROM SWholeIO IMPORT
  ReadCard;  (* and the other usual imports *)

```

TYPE

```
Name = ARRAY [0 .. 20] OF CHAR;
Marks = ARRAY [1 .. 4] OF REAL;
Student =
    RECORD
        moniker : Name;
        idnumber : CARDINAL;
        marks : Marks;
        grade : CHAR;
    END;  (* student *)
Class = ARRAY [1 .. 30] OF Student;
```

VAR

```
cmpt141 : Class;
classCount : CARDINAL;
```

BEGIN

```
FOR classCount := 1 TO 30
    DO
        WITH cmpt141 [classCount]
            DO
                WriteString ("What is the student's name? ");
                WriteLn;
                ReadString (moniker);
                SkipLine;
                WriteLn;
                WriteString ("and the i.d. number? ");
                ReadCard (idnumber);
                SkipLine;
                WriteLn;

                (* etc *)
            END; (* With *)
        END; (* For *)
```

```
END Mark2Book.
```

NOTE: Without the **WITH** statement, and assuming the declaration of a variable markCount to count through the small array of marks within the record, the correct qualified identifiers would be:

```
cmpt141 [classCount].moniker
cmpt141 [classCount].mark [markCount]
cmpt141 [classCount].grade
cmpt141 [classCount].idnumber
where: 1 <= markCount <= 4.
```


Arrays of records such as this one are particularly useful for keeping track of individual people and the information about them when those people are also organized into a specific grouping of some kind. Information about students, customers, clients, creditors, employees, and the like can all be arranged in such a fashion.

[Contents](#)

9.10 Getting Physical With Records

The discussion to this point has been rather abstract, and has dealt with records strictly in the context of representation and assignment. If records are to be of any use, they must be capable of being stored to and retrieved from the secondary storage (disk or tape drive). That is, one needs to be able to move records from one plane of existence to another in order to make efficient use of the machine. Just as in order to assign a record it is necessary to know the type of every field, so also one must take advantage of knowing the details of the structure in order to write one to a text stream. If one had:

TYPE

```
Name = ARRAY [0 .. 10] OF CHAR;
StudentData =
  RECORD
    lastName, firstName : Name;
    mark1, mark2 : REAL;
    grade : CHAR
  END;
```

and, supposing that *classMember* were of type *StudentData*, one would need to write the code:

WITH classMember

```
DO
  WriteString (outChan, firstName);
  WriteLn;
  WriteString (outChan, lastName);
  WriteLn;
  WriteReal (outChan, mark1, 12);
  WriteLn;
  WriteReal (outChan, mark2, 12);
  WriteLn;
  WriteChar (outChan, grade);
  WriteLn;
END;
```

to send the data to an output device, one item per line. This code would also suffice whether the output device were the screen, a printer, or a sequential file. This could be achieved in the usual ways:

1. Use *STextIO* routines without the channel parameter and the non-standard routine *OpenOutput* to redirect the standard output to a file, or
2. Use *StreamFile* or *SeqFile* as appropriate to open a specific stream channel to *outChan* and print as above.

As in earlier cases too, if one sent all that text to a disk file by either method, there would be quite a lot of valuable storage wasted because the ASCII text version of a real number would take up about six times as many storage locations as it the raw memory version does. Further, separate routines for displaying records contents on the screen would be required anyway, because one surely would want some static information to organize the data visually thus:

```
Name: Meier, Johanna
Marks: 84.7 and 92.6
Final Grade: A
```

and such static tags as *Name:* as may be necessary for screen output certainly ought not to be incorporated into every one of the

records as they go to the stored file.

So, in this case again, it would be better to use RawIO and send each record as an entire entity to the disk file in raw form thus:

```
RawIO.Write (classMember );
```

Likewise, if one had a large number of records stored in an array as:

TYPE

```
Name = ARRAY [0 .. 20] OF CHAR;  
Item =  
  RECORD  
    name : Name;  
    price : REAL;  
    quantity : CARDINAL;  
    bin : CHAR;  
  END; (* Item *)  
Items = ARRAY [1 .. max] OF Item;
```

VAR

```
stock : Items;  
dataChan : ChanId;
```

one could save such items to a disk file using:

```
SeqFile.Rewrite (dataChan);  
FOR count := 1 TO max  
  DO  
    RawIO.Write (dataChan, stock [count]);  
  END;  
SeqFile.Close (dataChan);
```

or, even just

```
RawIO.Write (dataChan, stock)
```

to send the entire array at once. Likewise, the items can be read using:

```
numItems := 0;  
REPEAT  
  RawIO.Read (dataChan, stock [numItems + 1]);  
  IF IOResult.ReadResult (dataChan) = allRight  
    THEN  
      INC (numItems);  
    END;  
UNTIL (numItems = max) OR (IOResult.ReadResult (dataChan) # allRight)
```

or, read the entire array, if that is the way the data was originally written out:

```
RawIO.Read (dataChan, stock);
```

This example is elaborated on in the extended illustration at the end of this chapter.

9.10.1 RawIO in Non-Standard Modula-2

As a last word on this example, it should be observed that the ISO procedures *Read* and *Write* in *RawIO* are high-level implementations of somewhat relatively low-level ideas. Corresponding procedures may or may not exist in some non standard implementations.

If there are such procedures, they are probably found in the module *Files/Filer/FileSystem* and have definitions such as:

```
PROCEDURE ReadRec(file : File; VAR rec : ARRAY OF BYTE);
  (* This is a safe high level procedure for reading records *)

PROCEDURE WriteRec(file : File; rec : ARRAY OF BYTE);
  (* This is a safe high level procedure for writing records *)
```

where *SYSTEM.BYTE* takes the place of *SYSTEM.LOC*.

On the other hand, it might be necessary to manufacture one's own procedures with these names from other routines that one already has in order to achieve results similar to this. For instance, there *may* be a low level regular or function procedure:

```
PROCEDURE ReadBytes (VAR f : FILE, addr : ADDRESS; count : CARDINAL) <:CARDINAL--
again, most likely in Files/Filer/FileSystem or in another module that might be
called ByteBlockIO.
```

The parameters are: a variable of type *File*, the starting address of the memory location to which one is going to read, and the number of bytes required.

If *ReadBytes* is implemented as a function procedure (check the manual) then the *CARDINAL* returned is the number of bytes actually read. If the latter is less than the number requested, a variable *EOF* will also be set to *TRUE*. Now, one could perhaps code:

```
PROCEDURE ReadRec (inFile: FILE; VAR item : ARRAY OF WORD);
BEGIN
  ReadBytes (inFile, ADR (person), SIZE (person))
END ReadRec;
```

and likewise for *WriteRec*.

Because such routines are tied to the machine on which they are executed, and because non-standard versions of Modula-2 have more variation in the area of file handling I/O than in anything else, the names and syntax will in those cases vary widely from one version of Modula-2 to another.

[Contents](#)

9.11 Records and Random Access Files

In the last chapter, and so far in this chapter as well the kind of file used in the examples was the sequential type, that is, the *stream* of data. It is important to remember that sequential files (whether rewindable or not) are only one of two possible models for files, the other being the random access file. The only difference introduced in the last section was that the stream consists of record items rather than text items, or individual binary items.

In the example of the last section, the records were sent to the output device in raw form. In the procedure *SaveFile* instead of:

```
FOR count := 1 TO numItems
DO
  Write (dataChan, stock [count]);
END;
```

one could have written the output as a text stream instead, though it would have taken more effort and more room on the storage device:

```
FOR count := 1 TO numItems
DO
  WITH Stock[count]
  DO
    WriteString (dataChan, name);
    WriteLn;
    WriteReal (dataChan, price, 1)
    WriteLn;
    WriteCard (dataChan, quantity, 1)
    WriteLn;
    WriteChar (dataChan, bin)
    WriteLn;
  END;
END;
```

The advantage would be that the resulting file would consist of text and could be read by other programs. Moreover, whether at the logical level one views the file as a sequence of records, or as a sequence of characters at the physical file level one still has a stream of text material--albeit organized into logical groupings determined by the structure of the program.

Since this logical organization makes so much more sense than thinking of such a file as merely a text stream, and since the record is the important abstraction here, it is natural to ask whether one can find a particular one of those records in the disk file without having to read them all into memory at once.

The answer is "yes," but there is a *caveat*. Any attempt to do this requires that one be able to reliably find the place on the disk where the record is stored. If the data is being read and written as text, this is tedious, because each record must be read back in one field at a time until the correct one is found. This is because there is no way to tell how much room such a record occupies. The text produced by a *WriteInt* could have any number of characters in it. The text produced by a *WriteString* has a number of characters up to one more than the HIGH of the character array parameter. Thus text records all occupy different amounts of space, and so must be read the way they are written--one text field at a time.

On the other hand, if the records are written raw, they all occupy the same amount of disk space, for they all occupy the same amount of memory. It is therefore possible to find a particular record by reading whole records sequentially. To find the inventory item whose name is *bolts*, proceed as follows: Assume that there is a program variable item with the same structure

as used above. This will not be an array, just a single record. Assume that the program has gone through the routine of opening the appropriate file and connecting the program to it. Now it must read through all the records until it finds the correct one:

REPEAT

```
Read (dataChan, item);
```

```
UNTIL (item.name = "bolts") OR (IOResult.ReadResult (dataChan) # allRight);
```

There is a better way, in those cases that one knows ahead of time which record one is looking for, or if one wishes only to handle in memory and edit a single record and then find the same position in the disk file to write it back. Suppose one has the number of the record *recnum* that must be found (starting with the first in the file as number zero), and also the size of the record (the number of storage locations each takes). The loop above could then be replaced with the following code:

```
recsize := SIZE (ItemType); (* the same for all reads and writes *)  
top := StartPos (dataChan);  
pos := NewPos (dataChan, recnum, recsize, top);  
SetPos (dataChan, pos);  
Read (dataChan, student);
```

where the variables *top* and *pos* are both of type *FilePos*, and the procedures *NewPos*, *StartPos*, and *SetPos* as well as the type *FilePos* are all imported from *RndFile*. The idea is to calculate with *NewPos* the position in the file where one wishes to begin reading or writing, and then set the actual file position marker to that position before doing the input or output. Once one begins to use position markers and to read and write whole records without regarding the expression of either their individual fields or entire records as streams, one has changed to a *random access model* for the file. Recall in the definitions given in chapter eight that the difference between a stream or sequential type of file and the random access type is the rules for reading and writing. It does not matter whether one uses a record as such in the process. What determines that a file is random access is whether or not one is allowed to calculate and set a position marker to read and write anywhere in the file. That is, the difference between the two has to do with the way one *thinks* about the files, not necessarily with the way they exist on the disk. Both can be thought of on one level as recordings of streams of data for all data is reduced to a series of raw binary numbers in any case. Furthermore, a sequential file *could* be thought of as a random access file whose records consist of single characters. Although the latter is not a particularly useful model, the various positioning functions described above will work on a file that was originally created as a sequence, because the I/O routines do not know or care what sort of structure the actual file has; those sorts of interpretations are up to the programmer.

File manipulation is subject to two markers.

The position marker is the point in the file at which the next piece of data will be read or written.

The End-Of-File, or EOF marker is defined to be one place after the last piece of data in the file.

A program cannot read past the EOF marker and an attempt to do so will result in *IOResult.ReadResult* becoming set to the value *endOfInput*. Further, any attempt to use *SetPos* to position reading beyond the EOF marker will generate an exception (run time error).

RndFile is the ISO module (device driver) that implements the random access file model. In order for the file positioning routines to work properly on the specified channel, the channel must actually be open by *RndFile*, just as *SeqFile.Rewrite* will only work on a file that has been opened (and is being modeled by *SeqFile* at the time). For reference, here is a listing of the contents of *RndFile*:

DEFINITION MODULE RndFile;

```
(* Random access files *)
```

```
( * =====
      Definition Module from
          ISO Modula-2
Draft Standard CD10515 by JTC1/SC22/WG13
      Language and Module designs © 1992 by
BSI, D.J. Andrews, B.J. Cornelius, R. B. Henry
R. Sutcliffe, D.P. Ward, and M. Woodman

      Implementation © 1993
          by R. Sutcliffe
      Trinity Western University
7600 Glover Rd., Langley, BC Canada V3A 6H4
      e-mail: rsutcl@twu.ca
      Last modification date 1993 10 20
=====*)
```

```
IMPORT IOChan, ChanConsts, SYSTEM;
```

TYPE

```
  ChanId = IOChan.ChanId;
  FlagSet = ChanConsts.FlagSet;
  OpenResults = ChanConsts.OpenResults;
```

```
CONST    (* Accepted singleton values of FlagSet *)
  read = FlagSet{ChanConsts.readFlag};    (* input operations are
requested/available *)
  write = FlagSet{ChanConsts.writeFlag}; (* output operations are
requested/available *)
  old = FlagSet{ChanConsts.oldFlag};      (* a file may/must/did exist before the
channel is opened *)
  text = FlagSet{ChanConsts.textFlag};    (* text operations are
requested/available *)
  raw = FlagSet{ChanConsts.rawFlag};      (* raw operations are
requested/available *)
```

```
PROCEDURE OpenOld (VAR cid: ChanId; name: ARRAY OF CHAR; flags: FlagSet; VAR res:
OpenResults);
  (* Attempts to obtain and open a channel connected to a stored random access file
of the given name.
  The old flag is implied; without the write flag, read is implied; without the
text flag, raw is implied.
  If successful, assigns to cid the identity of the opened channel, assigns
the value opened to res, and sets the read/write position to the start of the file.
  If a channel cannot be opened as required, the value of res indicates the
reason, and cid identifies the invalid channel.  *)
```

```
PROCEDURE OpenClean (VAR cid: ChanId; name: ARRAY OF CHAR; flags: FlagSet; VAR res:
OpenResults);
  (* Attempts to obtain and open a channel connected to a stored random access file
of the given name.
  The write flag is implied; without the text flag, raw is implied.
  If successful, assigns to cid the identity of the opened channel, assigns the
```

value opened to res, and truncates the file to zero length.

If a channel cannot be opened as required, the value of res indicates the reason, and cid identifies the invalid channel.

*)

PROCEDURE IsRndFile (cid: ChanId): **BOOLEAN**;

(* Tests if the channel identified by cid is open to a random access file. *)

PROCEDURE IsRndFileException (): **BOOLEAN**;

(* Returns TRUE if the current coroutine is in the exceptional execution state because of the raising of a RndFile exception; otherwise returns FALSE. *)

CONST

FilePosSize = 4; (* implementation defined constant *)

TYPE

FilePos = **ARRAY** [1 .. FilePosSize] **OF** SYSTEM.LOC;

PROCEDURE StartPos (cid: ChanId): FilePos;

(* If the channel identified by cid is not open to a random access file, the exception wrongDevice is raised; otherwise returns the position of the start of the file. *)

PROCEDURE CurrentPos (cid: ChanId): FilePos;

(* If the channel identified by cid is not open to a random access file, the exception wrongDevice is raised; otherwise returns the position of the current read/write position. *)

PROCEDURE EndPos (cid: ChanId): FilePos;

(* If the channel identified by cid is not open to a random access file, the exception wrongDevice is raised; otherwise returns the first position after which there have been no writes. *)

PROCEDURE NewPos (cid: ChanId; chunks: **INTEGER**; chunkSize: **CARDINAL**; from: FilePos): FilePos;

(* If the channel identified by cid is not open to a random access file, the exception wrongDevice is raised; otherwise returns the position (chunks * chunkSize) relative to the position given by from, or raises the exception posRange if the required position cannot be represented as a value of type FilePos. *)

PROCEDURE SetPos (cid: ChanId; pos: FilePos);

(* If the channel identified by cid is not open to a random access file, the exception wrongDevice is raised; otherwise sets the read/write position to the value given by pos. *)

PROCEDURE Close (**VAR** cid: ChanId);

(* If the channel identified by cid is not open to a random access file, the exception wrongDevice is raised; otherwise closes the channel, and assigns the value identifying the invalid channel to cid. *)

END RndFile.

NOTES: 1. A channel must be opened with *OpenOld* or *OpenClean* before it can be manipulated by the *RndFile* positioning procedures.

2. The flags employed upon opening come from the same initial source as for *SeqFile* and *StreamFile*, and have similar meanings.
 3. The "exceptions" mentioned are run time errors, about which more will be said later in the text.
 4. Observe that *NewPos* can compute a new position forward or backward from any position, because the number of chunks is an integer, and the position counted *from* can be anywhere in the file.
 5. *EndPos* is useful for calculating the place to do appending.
 6. A *read* or *write* of a record moves the position marker. To access the same record again, the position must be moved back to where it was before using *SetPos*.
 7. Non-standard Modula-2 implementations have similar procedures to these, though with slightly different names and syntax in some cases. Position variables may be just of type LONGINT, for example. These are usually found in *Files/Filer/FileSystem*, where also is located the one and only *Open* procedure. It is up to the programmer to enforce the separation between sequential file models and random access file models in such versions; there is no help from the library suite.
 8. If the position marker is at the end of the file then a *write* operation will extend the file, making it larger. If it is anywhere else, a *write* operation will lay down the new data on top of whatever was there before, erasing the old.
 9. One cannot set the current position marker past the end-of-file marker or before the beginning of the file; for there will then be a run-time error.
- These procedures are elaborated on in the second part of the extended example found in the next section.
-

[Contents](#)

9.12 An Extended Example (Inventory)

Here is a simple illustration of some of the ideas found in the last two sections. The program is designed to keep track of a small inventory. Each item is recorded by name, price, quantity and location (bin number). The records are kept in a disk file, and any new items are added to that disk file whenever the user chooses to save the data or exits the program.

9.12.1 Inventory with Raw Sequential I/O

The first version of the program keeps track of the inventory in a file that is manipulated by *SeqFile* as a rewindable stream, and the actual I/O is done by RawIO. Logically, the items in the stream consist of records; physically they are binary recordings. The planning information has been left out so as to save space.

```
MODULE Inventory;
(* Keeps track of a demonstration inventory in a file called "inventory.data." *)

(* Written by R.J. Sutcliffe *)
(* to demonstrate the use of records and RawIO *)
(* using ISO standard Modula-2 *)
(* last revision 1994 03 24 *)

FROM STextIO IMPORT
  WriteString, ReadString, SkipLine, WriteLn, ReadChar, WriteChar;
FROM SWholeIO IMPORT
  ReadCard, WriteCard;
FROM SRealIO IMPORT
  ReadReal, WriteFixed;
FROM RawIO IMPORT
  Read, Write;
FROM SIOResult IMPORT
  ReadResult, ReadResults;
IMPORT IOResult;
FROM SeqFile IMPORT
  OpenRead, OpenResults, raw, write, Reread, Rewrite, Close, ChanId;

CONST
  max = 10; (* a small inventory *)

VAR
  fileOpen, fileDirty : BOOLEAN; (* to know what the status is *)

TYPE
  Name = ARRAY [0 .. 20] OF CHAR;
  Item =
    RECORD
      name : Name;
      price : REAL;
      quantity : CARDINAL;
      bin : CHAR;
    END; (* Item *)
```

```

    Items = ARRAY [1 .. max] OF Item;
VAR
    emptyItem : Item;
    stock : Items;
    Ok : BOOLEAN;
    dataChan : ChanId;
    res : OpenResults;

(* This group of procedures displays current contents of fields on the screen for
viewing or editing. *)

PROCEDURE DisplayName (name: Name);
BEGIN
    WriteString ("Name ==");
    WriteString (name);
END DisplayName;

PROCEDURE DisplayPrice (price : REAL);
BEGIN
    WriteString ("Price ==");
    WriteFixed (price, 2,6);
END DisplayPrice;

PROCEDURE DisplayQuantity (quantity : CARDINAL);
BEGIN
    WriteString ("Quantity ==");
    WriteCard (quantity, 10);
END DisplayQuantity;

PROCEDURE DisplayLocation (bin : CHAR);
BEGIN
    WriteString ("Bin Location ==");
    WriteChar (bin);
END DisplayLocation;

PROCEDURE DisplayItem (item : Item);
(* calls the above to display an item. *)
BEGIN
    WITH item
    DO
        WriteString ("          Item");
        WriteLn;
        DisplayName (name);
        WriteString ("      ");
        DisplayPrice (price);
        WriteString ("      ");
        WriteLn;
        DisplayQuantity (quantity);
        WriteString ("      ");
        DisplayLocation (bin);
        WriteLn;WriteLn;
    END;
END DisplayItem;

```

```

VAR
    numItems : CARDINAL;  (* global *)

PROCEDURE ListItems; (* list all items *)
VAR
    count : CARDINAL;
BEGIN
    FOR count := 1 TO numItems
        DO
            WriteCard (count, 1);
            WriteString (". ");
            DisplayName (stock [count].name);
            WriteLn;
        END;
END ListItems;

PROCEDURE EditItem (VAR item : Item); (* change contents *)
VAR
    tempName : Name;
    tempPrice : REAL;
    tempQuantity : CARDINAL;
    tempBin : CHAR;

BEGIN
    WITH item
        DO
            WriteString (" Edit Item");
            WriteLn;
            DisplayName (name);
            WriteLn;
            ReadString (tempName);
            IF (ReadResult() = allRight)
                THEN
                    name := tempName;
                    fileDirty := TRUE;
                END;
            SkipLine;
            WriteLn;
            DisplayPrice (price);
            WriteLn;
            ReadReal (tempPrice);
            IF (ReadResult() = allRight)
                THEN
                    price := tempPrice;
                    fileDirty := TRUE;
                END;
            SkipLine;
            WriteLn;
            DisplayQuantity (quantity);
            WriteLn;
            ReadCard (tempQuantity);
            IF (ReadResult() = allRight)

```

```

    THEN
        quantity := tempQuantity;
        fileDirty := TRUE;
    END;
SkipLine;
WriteLn;
DisplayLocation (bin);
WriteLn;
ReadChar (tempBin);
IF (ReadResult() = allRight)
    THEN
        bin := tempBin;
        fileDirty := TRUE;
    END;
SkipLine;
END; (* with *)
END EditItem;

PROCEDURE AddItem;
(* make an empty item, and then edit it. *)
VAR
    temp : Item;
BEGIN
    IF numItems < max
    THEN
        temp := emptyItem;
        EditItem (temp);
        INC (numItems);
        stock [numItems] := temp;
    END;
END AddItem;

PROCEDURE GetItem (VAR itemNum : CARDINAL);
(* Find out which one to deal with *)
BEGIN
    IF numItems > 0 THEN
        WriteCard (numItems,1);
        WriteString (" ==");
        ReadCard (itemNum);
        SkipLine;
        Ok := (ReadResult() = allRight) AND (itemNum <= numItems);
        IF NOT Ok
        THEN
            WriteString ("Error in selection; try again");
            WriteLn;
        END;
        UNTIL Ok;
    ELSE
        WriteString ("No items to list");
        WriteLn;WriteLn;
        itemNum := 0;
    END;
END;

```

```
END GetItem;
```

```
PROCEDURE Menu (VAR menuNum : CARDINAL);
```

```
(* print a menu of program choices and get a valid choice *)
```

```
VAR
```

```
    Ok : BOOLEAN;
```

```
BEGIN
```

```
    REPEAT
```

```
        WriteString ("Do you wish to");WriteLn;
```

```
        WriteString ("1. Get existing/ open new disk file");WriteLn;
```

```
        WriteString ("2. Display an item");WriteLn;
```

```
        WriteString ("3. Add an item");WriteLn;
```

```
        WriteString ("4. Edit an item");WriteLn;
```

```
        WriteString ("5. Save disk file");WriteLn;
```

```
        WriteString ("6. Quit the program");WriteLn;
```

```
        WriteString ("Pick one 1 .. 6 ==");
```

```
        ReadCard (menuNum);
```

```
        WriteLn;
```

```
        SkipLine;
```

```
        Ok := (ReadResult () = allRight) AND (menuNum <7);
```

```
    IF NOT Ok
```

```
        THEN
```

```
            WriteString ("Error in menu selection; try again");
```

```
            WriteLn;WriteLn;
```

```
        END;
```

```
    UNTIL Ok;
```

```
END Menu;
```

```
PROCEDURE GetFile;
```

```
(* Open and read contents of Inventory.data, if any. *)
```

```
BEGIN
```

```
    numItems := 0;
```

```
    OpenRead (dataChan, "InventoryData", raw+write, res);
```

```
    IF (res = opened)
```

```
        THEN
```

```
            fileOpen := TRUE;
```

```
        REPEAT
```

```
            Read (dataChan, stock [numItems + 1]);
```

```
            IF IOResult.ReadResult (dataChan) = allRight
```

```
                THEN
```

```
                    INC (numItems);
```

```
                END;
```

```
            UNTIL (numItems = max) OR (IOResult.ReadResult (dataChan) # allRight)
```

```
        END;
```

```
END GetFile;
```

```
PROCEDURE SaveFile;
```

```
(* write out entire file.  If file not already open, opens and reads stuff in first. *)
```

```
VAR
```

```

    count : CARDINAL;
BEGIN
    IF NOT fileOpen
    THEN
        IF NOT fileDirty
        THEN
            WriteString ("No data collected & no file open");
            WriteLn;
            RETURN;
        END;
        GetFile;
    END;
    Rewrite (dataChan);
    FOR count := 1 TO numItems
    DO
        Write (dataChan, stock [count]);
    END;
    Close (dataChan);
    fileOpen := FALSE;
    fileDirty := FALSE;
    numItems := 0;

```

```

END SaveFile;

```

```

VAR (* main *)
    action, itemNum : CARDINAL;

```

```

BEGIN
    numItems := 0;
    fileOpen := FALSE;
    fileDirty := FALSE;
    (* make a default or empty item for later editing *)
    WITH emptyItem
    DO (* initialize one *)
        name := "";
        price := 0.0;
        quantity := 0;
        bin := "*"; (* nowhere *)
    END; (* with *)
    WriteString ("    Rick's Inventory program");
    WriteLn;WriteLn;

```

```

REPEAT
    Menu (action); (* print menu + get action *)
    (* take action according to request from menu *)
    IF (action = 1) AND NOT fileOpen
    THEN
        GetFile;
    ELSIF (action = 2) THEN
        GetItem (itemNum);
        IF itemNum # 0
        THEN
            DisplayItem (stock [itemNum]);

```

```

        END;
    ELSIF (action = 3) THEN
        AddItem;
    ELSIF (action = 4) THEN
        GetItem (itemNum);
        IF itemNum # 0
            THEN
                EditItem (stock [itemNum]);
            END;
    ELSIF (action = 5) THEN
        SaveFile;
    ELSIF (action = 6) THEN
        IF fileOpen
            THEN
                SaveFile;
            END;
        END;
    UNTIL action = 6; (* and then quietly exit *)
END Inventory.

```

NOTES: 1. Observe the free use of numerous small specialized procedures to encapsulate program tasks for easy debugging.
2. Note the use of a menu that is reprinted on the screen repeatedly whenever the current task is complete.
This program was run and the data file created, added to, and edited. The program was run again and more items added.
However, these runs are lengthy and the output is not reproduced here. It is left as an exercise to the student to make some improvements.

9.12.2 Inventory with Raw Random I/O

In this second module, the same file is opened and manipulated as in the last, except that a random access model is used, and only one record item is kept in memory at a time. The fact that a random access model can be imposed upon a file previously created with a sequential model device driver illustrates that the logical view of a file is independent of the physical recording. In the program listing below, many of the procedures from the first version are not duplicated.

```

MODULE RndInventory;
(* Keeps track of a demonstration inventory in a file called "inventory.data." *)

(* Written by R.J. Sutcliffe *)
(* to demonstrate the use of records, random access and RawIO *)
(* using ISO standard Modula-2 *)
(* last revision 1994 03 25 *)

FROM STextIO IMPORT
    WriteString, ReadString, SkipLine, WriteLn, ReadChar, WriteChar;
FROM SWholeIO IMPORT
    ReadCard, WriteCard;
FROM SRealIO IMPORT
    ReadReal, WriteFixed;
FROM RawIO IMPORT
    Read, Write;
FROM SIOResult IMPORT
    ReadResult, ReadResults;

```



```

IMPORT IOResult;
FROM RndFile IMPORT
    OpenOld, OpenResults, raw, read, write, Close, ChanId,
    FilePos, StartPos, NewPos, EndPos, SetPos;

VAR
    fileOpen : BOOLEAN; (* to know what the status is *)

TYPE
    Name = ARRAY [0 .. 20] OF CHAR;
    Item =
        RECORD
            name : Name;
            price : REAL;
            quantity : CARDINAL;
            bin : CHAR;
        END; (* Item *)

VAR
    emptyItem, currentItem : Item;
    Ok : BOOLEAN;
    dataChan : ChanId;
    res : OpenResults;

(* Put same display procedures here *)

VAR
    numItems : CARDINAL; (* global *)

PROCEDURE ListItems; (* list all items *)
(* Pre : file is open *)
VAR
    count : CARDINAL;
BEGIN
    SetPos (dataChan, StartPos (dataChan));
    count := 0;
    REPEAT
        Read (dataChan, currentItem);
        IF IOResult.ReadResult (dataChan) = allRight
            THEN
                WriteCard (count+1, 1);
                WriteString (". ");
                DisplayName (currentItem.name);
                WriteLn;
                INC (count);
            END;
        UNTIL IOResult.ReadResult (dataChan) # allRight;
    numItems := count;
END ListItems;

PROCEDURE PutItem (item: Item);
(* Write out item at current file position *)
BEGIN
    Write (dataChan, item);
END PutItem;

```

```

PROCEDURE FetchItem (itemNum: CARDINAL; VAR item : Item);
(* Obtain that item number in the file
Assume program numbering 1 ... and file numbering 0... *)

VAR
    pos : FilePos;
BEGIN
    pos := NewPos (dataChan, itemNum-1, SIZE (Item), StartPos (dataChan));
    SetPos (dataChan, pos);
    Read (dataChan, item);
    SetPos (dataChan, pos);
END FetchItem;

PROCEDURE EditItem (VAR item : Item); (* change contents *)
(* same as last time, but omit fileDirty lines *)

PROCEDURE GetFile;
(* Open or create Inventory.data *)
BEGIN
    OpenOld (dataChan, "InventoryData", raw+write+read, res);
    IF (res = opened)
        THEN
            fileOpen := TRUE;
            SetPos (dataChan, StartPos(dataChan))
        END;
END GetFile;

PROCEDURE CloseFile;
(* close file.  If file not already open, does nothing. *)
VAR
    count : CARDINAL;
BEGIN
    IF NOT fileOpen
        THEN
            WriteString ("No file open");
            WriteLn;
            RETURN;
        END;
    Close (dataChan);
    fileOpen := FALSE;
    numItems := 0;
END CloseFile;

PROCEDURE AddItem;
(* make an empty item, and then edit it. *)
VAR
    temp : Item;
BEGIN
    IF NOT fileOpen
        THEN
            GetFile;
        ELSE

```

```

        SetPos (dataChan, EndPos (dataChan));
    END;
    temp := emptyItem;
    EditItem (temp);
END AddItem;

```

```

PROCEDURE GetItem (VAR itemNum : CARDINAL);
(* Find out which one to deal with *)

```

```

BEGIN
    IF fileOpen
    THEN
        REPEAT
            ListItems;
            WriteString ("Pick one 1 .. "); WriteCard (numItems,1);
            WriteString (" ==");
            ReadCard (itemNum);
            SkipLine;
            Ok := (ReadResult() = allRight) AND (itemNum <= numItems);
            IF NOT Ok
            THEN
                WriteString ("Error in selection; try again");
                WriteLn;
            END;
        UNTIL Ok;
    ELSE
        WriteString ("File not open");
        WriteLn;WriteLn;
        itemNum := 0;
    END;

```

```

END GetItem;

```

```

PROCEDURE Menu (VAR menuNum : CARDINAL);

```

```

(* identical to last version *)

```

```

END Menu;

```

```

VAR (* main *)
    action, itemNum : CARDINAL;

```

```

BEGIN
    numItems := 0;
    fileOpen := FALSE;
    (* make a default or empty item for later editing *)
    WITH emptyItem
        DO (* initialize one *)
            name := "";
            price := 0.0;
            quantity := 0;
            bin := ""; (* nowhere *)
        END; (* with *)
    WriteString ("    Rick's Inventory 2 program");

```

```
WriteLn;WriteLn;
```

REPEAT

```
Menu (action); (* print menu + get action *)  
(* take action according to request from menu *)  
IF (action = 1) AND NOT fileOpen
```

THEN

```
    GetFile;
```

ELSIF (action = 2) **THEN**

```
    GetItem (itemNum);
```

```
    IF itemNum # 0
```

THEN

```
        FetchItem(itemNum, currentItem);
```

```
        DisplayItem (currentItem);
```

END;

ELSIF (action = 3) **THEN**

```
    AddItem;
```

ELSIF (action = 4) **THEN**

```
    GetItem (itemNum);
```

```
    IF itemNum # 0
```

THEN

```
        FetchItem(itemNum, currentItem);
```

```
        EditItem (currentItem);
```

END;

ELSIF (action = 5) **THEN**

```
    CloseFile;
```

ELSIF (action = 6) **THEN**

```
    IF fileOpen
```

THEN

```
        CloseFile;
```

END;

END;

```
UNTIL action = 6 (* and then quietly exit *)
```

```
END RndInventory.
```

Observe that only one item is kept in memory at a time, and that after it is read the file position is set back to its starting point so that if editing is done, the item will be written back to the correct location. Much of the other logic remains the same, but new procedures have been inserted to fetch an item by number and to return it to the disk.

[Contents](#)

9.13 Chapter Summary

This chapter covered these topics:

- a review of structured data
- SET structures
- set membership, set construction
- set union, intersection, difference, and symmetric difference
- bitsets and packedsets
- the RECORD structure
- how to fill records
- how to use the WITH statement to unqualify record field names
- when to use a record or an array
- arrays of records
- storing records as streams of text, of records, and of binary data
- storing records in random access files

It included discussion of the following Modula-2 built-ins:

Imports

Library Identifiers:

- Standard:
 - SYSTEM:
BITSPERLOC, SHIFT, ROTATE
 - ChanConsts:
ChanFlags, ChanFlags, readFlag, writeFlag, oldFlag, textFlag, rawFlag,
interactiveFlag, echoFlag , FlagSet , read, write, old, text, raw, interactive, echo
 - StreamFile:
FlagSet
 - RndFile:
ChanId, FlagSet, OpenResults, read, write, old, text, raw, OpenOld, OpenClean,
IsRndFile, IsRndFileException, Close, FilePosSize, FilePos, StartPos, CurrentPos,
EndPos, NewPos, SetPos.
 - Non-Standard:
 - Files/Filer/FileSystem:
ReadRec, WriteRec, ReadBytes, WriteBytes.
-

Contents

Reserved Words	Standard Identifiers	Symbols
SET	BITSET	+
OF	EXCL	-
IN	INCL	*
RECORD	PACKEDSET	/
WITH	SIZE	{ }
		=
		<<=
		>=

9.14 Exercises

Questions

1. What is the difference between a Modula-2 set and an abstract set?
2. What is the difference between a subrange of a scalar data type and a subset of a set type? Illustrate with an example of each.
3. The problem is to write a payroll problem. The monthly salaries are fixed, but daily salaries (for computing overtime) depend on the number of days in the month. Write out syntactically correct declarations for sets to contain month names. Months with the same number of days should be grouped into common sets.
4. Explain why sets were of limited use in older versions of Modula-2, and say what the ISO requirement is that makes them more useful.
5. Which ISO Modula-2 libraries that have been used so far export sets, and why are these items all compatible even though they are imported from different modules?
6. Name all the operations and operators that can be used on sets and explain their meaning.
7. What are packedsets and bitsets, and what procedures are there for manipulating them, either in the language proper, or in library modules?
8. What arithmetic operation is shifting one bit left equivalent to? One bit right? N bits left or right?
9. Assuming that the underlying base type is CharSet = SET OF CHAR or NumSet = SET OF [0..10] as appropriate for each question, work out the answers to the following Modula-2 set expressions: (Be sure to indicate any errors and explain)
 - a) NumSet{1,2,5,7} * NumSet{1,3,4,6} + NumSet{3,1,5}
 - b) NumSet{3,4,6,2} / NumSet{1,3,5,6} - NumSet{2}
 - c) NumSet{1,2} + NumSet{3,4} * NumSet{4,5} / NumSet{1} - NumSet{1}
 - d) CharSet{'a','b','2'} - CharSet{'b','c','d'} / CharSet{'c','d'} + CharSet{'2'}
 - e) NumSet{10,5,1} / NumSet{8,5,2} * NumSet{9,1,10}
 - f) CharSet{'k','f','j','n'} - CharSet{'t','j','n'} / CharSet{'t'} + {CharSet{'3','4','m'} * CharSet{'2','12','m'}}
 - g) NumSet{9,7,2} + NumSet{7,6,5} * NumSet{3,4,5} - NumSet{2}
 - h) NumSet{1,4,8} / NumSet{1,2,3} - NumSet{2,3}
 - i) CharSet{'r','g'} * CharSet{'1','2'} + CharSet{'y'}
 - j) CharSet{2, 3, 4} + CharSet{a, b, c}
10. Evaluate the following BOOLEAN expressions, if all the sets are of type Num.
 - a) (5 IN Num{1,3,5}) AND (Num{1,3} <= Num{1,3})
 - b) (Num{2,4,7} * CARDINAL * [0..100] * REAL.
15. Write a Modula-2 procedure to fill the fields of the above record from data typed in.
16. Give an example of a record contained within a record, and the code to fill its fields.
17. When should you use a record and an array? Give specific examples of real world data suitable for modeling by each.
18. What is a "qualified identifier" and under what circumstances does it arise in Modula-2? How is it unqualified in each case?
19. Describe three ways of storing records in files.
20. What is a random access file, and which module is used in ISO Modula-2 to implement this model?
21. Under what circumstances should you employ sequential files, and what circumstances should you employ random access files?
22. What is a file position variable? What procedures can manipulate this variable?
23. What other marker is important in manipulating random access files?
24. What is the difference between *OpenOld* and *OpenClean*?
25. If you read a record from a random access file, edit it, and then immediately write the record back, the file will be incorrect. Explain. What step is missing?

26. What happens if you attempt to position the read/write file to a place beyond the actual limits of the file?
27. Make a chart detailing the meanings of all the flags in the device drivers available in your system. (Some or all of *StreamFile*, *SeqFile*, *RndFile*, and *TermFile* should be included.)
28. Demonstrate that you understand the inventory example by producing a complete set of planning documents for it.
29. Further demonstrate this understanding by fully commenting the second version in [9.12.2](#)

Problems

30. Many previous programs have involved asking for keyboard input before proceeding. A typical case is one that prints a prompt

```
Do you want to do another (Y/N)? == "here ==" and then does a ReadChar and SkipLine,
and checks the answer against validAnswers
Post: If it is valid, WaitAnsOK returns TRUE. If not,
it prints an error message and asks the user to try again, then
repeats the prompt. It is only to do this retrys times,
however, before giving up and returning FALSE. *)
```

31. Write a program that will scan an input line of text and then print out all consonants and all vowels (separately, with headings) that have been found in the line.
32. Modify the [syllable counting program](#) in section 9.6 to improve the algorithm for determining what is or is not a syllable.
33. Modify the module [BitsetDemo](#) in section 9.5 to print out the contents of REAL numbers. Attempt to deduce the storage format of this type. After you give up, see if you can learn it from the documentation of your system. WARNING: This could take some research.
34. Re-implement the module [Points](#) in Chapter 6 using a record to represent the basic data type.
35. Implement the module [Vectors](#) in Chapter 7.11 using a record to represent the basic data type.
36. Modify the module [Inventory](#) in section 9.12 to employ a SET OF ["1" .. "6"] in checking for the correct range of answers to the menu prompt.
37. Write and test a module to enter data for a collection of people that is in the following categories into a suitable structure: name, height, mass, sex, hair colour, eye colour, church affiliation. It is not necessary to sort by name.
38. Now add a section to take someone specified at the keyboard (list them all and give a choice?) and search the rest of the list for the person of the opposite sex who matches in the most categories. (This kind of program has interesting possibilities.) Allow for a reasonable range in height and mass for a match.
39. A class of students, which the user may enter in alphabetical (or other) order (no sorting) has been given four marks expressed as whole number percentages. Write a program module to:
 - a) Enter all this data into a record. The record has to be able to hold a letter grade as well.
 - b) Find the average grade.
 - c) Find the median (middle) grade.
 - d) Scale the marks so that the class average is 70. (Do this by simply adding the difference 70 - Actual Average to every grade.)
 - e) Assign letter grades to the scaled marks according to: 86+ (A), 73+ (B), 54+ (C), 40+ (D), 39- (F).
 - f) Count the number of letter grades in each category.
 - g) Print out a summary of the results in parts b, c, e, and f.
40. Write a module to keep track of a chequebook. It will need to have the ability to read numbers in from the keyboard, tell the program whether the transaction is a cheque, or a deposit, update the balance, and record both the transaction and the new balance in a disk file for future reference. When the program is executed the first time, it should ask for the prior balance. On subsequent executions, it should read this from the disk file.
41. Add to the facilities in the previous question the ability to print out from the disk records the entire stored transaction history from the data file, along with the balance after each transaction.

42. Add to the module [Inventory](#) in section 9.12.1 the ability to delete a record from the data. Make other improvements that you think add to the usability, functionality and safety of this module. Detail any bugs you find in the process and document how you fixed them.
43. As in the previous question, modify the random access version of the inventory program. At least include the ability to delete an item. Observe that the file must be restructured and given a new end marker when this is done.
44. Make a different modification to the random access version of the inventory program, this time to do searching by having the user type the name of the item first and then find the item of that name in the disk file, rather than by scanning the disk file and printing all the names. When you have this working, modify the search capability further so as to present the user with a menu to search by item name or bin number before proceeding.
45. The Acme Pewter Tuning Fork Company makes four models of tuning fork (economy, standard, super, and deluxe). Each one can be made to play the notes A, B, C, D, E, F, or G, from a single octave and any of these could be sharp. Both wholesale and retail prices need to be recorded, as well as the current inventory stock. Because there are a fixed number of different items, it is not necessary to generate new items once the inventory is set up. The operations that are needed are
- a) search inventory for items low in stock (<10) to generate manufacturing orders
 - b) add newly manufactured items to inventory
 - c) fill wholesalers orders from stock
 - d) alter wholesale or retail pricing.
- Write an program module to keep track of all this activity. Use a random access file. (You can improve on the simplistic and slightly inaccurate musical notation if you know how.)
46. A Church address list needs to have the following information: Last name, first names, childrens' names, address, town, province, postal code, telephone number. The first name of anyone who is a member is to be printed (and may be stored) with an asterisk before it. Write a program to keep track of all this. You need to be able to add or delete records, and to edit any field.
47. A bibliography program is to store records of books used in writing papers. It must have the name of the author, title, place of publication, publisher, and year of publication. Items can be added or deleted, and any field can be edited. There should be a field to contain a mark that can be set or removed. When the bibliography is printed out, only the currently marked items should be printed (many sets of citations possible from one list) and citations should be written (without the italics) in the form:
- Granberg-Michaelson, Wesley, (ed.). *Tending the Garden--Essays on the Gospel and the Earth*. Grand Rapids, MI: Eerdmans, 1987
- Hofstadter, Douglas R. *Gödel, Escher, Bach: an Eternal Golden Braid*. New York: Basic Books, 1979
- Holmes, Arthur F. *All Truth is God's Truth*. Grand Rapids, MI: Eerdmans, 1977
48. The problem with using random access files with textual I/O is that the text fields are not all the same length (and therefore the records do not occupy the same amount of disk space). Devise a way to overcome this and rewrite one of the programs in this chapter (or one of the exercises) to employ your method. Carefully document the limitations of your technique.
-

Chapter 9

Structured Data--Intermediate Techniques

[9.0 Chapter Goals](#)

[9.1 Structured Data Revisited](#)

Part A--Sets

[9.2 Representation and Membership](#)

[9.2.1 Set Union](#)

[9.2.2 Set Intersection](#)

[9.2.3 Set Difference](#)

[9.2.4 Symmetric Set Difference](#)

[9.2.5 One Element at a Time](#)

[9.3 Set Comparisons](#)

[9.3.1 Set Equality and Inequality](#)

[9.3.2 Subset](#)

[9.3.3 Superset](#)

[9.4 Sets and the I/O Library](#)

[9.5 Sets at the Low Level](#)

[9.5.1 Bitsets](#)

[9.5.2 Packed Sets](#)

[9.6 An Extended Example](#)

Part B--Records

[9.7 Declaring and Assigning to Records](#)

[9.8 Using Records](#)

[9.9 Records and Arrays--Which, or Both?](#)

[9.10 Getting Physical With Records](#)

[9.10.1 RawIO in Non-Standard Modula-2](#)

[9.11 Records and Random Access Files](#)

[9.12 An Extended Example \(Inventory\)](#)

[9.12.1 Inventory with Raw Sequential I/O](#)

[9.12.2 Inventory with Raw Random I/O](#)

[9.13 Chapter Summary](#)

[9.14 Exercises](#)

[Contents](#)

10.0 Chapter Goals

The purpose of this chapter is to elaborate on a number of technical points related to topics covered in earlier chapters. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

No new data representation techniques are discussed in this chapter.

Data Manipulation Abstractions

General:

No new data manipulation techniques are discussed in this chapter except as side effects of programming methods.

Programming Abstractions

General:

advanced scope issues, transfer of control, error handling and exceptions, program finalization

Realized in the Modula-2 notation:

scope and blocks, modules, standard identifiers, opaque types, RETURN and LOOP, FINALLY clauses, HALT, Modula-2 exceptions employed in the language, in standard libraries, and in user programs

10.1 Introduction

A number of loose ends remain from previous sections of the text. As these are of a somewhat technical nature, they were not critical to the introductory material. However, in order to make further progress, it is necessary to consider some of the finer points relating to such matters as scope, some alternate mechanisms for program control, and some error handling methods.

[Contents](#)

Part A--Scope and Visibility Issues

10.2 Blocks, Global and Local Variables, Side Effects

Notations like Modula-2 and Pascal are sometimes called *block-structured languages*. What is meant by this is that code is organized or grouped into named sequences, called *blocks*. (In some languages, blocks do not need names, but they do in Modula-2). Both modules and procedures contain blocks, and these blocks provide the framework for organizing (structuring) a program. In general, the entire code following the line containing the name of the module or procedure is its main block, though subsidiary procedures inside it may contain other blocks.

A Modula-2 block consists of the declaration and body parts of a module or a procedure. It is declared by a heading that includes the block name. In the case of a procedure it also includes the parameter list. It contains

- (1) a declaration section--definitions of entities to be used inside the block*
- (2) a statement sequence known as the block body*

```
BEGIN
    statement sequence
END <block name>
```

As already seen, blocks must have names in Modula-2, and the identifier must be named in the first line (right after the reserved word MODULE or PROCEDURE, as the case may be), and also on the last line, right after the reserved word END.

A procedure may legally be written without a body:

```
PROCEDURE Disembodied;
END Disembodied;
```

but it would hardly be of any use. However, it may well happen that one would write an implementation part of a library module with no body, for it may have no need to do initializations of internal or exported items. The definition part of a library module is not permitted to contain a body.

A more substantial difference between modules and procedures is that the headings of procedures include parameter lists, and this is not applicable to the headings of modules. Module blocks, on the other hand may be preceded in the module by import lists, and these are not relevant to procedures.

The general purpose of blocks is to assist in structuring code for ease of organization. They also assist in segregating variables and other entities for use in specific parts of a program and, to contain code that may be used repeatedly in various contexts.

10.2.1 Procedure Blocks and Scope

As indicated in chapter three, any variables, constants, or other entities declared in a procedure have no existence whatsoever outside the scope of that procedure. Any attempt to refer to such identifiers in the main program will result in an *identifier not declared error* or some similar message being delivered by the compiler.

On the other hand, any variables that are declared in the scope surrounding a procedure--whether it be the main program module or another procedure--are available to all the procedures contained inside that scope. This is true at every level, so that if one had:

```
MODULE Visible;

VAR
  firstReal : REAL;

  PROCEDURE DoOne;

  VAR
    secondReal : REAL;

    PROCEDURE DoTwo;

    VAR
      thirdReal : REAL;

    BEGIN
      (* Body of DoTwo *)
    END DoTwo;

  BEGIN
    (* Body of DoOne *)
  END DoOne;

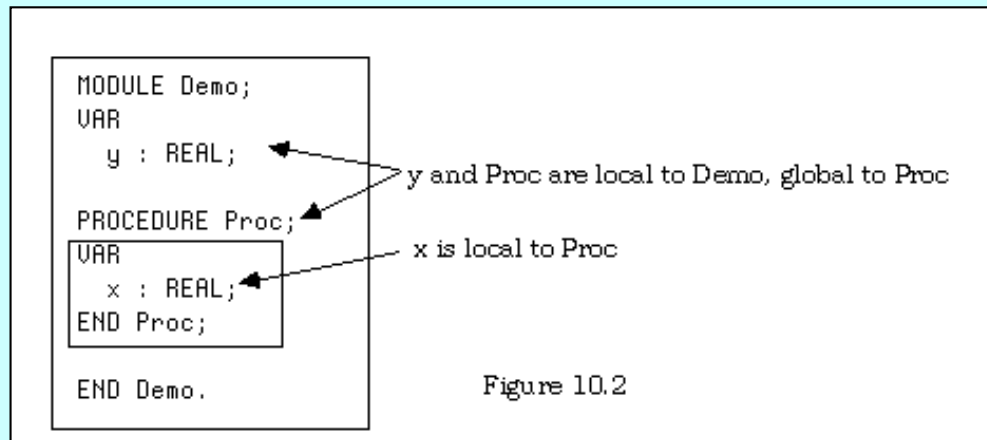
BEGIN
  (* Body of Visible *)
END Visible.
```

then the procedure *DoTwo* can use *DoOne* and all three of *firstReal*, *secondReal*, and *thirdReal*. procedure *DoOne* can use itself, *DoTwo*, *firstReal* and *secondReal*, but not *thirdReal*. The main program module can use only *DoOne* and just *firstReal* of the variables.

As far as any statement in the main module is concerned, the entities *DoTwo*, *secondReal* and *thirdReal* do not even exist. Whenever a procedure is entered, its own particular variables (whether formal parameters or declared in a VAR statement) come into existence and can have values assigned to them. As soon as this section is exited they cease to exist altogether, and the memory locations that they named are reclaimed for other uses by the program. The following summarizes these concepts in a couple of formal definitions :

An entity named in the parameter list of a procedure or in the declaration part of its block is said to be local to that block. The same entity is global to any block inside the procedure in which it is declared.

Thus in the module *Visible*, *secondReal* is local to *DoOne* and global to *DoTwo*. Some entities are more local than others. In fact, it is technically more correct to assert that Modula-2 does not have entirely global entities, just ones that are local to the outer block, because a program module executes in exactly the same manner as does one of its procedures. Perhaps it is best to refer to the surroundings and observe that an entity is local in the block in which it is declared, and global to any block contained within that. This is illustrated in figure 10.2.



The concept of locality applies not just to variables, but to constants, modules, procedures, and other entities as well. Clearly, it is important to know where in a program a given entity is in fact usable. Consider the following definitions:

If an entity (variable, constant, procedure, etc.) is available for use in a statement in some portion of a program, one says that it is visible in that portion of the program.

The scope of an entity is the portion of a program in which it is visible. Modules and procedures both define a scope for their entities, and an entity is visible in any procedure scope that is contained inside the one in which it is defined.

NOTES: 1. An entity is visible throughout the entire scope within which it is defined, not just from the point at which it is declared onward, as in some versions of Pascal. This characteristic of Modula-2 may have been sacrificed if the implementation you are using has a one-pass compiler.

2. The scope of a procedure starts immediately *after* its own name, and includes its formal parameters.

10.2.2 Side effects and Counting Loops

In general, it is a poor programming practice to declare all of a program's variables globally to the entire program. The larger the program is, the more likely this is to cause difficulties in keeping track of all the variables and their use. Failure to do so may result in writing code that modifies the value of a variable that is

important to the correct functioning of some other part of the program. This difficulty increases approximately with the square of the size of the program, and tends to be particularly acute with procedures, because these usually have variables of their own, and may be written long after the main program at a time when the creator of the original code has either departed her employment or forgotten what its variables were for. Given this, it is the product of a moment's carelessness or ignorance to write a procedure that, besides its intended action, also modifies some important program variable.

The modification by a procedure of some variable global to it is called a side effect of the procedure.

Observe that some side effects are desirable, and may furnish the very reason for writing the procedure in the first place. For instance, an invocation of `ReadChar(ch)` is made for the express purpose of changing the value of global variable `ch`. On the other hand, some side effects are quite deleterious. For instance, novice programmers, finding that they have many procedures containing counting loops such as:

```
WHILE count < 10
  DO
...
    INC (count)
  END;
```

may be tempted to declare a single variable `count` at the outermost level of a program and allow it to be shared by all procedures.

If this is done, then whatever the declared purpose of one of these procedures, its action has the additional side effect of changing `count`. On some occasions, this is not be a problem, but what if one of these procedures calls another?

```
VAR
  count: CARDINAL;
PROCEDURE DoOne ( );
BEGIN
  count := 1;
  WHILE count < 4
    DO
      DoTwo;
      INC (count);
    END (* while *)
  END DoOne;
```

```
PROCEDURE DoTwo ( );
BEGIN
  count := 12;
  WHILE count --without performing the rest of its loop. This would be a difficult
  bug to locate, especially if the code for the two procedures were separated in the
  source file.
```

If the variable is the loop control variable in a FOR statement, this code would not be allowed, because such loop control variables must be declared in the *same* scope as they are used. That is, code such as

```
MODULE TestFor;

VAR
    count : CARDINAL; (* unused in code shown, but available. *)

PROCEDURE One;

BEGIN
    FOR count := 1 TO 10
        DO
            END;
    END One;

PROCEDURE Two;

BEGIN
    FOR count := 1 TO 10
        DO
            One;
        END;
    END Two;

END TestFor.
```

should produce error messages like

```
#    9  FOR count := 1 TO 10
#####      ^ 205: illegal FOR variable
File "TestFor.MOD"; Line 9
#   17  FOR count := 1 TO 10
#####      ^ 205: illegal FOR variable
File "TestFor.MOD"; Line 17
```

This is because it is an error to *threaten* the value of a loop control variable in a FOR loop, and this code certainly does that. Note, however, that some compilers do not take this error checking as seriously as they should, and may incorrectly ignore this error. The correct way to write such code is:

```
MODULE CorrectFor;

VAR
    count : CARDINAL; (* unused in code shown, but available *)

PROCEDURE One;
VAR
    count : CARDINAL;
```

```

BEGIN
FOR count := 1 TO 10
  DO
    END;
END One;

PROCEDURE Two;
VAR
  count : CARDINAL;

BEGIN
FOR count := 1 TO 10
  DO
    One;
  END;
END Two;

END CorrectFor.

```

Notice that here each procedure has its own variable *count*. A procedure may legally re-use an identifier that already exists in the surrounding scope. A variable called *count* in the main program, and one called *count* in a procedure contained inside the program, are different entities--they name different cells of the computer's memory. In this example, all three items called *count* have their own memory, independent of the others, and so do not interfere with the others.

The only effect this will have is that the original variable will not now be visible in the inner procedure. The name is, after all, being used locally. So, the multiple use of *count* causes no conflicts--all are distinct.

The *non-threatening* rule only applies to FOR loops, not to WHILE or REPEAT loops, but the side effect problem can be fixed the same way, by declaring variables that are local to the scope containing the loop.

The point is that side effects can easily destroy the validity of a program, and that such side effects should therefore be avoided. It follows that the best way to use procedures is to construct them as *black boxes* that take in certain input, produce well-defined output and do not otherwise affect the surrounding program through their inner workings.

10.2.3 Other Global side effects

Apart from the damage done to loop control variables, there are other deleterious side effects involving local modification of global variables. To illustrate, consider the following pathological example:

```

MODULE BadSideEffect;
FROM STextIO IMPORT
  WriteLn;
FROM SWholeIO IMPORT
  WriteCard;

CONST

```

```

    two = 2;
VAR
    result, global : CARDINAL;

PROCEDURE Func (local : CARDINAL) : CARDINAL;
BEGIN
    local := global + local;
    INC (global);
    RETURN local;
END Func;

BEGIN (* main *)
    global := 5;
    result := Func (two) + Func (two);
    WriteCard (result, 0);
    WriteLn;
    global := 5;
    result := 2 * Func (two);
    WriteCard (result, 0);
    WriteLn;
END BadSideEffect.

```

When this program was run, the output was

```

15
14

```

Whoops! Surely it ought to be the case that any function procedure has the property that

$\text{Func}(\text{two}) + \text{Func}(\text{two})$ equals $2 * \text{Func}(\text{two})$

as it does in most reasonable mathematics, but here it does not. Here's why. When the statement

```
result := Func (two) + Func (two)
```

is executed, the procedure is entered twice. On first entry, *global* has the value 5, and on first exit the value returned by *Func* is 7 and that of *global* is now 6. The second call returns the function value 8. The sum of the two function results is 15. On the other hand, when the statement

```
result := 2 * Func (two)
```

is executed with *global* first set back to 5, the procedure is entered only once, returning 7; this is doubled; and 14 is output.

The example further illustrates that procedures should modify only local variables, not global ones. It also illustrates that a piece of code must be read for its *meaning* rather than simply for its *form* in order to understand the effect of

executing that code.

Here are two suggestions for avoiding deleterious side effects of this kind:

1: Avoid manipulating global variables within local routines if possible. Pass more parameters instead.

2: Declare only those variables as global that are necessary. If only a few procedures will use a variable, it may not belong in the outer block.

3: Re-use names where appropriate; the compiler will know to employ a different memory cell.

10.2.4 Nested Procedure Scopes

As noted in passing above, when a name is re-used in an inner (nested) scope, all references to that name in the inner scope are to the entity declared in that scope, not to the global entity of the same name. The re-use of the name *cuts off* the access to the global entity of the same name. Here is another illustration:

```
MODULE Nested;
FROM STextIO IMPORT
    WriteLn;
FROM SRealIO IMPORT
    WriteReal;
VAR
    number1, number2 : REAL;

PROCEDURE DoOne;
VAR
    number1 : REAL;

    PROCEDURE DoTwo;
    VAR
        number1, number2 : REAL;
    BEGIN      (* for DoTwo *)
        number1 := 1.0;  (* no change to number1 in DoOne or Nested *)
        number2 := 2.0;  (* doesn't affect number2 in main program *)
    END DoTwo;

BEGIN      (* for DoOne *)
    number1 := 3.0;  (* no effect on number1 in Main program *)
    number2 := 4.0;  (* does affect number2 in main program *)
    DoTwo;
END DoOne;

BEGIN
```

```
number1 := 5.0;  
number2 := 6.0;  
DoOne;  
WriteReal (number1, 5);  
WriteLn;  
WriteReal (number2, 5)  
END Nested.
```

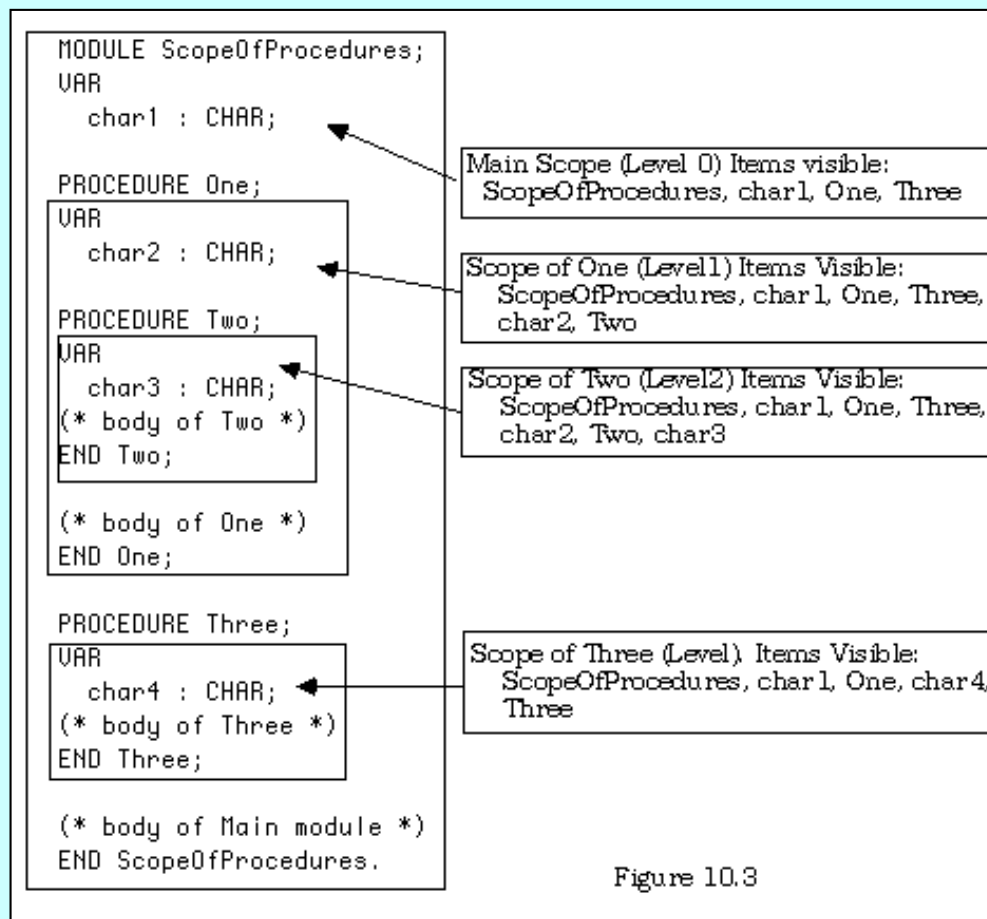
The output for the above module is

```
5.000  
4.000
```

for *number1* and *number2* respectively, for when an assignment is made to a variable inside a procedure, it will be done with the locally defined entity of that name. No search outside the procedure will be instituted unless the identifier cannot found locally.

*An entity declared in the main module is said to be at the outermost level, or level zero. An entity declared in a procedure belonging to the main module (one scope inside the main one) is at level one. An entity declared in a procedure nested *n* scopes inside the main scope is at level *n*.*

Naturally, if two procedures are declared at the same level, they cannot make any use of each other's entities--including any procedures hidden inside each other. Figure 10.3 illustrates:



NOTE: There may be an implementation specific restrictions on the use of the procedure *Three* inside the procedure *One* because of declare-it-before-you-use-it restrictions. If that is the case, declare *Three* FORWARD as shown in [section 4.8](#).

To illustrate the use of nested procedures, consider this little program that draws bar graphs on up to three quantities and up to 30 units wide. Observe the multiple use in different scopes of the counting variable and the use of subprocedures by a main procedure that do not need to be visible to the entire program (and so are not).

```

MODULE BarGraph;
(* by R. Sutcliffe
To illustrate nested procedures and scope
revised 1994 04 06 *)

FROM STextIO IMPORT
    WriteChar, WriteString, ReadChar, SkipLine, WriteLn;
FROM SWholeIO IMPORT
    ReadCard, WriteCard;
FROM SIOResult IMPORT
    ReadResult, ReadResults;

PROCEDURE DrawBar (wide : CARDINAL; desc : CHAR);
(* This procedure draws the entire bar using two subprocedures. *)

PROCEDURE MakeSide (howWide : CARDINAL);
(* This subprocedure makes one side of the bar. *)

VAR
    count : CARDINAL;
BEGIN

```

```

WriteChar (" ");
FOR count := 1 TO howWide-1
    DO
        WriteChar ("-");
    END; (* if *)
END MakeSide;

PROCEDURE MakeEnds (howWide : CARDINAL);
VAR
    count : CARDINAL;
BEGIN
    WriteChar ("|");
    FOR count := 2 TO howWide
        DO
            WriteChar (" ");
        END; (* for *)
    WriteChar ("|");
END MakeEnds;

BEGIN (* Main DrawBar procedure *)
    MakeSide (wide);
    WriteLn;
    MakeEnds (wide);
    WriteChar (desc);
    WriteLn;
    MakeSide (wide);
    WriteLn;
END DrawBar;

PROCEDURE GetInfo (VAR size : CARDINAL; VAR desc : CHAR);
VAR
    done : BOOLEAN;
BEGIN
    REPEAT (* for Range checking *)
        ReadCard (size);
        done := (ReadResult () = allRight) AND (size <= 30);
    IF NOT done
        THEN (* bad read or bad info *)
            WriteLn;
            WriteString (" Invalid width input for a bar graph");
        END; (* if beyond range for graph size *)
    SkipLine;
    UNTIL ReadResult () = allRight;
    WriteString ("Please type in a one character description ==");
    ReadChar (desc);
    SkipLine;
    WriteLn;
END GetInfo;

CONST
    numItems = 3;
TYPE
    Bar =
        RECORD
            width : CARDINAL;

```



```

        info : CHAR;
    END;
VAR
    bars : ARRAY [1..numItems] OF Bar;
    count : CARDINAL;

BEGIN    (* Main Program is a simple test of procedures *)
    FOR count := 1 TO numItems
        DO
            WITH bars[count]
                DO
                    WriteString ("How many units wide is bar # ");
                    WriteCard (count, 1);
                    WriteString (" ==");
                    GetInfo (width, info);
                END;
            END;
    FOR count := 1 TO numItems
        DO
            WITH bars[count]
                DO
                    DrawBar (width, info);
                END;
            END;
    END BarGraph.

```

Here is a sample run from this program with the user inputs in bold type .

How many units wide is bar # 1 ==> **a**

How many units wide is bar # 2 ==> **b**

How many units wide is bar # 3 ==> **c**

```

-----
|                                     |a
-----
-----
|          |b
-----
-----
|                                     |c
-----

```

10.3 Parameters Revisited

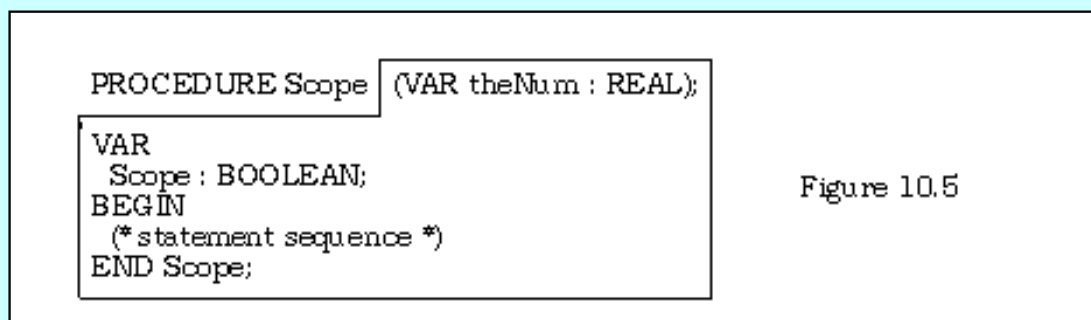
10.3.1 The Scope of Parameters

As a follow-up to the discussion of the last section, it is important to note that while the *name* of a procedure itself is part of the scope surrounding it, the parameters of a procedure are part of the procedure's own scope. The types of those parameters, however, must be available in the surrounding scope. These rules have several effects:

1. The name of the procedure is available for use in its own scope because of the inheritance rule. This makes recursion possible.
 2. The name of a procedure may be redefined inside the procedure, because it belongs to the outer scope.
 3. The names of parameters cannot be re-used as variables within the procedure, because they do belong to its scope.
 4. The parameters' types cannot be defined in the procedure itself; these types must be resolved in the surrounding scope.
- That is to say, the following is legal:

```
PROCEDURE Scope (VAR theNum : REAL);  
  
VAR  
    Scope : BOOLEAN;  
BEGIN  
    (* statement sequence *)  
END Scope;
```

and the scope diagram for a procedure, drawn to include this information about the parameters, would look like figure 10.5.



It is probably not very useful to do such a redeclaration; the information is included here only to emphasize the location of the scope boundary.

On the other hand, the following two are both *illegal*:

```
PROCEDURE BadScopeA (VAR card : CARDINAL);  
  
VAR  
    card : CARDINAL; (* can't do this *)  
BEGIN  
    (* statement sequence *)  
END BadScopeA;
```

```

PROCEDURE BadScopeB (VAR theThingy : ThingyType);

TYPE
    ThingyType = ARRAY [0..7] OF BOOLEAN; (* can't do this either *)
BEGIN
    (* statement sequence *)
END BadScopeB;

```

10.3.2 Parameters and side effects

Now consider another example of the potential for harm done by a procedure having side effects, this time caused by the inappropriate use of variable parameters. Sometimes, such parameters are used when one does not really intend to generate a side effect, and a value parameter would have been more appropriate. To illustrate, consider the following pathological example wherein the action of a variable parameter is misused to produce unexpected side effects:

```

MODULE ValVarDemo;
FROM STextIO IMPORT
    WriteLn;
FROM SWholeIO IMPORT
    WriteCard;

PROCEDURE ValAdd (card1, card2: CARDINAL; VAR result: CARDINAL);
BEGIN
    card2 := card1 + 1;
    result := card1 + card2
END ValAdd;

PROCEDURE VarAdd (VAR card1, card2: CARDINAL; VAR result: CARDINAL);
BEGIN
    card2 := card1 + 1;
    result := card1 + card2
END VarAdd;

```

(* notice that the only difference between the two is the fact that the two value parameters in the first are declared as variable parameters in the second. *)

```

VAR      (* main program material declared here *)
    mainNum, answer: CARDINAL;
BEGIN
    (* first, do it with value parameters *)
    mainNum := 10;
    ValAdd (mainNum, mainNum, answer);
    WriteCard (answer, 3);
    WriteLn;

    (* now, repeat with variable parameters *)
    mainNum := 10;
    VarAdd (mainNum, mainNum, answer);
    WriteCard (answer, 3);
    WriteLn;

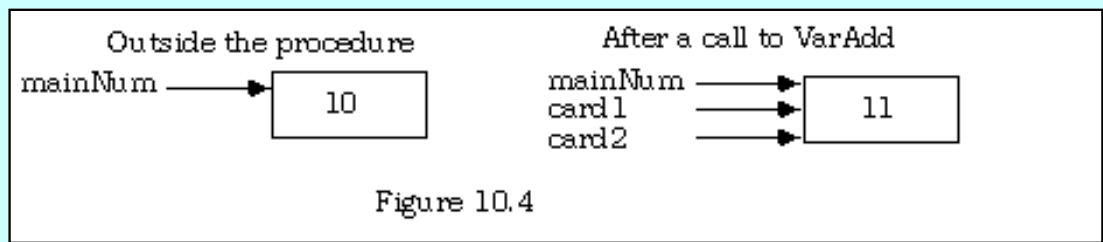
```

END ValVarDemo.

When this program is compiled and run, the output is:

21
22

The reason that the two answers are different is that when the second procedure is active, all three of *card1*, *card2*, and *mainNum* are names for the same memory location. (Recall that the action of a variable parameter is to *alias* the actual name to the same location named by the formal name.) Thus, when the first line in the procedure *VarAdd* changes the memory location named by *card2*, it does so for all three names (see figure 10.4), and the next line will add two copies of the value 11, rather than a 10 and an 11 as did the first procedure.



While this may seem to be an extreme example, it does show how the different nature of the two kinds of parameters affects the meaning of programs. Errors of this kind in real-life programs are likely to be much more subtle and difficult to find. This discussion also points to the rule of thumb:

Never use a variable parameter when a value parameter will do.

Note especially that function procedures are employed expressly to return a value to an expression; they can have no side effects at all--provided they have no variable parameters. Indeed, they should not, for there is no way to abandon the evaluation of the expression in which the function appears in order to check that side effect. For instance, if one has the syntactically legal.

```
PROCEDURE DoIt (a : REAL; VAR success: BOOLEAN) : REAL;
```

it is impossible to check on the (presumed) error result during a call like

```
x := DoIt (23.5, ok);
```

Likewise, the expression

```
a := Funct (x, y) + y;
```

if *y* is a variable parameter, may result in a change to *y* before the second instance of *y* in the expression is evaluated. This is unlikely to produce the expected result, and certainly makes the meaning of this line of code hard to understand. (Its meaning depends not on its own form, but on the inclusion of the variable parameter in the definition, and the logic of the execution, and this is not what is expected of functions.) Thus, the rules of thumb:

Never use a variable parameter in a function procedure.

Use function procedures in preference to regular procedures with variable parameters.

Contents

10.4 Procedure Types and their Variables

Modula-2 allows almost anything to be a variable type. (Exceptions include modules and types themselves). It is not by any means unique in this respect, but shares this ability with several other languages. What is allowed in Modula-2, though, and not in many others, is the assignment of procedures to a variable. That is, variables can be of a procedure type. As before, this is done by specifying the existence of the type first, and in this case, the parameter list types also form part of the type declaration.

There are several possible uses for this particular facility. One of the simplest is to accommodate the differences in the names of imported procedures in various versions. Suppose, for instance, one were to write a large program that contained the line:

```
FROM STextIO IMPORT
  WriteChar;
```

and there were several hundred references to *WriteChar* throughout the program. Later, it was desired to run the same program on a different machine where there were a similar module and procedure, but named *InOut* and *Write*, respectively, with *InOut.Write* having the same semantics as *STextIO.WriteChar*. The line above must be revised to:

```
FROM InOut IMPORT
  Write;
```

However, that is not enough. There are still all those references to *WriteChar* scattered through the program. For the program to work correctly (or even compile), one must evidently change them all to *Write*. But, how to ensure they are all done? And, how to change them all back again if, after further development and change, the first machine is to get the benefit of the changes as well? If the entity in question were a variable, all one would have to do is place an assignment statement at the beginning of the program. In Modula-2, one can do the same thing with procedures. All that is necessary is the declarations:

```
TYPE
  WriteProc = PROCEDURE (VAR CHAR);
VAR
  WriteChar : WriteProc;
```

to establish a variable of the appropriate procedure type, and the line of code:

```
WriteChar := Write;
```

somewhere near the start of the main program, and all will be well. Now, the calls to *WriteChar* are translated into calls to the procedure this variable equals. If it is necessary to port a new version of this second program back to the original system, the import line can be changed and the two lines of declaration and the assignment perhaps commented out. A second (more complex) example involves the assignment of procedures in order to improve the flow of a program itself. For instance, if one wishes to offer the user a choice of calculating the area and perimeter of either a square or a circle, one could offer a menu and then call one or the other of the procedures. Alternately, one can, using the facility here, set an appropriate procedure variable. This is realized below:

```
MODULE ProcVarDemo;
```

```

FROM STextIO IMPORT
    WriteString, WriteLn, ReadChar, SkipLine;
FROM SRealIO IMPORT
    ReadReal, WriteReal;

TYPE
    AreaPerim = PROCEDURE (REAL, VAR REAL, VAR REAL);

(* Specify only the types of the variables in the parameter list when declaring a
procedure Type. Every variable parameter must be individually specified as such *)

VAR
    SizeCalc : AreaPerim;
    dimension, area, perimeter : REAL;
    which : CHAR;

PROCEDURE Squares (side : REAL; VAR area, perim : REAL);
BEGIN
    area := side * side;
    perim := 4.0 * side;
END Squares;

PROCEDURE Circles (radius : REAL; VAR area, circ : REAL);

CONST
    pi = 3.14159265;
    twopi = 2.0 * pi;
BEGIN
    area := pi * radius * radius;
    circ := twopi * radius;
END Circles;

PROCEDURE GetNum ( ) : REAL;
VAR
    numToGet : REAL;
BEGIN
    WriteString ("Please type in the figure size here ==");
    ReadReal (numToGet);
    SkipLine;
    WriteLn;
    RETURN numToGet;
END GetNum;

BEGIN    (* Main program starts here. *)
    WriteString ("I will calculate the area and perimeter ");
    WriteLn;
    WriteString ("of either a circle or a square.");
    WriteLn;
    WriteString ("Type a 'C' or an 'S' here ===");
    ReadChar (which);
    SkipLine;
    WriteLn;
    (* assign procedure variable to correct procedure *)

```

```

IF CAP (which) = 'C'
  THEN
    SizeCalc := Circles;
  ELSE
    SizeCalc := Squares;
  END;  (* if *)

SizeCalc (GetNum( ), area, perimeter);
WriteString ("The area is ");
WriteReal (area, 15);
WriteLn;
WriteString ("And, the perimeter is ");
WriteReal (perimeter, 15);
WriteLn;
END ProcVarDemo.

```

One could of course leave out the TYPE part of the declaration and simply write:

```

VAR
  SizeCalc : PROCEDURE (REAL, VAR REAL, VAR REAL);

```

but as usual it is better to create a new type and then to use that type when declaring variables. This practice makes program modifications simpler and the text file becomes more readable and easier to follow.

Observe that the parameter type for each formal parameter must be separately specified, and the VAR must be indicated for each separately.

Modula-2 anticipates that procedure variables will be frequently used, and even has a built-in type PROC that denotes a parameterless procedure. (There are too many possibilities for procedure types that *do* take parameters to have any other procedure types built in.)

One can therefore assume that the language already has a definition equivalent to:

```

TYPE
  PROC = PROCEDURE ( );    (* no parameters *)

```

In other words, one can write

```

VAR
  Proc1, Proc2, Proc3 : PROC;

```

instead of having to write:

```

VAR
  Proc1, Proc2, Proc3 : PROCEDURE ( );

```

There have not been very many parameterless procedures in this text to date; however, they do play an important role in coprocesses and coroutines in Modula-2, and will surface several times later in the book. It is also worth noting that the assignment of procedure variables is distinguished from their invocation (calling) by the absence of a parameter list in the former case and its presence in the latter. Thus

```

Proc := Proc2

```


is an assignment of the procedure *Proc2* to the procedure variable *Proc*, whereas

```
x := Proc2 (<actual parameters>)
```

is an invocation of the function procedure *Proc2*, and assigns the returned value to *x*.

This is true even if the parameter list is empty, so such empty parameter lists must always be given when the intention is to call the procedure, in order to avoid confusion. Thus, if *x* is a numeric variable, the statement

```
x := Proc2
```

would be interpreted as an attempt to assign the procedure *Proc2* to the variable *x* and a type compatibility error would be generated.

NOTES: 1. Only procedures that are declared at the outermost level of a compilation unit can be assigned to procedure variables, not ones that are declared inside other procedures at level one or higher.

2. Standard procedures, such as *ABS*, cannot be assigned, though one can get around this (if one is a good-for-nothing hacker like the author) by enclosing the standard procedure inside another one with a slightly different name, and assigning the name of the procedure variable the enclosing procedure rather than (directly) the standard one. It is rather unlikely that many people would need to exploit this loophole.

[Contents](#)

10.5 Local Modules--Scope and Visibility Rules

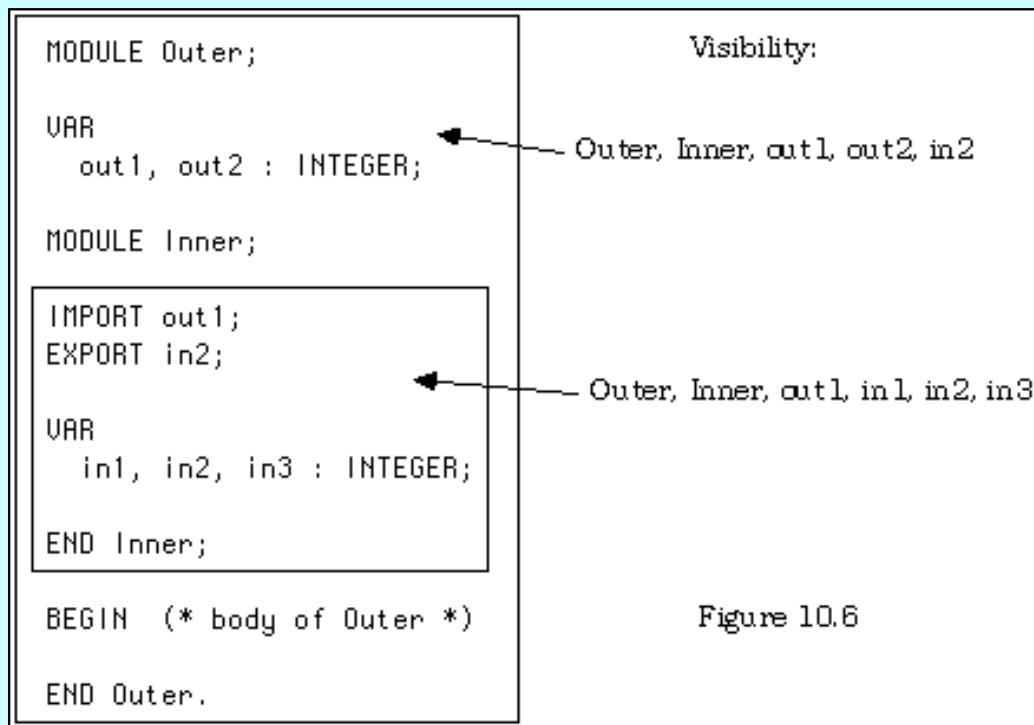
As mentioned in passing in [section 6.8](#), modules may themselves contain other (nested) modules and the collection consisting of these local modules together with the (global) program module is known as the program library.

A local module is any module entirely contained within some compilation unit.

Local modules resemble procedures in that they define a scope of visibility for the entities they contain. They do not, however have parameters, nor necessarily execute code. Such constants, variables or procedures may in turn be made visible in the scope of the surrounding module by naming them in an EXPORT statement. Conversely, they can be brought into a module from the surrounding scope by naming them in an IMPORT statement. It is important to note that module entities are not automatically visible in scopes created inside the ones in which they are declared as is the case for entities declared in or viewed from a procedure.

Visibility of entities declared in procedures is inherited hierarchically and automatically inward with increasing levels, but visibility of entities declared in modules must be explicitly and manually controlled.

Figure 10.6 illustrates the use of EXPORT and IMPORT in a simple case and lists the resulting visibilities in the two scopes.



As in the case of *Inner*, a module does not have to have a body with statements that actually execute (it need only be a collection of entities), but it does have to have an END even if there is no BEGIN. If it has a body, then that statement sequence will be executed at the time the main module containing it is activated. That is, at the same time that its entities come into existence, there is an opportunity to initialize them if the programmer so wishes. This applies to local as well as to library modules.

The body of a module may be referred to as its initialization section.

NOTE: Remember to initialize any module entities in its body if you are counting on them to have some value the first time they are called. This applies mainly to variables, including procedure variables, as they will be the ones used most commonly in this fashion.

It is important to realize that the wall erected around entities by a MODULE framework affects only the visibility of entities in the scopes on either side of the wall. It does not affect their existence between calls to procedures inside the module wall. Therefore, variables declared in a module (as opposed to one of its procedures) retain their values after the module is exited. This is quite different from the effect of declaring a procedure, for in that case variables declared inside are neither visible nor do they exist outside it. An entity is visible within procedure scopes created inside the one where it is defined, but not within module scopes created in this way, as the above example also illustrates. These rules apply to any combination of procedures and modules and the ones in which they are contained. See the next section for an illustration.

Observe that visibility is really only of concern at compile time, whereas existence is a run-time phenomenon. Since all variables declared at the outermost blocks of modules exist at run-time, one could even say that at run-time modules, unlike procedures, do not exist at all.

The net result of the module visibility rules is a further reduction in the number of global variables necessary, because the visibility of variables to any portion of the program can be strictly controlled with the IMPORT and EXPORT lists. This limits side effects, increases flexibility and improves the readability of programs. For instance, not only will a variable defined in a module keep its value until the next call to the module, it can be made visible outside the module without having to have (another) global name, whereas a procedure must pass a value to a global variable to communicate to the outer scope.

This can even be seen in the use of variables imported from library modules. One example is the boolean variable *Done* that is imported from the classical library *InOut*. One does not need to set and/or fetch the value of this variable whenever one does a read operation because its value is set by *InOut* itself whenever that module is called upon for an input operation. Consequently, the current value of *Done* is always available, once one has mentioned it in an import statement. On the other hand, the global availability of *Done* implies that it could be changed by other operations in the client program unrelated to read attempts, thus corrupting its meaning. That is why the ISO library makes available the enquiry function *ReadResult* to return a hidden error value instead. The hidden value cannot be changed outside the separate library module that owns it.

In any event, certain rules must be rigidly followed in the import and export of variables to and from modules. First, these lists must immediately follow the MODULE heading. There can be as many IMPORT statements as desired but only one EXPORT statement, and it must be the last of these lists.

```
MODULE MyModule;  
IMPORT variable1, variable2;  
IMPORT variable3;  
EXPORT myVariable1, myVariable2;  
  
(* declaration section comes next *)
```

Second, care must be taken to avoid an *identifier clash*. If two modules both export the same identifier into a common scope of visibility, then that entity will have been erroneously re-defined, and the compiler will report an error. For instance, module A and module B cannot both simply export a variable *number* into the surrounding module D, nor can module D import items named *number* into its own scope from two different sources.

To avoid such clashes when exporting, one can write:

```
MODULE Outer;
```

```
MODULE Show;  
EXPORT QUALIFIED item, number, sum;  
    etc.
```

When this is done, references in the surrounding scope must refer to the items by a *qualified identifier*. As in previous instances, this is written in the scope of module *Outer* as *Show.Item*, *Show.A*, etc. Now, a clash occurs only if one tries to give two modules the same name, and this problem should be easier to find and solve.

NOTE: Even if the EXPORT is not qualified, the entity *may* be referred to in the new scope with a qualified identifier, though this is unnecessary.

```
MODULE CheckModVisibility;  
  
MODULE Inner;  
EXPORT number;  
VAR  
    number : REAL;  
  
END Inner;  
  
BEGIN  
    number := 5.0;  
    Inner.number := 9.8; (* both references legal *)  
END CheckModVisibility.  
  
MODULE CheckModVisibility2;  
  
MODULE Insid1;  
EXPORT number1;  
VAR  
    number1 : REAL;  
  
BEGIN (* Only number1 is visible here *)  
    number1 := 5.4; (* number1 can be initialized here *)  
END Insid1;  
  
MODULE Inside2;  
IMPORT Insid1;  
VAR  
    number2 : REAL;  
  
BEGIN (* number2, number1 both visible here. *)  
    number1 := 3.4;  
    Insid1.number1 := 5.7;  
    number2 := 3.9  
END Inside2;  
  
BEGIN (* Test *)  
    (* only number1 visible here *)  
END CheckModVisibility2.
```

If an item is imported or exported, then some of its component names may be implicitly imported or exported as well. Implicit import was noted earlier in connection with the identifiers of an enumeration being made available in an importing scope.

The closure of an identifier is the set consisting of that identifier, together with any all identifiers that are implicitly exported or imported whenever that item is.

The identifiers of an enumeration are part of the closure of that enumeration. However, the exports of modules are *not* part of the closure of that module. Thus if an entire module is imported or exported, any of its exports are *not* automatically exported or imported as well. However, they can be referred to in the receiving scope of the exported or imported module in qualified fashion.

NOTES: 1. This is an ISO clarification, prior to which compilers (including those from ETH) followed a variety of rules in this regard.

2. The field names of a record are available when the record type is imported to another module scope but only qualified (unless unqualified by a WITH) so they are not, strictly speaking, part of the closure of the record type, though the net result is similar.

On the other hand, exporting (importing) a procedure does not make the types of the entities in its parameter lists visible in the receiving scope; these must be done separately. So if a module contains:

```
PROCEDURE Swap (VAR item1, item2 : ItemType);
```

then importing this procedure does not make *ItemType* visible in the receiving scope.

If the items of the exported module appeared in an EXPORT QUALIFIED list in a module being imported or exported as a whole, then they must be qualified in the receiving scope.

If the module is exported qualified, then it must itself be referred to in qualified fashion. In the following example, modules have been indented to better indicate their scope. This is not strictly necessary for good prettyprinting.

```
MODULE CheckModVisibility3;
```

```
  MODULE Shell1;
```

```
    EXPORT
```

```
      Inner1;
```

```
    MODULE Inner1;
```

```
      EXPORT thing;
```

```
      VAR
```

```
        thing: CARDINAL;
```

```
      END Inner1;
```

```
  END Shell1;
```

```
  MODULE Shell2;
```

```
    EXPORT Inner2;
```

```
      MODULE Inner2;
```

```
        EXPORT QUALIFIED thing;
```

```
        VAR
```

```
          thing: CARDINAL;
```

```
        END Inner2;
```

```

END Shell2;

MODULE Shell3;
EXPORT QUALIFIED Inner3;

    MODULE Inner3;
    EXPORT QUALIFIED thing;
    VAR
        thing: CARDINAL;
    END Inner3;

END Shell3;

```

```

(* here in this outer one, Inner1 is visible, so its exports may be qualified *)
(* Inner2 is visible, so its qualified exports may be qualified *)
(* however, Inner3 itself must be qualified *)
BEGIN
    Inner1.thing := 5;
    Inner2.thing := 4;
    Shell3.Inner3.thing := 8;
END CheckModVisibility3.

```

As this one illustrates, the export of the identifier *Inner1* from the scope determined by *Shell1* into the outermost scope also causes the exports of *Inner1* (namely *thing*) to become visible in the outermost module, but only qualified, not unqualified as some early compilers would have done.

The astute reader will probably realize by comparing previous experience with library modules that if a module's exports are QUALIFIED, then those identifiers can in turn be imported from the receiving scope (where they are qualified) into another scope by using

```

FROM modulename IMPORT item;

```

as in the following:

```

MODULE CheckModVisibility4; MODULE Inner1; EXPORT QUALIFIED thing; VAR thing: CARDINAL; END
Inner1; MODULE Inner2; FROM Inner1 IMPORT thing; BEGIN thing:= 5; (* available here now *) END Inner2;
END CheckModVisibility4.

```

Of course, one could not import (qualified or unqualified) the identifier *thing* into the outermost scope, because all imports to a scope are from the scope *outside* it. The only scope outside a program module is the universal scope that contains all the system and separate libraries, so all import statements in the main scope refer to the external libraries, not to the program library. An attempt to compile:

```

MODULE IllegalVisibility;

FROM Inner IMPORT (* illegal attempt to unqualify *)
    thing;

MODULE Inner;
EXPORT QUALIFIED thing;
VAR
    thing: CARDINAL;

```

```
END Inner;
```

```
END IllegalVisibility.
```

produced the compiler error message

```
No symfile found for module: Inner
--- could not open
---- symbolfiles missing
#      4      thing;
#####          ^   71: identifier not exported from qualifying module
File "RSFiles:Books:Modula
book:M2.3rdEdition:TextPrograms:Chal0:IllegalVisibility.MOD"; Line 4
Modula2 - Execution terminated!
```

because there was no library module called *Inner* found in the external environment.

10.5.1 Standard Identifiers and Scope

Standard identifiers, such as REAL, CARDINAL, ABS, and so on, are also called *pervasive*. They should be thought of as being visible one level outside every module when that module's body is thought of as a procedure. That is, they are automatically visible at level zero of all module scopes (and any procedures inside them) without having to be imported. Another way of thinking about this is to regard pervasives as being automatically imported into every module scope. However, these identifiers differ from user-defined ones in that they *may* be redefined even at the outermost level if desired. Thus

```
MODULE RedefineStdIdent;
VAR
    CARDINAL: REAL;

END RedefineStdIdent.
```

is legal, but probably not useful, as it cuts off access to the identifier CARDINAL in the module. Thus if

```
PROCEDURE try;
VAR
    card : CARDINAL;
END try;
```

were defined in the module above, there would be an error "wrong class of identifier" generated when the second CARDINAL was examined by the compiler, for it no longer is a type but a variable. However, since the rule is that standard identifiers are imported into every module scope, the following is legal, and the second use of CARDINAL inside the procedure *try* has its normal meaning.

```
MODULE RedefineStdIdent2;
VAR
    CARDINAL: REAL;
```

```

MODULE Inner;
  PROCEDURE try;
  VAR
    card : CARDINAL;
  END try;
END Inner;

```

```

END RedefineStdIdent2.

```

These somewhat strange examples are presented not so much because they have many practical applications, but to emphasize the visibility rules, and that they are universally applicable, even when rather odd things result. Recall of course, that there is a fundamental difference between standard identifiers and reserved words. As the latter are not identifiers at all, they can *never* be re-defined or re-used in any fashion.

10.5.2 Dynamic Modules

As envisioned so far, modules and their entities are entirely compile-time phenomena, and one statement above went so far as to say that they do not even exist at run time. However, there is an exception to this in any module that is declared inside a procedure. Since the entities in a procedure scope are only realized in memory when the procedure is invoked, such a module will also come into existence only when the procedure is active. The following is legal:

```

MODULE RedefineStdIdent3;
VAR
  CARDINAL, card: REAL;

  PROCEDURE NewScope;
    MODULE Inner;
      EXPORT card;
      VAR
        card : CARDINAL;
      END Inner;

  BEGIN
    card := 5;
  END NewScope;

END RedefineStdIdent3.

```

Observe that the export of the name *card* to the scope of *NewScope* is legal; it merely cuts off access to the *card* in the outer (procedure) scope, as one would expect. Moreover, the assignment *card* := 5 can be properly checked, even though the identifier CARDINAL with its built-in meaning is not visible to (and cannot be exported to) the procedure *NewScope*, because it does not belong to *Inner* and cannot be exported by it. That limitation can be gotten around of course, if the dynamic module exports its own name for the type CARDINAL as follows:

```

MODULE RedefineStdIdent4;
VAR
  CARDINAL, card: REAL;

  PROCEDURE NewScope;

```



```
MODULE Inner;  
EXPORT Cardinal;  
TYPE  
    Cardinal = CARDINAL;  
END Inner;
```

```
VAR  
    card : Cardinal;  
BEGIN  
    card := 5;  
END NewScope;
```

```
END RedefineStdIdent4.
```

The student should appreciate that these examples stretch the envelope of Modula-2 legalities nearly to the breaking point of common sense, but that they do so to illustrate the points being made about the scope rules rather than to inspire imitation.

[Contents](#)

10.6 An Extended Example--Fibonacci Sequences

No attempt will be made here to illustrate all the visibility details discussed so far in this chapter in practical examples. However, the principle that information can be kept hidden from portions of the program that do not need it is important, for code that cannot see some entities that it should not, can also not modify those entities, resulting in safer and easier-to-debug code. The next example illustrates the use of a module (local to the program module) to generate the next number in a special sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... known as the Fibonacci Sequence.

The Fibonacci Sequence is defined by
 $F_1 = 0, F_2 = 1$ and $F_{i+1} = F_{i-1} + F_i$
thereafter.

That is, after the first two numbers, all subsequent ones in the sequence are generated by adding the previous two. In the Modula-2 solution that follows, *FibGen* has a body that initializes the sequence, and local variables to maintain the (current) ultimate and penultimate values in the sequence. All of this is invisible to the surrounding program module, which is interested only in the exported procedure *NextFib*, and not any other details involved. If the module were declared within a procedure, by the way, that initialization body would execute whenever the procedure were entered. After all, the module would *exist* only inside the procedure, according to the scope rules for procedures. This way, the initialization takes place once.

```
MODULE Fibonacci;

(* by R. Sutcliffe
   To demonstrate hiding in local modules
   revised 1994 04 11 *)

(* This module generates and prints the first twenty numbers in the Fibonacci
   sequence. *)

FROM STextIO IMPORT
  WriteString, WriteLn;
FROM SWholeIO IMPORT
  WriteCard;

PROCEDURE OutputVal (index, num : CARDINAL);
BEGIN
  WriteCard (index, 2);
  WriteString ( " )  ");
  WriteCard (num, 5);
  WriteLn;
END OutputVal;

(*-----*)
```

```

MODULE FibGen;
EXPORT NextFib;
VAR
    ult, penult, temp: CARDINAL; (* hidden *)
PROCEDURE NextFib ( ) : CARDINAL;
BEGIN
    temp := ult;
    ult := ult + penult;
    penult := temp;
    RETURN ult;
END NextFib;

(* The module body initializes the two local variables before being used *)
BEGIN
    ult := 0;
    penult := 1;
END FibGen;

(*-----*)

(* Main Program Starts *)
VAR
    fib, count : CARDINAL;
BEGIN
    WriteString(" The first 20 Fibonacci Numbers are: ");
    WriteLn;
    OutputVal (1, 0);
    (* the first Fib num is written out before one starts calcs *)
    FOR count := 2 TO 20
        DO (* Then, the rest *)
            fib := NextFib( );
            OutputVal (count, fib);
        END;
    WriteString ("There you have it ");
END Fibonacci.

```

The output from this program was:

The first 20 Fibonacci Numbers are:

1)	0
2)	1
3)	1
4)	2
5)	3
6)	5
7)	8
8)	13

```
 9 )      21
10 )      34
11 )      55
12 )      89
13 )     144
14 )     233
15 )     377
16 )     610
17 )     987
18 )    1597
19 )    2584
20 )    4181
```

There you have it

Notice how one keeps the entities being manipulated hidden from the surrounding scope and update them only through an exported procedure. One uses the fact that *ult* and *penult* are global to *FibGen* to maintain their values between calls to the procedure *NextFib*. However, these two variables are not visible in the outer scope and so the main program cannot change them except by calling *NextFib*.

[Contents](#)

10.7 Library Modules--Scope and Visibility Rules

Consider for a moment the modules already used, and recall that the *WriteString* of *STextIO* is not the same procedure as the *WriteString* of *TextIO*. A number of previous examples have utilized the fact that when both are needed, a clash can be avoided by writing:

```
IMPORT STextIO;
```

among the other IMPORT lists and then referring to the qualified identifier *STextIO.WriteString* when it was required instead of the *WriteString* from *TextIO*.

The astute reader should realize that this treatment must be a consequence of the very nature of library modules, for since their exported items are available to be imported into unknown scopes, all those exports must in fact be QUALIFIED. Not only is this true, but in earlier versions of Modula-2, it was also necessary to include the line

```
EXPORT QUALIFIED <list of entities"I can do this here.">;
  WriteLn;
END Inner;

(* WriteLn visible only as STextIO.WriteLn *)

END LibVisibilityDemo.
```

Note that if the program module requires entities from *STextIO*, it can import them unqualified as well as qualified (FROM). Since only one copy of an item obtained from an imported module lives in the computer's memory at any one time, no additional storage space is required to do both types of imports.

Any items not directly needed in the outer scope ought to be imported only as part of whole libraries, then imported into inner scopes as needed. Then, only those items actually being used are available in each scope.

10.7.2 Visibility in Library Modules

Like any other modules, library modules may employ only those entities that they either define themselves, or that they import from elsewhere. These imports are subject to some special rules that apply only to library modules. These were referred to in Chapter 6, and are included here for the sake of completeness.

1. Any entities declared in the DEFINITION part of the library module are automatically visible to the IMPLEMENTATION part of the module.
2. The converse is false. Items declared only in the implementation part are visible only in the implementation part.
3. Items imported from elsewhere by the definition part (for instance, types for parameter lists) are available only in the definition part. They must be imported separately by the implementation part.
4. Even if the definition part redefines an enumerated import as its own, that does not make the field names of the original item visible in the implementation.

Thus, one has, for instance:

```
DEFINITION MODULE RndFile;

IMPORT IOChan, ChanConsts, SYSTEM;
```

TYPE

```
ChanId = IOChan.ChanId;
FlagSet = ChanConsts.FlagSet;
OpenResults = ChanConsts.OpenResults;

(* Accepted singleton values of FlagSet *)
```

CONST

```
read = FlagSet{ChanConsts.readFlag};
write = FlagSet{ChanConsts.writeFlag};
old = FlagSet{ChanConsts.oldFlag};
text = FlagSet{ChanConsts.textFlag};
raw = FlagSet{ChanConsts.rawFlag};
```

```
PROCEDURE OpenOld (VAR cid: ChanId; name: ARRAY OF CHAR; flags: FlagSet; VAR res:
OpenResults);
```

```
(* more stuff *)
```

```
END RndFile.
```

where assorted items are imported for the use of the definition module, and a number of them are redefined for the convenience of other modules that might want to import these common items from here, the redefined items are all available in the implementation part, but it must import its own copy of SYSTEM, and it does not have access to the enumeration values of the re-defined OpenResults directly; it must refer to them by importing ChanConsts.OpenResults itself and employing qualified identifiers of that module. Here is a portion of the author's implementation:

```
IMPLEMENTATION MODULE RndFile;
```

```
IMPORT ChanConsts, IOConsts, IOLink, IOChan, SYSTEM;
```

```
IMPORT Filer;
```

```
FROM Filer IMPORT
```

```
FileErr;
```

```
FROM Strings IMPORT
```

```
Assign;
```

```
(* all the rest here *)
```

```
END RndFile.
```

A type that is exported from a definition module, but whose details are hidden in the implementation module is called an opaque type. Types whose details are visible (normal types) are called transparent types.

These last two definitions are presented for information at this point. This book will not be making use of opaque types for a while yet. Suffice it to say for now that opaque types allow hiding the details of data from the view of the main program and prevent it from becoming corrupted inadvertently. All access to

that data is controlled by the procedures written specifically for that purpose.

[Contents](#)

Part B--Program Control and Error Handling Issues

10.8 Transfer of Control

10.8.1 GOTO and Other Noxious Weeds

Unlike most other programming languages, Modula-2 does not allow statements to be given numbers or labels nor does it permit a *GOTO line number* or a *GOTO label* to be used.

Statements like the latter are often used by programmers in some languages to transfer control indiscriminately all over a program. Their use therefore results in programs that are hard to read, harder still to modify, and nearly impossible to debug. Programmers who think in an organized fashion and design in well-defined modules with pre-planned interfaces, and carefully packaged procedures do not need unconditional transfers of control except under very special circumstances. Modula-2 does provide for those special circumstances where a regulated transfer of control is permitted.

10.8.2 RETURN in a Regular Procedure

In Chapter 4, RETURN <value> statement in a regular procedure.

Note also that since program module execution is essentially similar to that of procedure execution, the use of RETURN in the body of a module will cause a transfer of control to a point just after the last statement in the body associated with that scope--in this case, terminating the module.

```
MODULE ReturnDemo;
(* program by R. Sutcliffe
   to demonstrate return from a module
   revised 1994 04 15 *)
FROM STextIO IMPORT
    WriteString, WriteLn, ReadChar, SkipLine;
FROM SIOResult IMPORT
    ReadResults, ReadResult;

PROCEDURE FinishUp;
BEGIN
    WriteLn;
    WriteString ("Press return to end == >");
    ReadChar (ch);
    SkipLine;
```



```

END FinishUp;

VAR
    ch : CHAR;

BEGIN
    WriteString ("This program written to demonstrate");
    WriteString (" returning from a module body");
    WriteLn;
    REPEAT
        WriteString ("Do you wish to:");
        WriteLn;
        WriteString ("1. continue without returning?");
        WriteLn;
        WriteString ("2. return from the module");
        WriteLn;
        WriteString ("3. terminate normally");
        WriteLn;
        ReadChar (ch);
        IF (ReadResult () # allRight) OR (ch = "2")
            THEN
                SkipLine;
                WriteString ("returning from module");
                FinishUp;
                RETURN
            ELSE
                SkipLine;
            END;
    UNTIL (ReadResult () # allRight) OR (ORD (ch) > ORD ("3"));
    WriteString ("Regular termination");
    FinishUp;
END ReturnDemo.

```

When this program was run, the action of executing the **RETURN** from within the body of the module was indistinguishable from the action of merely concluding the program at the **END** in the normal way. This is as it should be, because **RETURN** transfers control to a point just beyond the call to *FinishUp* and before the **END** of the enclosing procedure scope, which is in this case the module. What happens *after* a program terminates depends on the operating system being used; the user should be returned to the environment from which the program was run in the first place.

Either of

(i) allowing a program to run to its end, or

*(ii) transferring control to the end by a simple **RETURN** statement*

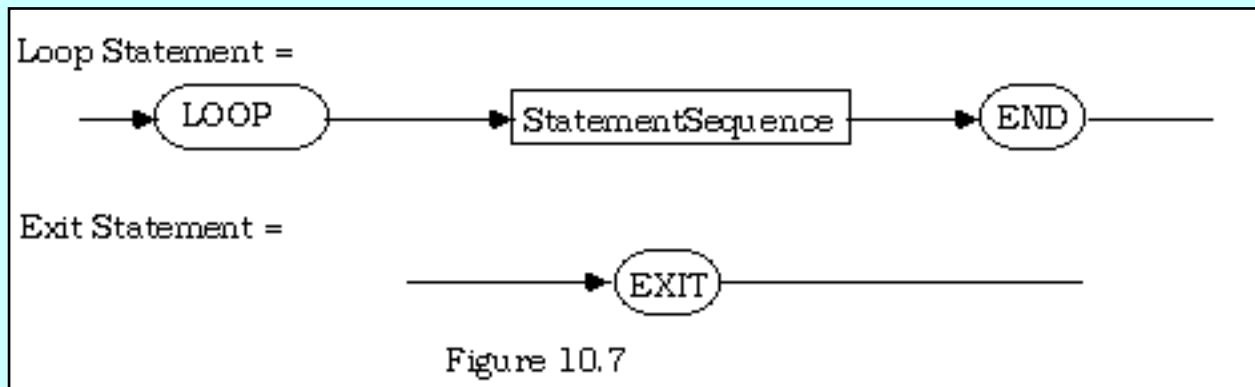
is called normal program termination.

10.8.3 Repetition Revisited--The Generalized LOOP Statement

Besides the WHILE, REPEAT, and FOR loops, Modula-2 allows a generalized type of loop from which there may be several possible exits. Here is an outline example that indicates one possible form in which this could exist:

```
LOOP
  Statement Sequence 1;
IF Condition 1
  THEN
    EXIT
  END;    (* if *)
END;    (* loop *)
```

The syntax diagrams are simple:



An EXIT statement within a LOOP serves the same purpose as a RETURN in a regular procedure, in that it transfers control to the END and thence out of the LOOP. It differs, however, in that it is the only way to exit a LOOP, for this repetition will execute indefinitely if some EXIT statement is not encountered. To put it another way, the EXIT provides a mid-loop test in the same way that the WHILE loop has a top-of-loop test and the REPEAT loop has a bottom-of-loop test. The following example reads in a Y or N, printing out a message only if some other character is pressed:

```
LOOP
  WriteString ('Enter <Y> or <N> please ... ');
  Read (ch); SkipLine;
  ch := CAP (ch);
  IF (ch = 'Y') OR (ch = 'N')
  THEN
    EXIT
  END; (* If *)
```

```
    WriteString ('Error. You must '); (* and go around again *)  
END; (* Loop *)
```

Other more familiar repetition structures can be simulated with the LOOP.

```
REPEAT  
    Statement Sequence  
UNTIL Condition;
```

could be simulated by

```
LOOP  
    Statement Sequence;  
  
    IF Condition  
        THEN  
            EXIT  
        END;  
END;
```

Moreover, there may be multiple test points within a LOOP for a possible exit from the structure, provided that the run-time logic eventually allows one of the exit points to be reached.

```
LOOP  
    Statement Sequence1;  
    IF Condition1  
        THEN  
            EXIT  
        END;  
    Statement Sequence2;  
    IF Condition2  
        THEN  
            EXIT  
        END;  
    Statement Sequence3;  
END;
```

Note, however, that in most cases the LOOP structure is not needed. It should be used when

- there must be more than one exit point from a loop
- the flow of the program naturally lends itself to a loop structure
- one must exit from a deeply nested loop or selection structure

The latter case, where one uses LOOP..EXIT as a means to leave a very deeply nested loop within a loop without having to transfer control out one level at a time, can be illustrated in the outline:

```
LOOP
  REPEAT
    (* some code *)
  REPEAT
    (* some more code *)
  CASE selector OF
    one:
      (* more code *) |
    two:
      (* alternate *)
  ELSE
    EXIT
  END;
  UNTIL condition1
UNTIL condition2;
(* yet another statement sequence *)
EXIT; (* if we get here, processing done *)
END;
```

In cases like these, a normal repetition structure from which one wants to have an alternate exit points surrounded with a LOOP that has an EXIT at the very end. If processing reaches that end point, normal termination of the repetition is assured by the terminating EXIT. Alternately, it is possible to exit without concluding the repetition if necessary.

LOOP structures are often used when processing data from or to sequential files, though they are usually not strictly necessary, and just save a little writing. Typical code from Chapter 8 such as

```
REPEAT
  WriteChar (""); (* prompt *)
  ReadInt (number);
  IF ReadResult() = allRight
  THEN
    SkipLine;
    WholeIO.WriteInt (outfile, number, 0);
    TextIO.WriteLine (outfile); (* separate the numbers *)
  END; (* if *)
UNTIL ReadResult() # allRight;
```

could be written:

LOOP

```
WriteChar (>"); (* prompt *)
ReadInt (number);
IF ReadResult() = allRight
  THEN
    SkipLine;
    WholeIO.WriteInt (outfile, number, 0);
    TextIO.WriteLine (outfile); (* separate the numbers *)
  ELSE
    EXIT
  END; (* if *)
END; (* loop *)
```

Notice that the only saving here was not having to write the condition on *ReadResult* twice, and indeed, if the conditions for processing were slightly different from those for exiting, there would be no gain at all. In general, it is better to try to employ one of the more conventional loop structures, and this can be done in the vast majority of situations.

EXIT and RETURN are the only statements in Modula-2 that cause an immediate transfer of the normal flow of control, and these will transfer only to a point after the last statement in the statement sequence associated with the loop or the body of the current PROCEDURE. There are *abnormal* transfers resulting from errors; these will be considered in the section on exceptions later in this chapter.

[Contents](#)

10.9 Error Handling Revisited

Careful planning, accurate coding in a structured notation, and diligent care in debugging can greatly increase confidence in the reliability and correctness of a piece of code. It must be understood, however, that testing is undertaken to *find* errors. No amount of testing *proves* that software is error free.

There is always one more bug. --a Corollary of Murphy's Law

The first thing that *any* engineer must learn is that Murphy's Law is not a joke. The good engineer *knows* there will be failures, and takes that into account. Specifically, it is not enough to plan what is thought to be error free software and test it thoroughly after it has been produced. Provision must still be made to handle the errors the programmer thinks cannot occur when they do in fact occur. No compiler can alert the programmer to logical errors; the ability to do so would mean that all possible logically correct programs were explicitly or implicitly known to the compiler, which is not possible. Indeed, it can be shown mathematically that it is not possible to build a program that can examine all programs to determine if they will be successful in their tasks or not.

Thus, one is always required to develop a variety of error handling techniques. Of course, the best kinds of errors are those that are prevented from happening in the first place. That is why section 6.6 emphasized precondition checking as a means of reducing errors. However, even if the software requiring certain preconditions is itself correct, and those preconditions are carefully met by the calling code, there may still be errors of two kinds:

1. errors caused by the failure of external library module routines, such as those attempting to read information from a file
2. unforeseen errors in either the client code or the pre-conditional code

The first kind is outside the programmer's direct control, and as for the second, there are always conditions the programmer has not thought of, and so there are possible errors unrelated to the checked conditions.

10.9.1 Typical Library Errors

For such reasons, library modules often export an error type so as to allow postcondition checking once the library routine has returned control to the client program. The simplest method is to export a boolean variable, say, *done*, which is set to TRUE whenever an invoked library routine succeeded, and to FALSE whenever it failed. This is the method employed by the classical libraries *InOut* and *RealInOut*.

There are two disadvantages to such a technique. The first is that a Boolean variable cannot convey information about *why* the failure took place. This problem can be solved by making the error type an enumeration, as in the ISO libraries, where one has:

DEFINITION MODULE IOConsts;

TYPE

```
ReadResults =
( notKnown,      (* no data read result is set *)
  allRight,      (* data is as expected or as required *)
  outOfRange,    (* data cannot be represented *)
  wrongFormat,   (* data not in expected format *)
  endOfLine,     (* end of line seen before expected data *)
  endOfInput     (* end of input seen before expected data *)
);
```

END IOConsts.

Likewise, the module *ChanConsts* exports (for the import and re-export of device drivers):

TYPE

```

    OpenResults =          (* Possible results of open requests *)
        (opened,           (* the open succeeded as requested *)
         wrongNameFormat,  (* given name is in the wrong format for the implementation *)
         wrongFlags,       (* given flags include a value that does not apply to the
device *)
         tooManyOpen,      (* this device cannot support any more open channels *)
         outOfChans,        (* no more channels can be allocated *)
         wrongPermissions, (* file or directory permissions do not allow request *)
         noRoomOnDevice,    (* storage limits on the device prevent the open *)
         noSuchFile,        (* a needed file does not exist *)
         fileExists,        (* a file of the given name already exists when a new one is
required *)
         wrongFileType,     (* the file is of the wrong type to support the required
operations *)
         noTextOperations,  (* text operations have been requested, but are not supported
*)
         noRawOperations,   (* raw operations have been requested, but are not supported
*)
         noMixedOperations,(* text and raw operations have been requested, but they are
not supported in combination *)
         alreadyOpen,       (* the source/destination is already open for operations not
supported in combination with the requested operations *)
         otherProblem       (* open failed for some other reason *)
        );

```

The second difficulty with using variables is that an error *variable* is not safe, for it can be changed by the client program. Programmers are fond of reusing such variables by attaching additional conditions to the *read* in the following manner:

```
Done := Done AND (num >= 10);
```

thus destroying the original value of a variable that does not belong to the module making the change. To prevent this, library modules should not export error variables (and should rarely, if ever, export variables at all). Rather, they should export only enquiry procedures that return a value of the error type as is done in the ISO I/O library module:

```
DEFINITION MODULE IOResult;
```

```
IMPORT IOConsts, IOChan;
```

TYPE

```
    ReadResults = IOConsts.ReadResults;
```

```
PROCEDURE ReadResult (cid: IOChan.ChanId): ReadResults;
```

```
END IOResult.
```

Likewise, a module like StreamFile has:

```
IMPORT IOChan, ChanConsts;
```

TYPE

```
    ChanId = IOChan.ChanId;
```

```
    FlagSet = ChanConsts.FlagSet;
```

```
OpenResults = ChanConsts.OpenResults;
```

```
PROCEDURE Open (VAR cid: ChanId; name: ARRAY OF CHAR; flags: FlagSet; VAR res:  
OpenResults);
```

so that, in this case, rather than an inquiry procedure, the error result is returned in a variable parameter of the module's typical procedure activities.

When both these suggestions are followed, a substantial amount of information can be obtained about the reasons why a file open or a disk read failed, without compromising the value of the variables where this information is stored (because the client program cannot see that variable).

[Contents](#)

10.10 Controlling Program Termination

Besides the normal program termination events triggered by concluding the program and arriving at the END or by a RETURN from the program module body, it is possible to terminate programs in other ways. This section discusses one of those ways, and also presents some strategies for action *after* a termination event has taken place.

10.10.1 HALT and Abnormal Termination

Some programmers may feel it necessary to *kill* a program that has encountered serious difficulties of some kind. Modula-2 provides the HALT command for such cases. It is similar to a simple RETURN, except that HALT:

- is not thought of as a transfer of control, but as a program abort
- will kill a program from anywhere, not just the main module body
- is reported to the user by many (but not all) operating systems (non-silent termination)
- is a termination-with-extreme-prejudice or *abnormal termination*
- should be used even less frequently than RETURN

For instance, when the following Module was run, the memory allocated to the boolean flag happened to be interpreted as TRUE, and the program halted.

```
MODULE HaltDemol;  
VAR  
  reallyAwfulCondition : BOOLEAN;  
  
BEGIN  
  (* some code *)  
  IF reallyAwfulCondition  
    THEN  
      HALT  
    END;  
  (* some more code *)  
END HaltDemol.
```

In one specific version used, a debugging environment was entered, part of the display for which is shown in figure 10.8.

Procedure Trace	Locals to HaltDemo1
Status: Program Halt HaltDemo1 56 in HaltDemo1	Name = HaltDemo1 Type = PROGRAM reallyAwfulCon...BOOLEAN TRUE
Figure 10.8	

Of course, the precise behaviour *after* the HALT is called is implementation defined, and some implementations will report nothing at all. In such implementations, a separate non-standard command such as BREAK (perhaps in a compiler directive) may be employed to invoke the debugging environment. There is a *language* difference between HALT and a normal termination, however, and this will be detailed as part of the next section.

[Contents](#)

10.11 FINALLY: Termination, Detection and Cleanup

In [section 10.8.2](#) it was pointed out that immediate normal program termination could be triggered manually through a RETURN from the body of the program module. The transfer of control was described as being to a point just after the last statement in the block body of the program module, rather than simply "to the END of the module." In fact, there can be additional code between the two in what is called a FINALLY clause, and this code can be used for the purpose of cleanup after the program, (such as the closing of files). Here is a sketch:

```
MODULE Program;
(* imports *)
(* declarations *)
BEGIN (* the body *)
  (* statement sequence *)
FINALLY
  (* statement sequence *)
END Program.
```

NOTES: 1. The statement sequence in the FINALLY clause always executes when a program is terminated, regardless of the reason for this termination.

2. Every module may have a FINALLY clause.

If both the program module and one or more library modules that it imports have FINALLY clauses, then when the program terminates, the FINALLY clauses are executed in the reverse order that the module bodies were run (initialized) in the first place. This order will of course depend on the order of the imports in the program module and in the modules they in turn import and can only be known to the programmer of all the modules in the suite.

The orderly execution of the finalization clauses of the various modules constituting a program is termed program finalization.

NOTE: If a program terminates because of a HALT or some error condition encountered by a modules, only the modules that have at least *begun* to run their bodies will be finalized. The FINALLY clause of any module whose (initialization) body has not yet been commenced will not be invoked.

Whatever *cleanup* has to be done by a module should therefore be related only to what that module itself has done. If a program module, for instance, opens files, that module ought at some point close those files. In some cases, this might best be done in a FINALLY clause, depending on the logic of the rest of the code. Here is a simple example, based on the chapter eight program *ReadNAdd*:

```
MODULE FinallyDemo;

(* Written by R.J. Sutcliffe *)
(* to illustrate the use of Finalization *)
(* using ISO standard Modula-2 *)
(* last revision 1994 04 22 *)

(* This module, like ReadNAdd, reads a series of Integers from the disk file called
"numbers". It sums them and prints out the sum. *)

FROM StreamFile IMPORT
  ChanId, Open, read, Close, OpenResults;
```

```

FROM TextIO IMPORT
    WriteLn, WriteString, SkipLine, ReadChar;
FROM IOResult IMPORT
    ReadResult, ReadResults;
FROM WholeIO IMPORT
    ReadInt, WriteInt, WriteCard;
FROM StdChans IMPORT
    StdInChan, StdOutChan;

VAR
    infile, stdOut, stdIn : ChanId;
    number, sum : INTEGER;
    res : OpenResults;
    ch : CHAR;

BEGIN
    stdOut := StdOutChan(); (* to force screen output *)
    stdIn := StdInChan(); (* to force keyboard input *)

    Open (infile, "numbers", read, res);
    IF res = opened
    THEN
        sum := 0; (* initialize sum *)
        REPEAT (* Collect the numbers from the file *)
            ReadInt (infile, number);
            IF ReadResult (infile) = allRight
            THEN
                SkipLine (infile);
                INC (sum, number);      (* ok, so add to sum *)
            END; (* if *)
        UNTIL ReadResult (infile) # allRight;

        WriteString (stdOut, "The sum of the numbers is ");
        WriteInt (stdOut, sum, 6);
        WriteLn (stdOut);
    ELSE
        RETURN
    END; (* if *)
    WriteString (stdOut, "Processing concluded normally");
    WriteLn (stdOut);

FINALLY
    IF res # opened
    THEN
        WriteString (stdOut, "Sorry, couldn't open the file");
        WriteLn (stdOut);
        WriteString (stdOut, " because of error number ");
        WriteCard (stdOut, ORD (res), 1);
        WriteLn (stdOut);
    ELSE
        Close (infile);
    END;
    WriteString (stdOut, "type a character to continue==");

```

```
ReadChar (stdIn, ch);
```

```
END FinallyDemo.
```

HALT interacts with finalization in the following ways:

1. HALT is a termination event, and like a simple RETURN in a program module will initiate the finalization of the modules whose initialization has begun.
2. If HALT is called from a dynamic module (i.e. a module inside a procedure) any FINALLY clause in that dynamic module is not executed.
3. If HALT is called in a FINALLY clause, that particular clause is immediately concluded (transfer to a point just beyond its last statement) and the finalization of the program suite carries on with the next scheduled FINALLY clause in its proper order.

Because HALT and RETURN are not the same kind of termination event, it *may* be necessary to distinguish between them in code included in a FINALLY clause. This is done as in the following sketch outline:

```
MODULE DistinguishHalt;
(* By R. Sutcliffe
   To demonstrate HALT/FINALLY interaction
   revised 1994 04 22 *)

FROM TERMINATION IMPORT
  HasHalted;
IMPORT STextIO;

VAR
  ch : CHAR;
BEGIN
  STextIO.WriteString ("Halt? Y/N =");
  STextIO.ReadChar (ch);
  STextIO.SkipLine;
  IF CAP (ch) = "Y"
    THEN
      HALT
    END;
  FINALLY
    IF NOT HasHalted ()
      THEN
        STextIO.WriteString ("Not ");
      END;
    STextIO.WriteString ("Halted");
    STextIO.WriteLine;
    STextIO.WriteString ("Press a return to continue");
    STextIO.SkipLine;
  END DistinguishHalt.
```

The module TERMINATION is a required system module in the ISO libraries and provides services to distinguish termination events.

10.12 Exceptions

The facilities mentioned in this section are present only in ISO compliant versions of Modula-2 although they are similar to ones that were used in a few early commercial flavours. Some of the ideas were described in the first of the author's books on Modula-2, and then finally adopted by the ISO committee, though with many syntax changes and additions. These methods constitute a considerable extension of the language itself--they are not simply a collection of library based error handlers or variables. Other languages also make use of exceptions for signalling and handling errors, though the details vary somewhat. There is a class of error that can not always be anticipated prior to program execution and that in many systems (hardware and software combined sense) cause the program to "crash" in a manner that results in an immediate and premature exit to the surrounding environment.

An exception is a violation of the run-time meaning of a program that when detected automatically alters the normal flow of control in the procedure or module body where it occurs, immediately transferring control to an exception handler for that procedure or module body, if one exists.

As soon as an exception occurs, the procedure or module body in which it takes place loses control of the machine. In some languages, the result is that the program terminates immediately in an error condition. ISO Modula-2, however, makes provision for the detection and possible handling of exceptions within the context of the program itself.

A Modula-2 exception changes the state of the program from normal execution to exceptional execution.

A Modula-2 exception handler is a code clause that may be attached to any procedure.

The change in state to exceptional execution is called raising an exception and it immediately transfers control to the nearest (nested) exception handler.

If an exception is not trapped (handled) even by the program module then termination of the program commences with the program state still exceptional. (Exceptional termination).

This kind of action is the most automatic of all types of error handling. If the procedure or module body being executed at the time the exception is raised has an exception handler of its own, then this will be executed. If it does not, or if its handler raises the same exception or a new one, control exits to the next outer procedure, and in fact will cascade up the chain of calls until it finds a handler. As soon as it does, this handler is automatically invoked--an action which places the burden on the program to ensure that errors causing exceptions are properly handled at the appropriate level. If no handler can be found even in the program module body, the program terminates exceptionally.

What happens *after* a program terminates in an exceptional state depends on the implementation, but most are likely to have an automatic handler at the outermost level, so that just as control returns to the surrounding environment there will at least be some report made to the user of the program.

There are several stages to this kind of activity:

1. The conditions under which exceptions are raised must be defined.
2. All exceptions must be named.
3. The source of the exception must be defined and registered.
4. The conditions producing the exception must be detected.
5. The exception must be raised, providing the source and a message.
6. Exceptions raised by a program *should* be handled by the program.
7. Those raised by the language or a library *may* be handled in the program.
8. Handling exceptions *may* result in re-trying code that caused them. or
9. It may result in re-raising the same exception for calling code to handle. *or*

10. It may result in the program terminating (normally or exceptionally).

In order to achieve all this, the ISO committee made some additions to the syntax of Modula-2, for the language as defined by Wirth has no facility like the one just described.

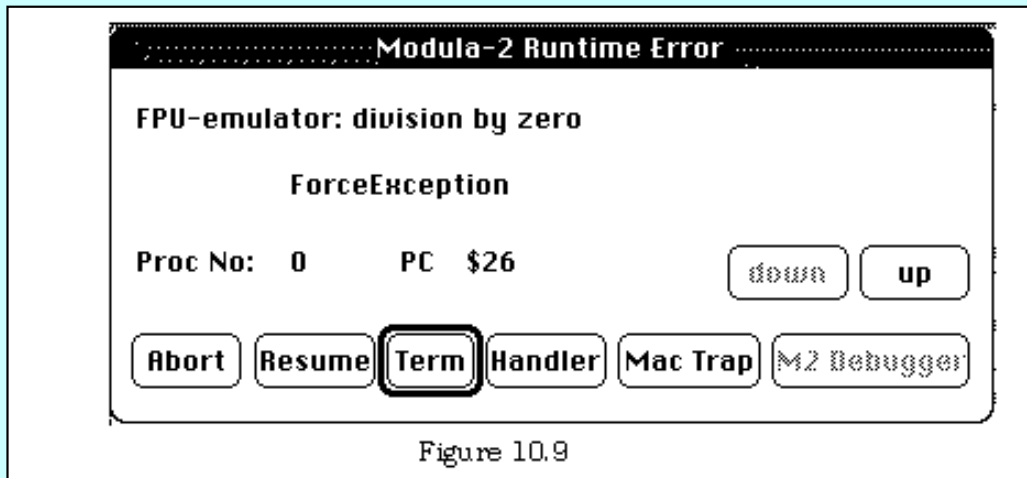
In the sections that follow, the points in the above list will be elaborated, though not entirely in the order given here, which in its entirety applies only to exceptions both defined and detected by user programs.

10.12.1 Language Exceptions

Some exceptions are defined in the language itself, and all ISO Modula-2 implementations are required to detect them at run time. An example is division by zero.

```
MODULE ForceException;
VAR
  a : REAL;
BEGIN
  a := 0.0;
  a := 5.0 / a
END ForceException.
```

Whether or not the program does anything with these exceptions once they do occur depends on whether there has been a handler written into the program. There is none in the code above, and in one ISO compliant system in which it was run, the display in figure 10.9 was produced by the run time system, in accordance with the post-exceptional-termination behaviour defined for that specific implementation :



The complete list of language defined exceptions is found in the System module M2EXCEPTION. Recall that a system module is a segregated part of the compiler; it behaves, however, as *if* it had a definition module.

```
DEFINITION MODULE M2EXCEPTION;
```

```
(* Provides facilities for identifying language exceptions *)
TYPE
  M2Exceptions =
    (indexException, rangeException, caseSelectException,
     invalidLocation, functionException, wholeValueException,
     wholeDivException, realValueException, realDivException,
     complexValueException, complexDivException, protException,
     sysException, coException, exException *)
```

```
PROCEDURE M2Exception () : M2Exceptions;
```

```
(* If the current coroutine is in the exceptional execution state because of a language exception, returns the corresponding enumeration value, and otherwise raises an exception *)
```

```
PROCEDURE IsM2Exception () : BOOLEAN;
```

```
(* If the current coroutine is in the exceptional execution state because of a language exception, returns TRUE, and otherwise returns FALSE *)
```

```
END M2EXCEPTION.
```

The particular exception raised by the sample code is *realDivException*, and as may be readily be seen, there are a variety of others, some with rather obvious meanings in the light of previous discussions in this text (for invalid ranges or array indicies, overflow values, and improper case selection). Others have not yet been encountered, and can await further developments, as can the meaning of the term *coroutine* in the two procedure definitions. The exception raised by this module itself when it is asked for the number of an exception it is not responsible for is *exException*. Note that *exException* need never occur, if the request for the enumeration value is formulated:

```
IF IsM2Exception ()  
  THEN  
    exVariable := M2Exception()  
  END;
```

This system module also sheds light on what kind of entity an exception is from a language point of view--it is simply a member of an appropriate enumeration.

10.12.2 Library Exceptions

Similar facilities are incorporated into many standard libraries, and consist of three parts:

1. an enumeration of the possible exceptions,
2. an enquiry function to discover which of those was raised,
3. an enquiry function to determine whether the module did raise one.

For instance, the standard I/O library module IOChan contains the following items not previously mentioned:

```
TYPE
```

```
  ChanExceptions =  
    (wrongDevice,      (* device specific operation on wrong device *)  
     notAvailable,     (* operation attempted that is not available on that channel  
*)  
     skipAtEnd,        (* attempt to skip data from a stream that has ended *)  
     softDeviceError,  (* device specific recoverable error *)  
     hardDeviceError,  (* device specific non-recoverable error *)  
     textParseError,   (* input data does not correspond to a character or line mark -  
optional detection *)  
     notAChannel       (* given value does not identify a channel - optional  
detection *)  
    );
```

```
PROCEDURE IsChanException () : BOOLEAN;
```

```
(* Returns TRUE if the current coroutine is in the exceptional execution state because of the raising of an exception from ChanExceptions; otherwise returns FALSE. *)
```

```
PROCEDURE ChanException () : ChanExceptions;
```



```
(* If the current coroutine is in the exceptional execution state because of the
raising of an exception from ChanExceptions, returns the corresponding enumeration
value, and otherwise raises an exception. *)
```

```
(* When a device procedure detects a device error, it raises the exception
softDeviceError or hardDeviceError. If these exceptions are handled, the following
facilities may be used to discover an implementation-defined error number for the
channel. *)
```

TYPE

```
DeviceErrNum = INTEGER;
```

```
PROCEDURE DeviceError (cid: ChanId): DeviceErrNum;
```

```
(* If a device error exception has been raised for the channel cid, returns the
error number stored by the device module. *)
```

The items marked in the standard as optional may not be detected in some implementations. Likewise, if *DeviceErrNum* is used, the number obtained after an error on enquiry by *DeviceError* depends on the implementation. The exception raised by *ChanException* when asked for an exception that it has not raised is the language exception *exException*. Once again, this exception need never occur if the user always inquires of *IsChanException* before calling *ChanException*.

10.12.3 Handling Exceptions

The material thus far only illuminates the nature of exceptions. This section deals with how to trap (handle) them, gives some suggestions about what can be done once they are detected by the user program, and some better advice about error handling code in final products.

Any procedure or module body can have an EXCEPT clause attached to it. This attachment is immediate; that is, it must come before a FINALLY part, and indeed a FINALLY part can have an EXCEPT clause of its own. In the first example that follows, when the code shown is run in the absence of the file requested, either the optional *notAChannel* exception or the exception *wrongDevice* will be raised; the message in the EXCEPT clause will be printed; and the program will exit with the exception still raised. Most systems will then provide a further diagnostic message of their own, as shown in the case of raising the *realDivException* in 10.11.1 above.

```
MODULE ExceptionDemo;
```

```
(* Written by R.J. Sutcliffe *)
(* to illustrate the trapping of exceptions *)
(* using ISO standard Modula-2 *)
(* last revision 1994 05 09 *)
```

```
FROM StreamFile IMPORT
```

```
ChanId, Open, write, Close, OpenResults;
```

```
FROM IOChan IMPORT
```

```
IsChanException, ChanException, ChanExceptions;
```

```
FROM STextIO IMPORT
```

```
WriteString, WriteLn, SkipLine;
```

```
VAR
```

```
outfile : ChanId;
```

```
res : OpenResults;
```

```
BEGIN
```

```
Open (outfile, "numbers", write, res);
```

```
(* some code *)
```

```

    Close (outfile);
EXCEPT
    IF IsChanException () AND ((ChanException() = wrongDevice)
                                OR (ChanException() = notAChannel))
    THEN
        WriteString ("Can't close; perhaps file was never opened");
        WriteLn;
        WriteString ("Press return to continue");
        SkipLine;
    END;

END ExceptionDemo.

```

Two things need to be noticed about this demonstration code. First, the error was preventable. That is, the *Open* ought to have been followed by

```

IF res = opened
THEN

```

and the exception can only take place because of sloppy logic in the program. However, the presence of such an EXCEPT clause can assist the programmer in discovering such lapses and correcting them before the finished code passes into the hands of a user.

Second, in this example, the exception was caught and described but nothing more. There are three possible results of handling an exception:

1. Control runs off the end of the handler (as here) with the exception still raised. In this case, control passes to the next outer handler, and if nothing more is done, the program will terminate, still in the exceptional state.
2. There is a RETURN from the exception handler, in which case the exception is cleared; normal execution resumes; and control passes back out to the caller of the procedure in which the exception occurred. In the example above, that would have meant a normal termination.

3. There is a RETRY in the exception handler, in which case the exception is cleared; normal execution resumes; and control passes back to the beginning of the procedure in which the exception occurred. In the example above, that would have meant attempting to run the entire module body again (but without re-initializing imported or local module bodies).

In both the second and third cases, the conditions causing the exception are all taken care of, and the logic allows it to be regarded as completely handled. Of course, in the third case, some care must be taken not to cause an infinite loop of exception raising and retrying to be entered--and that is exactly what *would* have happened if this strategy had been used in the example above.

One might follow the second strategy if the exception were the result of an error after a menu choice, and control could pass back to the menu reliably. Here is an outline:

```

MODULE ExceptionDemo2;
(* Written by R.J. Sutcliffe *)
(* to illustrate the trapping of exceptions *)
(* using ISO standard Modula-2 *)
(* last revision 1994 05 09 *)

FROM StreamFile IMPORT
    ChanId, Open, write, Close, OpenResults;
FROM IOChan IMPORT
    IsChanException, ChanException, ChanExceptions;
FROM STextIO IMPORT
    WriteString, WriteLn, SkipLine, ReadChar, ReadToken;

VAR

```

```
choice : CHAR;  
outfile : ChanId;  
res : OpenResults;
```

```
PROCEDURE OpenFile;
```

```
VAR
```

```
    name : ARRAY [0..30] OF CHAR;
```

```
BEGIN
```

```
    WriteString ("What is the file name to open?");
```

```
    ReadToken (name);
```

```
    SkipLine;
```

```
    Open (outfile, name, write, res);
```

```
END OpenFile;
```

```
PROCEDURE CloseFile;
```

```
BEGIN
```

```
    Close (outfile);
```

```
EXCEPT
```

```
    IF IsChanException () AND ((ChanException() = wrongDevice)  
                                OR (ChanException() = notAChannel))
```

```
    THEN
```

```
        WriteString ("Can't close; perhaps file was never opened");
```

```
        WriteLn;
```

```
        WriteString ("Press return to continue");
```

```
        SkipLine;
```

```
        RETURN;
```

```
    END;
```

```
END CloseFile;
```

```
BEGIN
```

```
    REPEAT
```

```
        WriteString ("        MENU");
```

```
        WriteLn;
```

```
        WriteString ("Do you want to:");
```

```
        WriteLn;
```

```
        WriteString ("O. Open a new file");
```

```
        WriteLn;
```

```
        WriteString ("C. Close the file");
```

```
        WriteLn;
```

```
        (* more choices *)
```

```
        WriteLn;
```

```
        WriteString ("Q. Quit the program");
```

```
        WriteLn;
```

```
        WriteString ("Pick one ==");
```

```
        ReadChar (choice);
```

```
        SkipLine;
```

```
        choice := CAP (choice);
```

```
        CASE choice OF
```

```
            "O" :
```

```
                OpenFile; |
```

```
            "C" :
```

```
                CloseFile
```

```
        ELSE
```

```
            (* default is fall off the end *)
```

```
    END;  
    UNTIL choice = "Q";
```

```
END ExceptionDemo2.
```

Once again, however, the exception detection code is only a stopgap; it would be far better to flag a successful open operation than to detect a failure to close properly.

The third option, RETRY, is outlined in the following sketch--another version of an earlier program for making computations based on Ohm's law. Here, it is supposed that an exception might take place as a result of the operation of a function procedure that returns values directly to a division, resulting in the possibility of a divide-by-zero error. Of course, this error too can be prevented from ever taking place if the code is written a little differently.

```
MODULE OhmsLaw2;
```

```
(* Written by R.J. Sutcliffe *)  
(* to illustrate an exception retry *)  
(* using ISO Modula-2 *)  
(* last revision 1994 05 17 *)
```

```
FROM STextIO IMPORT  
    WriteString, WriteLn, ReadChar, SkipLine;  
FROM SRealIO IMPORT  
    ReadReal, WriteFixed;  
FROM SIOResult IMPORT  
    ReadResult, ReadResults;  
FROM M2EXCEPTION IMPORT  
    M2Exception, IsM2Exception, M2Exceptions;  
IMPORT TermFile, Strings;  
TYPE  
    String = ARRAY [0..80] OF CHAR;
```

```
VAR  
    retrys : CARDINAL;
```

```
PROCEDURE GetNum (prompt : String) : REAL;
```

```
VAR  
    readOK : BOOLEAN;  
    theNum : REAL;
```

```
BEGIN  
    REPEAT  
        WriteString (prompt);  
        WriteLn;  
        WriteString ("Type the number here ==");  
        ReadReal (theNum);  
        readOK := (ReadResult() = allRight);  
        WriteLn;  
        IF NOT readOK  
            THEN  
                WriteString ("error in input number; try again.");  
                WriteLn;  
            END;  
        SkipLine;  
    UNTIL readOK;
```

```

    RETURN theNum
END GetNum;

PROCEDURE Ohms;
(* No parameters;  modifies global variables only *)
VAR
    voltage : REAL;
BEGIN
    voltage := GetNum ("What is the current in amperes?")
        / GetNum ("What is the resistance in ohms?");
    WriteString ("This current and resistance ");
    WriteLn;
    WriteString (" produce a voltage of ");
    WriteFixed (voltage, 2, 0);
    WriteString (" volts. ");
    WriteLn;
    WriteLn;
EXCEPT
    IF IsM2Exception () AND (M2Exception() = realDivException)
    THEN
        WriteString ("Can't have zero resistance");
        WriteLn;
        IF (retrys = 0)
        THEN
            WriteString ("There seems to be a little problem.");
            WriteLn;
            WriteString ("Do you want to begin again? (y/n) ");
            ReadChar (answer);
            SkipLine;
            IF CAP(answer) = "Y"
            THEN
                RETURN
            END (* else kill program;  persistant offender *)
        ELSE
            DEC (retrys);
            WriteString ("Please try again");
            WriteLn;
            RETRY;
        END;
    END;
END Ohms;

VAR
    answer : CHAR;
BEGIN
    REPEAT
        retrys := 2;
        Ohms;
        WriteString ( "Type 'Y' to do another ");
        ReadChar (answer);
        SkipLine;
        WriteLn;
    UNTIL CAP (answer) # "Y"

```

```
END OhmsLaw2.
```

Observe the necessity of controlling the `RETRY` so as to avoid an infinite number of them. (There are other ways to do this, besides keeping track of the number of times tried.) Here is the output from this code, as produced by a logging utility that monitored the information going through the module *TermFile*.

```
** Run log starts here **
What is the current in amperes?
Type the number here ==> 7

This current and resistance
  produce a voltage of 0.57 volts.

Type 'Y' to do another y

What is the current in amperes?
Type the number here ==> 0

Can't have zero resistance
Please try again
What is the current in amperes?
Type the number here ==> 0

Can't have zero resistance
Please try again
What is the current in amperes?
Type the number here ==> 0

Can't have zero resistance
There seems to be a little problem.
Do you want to begin again? (y/n) y
Type 'Y' to do another y

What is the current in amperes?
Type the number here ==> 9

This current and resistance
  produce a voltage of 0.56 volts.

Type 'Y' to do another n
```

Clearly, this code could (and should!!) have been written so as to check the data before putting it into the formula. The point is that *sometimes* it may not be possible to guarantee the validity of data by asking the user to repeat it, and an exception will be raised. This code shows one possible way of handling the exception through a retry of the offending block.

10.12.4 Exceptions and Termination

An exception handler can be attached to the body of a procedure or module, or to the `FINALLY` clause of a module, or to both. Here is a sketch:

```
MODULE ExceptionTermination1;

(* imports *)
```

```

(* declarations *)

BEGIN
    (* statement sequence *)
EXCEPT
    (* exception handling for main body *)

FINALLY
    (* termination/cleanup code goes here *)
EXCEPT
    (* exception handling for termination time *)

END ExceptionTermination1.

```

The sequence of events is:

1. If an exception is encountered in the body, control is transferred to the exception handler for that body.
- 2a. If the exception is handled and there is a RETRY, control reverts to the beginning of the body in which the exception took place.
- 2b. If the body was that of a procedure, control transfers outward to the caller of the procedure until an exception handler is encountered. If none is, step (2c) applies.
- 2c. If the body was that of a module, and the exception remains raised, or there is a RETURN, termination commences.
3. Once termination has commenced, control transfers to the FINALLY clause, and this clause executes.
4. If a new exception is raised, it is immediately handled by the exception handler for the FINALLY clause, but if the exception from the main body is still raised, it receives no further handling.

Thus, the module:

```

MODULE ExceptionTermination2;
IMPORT STextIO;

VAR
    b: CARDINAL;

BEGIN
    b := 0;
    b:= 1/b; (* force an exception *)
EXCEPT
    STextIO.WriteString ("Entered exception handling for main body");
    STextIO.WriteLine;
FINALLY
    STextIO.WriteString ("Program now terminating.");
    STextIO.WriteLine;
EXCEPT
    STextIO.WriteString ("Entered exception handling at termination time.");
    STextIO.WriteLine;

END ExceptionTermination2.

```

which raises an exception in the main body but does not handle it and terminates in an exceptional state without executing the EXCEPT clause of the FINALLY part, producing the output:

```

Entered exception handling for main body
Program now terminating.

```

On the other hand, the module:

```
MODULE ExceptionTermination3;
IMPORT STextIO;

VAR
    b: CARDINAL;

BEGIN
    b := 0;
    b:= 1/b; (* force an exception *)
EXCEPT
    STextIO.WriteString ("Entered exception handling for main body");
    STextIO.WriteLine;
    RETURN (* kill the first exception and move along *)
FINALLY
    b:= 1/b; (* force another exception *)
    STextIO.WriteString ("Program now terminating.");
    STextIO.WriteLine;
EXCEPT
    STextIO.WriteString ("Entered exception handling at termination time.");
    STextIO.WriteLine;

END ExceptionTermination3.
```

which raises a new exception at the beginning of the **FINALLY** clause (and so does not conclude it), and then produces the output:

```
Entered exception handling for main body
Entered exception handling at termination time.
```

It would make no difference to the output in this case whether the **RETURN** were executed or not, as it is the raising of a exception in the **FINALLY** part that triggers transfer of control to its **EXCEPT** clause.

In the event that there is no finally clause to a module, but there is an except clause, then the except clause applies to exceptional events regardless of whether they took place during execution of the body of the module, or after termination had been initiated. In such situations, it may be desirable to determine whether one has entered the exception handler prior to or subsequent to the commencement of termination. In order to do this, one could write code as in the following outline:

```
MODULE ExceptionTermination4;
FROM TERMINATION IMPORT
    IsTerminating, HasHalted;

BEGIN
    (* statement sequence *)
EXCEPT
    IF IsTerminating ()
        THEN
            (* action to handle exception after termination started *)
            IF HasHalted ()
                THEN
                    (* specific action to handle exception after HALT *)
                END;
            ELSE
```



```

    (* action to handle exception before termination commenced *)
END;
END ExceptionTermination4.

```

10.12.5 User-Defined Exceptions

This subsection addresses the first five points in the list at the beginning of section 10.11, and that apply only to code in which user defined exceptions are to be defined, detected, and raised, namely:

1. The conditions under which exceptions are raised must be defined.
2. All exceptions must be named.
3. The source of the exception must be defined and registered.
4. The conditions producing the exception must be detected.
5. The exception must be raised, providing the source and a message.

First, the programmer must determine whether it is necessary to use exceptions at all. As illustrated above, most exceptional circumstances can be avoided. However, the code in a library cannot itself guarantee that the user will always employ its routines correctly, despite the stated preconditions. Thus, it may be necessary for code to be able to raise an exception if it is misused. As in all aspects of planning, it is essential that the logic of this be carefully thought out before being committed to code.

Second, exceptions need to be declared. As illustrated by the interface to the standard libraries, this is just an enumeration:

```

TYPE
  MyExceptions = (JohnGough, KeithHopper, AlbertMeier);

```

If this is done as part of a library, enquiry functions should be provided as well, and these may be modeled on the ones found in *IOChan*. The enumeration, and the second function are of course unnecessary if the library module only has one exception, and both procedures are unnecessary if the exceptions are defined, detected, and raised within the confines of the main program module.

```

PROCEDURE IsMyException (): BOOLEAN;
  (* Returns TRUE if the current coroutine is in the exceptional execution state
  because of the raising of an exception from MyExceptions; otherwise returns FALSE. *)

```

```

PROCEDURE MyException (): MyExceptions;
  (* If the current coroutine is in the exceptional execution state because of the
  raising of an exception from MyExceptions, returns the corresponding enumeration
  value, and otherwise raises an exception. *)

```

Third, in order to protect the language itself and individual library modules from having their exceptions misused by some code other than that entity entitled to use them, every *source* of exceptions must register itself with the module EXCEPTIONS (another system module). In so doing, the source will receive a unique value for an identifier of type *ExceptionSource* that must be supplied whenever actually raising an exception. Since there is no way for another module to discover this value, only the code that registers itself as a source can raise the exceptions that belong to that source. This is all done by

```

FROM EXCEPTIONS IMPORT
  ExceptionSource, AllocateSource;
VAR
  myExSource : ExceptionSource;
BEGIN
  AllocateSource (myExSource);

```

Fourth, the conditions determined in the planning stage are coded appropriately.

Fifth, the exception is actually raised as a result of the detected conditions. When this is done, the *source* must be provided,

along with a message. Typically the message is a brief description of the problem that can be printed during the course of handling the exception. These tasks are accomplished using:

```
FROM EXCEPTIONS IMPORT
    RAISE; (* in addition to the stuff above *)

and then in the actual code:

(* statement sequence *)
IF brokenAussie
    THEN
        RAISE (myExSource, ORD (JohnGough), "Down under task bad.")
    ELSIF badKiwiError THEN
        RAISE (myExSource, ORD (KeithHopper), "Hopper already Empty")
    ELSIF AlpineSlide THEN
        RAISE (myExSource, ORD (AlbertMeier), "Matterhorn has fallen")
    END;
```

A **FINALLY** clause can determine whether the current exception state is exceptional or not, and if so (either it or possibly an exception handler for a module or procedure), can enquire of the module **EXCEPTIONS** whether or not it is the source of an exception, if so, which one it is, and what is the error message. (The error message may be used by an automatic handler at the outermost level as well). This is done as in the following illustration:

```
FROM EXCEPTIONS IMPORT
    CurrentNumber, GetMessage, IsCurrentSource, IsExceptionalExecution,
    ExceptionNumber;
    (* in addition to all the stuff above *)
VAR
    theErrorNumber : ExceptionNumber; (* just a cardinal *)
    stringVar : ARRAY [0..50] OF CHAR
```

and in the code

```
FINALLY
    IF (IsExceptionalExecution() ) AND (IsCurrentSource (myExSource) )
        THEN
            theErrorNumber := CurrentNumber (myExSource);
            GetMessage (stringVar);
            (* take further action *)
        END;
```

As one might expect, an attempt to obtain a value from *CurrentNumber* when *myExSource* is *not* in fact the source of the exception will in itself cause an exception to be raised, for this is interpreted as an attempt to steal information that belongs to some other source of exceptions. Note, however, that the message of the current exception is accessible to any caller without having to provide a source identity check.

Observe that apart from the enquiry function *IsExceptionalExecution* (and possible *GetMessage*), that might be useful in any **FINALLY** clause, none of the other items in **EXCEPTIONS** need to be imported unless the code in question is to define, detect, and raise its own exceptions. Indeed, exception *handling* need not use anything at all from this module.

10.13 An Extended Example--Fractions and Exceptions

The purpose of this section is to illustrate some of the ideas presented in the last section in a longer piece of code. For simplicity, easy comparison, and to avoid having to present planning steps not relevant to this section, consider one way of adding appropriate exceptions to the Module *Fractions* first developed in chapter six.

Three operations with fractions can be identified as ones that, if proper data is provided, there will be no error, but if improper data is passed the fractions are invalid. (This makes the module a good candidate for having some exceptions defined--it can control its own code, but not the mistaken calls of clients using erroneous data.) These are:

1. assignment of a zero denominator,
2. taking the inverse when there is a zero numerator, and
3. dividing when the divisor has zero numerator.

These are similar errors, and all could be given the same exception identifier, but for the purpose of illustrating features of exceptions, all three will be treated separately.

In the definition module that follows, alterations have been made to the original in accordance with the remarks in the last section.

DEFINITION MODULE Fractions;

```
(* Written by R.J. Sutcliffe *)
(* using ISO Modula-2 *)
(* to illustrate exceptions use in libraries *)
(* last revision 1994 05 31 *)
```

TYPE

```
Fraction = ARRAY [1 .. 2] OF INTEGER;
(* the first component is the numerator; the second the denominator *)
```

PROCEDURE Assign (m, n : **INTEGER**) : Fraction;

```
(* If n is not equal to zero, then the fraction returned has m as numerator and n
as denominator. Otherwise the exception zeroDenominator is raised. *)
```

PROCEDURE Numerator (x : Fraction) : **INTEGER**;

```
(* the numerator of the fraction is returned *)
```

PROCEDURE Denominator (x : Fraction) : **INTEGER**;

```
(* the denominator of the fraction is returned *)
```

PROCEDURE Neg (x : Fraction) : Fraction;

```
(* Pre: the fraction returned has the numerator negated *)
```

PROCEDURE Inv (x : Fraction) : Fraction;

```
(* If the numerator of x is not equal to zero then the fraction returned has
numerator and denominator swapped. Otherwise the exception noInverse is raised. *)
```

PROCEDURE Add (x, y : Fraction) : Fraction;

```
(* The fraction returned is the sum x plus y *)
```

PROCEDURE Sub (x, y : Fraction) : Fraction;

```
(* The fraction returned is the difference x minus y *)
```

```

PROCEDURE Mul (x, y : Fraction) : Fraction;
    (* The fraction returned is the product of x and y *)

PROCEDURE Div (x, y : Fraction) : Fraction;
    (* If the numerator of y is not equal to 0, then the fraction returned is the
    quotient of x by y.  Otherwise, the exception zeroDivide is raised *)

TYPE
    FracExceptions = (zeroDenominator, noInverse, zeroDivide);

PROCEDURE IsFracException (): BOOLEAN;
    (* Returns TRUE if the current coroutine is in the exceptional execution state
    because of the raising of an exception from FracExceptions; otherwise returns FALSE.
    *)

PROCEDURE FracException (): FracExceptions;
    (* If the current coroutine is in the exceptional execution state because of the
    raising of an exception from FracExceptions, returns the corresponding enumeration
    value, and otherwise raises an exception. *)

END Fractions.

```

Note in the following implementation that a choice has been made to report any exception occurrence and print the associated string in the termination part of the module.

```

IMPLEMENTATION MODULE Fractions;

(* Written by R.J. Sutcliffe *)
(* using ISO Modula-2 *)
(* to illustrate exceptions use in libraries *)
(* last revision 1994 05 31 *)

FROM EXCEPTIONS IMPORT
    ExceptionSource, AllocateSource, RAISE, IsExceptionalExecution, IsCurrentSource,
    CurrentNumber, GetMessage;
FROM STextIO IMPORT
    WriteString, WriteLn, SkipLine, ReadChar;

VAR
    fracExSource : ExceptionSource;

PROCEDURE Assign (m, n : INTEGER) : Fraction;

VAR
    temp : Fraction;

BEGIN
    IF n = 0
    THEN
        RAISE (fracExSource, ORD (zeroDenominator), "Cannot assign fraction with zero
        denominator");
    ELSE
        temp [1] := m;
    END
END

```

```

        temp [2] := n;
    RETURN temp;
END;
END Assign;

PROCEDURE Numerator (x : Fraction) : INTEGER;

BEGIN
    RETURN x [1];
END Numerator;

PROCEDURE Denominator (x : Fraction) : INTEGER;

BEGIN
    RETURN x [2];
END Denominator;

PROCEDURE Neg (x : Fraction) : Fraction;

BEGIN
    x [1] := -x [1];
    RETURN x;
END Neg;

PROCEDURE Inv (x : Fraction) : Fraction;

VAR
    temp : INTEGER;

BEGIN;
    IF Numerator (x) = 0
    THEN
        RAISE (fracExSource, ORD (noInverse), "Cannot invert fraction with zero
numerator");
    ELSE
        temp := x [1];
        x [1] := x [2];
        x [2] := temp;
        RETURN x;
    END;
END Inv;

PROCEDURE Add (x, y : Fraction) : Fraction;

VAR
    temp : Fraction;

BEGIN
    temp [1] := x [1] * y [2] + x [2] * y [1];
    temp [2] := x [2] * y [2];
    RETURN temp;
END Add;

PROCEDURE Sub (x, y : Fraction) : Fraction;

```

```

BEGIN
    RETURN Add (x, Neg (y) );
END Sub;

PROCEDURE Mul (x, y : Fraction) : Fraction;

BEGIN
    RETURN Assign (x [1] * y [1], x [2] * y [2]);
END Mul;

PROCEDURE Div (x, y : Fraction) : Fraction;

BEGIN
    IF Numerator (y) = 0
    THEN
        RAISE (fracExSource, ORD (zeroDivide), "Cannot divide by zero");
    ELSE
        RETURN Mul (x, Inv (y) );
    END;
END Div;

PROCEDURE IsFracException (): BOOLEAN;
BEGIN
    RETURN (IsExceptionalExecution() ) AND (IsCurrentSource (fracExSource) )
END IsFracException;

PROCEDURE FracException (): FracExceptions;
(* The call to CurrentNumber will raise ExException automatically if this source
didn't raise an exception. *)
BEGIN
    RETURN VAL (FracExceptions, CurrentNumber (fracExSource) );
END FracException;

VAR
    errorMessage : ARRAY [0..255] OF CHAR;

BEGIN (* initialize *)
    AllocateSource (fracExSource);

FINALLY
    IF IsFracException ()
    THEN
        GetMessage (errorMessage);
        WriteString ("Program terminating because of exception");
        WriteLn ;
        WriteString (errorMessage);
        WriteLn;
        WriteString ("Type return to continue");
        SkipLine;
    END;

```

END Fractions.

There are slight dangers in putting code into a **FINALLY** clause that depends on the importation of some other module, as in this case.

First, if termination is caused by an exception during initialization, then it may be that the module required (*STextIO* here) has not yet been initialized (and therefore cannot be used correctly). This should not happen in this case, because all the exceptions that can be raised by this module are in code that is unlikely to be called during the initialization of library modules.

Second, modules are terminated in the reverse order that they are initialized. If the module being employed in **FINALLY** clause has already been terminated, it *may* be that some facilities it offers are no longer available. (An implementation may choose to close all channels during finalization, for instance). This latter problem can only be determined to exist by examining the implementation documentation.

Note that if either problem *does* exist, there is no work around, because:

The correctness of a program depends on the meaning of its individual modules, but any program whose meaning depends on the order of initialization (or finalization) of its modules is incorrect.

In this case, the following simple application was run to do a limited test of the new exceptions, by forcing one of them to be raised:

```
MODULE TestFractions;  
  
IMPORT Fractions;  
  
VAR  
    p, q, r : Fractions.Fraction;  
  
BEGIN  
    p := Fractions.Assign (0, -3);  
    q := Fractions.Assign (4, -3);  
    r := Fractions.Div (q, p);  
END TestFractions.
```

The exception was triggered, and checking the output log after the run was complete revealed its contents as:

```
** Run log starts here **  
Program terminating because of exception  
Cannot divide by zero  
Type return to continue
```

[Contents](#)

10.14 Chapter Summary

This chapter covered these topics:

- more about scope and visibility and how to control them

It included discussion of the following Modula-2 built-ins:

Reserved Words	Standard Identifier
FINALLY	EXCEPT
LOOP	EXIT
EXPORT	HALT
IMPORT (new sense)	RETURN (new sense)
QUALIFIED	RETRY

Standard Library Imports

From TERMINATION:

HasHalted, IsTerminating

From M2EXCEPTION:

M2Exceptions, M2Exception, IsM2Exception

From EXCEPTIONS

ExceptionSource, AllocateSource, RAISE, CurrentNumber, GetMessage, IsCurrentSource, IsExceptionalExecution, ExceptionNumber

From IOChan:

ChanExceptions, IsChanException, ChanException, DeviceErrNum, DeviceError

10.15 Assignments

Questions

1. What is a block?
2. What differences are there between a procedure block and a module block?
3. When *must/must not* a module have a block body?
4. Define *scope*.
5. Which of (a) a Procedure's name (b) its parameters (c) its local variables are part of its scope?
6. Under what circumstances might a procedure's name *not* be visible in its body?
7. Define the terms *local* and *global* as they pertain to procedures.
8. Why is it important to avoid using many global variables?
9. Suppose the declaration `VAR z : BOOLEAN` were contained in a program module, again in a procedure, and yet again in another procedure inside this one. Does an assignment to *z* in the innermost procedure change the value of the variable in the main program? in the first procedure? in some other procedure declared in the main program? in some other procedure defined inside the first one? What if yet another procedure were declared inside the one innermost above, this one with no `BOOLEAN`s declared in it, but also with an assignment to *z*. At what level(s) in the above hierarchy is *z* affected?
10. What does *visibility* mean?
- 11 Define the term *side effect*.
12. Under what circumstances are side effects good? bad?
13. Under what circumstances might a program have many variables with the same name, and how does the compiler distinguish among them?
14. Describe the differences between the scope rules for nested procedures and those for nested modules.
15. What is meant by "the outermost scope level?"
16. How do modules import two items of the same name into the same scope?
17. Can two modules `EXPORT` items of the same name into the same scope?
18. What is the difference between a variable that is `EXPORTed` and one that is `EXPORTed` as `QUALIFIED`? Illustrate with examples.
19. A program module cannot have an `EXPORT` list. Why?
20. What is the difference between a standard identifier and a reserved word?
21. What is a dynamic module?
22. How may the visibility of identifiers inside a dynamic module be different from that in the scope immediately surrounding it?
23. What is a Fibonacci sequence?
24. What is the difference between an opaque type and a transparent type?
25. What is a procedure type?
26. Show how to define procedure types in each case that could have the following assigned to it.
(a) `SeqFile.OpenWrite`

- (b) `STextIO.WriteString`
 - (c) `RealIO.WriteFixed`
 - (d) `Strings.Append`
 - (e) `LongMath.ln`
27. When should you use or not use the `LOOP .. EXIT` construction?
 28. What two Modula-2 commands cause the program to unconditionally transfer control to some other line of code?
 29. Which two commands can cause a program to terminate?
 30. In what other way, besides the issuing of a direct command, might a program be caused to terminate?
 31. How is a termination event detected?
 32. How does one determine whether a program is in normal or exceptional execution state?
 33. What is an exception?
 34. How is an exception trapped?
 35. What are the three strategies that can be followed in an exception handler?
 36. What specific tools are necessary, and how are they used, to declare and detect exceptions in user-written code?
 37. How do the exceptions facilities prevent the raising of exceptions by code that does not contain the definition of those exceptions in its own scope?
 38. How many exception handlers may a program module contain?
 39. How many exception handlers may the body of a module itself have attached?
 40. What is the output of the following program?

```
MODULE Scramble;  
FROM SWholeIO IMPORT  
    WriteCard;  
FROM STextIO IMPORT  
    WriteLn;  
VAR  
    w, x, y, z : CARDINAL;  
  
PROCEDURE Scram (x, y : CARDINAL);  
  
VAR  
    z : CARDINAL;  
BEGIN  
    w := 7;  
    x := 8;  
    y := 9;  
    z := 10;  
END Scram;  
  
BEGIN    (* Main *)  
    w := 1;
```

```

x := 2;
y := 3;
z := 4;
WriteCard (w, 6);
WriteCard (x, 6);
WriteCard (y, 6);
WriteCard (z, 6);
WriteLn;
Scram (x, y);
WriteCard (w, 6);
WriteCard (x, 6);
WriteCard (y, 6);
WriteCard (z, 6);
WriteLn;

```

END Scramble.

41. Make a list of the variables visible at the places marked *point 1*, *point 2*, etc. If some are visible provided that they are written as qualified identifiers, please indicate this also. This question of course assumes a multi pass compiler. Why?

MODULE Exercise;

VAR

```

    outA, outB : REAL;

```

MODULE Inner1;

EXPORT inner1C, days;

TYPE

```

    days = (Sat, Sun);

```

VAR

```

    inner1C, inner1D : REAL;
    (* point 1 *)

```

END Inner1;

MODULE Inner2;

EXPORT QUALIFIED inner2E, Inner3;

VAR

```

    inner2E, inner2F : REAL;
    (* point 2 *)

```

```

MODULE Inner3;

EXPORT inner3H;

VAR
    inner3G, inner3H : REAL;

END Inner3;
(* point 3 *)

END Inner2;

(* point 4 *)
END Exercise.

```

Problems:

42. Give an example of a program (not the same as in the text) in which unexpected results are obtained because a procedure modifies a variable global to it.
43. Give an example of a program (not the same as in the text) in which unexpected results are obtained because a procedure has a variable parameter when a value parameter would do.
44. Expand upon the [bar graph program](#) in section 10.2 as follows: Write a module that allows a user to enter data in several (1 - 10) subcategories of some main category and that then prints out an appropriately scaled bar graph that pictures the data. Make the program module general, but test it on the following data:

The number of students at TWU taking each of the following courses are in parentheses: Chemistry (120), Physics (70), Mathematics (200), Biology (130), and Computer Science (200).

45. Write a program module to examine the contents of Modula-2 program files and count the number of occurrences of the various reserved words and standard identifiers. Use a LOOP in the routine that reads the file.
46. Select a program from [chapter 8](#) that employed *StreamFile* and add to it a FINALLY clause that closes any channels still open.
47. Select a program from [chapter 8](#) that employed *SeqFile* and add to it a FINALLY clause that closes any channels still open.
48. Select a program from [chapter 8](#) that employed *RndFile* and add to it a FINALLY clause that closes any channels still open.
49. Revise the module [PointToPoint](#) in chapter 6 to add a procedure to compute the slope (using the slope formula) of a segment joining two points. Forbid vertical segments with an appropriate precondition, but add an exception to cover cases where client software ignores the rules. Encapsulate the whole thing in a library module and test the result.
50. Revise the module [Fractions](#) in this chapter to raise a different exception *indeterminate* in cases where both the numerator and denominator are zero. Do a complete test of the result.

51. Add exceptions as appropriate to the module [Stats](#) in chapter 7, and test the result.
52. Write a program module to demonstrate (by catching and handling the exception and printing the message) that an ISO library module that tries to close a channel it did not open causes an exception.
53. Write a program module to demonstrate (by catching and handling the exception and printing the message) that a program that attempts to access an array element with too large an index causes an exception.
54. Write a program module to demonstrate (by catching and handling the exception and printing the message) that a program that attempts to obtain the error number of an exception but provides the wrong source identifier causes an exception.
55. Write a program module to demonstrate (by catching and handling the exception and printing the message) that a program that attempts to use *RndFile.NewPos* to compute a file position past the end of the file causes an exception.

Challenges

56. Modify the program in #15 to print the bar graphs with vertical bars instead of horizontal ones.
 57. Write a new version of the ISO library module [Strings](#) with the same content as in the standard, but with exceptions added to cover all those cases where attempts are made to overfill strings. That is, require the user of the module to employ such procedures as *CanDeleteAll* and *CanInsertAll*, before *Delete* and *Insert* are used, by removing the silent truncation semantics from the latter and replacing them with the raising of exceptions when the limits are violated.
-

[Contents](#)

Chapter 10

Intermediate Program Structuring

[10.0 Chapter Goals](#)

[10.1 Introduction](#)

Part A--Scope and Visibility Issues

[10.2 Blocks, Global and Local Variables, Side Effects](#)

[10.2.1 Procedure Blocks and Scope](#)

[10.2.2 Side effects and Counting Loops](#)

[10.2.3 Other Global side effects](#)

[10.2.4 Nested Procedure Scopes](#)

[10.3 Parameters Revisited](#)

[10.3.1 The Scope of Parameters](#)

[10.3.2 Parameters and side effects](#)

[10.4 Procedure Types and their Variables](#)

[10.5 Local Modules--Scope and Visibility Rules](#)

[10.5.1 Standard Identifiers and Scope](#)

[10.5.2 Dynamic Modules](#)

[10.6 An Extended Example--Fibonacci Sequences](#)

[10.7 Library Modules--Scope and Visibility Rules](#)

[10.7.1 Access to Imported Libraries at Inner Scopes](#)

[10.7.2 Visibility in Library Modules](#)

[10.7.3 Opaque Types--a Brief Introduction](#)

Part B--Program Control and Error Handling Issues

[10.8 Transfer of Control](#)

[10.8.1 GOTO and Other Noxious Weeds](#)

[10.8.2 RETURN in a Regular Procedure](#)

[10.8.3 Repetition Revisited--The Generalized LOOP Statement](#)

[10.9 Error Handling Revisited](#)

[10.9.1 Typical Library Errors](#)

[10.10 Controlling Program Termination](#)

[10.10.1 HALT and Abnormal Termination](#)

[10.11 FINALLY: Termination Detection and Cleanup](#)

[10.12 Exceptions](#)

[10.12.1 Language Exceptions](#)

[10.12.2 Library Exceptions](#)

[10.12.3 Handling Exceptions](#)

[10.12.4 Exceptions and Termination](#)

[10.12.5 User-Defined Exceptions](#)

[10.13 An Extended Example--Fractions and Exceptions](#)

[10.14 Chapter Summary](#)

[10.15 Assignments](#)

[**Contents**](#)

Chapter 11 Intermediate Programming-- Data and Techniques

[11.0 Chapter Goals](#)

[11.1 Introduction](#)

Part A--Programming Techniques

[11.2 Recursion Revisited](#)

[11.2.1 The Knight's Tour--An Extended Example](#)

[11.3 Selection Revisited-The CASE Statement](#)

[11.4 Pragmas](#)

[11.5 Efficiency in Large Programs](#)

[11.5.1 Controlling Run-Time Checking](#)

[11.5.2 Compiling For Specified Environments](#)

[11.5.3 Fine Tuning Loops](#)

[11.5.4 Linking, Program Libraries and Speed](#)

[11.5.5 Efficiency--A Summary](#)

Part B--Intermediate (Structured) Data Issues

[11.6 Structure Constructors](#)

[11.6.1 Array Constructors](#)

[11.6.2 Record Constructors](#)

[11.7 The Variant Record--a Chameleon](#)

[11.8 An Extended Example--Variant Records](#)

[11.9 Chapter Summary](#)

[11.10 Assignments](#)

[Contents](#)

12.0 Chapter Goals

The purpose of this chapter is to introduce the use of pointers and some applications that use them. As well, the concept of dynamic data is explored and illustrated with some elementary dynamic structures. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

Static and dynamic representations of existing data types are discussed and the concept of a dynamic list is introduced.

Realized in the Modula-2 notation:

pointers, handles, dynamic data structures and linked lists

Data Manipulation Abstractions

General:

manipulation of items stored in dynamic memory

Realized in the Modula-2 notation:

the dereferencing operator, NEW, DISPOSE, Storage.ALLOCATE and Storage.DEALLOCATE

Programming Abstractions

General:

using stack and heap memory, applications of pointers, dynamic types and ADTs

Realized in the Modula-2 notation:

opaque types and variant dynamic record memory allocation

12.1 Pointers

Pointers were briefly mentioned in [section 8.3.1](#) in the discussion of the contents of the pseudo-module SYSTEM, which uses the definition:

TYPE

```
ADDRESS = POINTER TO LOC;
```

Because this definition is phrased as a specific instance of the pointer type, it should be apparent that a pointer may be something more general than the address of a LOC (smallest storage location). In fact a pointer may hold the location of any data. This is only a slight conceptual generalization, however, as the pointer will still hold the address of a LOC--the first unit of storage belonging to the data in question.

A pointer or reference variable identifies a memory location that holds the address of some other entity. It points to that other entity.

Although an addressible location might not be called a LOC in other programming notations, this definition is a general one, and is not specific to Modula-2.

12.1.1 Pointer Variables

Of course in Modula-2, identifiers for pointer variables have to be declared using the usual syntax, for instance, for a type whose entities will point to integers:

TYPE

```
IntPoint = POINTER TO INTEGER;
```

VAR

```
iPoint : IntPoint;  
int : INTEGER;
```

Following these declarations, assignments could be made such as:

```
iPoint := SYSTEM.ADR (int);
```

Conceptually, items of the type *IntPoint* point to an entire integer, whatever number of memory locations an integer occupies. On the other hand, there is a sense in which all pointers are the same type (ADDRESS) even though conceptually each pointer type is different, depending on the type of data they point to. Thus the Modula-2 compatibility rule is:

Items of any pointer type are assignment compatible to the type SYSTEM.ADDRESS. Two different pointer types are not assignment compatible with each other, but can be CAST to another if required.

To illustrate, if one also had

TYPE

```
RealPoint = POINTER TO REAL;
```

VAR

```
rPoint, rPoint2 : RealPoint;  
re : REAL;  
adr: SYSTEM.ADDRESS;
```

then the following are all legal:

```
adr := iPoint;  
adr := rPoint;  
rPoint := SYSTEM.CAST (RealPoint, iPoint); (* most have a good reason for this *)
```

but the following are all illegal because the types pointed to are incompatible, and therefore so are the pointer types:

```
iPoint := rPoint;  
rPoint := iPoint;
```

Pointer types *may*, as in the examples, point to numeric entities, but are not themselves numeric, and therefore *none* of the usual numeric operations (+ - * /) can be performed on variables of these types. Two pointers of the same type can be compared to one another using either "=" or "#" but not with "<" or any other comparison operators. Thus,

```
IF rPoint = rPoint1
```

is legal, but

```
IF rPoint < iPoint
```

is not allowed.

12.1.2 Pointer References

It is worth observing that the memory pointed to by a pointer type has a type of its own, but not a name of its own; its name may be regarded as anonymous. Actual references to the memory pointed to are made by using the pointer name, followed by the symbol "^."

Using a pointer to access the memory to which it points is called dereferencing the pointer.

In the cases shown, one could initialize the contents of a memory location pointed to in any of the following ways (among others):

```
iPoint^ := int;  
iPoint^ := -5;  
rPoint^ := 3.24E-7;  
TextIO.ReadReal (rPoint2^);
```

NOTES: 1. On some older systems, the ^ or *caret* character was written as the *up-arrow* character ↑, but on most, these are two distinct characters. Where both exist, the one desired here is the *caret*.

2. The name *caret* is often shortened to *hat* and one then pronounces *point*^ as *point-hat*.

3. Recall that an ADDRESS is a POINTER TO LOC. Thus, if *Ad* is of type ADDRESS, then *Ad*^ is of type LOC.

It is easiest to remember the meaning of *point*^ if one thinks of it as "the thing pointed to by *point*". It must be kept very

clear that *point* and *point*[^] are two entirely different entities--the first is the name of the pointer variable whose contents are the number or address of a memory location, and the second is the name of the entity situated at that location. So, when the *point* is declared, only enough memory for the pointer is set aside. The space for *point*[^] must be obtained separately, either by declaring an entity of the type that can be pointed to, or by executing code that allocates memory at run time (see [section 12.5](#)).

The Modula-2 symbol "[^]" when affixed to an identifier is called the dereferencing operator.

To further illustrate by a map or picture of memory, suppose a program contains the declarations and code fragment:

```
TYPE
  sPoint = POINTER TO Student;
  Student =
    RECORD
      name : ARRAY [0..80] OF CHAR;
      number : CARDINAL;
    END;

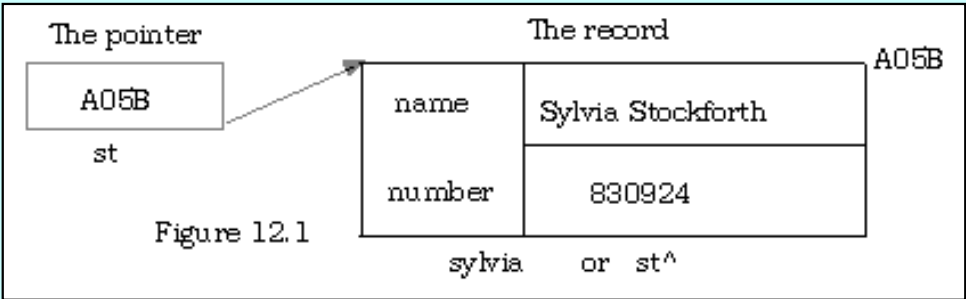
VAR
  st : sPoint;
  sylvia : Student;

BEGIN
  st := SYSTEM.ADR (sylvia);
  st^.name := "Sylvia Stockforth";
  st^.number := 830924;
```

Note, by the way, that *st*[^] is a record, so its field names are referred to in the usual way; one could also use

```
WITH st^
DO
  name := "Sylvia Stockforth";
  number := 830924;
END;
```

At this juncture, one could envision the memory contents looking as in figure 12.1 (the address A05B has been chosen arbitrarily).



At this point, one might legitimately ask why a programmer should go to this trouble when the variable *sylvia* can be more easily referred to directly than via a pointer whose value consists of its address. The answer is that not all situations are as simple as the one used here to illustrate the basic ideas; there are others in which pointers are quite useful, or even necessary.

12.1.3 The value NIL

Sometimes it is useful to initialize pointer variables to a "safe" value, that is, to a value that does not actually point anywhere. This special value has the standard identifier *NIL*. Just as declaring a Modula-2 numeric variable does not set its value (say, to 0) without an explicit initialization statement in the code, declaring a pointer variable does not set its value to *NIL*--a specific value--but to an indeterminate value. A not-intentionally-initialized variable has whatever value found in memory when that memory is allocated, and that could be anything.

The Modula-2 standard identifier NIL has an anonymous type called the nil-type, and all pointer types are compatible with the nil-type.

This rule means that a variable of any pointer type, including the type ADDRESS, can be given the value *NIL*.

NOTES: 1. It causes a run time error (raises an exception) to make a reference to *NIL*[^]. Thus, a reference to *point*[^] when *point* happens to equal *NIL*, will always cause an error.

2. Some implementations have a compiler option (command line and/or pragma) to force automatic initialization of numeric variables to *zero* and pointers to *NIL*. This should not be relied upon, as such code would not be portable.

The short-circuited Boolean expression evaluation feature of Modula-2 can come in handy to prevent such erroneous references as this. Rather than writing

```
IF p^ = theInterestingValue
```

and taking the chance that the pointer might be *NIL*, one should always write:

```
IF (p # NIL) AND (p^ = theInterestingValue)
```

so that if in fact *point* does equal *NIL*, the right side of the expression will not be evaluated at all, neatly avoiding the potential error. The potentially dangerous evaluation has been *guarded* by the prior boolean condition that prevents the problematic evaluation from taking place if it would be erroneous. This correction from the Pascal rule for Modula-2 of the logical rules for evaluating Boolean expressions was designed primarily with this very situation in view.

[Contents](#)

12.2 Applications of Pointers

12.2.1 Pointers and Parameters

Sometimes procedures need to operate on rather large data collections. If these are passed as value parameters, considerable resources of both time and memory may be consumed in copying the information from the main memory to the portion employed by the procedure. There are two possible ways around this problem.

One is to use a variable parameter, contrary to the usual recommendation for parameter passing. There is no copying, and the procedure's formal parameter becomes an alias to the actual parameter.

A second solution would be to pass a pointer to the data as a value parameter, rather than the data itself. For instance, if one had:

TYPE

```
DataArray = ARRAY [1 ..100000] OF REAL;
```

VAR

```
theData : DataArray;
```

PROCEDURE Mean1 (data : DataArray; lBound, uBound : **CARDINAL**) : **REAL**;

(* find the mean of the array data from item #lBound to item #uBound *)

VAR

```
count, number : CARDINAL;
```

```
total : REAL;
```

BEGIN

```
number := uBound - lBound + 1;
```

```
IF number <= uBound
```

```
    DO
```

```
        total := total + data [count];
```

```
        INC (count);
```

```
    END;
```

```
    RETURN total / FLOAT (number)
```

```
ELSE
```

```
    RETURN 0.0
```

```
END
```

```
END Mean1;
```

and invoked this with

```
mean := Mean1 (theData, 1, 100000);
```

it might be useful to pass a value parameter pointer to the data array, rewriting things as:

TYPE

```
DataPoint = POINTER TO DataArray;
```

VAR

```
dataP : DataPoint;
```

```
PROCEDURE Mean2 (data : DataPoint; lBound, uBound : CARDINAL) : REAL;
```

with all as above except the line

```
total := total + data^ [count];
```

and invoke this by:

```
dataP := SYSTEM.ADR (theData);  
mean := Mean2 (dataP, 1, 100000);
```

In this manner, the procedure is referring to the original data, and only a pointer to that data needs to be passed rather than copying all of it to a value parameter.

The astute student will note at this juncture that the effect is exactly the same as if a variable parameter had been used in the first place, and wonder why one would go to all this trouble to imitate something that is already in the language. The answer is that in this case one would decidedly not go to the trouble.

However, many, if not most systems actually implement variable parameters by passing a pointer.

Although this action is transparent to the user, it is useful to know how it can be done (probably is done).

12.2.2 Pointers and Sorting

The example just concluded ought to make the same astute reader ask whether there are any other situations involving the moving of large quantities of data that can be avoided by the judicious use of pointers.

When one is sorting cardinals as will be done in the examples of chapter 13, the data movement involves rather small items. However, suppose that instead of sorting an array of cardinals, one is sorting arrays of rather large records. In such a situation, the movement of the data items can become so large a task that it overwhelms that of making comparisons, and any sorting method would quickly run into performance difficulties. In such cases, it may be best to keep an array of pointers to the data, make comparisons on the original data, and swap around only the pointers. This is especially true if one wants to sort the records according to different field values at different times (or even at the same time). To see the details of how this is achieved in a particular instance can be seen in [section 13.6.2](#).

Contents

12.3 Pointer Arithmetic

It was noted above that operations (other than assignment) on pointer variables are not allowed. However, the pseudo-module **SYSTEM** contains the function procedures **ADDADR**, **SUBADR** and **DIFADR** for manipulating address variables. Although these take parameters and return values of type **ADDRESS**, the parameters are value parameters, not variable parameters. Thus, the assignment compatibility between pointer types and address types means that these function procedures can be applied to pointers and can return values to be assigned to pointers in expressions. In addition, these items are in **SYSTEM**, and would not necessarily be limited by such considerations, in any case, as the routines of **SYSTEM** can almost be expected to behave by unique rules. Here are some fragments of correct code:

```
FROM SYSTEM IMPORT
    ADDRESS, ADR, ADDADR, SUBADR, DIFADR;

VAR
    point, point2 : POINTER TO INTEGER;
    addr : ADDRESS;
    int : INTEGER;

BEGIN
    point := ADDADR (point, 10);
    point := SUBADR (point, 10);
    int := DIFADR (point, point2);
```

It is important to note that manipulating pointers in this fashion requires a thorough knowledge of the specific machine being used and particularly the manner in which it uses addresses and pointers. Because of potential portability problems it is therefore not likely that many programmers will have a great deal of use for this facility. However, one could take advantage of the fact that the elements of an array are stored contiguously to set up the addresses in the example above as follows:

```
theStuffNumAdrs [1] := ADR (theStuff [1]);
FOR count := 2 TO 13 (* set up addresses *)
    DO
        theStuffNumAdrs [count] := ADDADR (theStuffNumAdrs [count - 1], SIZE (Data))
    END;
```

This facility can also be useful when a particular machine is known to have certain specific memory set aside for special purposes, and it is desired to access that memory via pointers. Of course, such code is non-portable, as it makes assumptions about particular hardware.

12.4 Dynamic and Static Memory

Readers who are familiar with the concepts of dynamic memory and pointers may wish to skip to the next section of this chapter. In this one, the general concepts of static and dynamic memory is outlined. How these issues are specifically handled in Modula-2 is not taken up again until after the rest of the preliminary discussions are complete.

The distinction made in this section is based on the timing and manner for the setting aside of memory for the use of a program. Some memory use is predetermined by the compiler and will always be set aside for the program in exactly the same manner at the beginning of every run of a given program. Other memory is obtained by the program at various points during the time it is running. This memory may only be used by the program temporarily and then released for other uses.

The allocation of memory for the specific fixed purposes of a program in a predetermined fashion controlled by the compiler is said to be static memory allocation.

The allocation of memory (and possibly its later deallocation) during the running of a program and under the control of the program is said to be dynamic memory allocation.

12.4.1 Static Memory Use

By the time a program begins to execute, there must be some specific blocks of memory set aside for its use that cannot be trespassed upon by any other program, by the system, or even by the program itself. This includes, for instance, the memory containing the program's own code. While it is possible (in machine language) to write a program that can modify its own code, this is a very dangerous practice, and should never be employed. It is not possible to do this at all in most high level languages such as Modula-2.

Moreover, any variables named in the declaration section must have specific memory set aside for their contents, and this action can not be controlled or changed in any way by the programmer, except by declaring more or fewer variables in the first place. The memory in question cannot itself be relocated to some other place or expanded or contracted.

Static variables are those that are created in the declaration section of a program and continue to exist (whether visible or not) and to require that space until its conclusion. Their space is allocated at the beginning of the program run.

To be very specific:

1. The only changes that can take place to static variables are to their *contents* and this is done by assignment statements.

2. The Modula-2 procedure `SIZE` can be employed to determine the number of LOCs set aside for a particular variable, but the compiler itself can do this arithmetic as it must "know" these amounts ahead of time. This means, for instance, that `SIZE` can be employed in a constant expression.

3. The procedure `SYSTEM.ADR` may be employed to determine the location of this memory, but the location cannot be changed as the code generated by the compiler for such things as assignments depends on the pre-determined location.

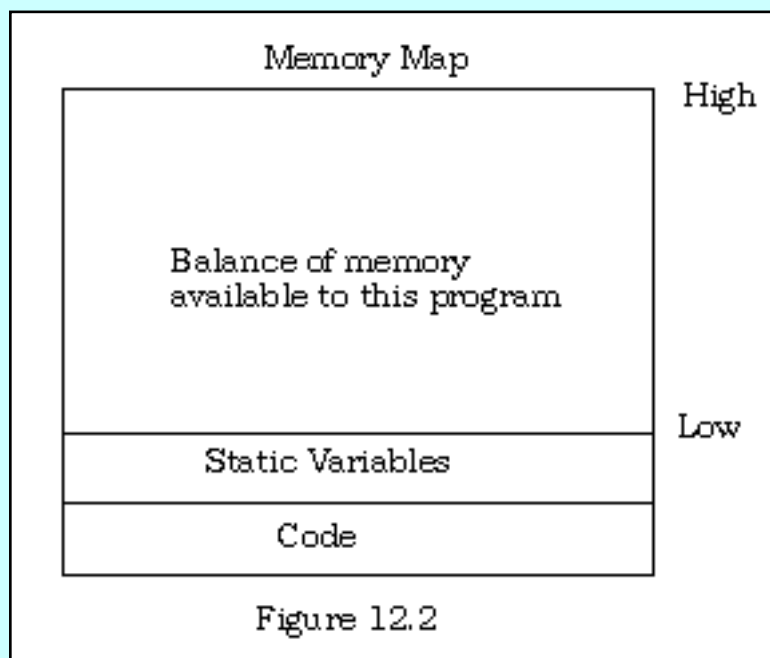
4. As indicated in Chapter 8, it *may* be useful in some cases to declare a variable at a fixed address, but this too must be done ahead of time, and cannot be changed while the program is running.

Items of all the data types considered so far are of the static kind--once declared, they will be at a fixed location and consume a specific amount of memory during the running of the program. Both size and location (relative to the start of the code) are predetermined at the time the program is compiled. The location is relative, and not absolute because there is no way for the compiler to determine ahead of time how many programs will be running and what memory will be already in use when the new program is loaded. However, with respect to program starting address, it is fixed and cannot be changed by the program.

One of the interesting consequences of this is that arrays cannot have their dimensions changed during the running of the program, and therefore can hold only the type and only the number of entities indicated when they were declared. A Modula-2 `ARRAY [0..79] OF CHAR`, for instance, can never hold more than 80 characters.

NOTE: These observations may not be true in other languages, some of which do allow arrays to be redimensioned by the program on the fly.

Figure 12.2 illustrates a common method of allocating memory. A block is of memory set aside for the program's use, and within this, the code is placed first (at the lowest address) and this is followed by the static variable space.



12.4.2 Procedures and the Stack--Automatic Dynamics

The next step in understanding memory allocation is to observe that what has been said so far about static memory for program variable also applies to procedures in exactly the same way. Just as whenever a program is run static memory is set aside, so also whenever a procedure is entered, memory must be set aside for its parameters and variables. (The code for the procedure has memory within the block of memory for the code of the entire program.) When the procedure is running, the memory it employs for its variables can be regarded as static. Indeed, the relative positions of the memory for each variable with respect to the start of this memory block is all worked out by the compiler ahead of time so that assignments to such variables may be properly coded.

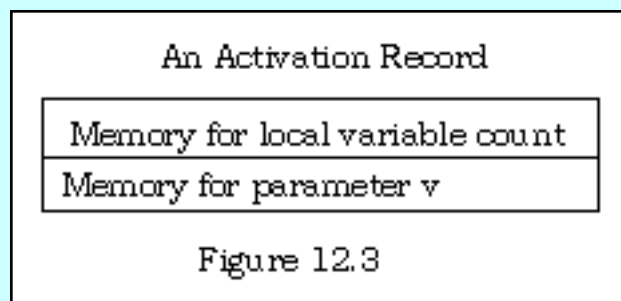
The memory assigned to a procedure for its parameters and variables when it is invoked is called an activation record for the procedure.

Example:

TYPE

```
Vector = ARRAY [1..2] OF REAL;  
PROCEDURE DisplayVector (v : Vector);  
VAR  
    count : CARDINAL;  
    (* etc *)  
END DisplayVector;
```

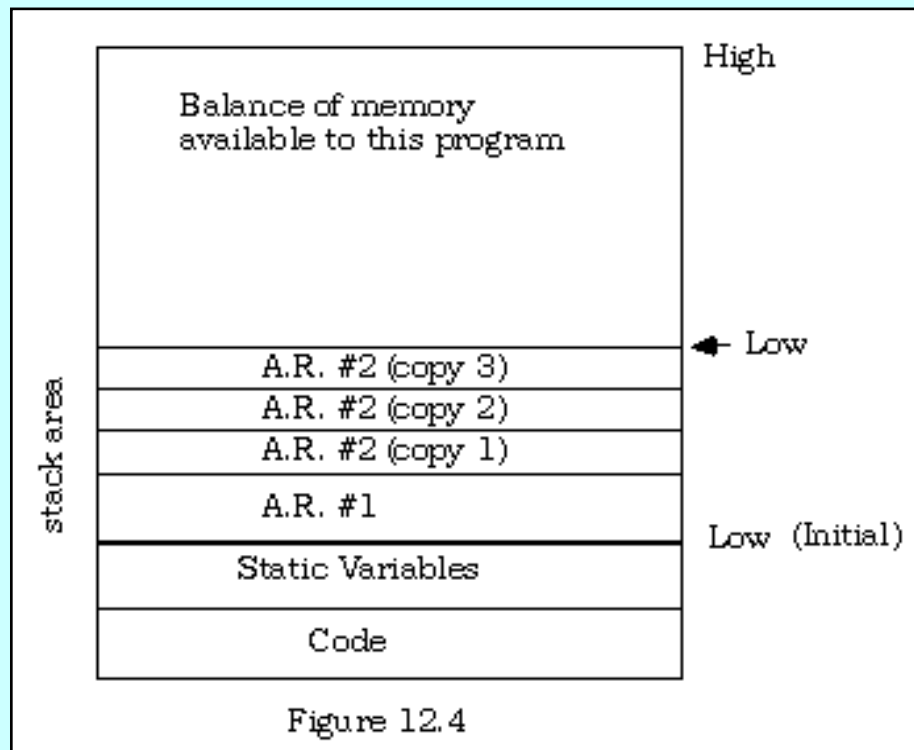
This procedure would have an activation record something like that shown in figure 12.3 below.



One could also think of the static variable memory for a program as its activation record in the sense that a program *is* a procedure. However, the program's activation record is present all the time the program is active, whereas the activation record for a procedure is only present when the procedure is active. If one procedure invokes another one (including itself) a second activation record is placed in memory. As each one is exited, the memory employed for its activation record is given back. Since such procedure invocations must be in a particular order, and no procedure can be exited before one that it has itself invoked, these activation records are simply stacked one on top of another above the one for the program. The area of memory available for the use of the program will shrink and grow, depending on the amount currently set aside for activation records, which in turn depends on how deep is the current chain of procedure calls.

Example:

Suppose the main program calls *procedure1*, which in turn calls *procedure2*, and this is recursive, calling itself two more times. At the most deeply nested level of procedure calls, the memory map looks like figure 12.4.



As each procedure is exited in the chain of calls, one activation record is released for later use, and the marker for the low end of the available memory is moved back down. If another procedure call is made, its activation record is placed on top of the stack of existing ones.

NOTE: The three copies of A.R.#2 are *structurally* the same. That is, they have the same memory allocation for variables of the same names. The *contents* of those memory locations depend, however, on the logic of the procedure as it runs, and may all be different.

It is this behaviour that makes recursion feasible in Modula-2 (and other languages that use a procedure activation record stack), for each recursive invocation of a procedure yields a new activation record that is specifically for the use of that entry to the procedure. Value parameters and locally defined variables are therefore different on each new invocation, and so the procedure may work on data without affecting the variables or data of the next outermost invocation of the same procedure. Of course, variable parameters are merely treated as aliases to a variable in an outer scope, so each invocation of a procedure with one of these will work on the same copy of the data. If the procedure is a function procedure, then

the stack is probably also used to pass back the return value to the expression from which it was invoked, though the details of this may vary from implementation to implementation.

It is important to realize that although this activity is dynamic (it takes place at run time and depends on the logic of the program, and not just on the declarations) it is automatic, and cannot be controlled by the programmer. Note also that this is only a *model* of the memory management; an actual machine might start the stack at the highest available memory location and grow it down--the opposite of what is shown here.

12.4.3 Dynamic Memory and the Heap--Program Controlled Dynamics

The description thus far is only an explanation of what has already been done; it offers nothing new. In particular, all the variable kinds employed thus far must have a predetermined or static memory allocation. If the programmer decides that the maximum number of students in a class for a marks program will be 100, the program simply will not allow entry number 101.

However, real life is different. Take airline reservations, for example. The number of active passenger records may well grow and shrink according to the time of day, the day of the week, season of the year, and whether the attendant is wearing green or the wind is blowing from the East. The point is that it is not always possible to predetermine the maximum number of items of a data type representing a reservation ahead of time.

It is natural therefore to ask whether it is possible to create some data types that do not have a fixed number of instances, nor therefore a predetermined memory allocation, but can have instances created and the memory allocated by the running program, so that the maximum number of entities is limited only by the available memory rather than by the program.

It is, and the main purpose of this chapter is to show how this is done in Modula-2. First, the preceding discussion is formalized in the following definition:

Dynamic variables are those that can have the space allocated to them as needed at some point during the execution of a program or procedure and that can also be disposed of and have their space given back to the system by the program.

The only place where such an activity can take place in the memory model under discussion here is in the stretch of memory from the top of the stack to the end of the region allocated to the program. Within this region, some languages (including Modula-2) permit manual allocation and deallocation of memory under program control.

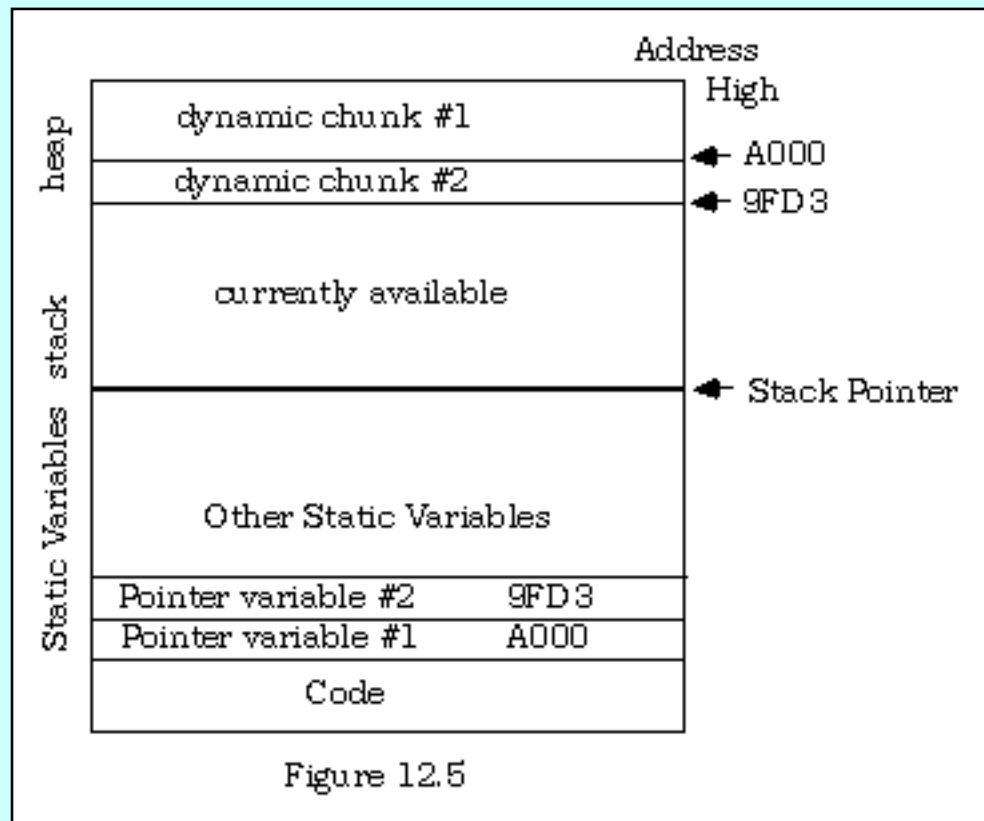
The region of memory above the stack and in which program-controlled dynamic allocation and deallocation of memory can take place is called the heap.

In order to employ the heap, a program needs to have:

- one or more static pointer variables to hold the locations of dynamic variables

- a method to obtain heap memory and to return it (allocate/deallocate)
- a method to assign the address of such memory to the static pointer variable
- a method to refer to the dynamic memory through the static variable

The means by which the first and last points are achieved have already been covered. Modula-2 pointer variables are used as the static variables to hold the addresses of the dynamic chunks of memory, and they are dereferenced to refer to the actual data. The details of memory allocation and deallocation in Modula-2 are the subject of the next section. However, at this point, the memory map in this model of memory management could look something like figure 12.4, in which it is assumed that the program has two static pointer variables to hold dynamic memory locations, and has already obtained the memory and assigned the addresses to those variables.



As a parting remark, it ought to be noted that the model of memory presented here is not necessarily used in this way by all systems. It may be that the only memory allocated permanently to a program's use is that for its code and static variables, and that beyond this, additional memory is assigned for activation records and/or dynamic uses by the operating system's memory manager from a common pool it organizes. In this more abstract model, a request for memory is dispatched by the program to the memory manager and if met, it in turn hands back the appropriate memory to the program. Indeed, it would be best to assume that nothing can be known by the programmer ahead of time, or determined by the program at run time, about the configuration of allocated memory at any given point. However, the model here is common, simple, and easy to understand, even if an actual system may be doing something a little different.

Contents

12.5 Managing Dynamic Memory in Modula-2

All that now remains to explore are the specific notational abstractions employed in Modula-2 to manage dynamic memory (de)allocation. One has to be able to obtain a stretch of memory for an item at run time, and to keep on hand a pointer to let the program know where it starts. One also needs to be able to give the memory back when it is no longer needed, so that a subsequent allocation request by another part of the program may lay claim to that territory for its own. That is, dynamic memory management involves handling directly some of the things that are done automatically for static types.

Dynamic memory allocation is performed using the standard Modula-2 procedure NEW.

Suppose one has:

TYPE

```
CardPoint = POINTER TO CARDINAL;  
RecPoint  = POINTER TO MyRecType;
```

VAR

```
cPoint : CardPoint;  
rPoint : RecPoint;  
theRec : MyRecType;
```

The code

```
NEW (cPoint);
```

will obtain a stretch of dynamic (heap) memory sufficient to store an item the size of a `CARDINAL`. Simultaneously, the address of that chunk of memory will be assigned to `cPoint`, so that future use of `cPoint^` will give access to the new piece of memory. Likewise, the code

```
NEW (rPoint);
```

will obtain `SIZE(MyRecType)` number of LOCs of dynamic memory to store a an item of type `MyRecType`. Simultaneously, the address of that chunk of memory will be assigned to `rPoint`, so that future use of `rPoint^` will give access to the new piece of memory. Once this is done, it is legitimate to use such code as:

```
cPoint^ := 34;  
rPoint^ := theRec;
```

Naturally, there is no point in making these latter (normal) type of assignments until after a valid pointer has been initialized via the procedure `NEW`, for `NEW` does more than just give `cPoint` and `rPoint` their values; it simultaneously obtains and sets aside enough memory for one item of the type of data to which the respective pointer points.

It was noted earlier that this type of data is called *dynamic* because the number of items of such types can be expected to grow and shrink as the program runs. This implies the existence of a procedure to reclaim for other use the chunk of memory pointed to by a pointer variable.

Dynamic memory deallocation is performed using the standard Modula-2 procedure DISPOSE.

Continuing from above, if at some point in the program one were to execute the code

```
DISPOSE (rPoint);
```

then SIZE (MyRecType) number of LOCs of dynamic memory at the address *rPoint* and referred to as *rPoint*[^] would be returned to the system for later use by some other client of dynamic memory.

There are several rather important points to note in connection with the use of the two procedures NEW and DISPOSE.

First, these are relatively low-level routines, and there are few safeguards against their misuse. If an assignment has been done so that two pointers *point* and *pointer2* of the same type both happen to point to the same section of memory (and thus to the same entity) and one writes, say,

```
DISPOSE (point),
```

the memory at *point* is reclaimed by the system and could be used for something else (because it can be reassigned at any time). However, at this juncture, *pointer2* still points to that deallocated memory. Logically, *pointer2*[^] must be regarded as undefined, even though *pointer2* apparently still has a legitimate value. Indeed, in older versions of Modula-2 *point* also still points to the deallocated memory, because in classical versions of Modula-2 the procedure DISPOSE (p) does not necessarily change the value of p (Wirth did not say what happens to p). However, the rule in ISO Modula-2 is that

The ISO Modula-2 procedure DISPOSE (p) not only deallocates the memory pointed to by p, it also assigns NIL as the value of p.

Thus, although references to the deallocated memory are not possible via *point*[^], they are through *pointer*[^] even though some other data may now be occupying that memory. (One says that *pointer2*[^] has been left *dangling*.) The compiler cannot check these logical aspects of the program, so such errors will only be revealed when the program crashes. Experienced programmers are well aware of the fact that some of the most difficult-to-find errors are generated from logically invalid pointer references. Thus, programs that employ pointers can be extremely difficult to debug and maintain, and programmers who use pointers must do so in an extremely careful and disciplined manner.

If the student has taken to heart the lessons about planning and program structure presented thus far in this text, much progress will already have been made toward the goal of writing correct programs. If however, the student has developed a careless and unprofessional approach to the detail work needed to write good programs, it is at this point that she will part company with the professional and remain an amateur, for her programs will be less and less likely to work as she employs more pointers in such a manner.

At the risk of belabouring the matter, it is also worth noting that languages whose structure promote or requires the routine and indiscriminate use of pointers when they are not really necessary are inherently poor tools for writing properly engineered and maintainable software. This needs to be remembered when the student begins to use commercial tools, for they will not necessarily be crafted to the professional standards being upheld here. Moreover, the marketplace has a way of insisting upon the use of tools (hardware as well as software) because they are popular, even in the face of the fact that they are inferior. Some professional programmers even find it efficient to design and write code first in Modula-2 and then translate or re-write it in some other notation that happens to be demanded in the workplace in order to minimize errors and reduce debugging time.

For such people, Modula-2 becomes their last pseudocode before producing the flavour of code required of the installation.

Second, the astute reader may have observed the fact that NEW and DISPOSE are low level (machine specific) and been expecting to be told that they are in a separate module and must be imported in order to improve portability. Yet both are standard identifiers in the language proper--isn't that somewhat of a contradiction?

This is indeed an inconsistency, and has a unique explanation (for Modula-2). Calls to NEW and DISPOSE are not executed directly, but are translated by the compiler into calls upon ALLOCATE and DEALLOCATE (respectively). The definitions are:

```
PROCEDURE ALLOCATE (VAR addr: SYSTEM.ADDRESS; amount: CARDINAL);
```

```
  (* Allocates storage for a variable of size amount and assigns the address of this
```

variable to `addr`. If there is insufficient unallocated storage to do this, the value `NIL` is assigned to `addr`. *)

```
PROCEDURE DEALLOCATE (VAR addr: SYSTEM.ADDRESS; amount: CARDINAL);  
  (* Deallocates amount locations allocated by ALLOCATE for the storage of the  
  variable addressed by addr and assigns the value NIL to addr. *)
```

That is

```
NEW (p);  
DISPOSE (p);
```

automatically are translated into

```
ALLOCATE (p, SIZE (type of p^));  
DEALLOCATE (p, SIZE (type of p^));
```

If the programmer has an intimate knowledge of the way in which memory is handled on a particular machine and is capable of writing a manager to handle the details, it is possible to write a separate module that exports two procedures called `ALLOCATE` and `DEALLOCATE`, and then import them into programs that employ `NEW` and `DISPOSE`. It is also possible to include procedures called `ALLOCATE` and `DEALLOCATE` in the scope of the program (either by defining them there or by import) that do something entirely different, or that do nothing at all. In such cases, calls to `NEW` and `DISPOSE` would compile correctly but would not actually perform as they should, and this would presumably make the programs incorrect, or at least misleading.

For most mortals, both the ISO standard and classical Modula-2 require implementors to supply a predefined module *Storage* from which `ALLOCATE` and `DEALLOCATE` are exported and can be imported into programs. Thus, any program module that makes use of dynamic data will normally have to include the line:

```
FROM Storage IMPORT  
  ALLOCATE, DEALLOCATE;
```

In this way one preserves the portability of the Modula-2 code, while at the same time being able to perform the necessary low level functions by employing a library to encapsulate code that must be platform-specific.

WARNING: For one edition of *Programming in Modula-2* Wirth apparently decided that the automatic translation to `ALLOCATE` and `DEALLOCATE` was too sly, or possibly too magical, so he removed `NEW` and `DISPOSE` from the language altogether, requiring programmers to both import and write code in terms of `ALLOCATE` and `DEALLOCATE`. Although most implementations (and the ISO committee) declined to follow this new rule, some compiler writers did, so there are a few versions that lack `NEW` and `DISPOSE` entirely.

Another possible difference between classical and ISO versions relates to the handling of situations where there is no memory left to allocate. Some classical versions of the module *Storage* had something resembling

```
PROCEDURE Available (memoryNeeded : CARDINAL) : BOOLEAN;
```

and the cautious user of dynamic memory could call this enquiry procedure before attempting to use `Storage.ALLOCATE` (whether directly or via `NEW`). Other versions (including the ISO) do not have such an enquiry procedure. Instead, they merely specify (as noted above in the procedure definitions) that an attempt to use `Storage.ALLOCATE` that fails results in the return of the pointer `NIL`. If `NEW` was used, this value is returned in its parameter as well. It is then up to the program to check the return value to ensure that it is proceeding under the assumption that memory has been obtained.

12.6 An Example--Dynamic Records and Files

Dynamic memory is useful when working with records initially stored in a file because the number of records in the file is not known by the program ahead of time. Therefore, it may be worthwhile to allocate the memory for the items in the file as they are read, instead of declaring ahead of time an array of records large enough to hold all the data. The example below is a simple illustration of this idea. As in the sorting example of a previous section, the dynamic data is tracked by a collection of pointers to the data items held in an array and each item is sorted to its correct place in the list by manipulating the pointers in the array, not by reassigning any of the data elements themselves. As in that example, this is more efficient than keeping an array of the entire records, because memory reshuffling is limited to pointers, which take far fewer storage locations and therefore less time to swap or otherwise reassign. Here, a small record consisting of people's names, initial, and age is kept in a file on the disk. For simplicity, it is supposed that the maximum number of these records is 101. The purpose this simple module serves is to read the items into memory, simultaneously sorting their pointers by the name in the record, and then printing out the entire list. The module also illustrates some elementary interactions between files and dynamic records, though this is by far the last word on the subject. First a very simple module was written to create a data file. It has no dynamic data or pointers of any kind.

```
MODULE MakeRecords;
```

```
(* very crude data entry  
to produce a file for the program GetPrintRecords  
by R. Sutcliffe -- modified 1995 05 12 *)
```

```
FROM STextIO IMPORT
```

```
    WriteString, ReadString, ReadChar, WriteLn, SkipLine;
```

```
FROM SWholeIO IMPORT
```

```
    ReadCard, WriteCard;
```

```
FROM RndFile IMPORT
```

```
    ChanId, OpenClean, OpenResults, Close, raw;
```

```
FROM RawIO IMPORT
```

```
    Write;
```

```
FROM SIOResult IMPORT
```

```
    ReadResult, ReadResults;
```

```
CONST
```

```
    max = 100;
```

```
TYPE
```

```
ShortAry = ARRAY [0..15] OF CHAR;
```

```
RecordData =
```

```
  RECORD
```

```
    last, first: ShortAry;
```

```
    initial: CHAR;
```

```
    age: CARDINAL;
```

```
  END;
```

```
VAR
```

```
  person: RecordData;
```

```
  numActive : CARDINAL;
```

```
  outfile: ChanId;
```

```
  res : OpenResults;
```

```
BEGIN    (* main program *)
```

```
  OpenClean (outfile, "data", raw, res);
```

```
  IF res = opened
```

```
    THEN
```

```
      numActive := 0;
```

```
      LOOP
```

```
        WriteString ("reading record number");
```

```
        WriteCard (numActive +1, 5);
```

```
        WriteLn;
```

```
        WriteString ("last name?");
```

```
        ReadString (person.last);
```

```
        IF ReadResult () # allRight
```

```
          THEN
```

```
            EXIT
```

```
          END;    (* if *)
```

```
        SkipLine; WriteLn;
```

```
        WriteString ("first name?");
```

```
        ReadString (person.first);
```

```
        IF ReadResult () # allRight
```

```
          THEN
```

```
            EXIT
```

```
          END;    (* if *)
```

```
        SkipLine; WriteLn;
```

```
        WriteString ("initial?");
```

```
        ReadChar (person.initial);
```

```
        IF ReadResult () # allRight
```

```
          THEN
```

```
            EXIT
```

```
          END;    (* if *)
```

```

SkipLine; WriteLn;
WriteString ("age?");
ReadCard (person.age);
IF ReadResult () # allRight
    THEN
        EXIT
    END;    (* if *)
SkipLine;
Write (outfile, person);
INC (numActive);
END; (* loop *)
Close (outfile);
ELSE
    WriteString ("could not open file");
END (* initial if res *)
END MakeRecords.

```

Next, the main module was produced to read, sort and print this data. To keep things simple so that the focus can be on the dynamic nature of the memory allocation error checking is primitive or non-existent. The sorting routine should be considered carefully, however. This particular one is called an insert sort, and will be discussed in detail in chapter 13, but something like it will be used in other places in the chapter. It works by examining each item in a sorted list and then inserting the new one in its appropriate place.

```

MODULE GetPrintRecords;

(* By R. Sutcliffe
to illustrate dynamic records
modified 1995 05 12 *)

FROM STextIO IMPORT
    WriteChar, WriteString, WriteLn;
FROM SWholeIO IMPORT
    WriteCard;
FROM RndFile IMPORT
    ChanId, OpenOld, OpenResults, Close, raw;
FROM RawIO IMPORT
    Read;
FROM IOResult IMPORT
    ReadResult, ReadResults;
FROM Storage IMPORT
    ALLOCATE;
FROM Strings IMPORT

```

Compare, CompareResults;

CONST

max = 100; (* maximum number of records *)

TYPE

ShortAry = **ARRAY** [0..15] **OF** **CHAR**;

RecordData =

RECORD

last, first: ShortAry;

initial: **CHAR**;

age: **CARDINAL**;

END;

DataPoint = **POINTER TO** RecordData;

ListType = **ARRAY** [1..max] **OF** DataPoint; (* pointers to records *)

VAR

person: RecordData; (* one item for I/O *)

list: ListType;

count, numActive : **CARDINAL**;

infile: ChanId;

res : OpenResults;

dataArrayFull : **BOOLEAN**;

PROCEDURE InitPointers; (* Set all pointers in the list to **NIL** *)

VAR

count: **CARDINAL**;

BEGIN

FOR count := 1 **TO** max

DO

list [count] := **NIL**

END

END InitPointers;

PROCEDURE SortOneIn (name: RecordData);

VAR

moveCount, count: **CARDINAL**;

BEGIN

IF numActive = 0 (* perhaps this is the first one *)

THEN

INC (numActive); (* get it all started *)

NEW (list [1]); (* get memory *)

list [1]^ := name; (* and occupy it *)

ELSIF numActive = max **THEN** (* no room *)

dataArrayFull := **TRUE**;


```

    RETURN;
ELSE (* add this one to the array *)
    count := 0;
    REPEAT
        INC (count);
    UNTIL (count "data", raw, res);
IF res = opened
    THEN
        numActive := 0;
        LOOP
            Read (infile, person);
            IF ReadResult (infile) # allRight
                THEN
                    EXIT
                END; (* if *)
            SortOneIn (person);
            IF dataArrayFull
                THEN
                    WriteString ("Could not read entire file");
                    WriteLn;
                    EXIT;
                END; (* if *)
        END; (* loop *)
        Close (infile);
(* now ready to print it out *)
        FOR count := 1 TO numActive
            DO
                WITH list [count]^
                    DO
                        WriteString (first);
                        WriteString (' ');
                        WriteChar (initial);
                        WriteString ('. ');
                        WriteString (last);
                        WriteString (' is ');
                        WriteCard (age, 3);
                        WriteString (' years old. ');
                        WriteLn;
                    END (* with *)
            END; (* for *)
        END (* if res *)
END GetPrintRecords.

```

Examples of this general nature will be revisited and improved on substantially from time to time in later sections, but for now, the reader will be invited to incorporate at least some cosmetic improvements in the exercises. One set of output from this latter module, after entering some data for a file with the first one, was:

```
jim t. babchuk is  56 years old.  
ernest y. borgnine is  89 years old.  
catherine m. davis is  15 years old.  
dawn t. moore is  47 years old.  
garth r. moorehead is  34 years old.  
tom r. rowan is  26 years old.  
william r. shakespeare is 390 years old.  
joel r. sutcliffe is  15 years old.  
nathan p. sutcliffe is  15 years old.
```

[Contents](#)

12.7 Towards A Dynamic Array ADT

In the discussion of static data types in [section 12.4.1](#) it was observed that because Modula-2 arrays are static, they cannot be redimensioned at run time. In this section, the first steps are taken toward finding a way around that limitation via a user-defined dynamic array type.

Here, a one-dimensional array type is defined as a record that holds the number of items (the length of a vector) and the address of the first item. The address of subsequent items is calculated using an offset from that first address worked out from the size and position number of the element in the array.

```
adr := ADDADR (vector.first, ( pos - 1 ) * sizeofElement);
```

In this example, the array holds real values and *sizeofElement* is just SIZE (REAL). Since every creation of such a one-dimensional vector could use a different length, the location must be of type ADDRESS rather than a specific pointer type. This in turn means that ALLOCATE and DEALLOCATE are employed directly in creating instances of the dynamic array, rather than indirectly through NEW and DISPOSE.

```
ALLOCATE (vector.first, sizeofElement * numElements);
```

This module contains only the beginning steps toward creating a proper dynamic array ADT. Operations on the type still need to be defined, error handling added, and the whole code encapsulated in a library module and tested. It could also be made more versatile (generic) by extending the record to a third field, containing the size of the elements in the array, and adding a parameter to the *Create* procedure specifying this value. Such a module could then be used to define dynamic arrays of any type or length. Two-dimensional versions could also be constructed. Here is the code:

```
MODULE DynVectorDemo;
```

```
(* This Module implements a one-dimensional dynamic array of real  
   It is a demonstration of dynamic memory use  
   by R. Sutcliffe  
   modified 1995 05 14 *)
```

```
FROM SYSTEM IMPORT
```

```
  ADDRESS, ADDADR;
```

```
FROM Storage IMPORT
```

```
  ALLOCATE, DEALLOCATE;
```

```
FROM STextIO IMPORT
```

```
  WriteString, WriteChar, WriteLn;
```

```

FROM SRealIO IMPORT
    WriteFixed;

CONST
    sizeofElement = SIZE (REAL);

TYPE
    DynVector =
        RECORD
            size   : CARDINAL;           (* the size of the vector *)
            first  : ADDRESS;           (* the address of the first element *)
        END;

VAR
    vector : DynVector;
    re     : REAL;  (* need one of the items *)
    adr    : POINTER TO REAL;  (* and a pointer to such *)
    count  : CARDINAL;
    numElements : CARDINAL; (* for the demo vector *)

PROCEDURE WriteVector;
(* write out all the items in the vector *)
VAR
    count : CARDINAL;
BEGIN
    FOR count := 1 TO numElements
    DO
        WriteFixed (Fetch (count), 2, 8 );
        WriteChar (" , ");
    END;
    WriteLn;
END WriteVector;

PROCEDURE Create (numElements : CARDINAL );
(* create a new dynamic array with 'length' elements *)
BEGIN
    vector.size := numElements;
    ALLOCATE (vector.first, sizeofElement * numElements);
    (* get enough memory for this many items *)
END Create;

PROCEDURE Destroy (numElements : CARDINAL );
(* create a new dynamic array with 'length' elements *)
BEGIN

```

```

vector.size := 0;
DEALLOCATE (vector.first, sizeofElement * numElements);
    (* give back the memory *)
END Destroy;

PROCEDURE Update (pos : CARDINAL; newValue : REAL );
    (* Pre : pos <= vector.size
       Post : if so, item #pos is updated with newValue
              if not, nothing happens, no error *)
BEGIN
    IF (pos <= vector.size)
        THEN    (* compute address of this index *)
            adr := ADDADR (vector.first, ( pos - 1 ) * sizeofElement);
            adr^ := newValue;  (* and put the value there *)
        END; (* if *)
END Update;

PROCEDURE Fetch (pos :CARDINAL) : REAL;
    (* Pre : pos <= vector.size
       Post : if so, item #pos is returned
              if not, returns maximum real but no error *)
VAR
    temp : REAL;
BEGIN
    IF (pos <= vector.size)
        THEN    (* compute address of this index *)
            adr := ADDADR (vector.first, ( pos - 1 ) * sizeofElement);
            temp := adr^; (* get the value there *)
        ELSE
            temp := MAX (REAL);
        END;
    RETURN temp;
END Fetch;

BEGIN (* main part of demo *)
    numElements := 7;
    Create (numElements);

    (* load any old values into the vector *)
    re := 45.0;
    FOR count := 1 TO numElements
        DO
            Update (count, re + FLOAT (count));
        END;

```

```

WriteVector;    (* now read them back to see if all is ok *)
Destroy (numElements);  (* wipe it all out *)

numElements := 5; (* and start over *)
Create (numElements);
Update (1, 4.5);  (* leave rest uninitialized *)
WriteVector;    (* and see what happens *)
Destroy (numElements);  (* erase again before quitting *)

```

END DynVectorDemo.

Observe that in the testing harness (the main module body here) memory was allocated to a vector of length seven, given back to the system, and then a request for memory for a smaller vector was issued. While one might expect that that a portion of the original allocation might be re-used and some of the values survive from the original use, here is the actual output when this module was run:

```

** Run log starts here **
  46.00,   47.00,   48.00,   49.00,   50.00,   51.00,   52.00,
   4.50,    0.00,    0.00,1049841000000.00,26678050000000000000.00,

```

If any of the first piece of memory was indeed used the second time, the starting point of the real numbers was evidently different. In any case, this illustrates that once memory has been deallocated and returned to the system, no assumptions whatever can be made about it afterwards. Even if the second request is for the same amount of memory, the programmer cannot know that the same piece of memory just given back will be reclaimed. No assumptions whatever can be made about the memory manager's handling of requests for dynamic memory.

[Contents](#)

12.8 Pointers and Return Types

On occasion, it might be more practical to have a function return a pointer to a RECORD or an ARRAY, rather than the data item itself. Indeed, early versions of Modula-2 did not allow a function procedure to return a structured type, and a work-around returning a pointer was often necessary. More commonly, a library module might implement an ADT as a pointer, and its function procedures that return this type will be returning pointers. (As shall be seen in succeeding sections, this is in fact the standard way of working). To illustrate some of the points relating to such methods, the following module provides a partial implementation of a dynamic (but fixed length) string type, together with a function procedure that reads a string and returns a pointer to the string. There is a brief testing harness at the end.

NOTE: The choice of *ReadString* to do the assignment is for the sake of illustration only. The analysis and comments apply to *any* procedure that is to assign something to dynamic memory and communicate back to a caller.

```
MODULE DynStringDemo;

(* A crude demonstration of pointer return styles
   by R. Sutcliffe  1995 05 17 *)

FROM STextIO IMPORT
    WriteString, WriteLn, SkipLine, ReadString;
FROM SIOResult IMPORT
    ReadResult, ReadResults;
FROM Storage IMPORT
    ALLOCATE;
CONST
    maxChars = 80;
TYPE
    String = ARRAY [0..maxChars-1] OF CHAR;
    StrPoint = POINTER TO String;

VAR
    sPoint : StrPoint;

PROCEDURE ReadDynStr1 () : StrPoint;
VAR
    answer : StrPoint;
BEGIN
    NEW (answer);
    ReadString (answer^);
```

```

SkipLine;
RETURN answer
END ReadDynStr1;

BEGIN
  WriteString ("type string1 here=");
  sPoint := ReadDynStr1 ();
  WriteLn;
  WriteString (sPoint^);
  WriteLn;

END DynStringDemo.

```

In place of the first two lines of the main block, one might be tempted to reason that since the procedure *ReadDynStr1* returns a pointer to an item that is the correct type for the parameter to *ReadString* one could write:

```
WriteString (ReadDynStr1 ()^);
```

However, this is not the case in Modula-2. The rule is:

Modula-2 function procedures cannot be selected. It is illegal to write:

Proc (params)^

Proc (params) [index]

Proc (params).fieldname

This style of working may be adequate for some purposes; however there may also be some difficulties. Repeated calls to a procedure that must create a new dynamic string in order to have something to return might litter up the memory quite unnecessarily. Moreover, the creation of a new dynamic entity has been combined with the act of reading one, and this seems rather unstylish. One possible modification would be to separate the creation of the dynamic entity from the reading of one by having two procedures.

```

PROCEDURE MakeDynStr () : StrPoint;
VAR
  answer : StrPoint;
BEGIN
  NEW (answer);
  RETURN answer;
END MakeDynStr;

PROCEDURE ReadDynStr2 () : StrPoint;
VAR
  answer : StrPoint;

```


BEGIN

```
  ReadString (answer^);  
  SkipLine;  
  RETURN answer  
END ReadDynStr2;
```

One would call them as follows:

```
WriteString ("type string2 here=");  
sPoint := MakeDynStr ();  
sPoint := ReadDynStr2 ();  
WriteLn;  
WriteString ( sPoint^);  
WriteLn;
```

However, this style too has its disadvantages, for it requires the use of a variable global to both procedures, a solution that is sure to be unsatisfactory in a library module. Once such a global variable had its contents passed back to a program, the next time the procedure was called it would alter that global variable, and the calling program could be thereby invalidated, for whatever one of its pointer variables had been pointed at the global item would now also be altered.

Perhaps a more satisfactory style to use when it is necessary to return a pointer to a calling procedure is the variable parameter:

```
PROCEDURE ReadDynStr3 (VAR strPoint: StrPoint);
```

BEGIN

```
  ReadString (strPoint^);  
  SkipLine;  
END ReadDynStr3;
```

Here, it is the responsibility of the caller to provide the (static) memory for the pointer variable, and to allocate memory to which it can point. The function procedure accesses this via a variable parameter and assigns directly into the supplied memory; it does not obtain any more of its own. This one would be called in the more conventional style shown below (where the call to NEW with *sPoint* has already been made):

```
WriteString ("type string3 here=");  
ReadDynStr3 (sPoint);  
WriteLn;  
WriteString (sPoint^);  
WriteLn;
```

It may be easier to declare a local variable in the procedure to hold the data temporarily and then pass this to the space pointed to by the variable parameter; this is very little different from the third variation above and is called the same way.

```
PROCEDURE ReadDynStr4 (VAR strPoint: StrPoint);  
VAR  
    result : String;  
  
BEGIN  
    ReadString (result);  
    SkipLine;  
    strPoint^ := result;  
END ReadDynStr4;
```

[Contents](#)

12.9 Opaque Types Revealed

There was a brief introduction to the concept of opaque types in [section 10.7.3](#), and the reader should review that comment before continuing.

Here, the goal will be to implement an ADT in an opaque way, that is, with the structural details of the type deliberately hidden away in the implementation module so that clients cannot know what they look like.

Here, this will be achieved by revisiting an earlier example. In [section 6.9](#), in the introduction to library modules, the ADT *Point* was implemented in a transparent manner, that is, with the implementation details available for all to see in the definition module.

TYPE

```
Point = ARRAY [1 .. 2] OF REAL;
```

Client programs "know" the names of the parts of the type *Point*. The way that they are written may depend heavily on this fact. It is a better idea in practice to keep such details hidden, so that the client program does not have access to such information, but rather require that such data items can be operated upon only by the procedures that are also contained in the module along with the data type definition. This has been referred to, and there have been several instances of predefined library modules containing such opaque types throughout the text.

Object oriented design concentrates on the need for a type of data. The data type and all the operations on it are implemented in such a way that an item of that type can be modified only through those operations, and not by any client program.

There is somewhat more to object oriented design than this in practice, especially if one is working in a notation built from the ground up on such a philosophy. However, this is a good working start for such thinking.

Here is the revised definition of the ADT *Point* with some procedures changed from functions so that they can return *Point* in a variable parameter.

DEFINITION MODULE OpaquePoints;

```
(* based on an original design by R. Sutcliffe  
   modified to opaque form by Gord Tisher  
   last modification 1995 05 17 *)
```

```
(* angles are measured in radians counterclockwise from the positive x-axis *)
```

TYPE

```
Point;
```

```
PROCEDURE newPoint (x, y : REAL; VAR p : Point);  
  (* allocates memory for a new point *)
```

```
PROCEDURE killPoint (VAR p : Point);  
  (* deallocates memory the point variable -- using the variable after a call to  
  killPoint is wrong. *)
```

```
PROCEDURE assign (x, y : REAL; VAR p : Point);  
  (* returns the abstract point with coordinates x and y *)
```

```

PROCEDURE abscissa (p : Point) : REAL;
    (* returns the first, or x-coordinate of the point *)

PROCEDURE ordinate (p : Point) : REAL;
    (* returns the second, or y-coordinate of the point *)

PROCEDURE abs (p : Point) : REAL;
    (* returns the distance from the point to the origin *)

PROCEDURE arg (p : Point) : REAL;
    (* returns the angle to the positive x-axis subtended by a line segment from the
    origin to the point measured in the range 0 to 2 $\pi$  radians *)

PROCEDURE polarToRect (abs, arg : REAL; VAR p : Point);
    (* returns the point with the given absolute value and argument *)

PROCEDURE reflectX (VAR p : Point);
    (* returns the reflection of the point in the x-axis *)

PROCEDURE reflectY (VAR p: Point );
    (* returns the reflection of the point in the y-axis *)

PROCEDURE reflect0 (VAR p : Point);
    (* returns the reflection of the point in the origin *)

PROCEDURE reflect45 (VAR p : Point);
    (* returns the reflection of the point in the line y = x *)

PROCEDURE scale (VAR p : Point; scaleFactor : REAL);
    (* returns the point with the same argument as p and its absolute value multiplied
    by the scale factor *)

PROCEDURE rotate (VAR p : Point; rotAngle : REAL);
    (* returns the point with the same absolute value as p and with its argument
    increased by rotAngle *)

PROCEDURE translate (VAR p : Point; deltaX, deltaY : REAL);
    (* returns the point obtained by shifting the given point deltaX horizontally and
    deltaY vertically *)

END OpaquePoints.

```

It is time to clear up a little mystery regarding opaque types that may be troubling the thinking reader. Such types are named in the definition module, but their structure is not given there. How then can the compiler know how much memory to assign statically to such a variable? After all, the earliest that the details can become available is the time when the implementation module is linked to the client program.

In defining Modula-2, Wirth replied that the only possible course of action for the compiler in this case is to assume that opaque types are always a certain fixed size, never more nor less. He concluded that this in turn meant that the opaque type itself would in practice have to be a pointer, and that the data represented by the abstraction would have to be contained in the structure to which it points. Wirth left open the possibility that someone might find some other way to achieve this that would allow opaque types to be other than pointers, and some implementations took advantage of this to the extent that they allowed opaque types to be any type that occupied the same amount of memory as a pointer type. That is, this limitation was a practical one, but not a theoretical one. Although the restriction of opaque types to pointers is clearly an artifact of current technology, in the sense that the limitation results from the standard view of what must happen at compile-link-run time, the ISO standards

committee took a much narrower view of the matter:

In an ISO standard conforming implementation, any opaque type defined in a definition module must be redefined in the corresponding implementation module either as another opaque type or as a pointer type.

Any redefinition in terms of another opaque type merely postpones the inevitable--an opaque type is, in the (ISO) end a pointer type. This can be clearly seen in the implementation of the *OpaquePoints* module below. The type `Point` is a pointer to the data structure that contains the coordinates of the point, and all access to the actual data must employ the dereferencing operator. Observe that the student who implemented the design chose to use `ALLOCATE` directly rather than `NEW`. Note, however, that `ALLOCATE` and `DEALLOCATE` appear only once in the entire module, confining their presence to as small a scope as possible, and ensuring good control over memory use.

```
IMPLEMENTATION MODULE OpaquePoints;

(* based on an original design by R. Sutcliffe
   modified to opaque form by Gord Tischer
   last modification 1995 05 18 *)

FROM Storage IMPORT
    ALLOCATE, DEALLOCATE;

FROM RealMath IMPORT
    sqrt, arctan, sin, cos, pi;

TYPE
    Point = POINTER TO PointData;
    PointData = ARRAY[1..2] OF REAL;

PROCEDURE newPoint (x, y : REAL; VAR p : Point);
BEGIN
    ALLOCATE (p, SIZE (PointData));
    p^[1] := x;
    p^[2] := y;
END newPoint;

PROCEDURE killPoint (VAR p : Point);
    (* deallocates memory the point variable -- using the variable after a call to
    killPoint is wrong *)
BEGIN
    DEALLOCATE (p, SIZE (PointData));
END killPoint;

PROCEDURE assign (x, y : REAL; VAR p : Point);

BEGIN
    p^[1] := x;
    p^[2] := y;
END assign;

PROCEDURE abscissa (p : Point) : REAL;

BEGIN
    RETURN p^[1];
```

```

END abscissa;

PROCEDURE ordinate (p : Point) : REAL;

BEGIN
    RETURN p^[2];
END ordinate;

PROCEDURE abs (p : Point ) : REAL;

BEGIN
    RETURN sqrt (p^[1] * p^[1] +  p^[2] * p^[2]);
END abs;

PROCEDURE arg (p : Point) : REAL;
(* if both coordinates are zero, the angle is not defined, but this procedure will
   return zero.  No errors are generated. *)

VAR
    temp : REAL;

BEGIN
    IF p^[1] = 0.0
    THEN
        IF p^[2] < 0.0 THEN
            RETURN 1.5* pi (* case of point on negative y-axis *)
        ELSE
            RETURN 0.0;
        END;
    END;
    temp:=  arctan (p^[2]/p^[1]); (* returns first and fourth quadrants only *)
    IF p^[1] >= 0.0
    THEN
        RETURN temp;
    ELSE
        RETURN (2.0 * pi) + temp;
    END; (* if *)
    ELSE
        RETURN pi + temp (* adjust for second and third quadrants *)
    END;
END arg;

PROCEDURE polarToRect (abs, arg : REAL; VAR p : Point);

VAR
    temp : PointData;
BEGIN
    temp[1] := abs * (cos (arg));
    temp[2] := abs * (sin (arg));
    p^ := temp;
END polarToRect;

PROCEDURE reflectX (VAR p : Point );

```

```

BEGIN
    p^[2] := -p^[2];
END reflectX;

PROCEDURE reflectY (VAR p : Point );

BEGIN
    p^[1] := -p^[1];
END reflectY;

PROCEDURE reflect0 (VAR p : Point );

BEGIN
    p^[1] := -p^[1];
    p^[2] := -p^[2];
END reflect0;

PROCEDURE reflect45 (VAR p : Point );
VAR
    temp : REAL;
BEGIN
    temp := p^[1];
    p^[1] := p^[2];
    p^[2] := temp;
END reflect45;

PROCEDURE scale (VAR p : Point; scaleFactor : REAL);

BEGIN
    p^[1] := scaleFactor * p^[1];
    p^[2] := scaleFactor * p^[2];
END scale;

PROCEDURE rotate (VAR p : Point; rotAngle : REAL);
VAR
    r, theta : REAL;
BEGIN
    r := abs (p);
    theta := arg (p) + rotAngle;
    polarToRect (r, theta, p);
END rotate;

PROCEDURE translate (VAR p : Point; deltaX, deltaY : REAL);

BEGIN
    p^[1] := p^[1] + deltaX;
    p^[2] := p^[2] + deltaY;
END translate;

END OpaquePoints.

```

Observe that if the implementor had later decided to change the type *PointData* and implement it as a record instead, only this implementation module, and not any client program, would have to be re-compiled. Client programs would only have to be re-linked to take advantage of the new code. They need no re-coding themselves to do this, since their code does not depend on

the structural details of the opaque type. Indeed, it cannot, for those details are deliberately concealed from the client programs.

[Contents](#)

12.10 Pointers and Lists

All that has been said so far is still preliminary to the main event, for the data structures implemented so far, though dynamic, have not been of the kind whose number of entities could grow and shrink with the needs of the program at run time. In each instance to date in this chapter, there were static variables related to each dynamic instance. It is useful, but does not reach the goals set for this chapter, to declare large numbers of pointer variables (or opaque ones). In order to have truly dynamic collections, there must be a way to free the program from any presuppositions about the number of dynamic entities. Declaring a 500-item static array of pointers does not respond adequately to this challenge.

12.10.1 Declaring The Linked List Apparatus

The perceptive student will respond that the solution is to make new pointer variables as the program goes along (dynamically). In order for this to happen, the pointer type must itself be a field in the record holding the data. Then, every time memory is allocated for a new data record, so is a new pointer variable. Consider the declarations:

```
TYPE
  ItemPointerType = POINTER TO ItemData;
  ItemData =
    RECORD
      data : INTEGER;
      (* more data fields here *)
      toPoint : ItemPointerType;  (* Pointer field *)
    END;
VAR
  itemPoint : ItemPointerType;
```

A call to obtain dynamic memory via

```
NEW (itemPoint);
```

will obtain enough memory not only for the requisite data, but also for another pointer, via the additional field at the end of the data record. The next time a dynamic item is needed, a program variable is not necessary, for one can call

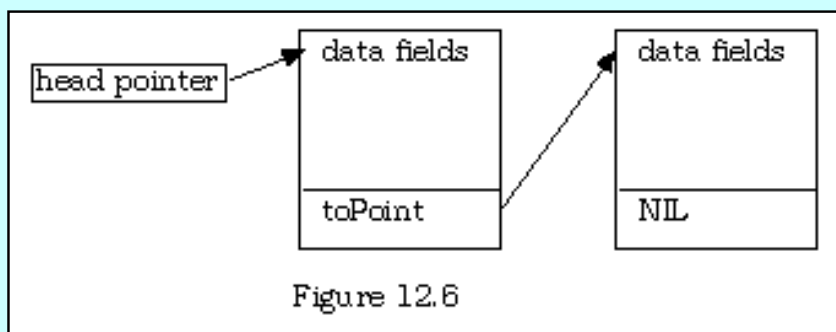
```
NEW (itemPoint^.toPoint);
```

and the time after that

```
NEW (itemPoint^.toPoint^.toPoint);
```

and so on, indefinitely. The static program variable *itemPoint* points to the first of the dynamic records, and then each in turn points to the next one, and so on. Everything beyond the initial pointer is dynamic. Such a chain of records is called a linked list, and with a little work, does have the potential to be used to hold the data in the airline reservation problem mentioned in [section 12.4.3](#).

A linked list consists of a sequence of dynamically allocated data items, each element of which includes a pointer to the next item on the list. The first item on the list is called the head; the last item is called the tail, and the program pointer that points to the head item is called the head pointer.



Notice that the last item on the list cannot point anywhere. The value of its pointer variable should be set to NIL as a terminal flag.

Also, look carefully at the declarations above. Do you notice anything unusual? Those readers who are operating with a one-pass compiler, and those who believe in *declare-it-before-you-use-it* (which should be everyone else), might object that this declaration makes use of an entity of type *ItemData* (to declare the type *ItemPointerType*) before declaring it. Of course, if one writes it the other way around one has to use *ItemPointerType* before declaring it. This appears to violate a rule that this text has been very careful about following up to now.

Indeed, even in multi-pass compilers, there is still a *declare-it-before-you-use-it-in-a-declaration* rule, and the ISO standard actually distinguishes between what are here called *single-pass* and *multiple-pass* compilers by referring to them as following either a *declare-before-use* rule or a *declare-before-use-in-declarations* rule, respectively.

Clearly the former is a greater restriction than and includes the latter.

ISO Modula-2 expresses the rule that allows such circular references in this case in this manner:

*The use of an identifier within a new pointer type follows the keywords **POINTER TO** and is shielded from the declare-before-use-in-declarations that normally applies.*

This requires the declaration of the pointer to be the first of the two. Although this rule does not strictly say how far away in the program text the declaration of the type pointed to must be located, it should be placed immediately adjacent to the declaration of the pointer type as in the example above, for the sake of clarity. Indeed, some implementations may have an additional rule requiring either this proximity, or at least that the second declaration be under the same TYPE heading as the first.

12.10.2 Maintaining The Linked List

Besides the beginning step of initialization, the maintenance of a linked list needs to take into consideration:

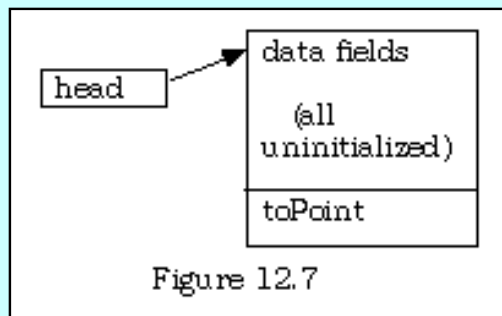
- how to append items to the list
- how to insert items into the list

- how to remove items from the list.

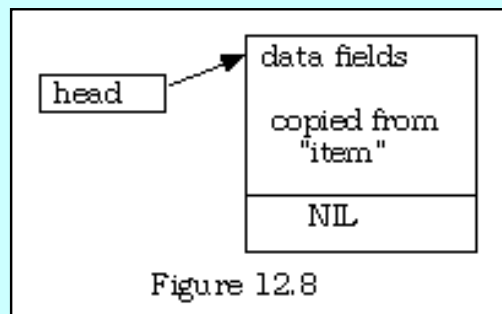
It is probably best to begin by considering the task of appending, that is of adding at the end of a list. Quite apart from the fact that most lists have things added in that position from time to time, one needs to add the first item to the end of the list before one will even have a list. Assume that, in a similar fashion to the last section, the following declarations have been made:

```
TYPE
  ItemPointerType = POINTER TO ItemData;
  ItemData =
    RECORD
      number : CARDINAL;
      (* more data fields here *)
      toPoint : ItemPointerType;  (* Pointer field *)
    END;
VAR
  head, point : ItemPointerType;
  item : ItemData;
```

At this stage, there is no list, just an uninitialized pointer and data item of the correct type to be a list element. The execution of `NEW (head)` creates an instance of the *ItemData* type that can be the first item in the list.



Next assign a copy of *item* (perhaps filled in by keyboard inputs) to *point*[^]. All the data fields are filled in, and so is the pointer field, but with no particular initialized value. Give it the value NIL to begin with.



Here is the code:

```
PROCEDURE AppendFirst (item : ItemData);
(* Pre: the data fields of the item have been filled in
   Post : head now points to the first record, which points nowhere *)
BEGIN
```

```

NEW (head);
head^ := item;  (* "point^" is the actual dynamic record *)
head^.toPoint := NIL;  (* this item points nowhere yet *)
END AppendFirst;

```

Now if one assumes that at some point the list has one or more items in it with the *toPoint* field of the last one containing the value *NIL*, the strategy for appending an item is to search through the list until the last one is found (*previousItem.toPoint = NIL*), then create space for and assign the new item, change the last *toPoint* previously present to point to the new item, and change the *toPoint* of the new one to *NIL*. Here is the code:

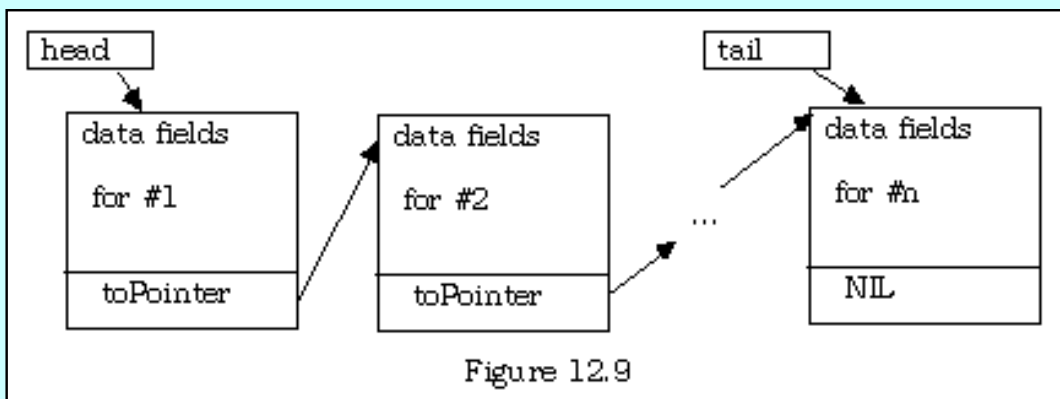
```

PROCEDURE Append (item : ItemData);
(* Pre: the data fields of the item have been filled in
   Post : old last one now points to the item record, which points nowhere *)
VAR
    point, temp : ItemPointerType ;
BEGIN
    temp := head;

WHILE temp^.toPoint # NIL
    DO      (* search through the list *)
        temp := temp^.toPoint      (* for the last item *)
    END;
    NEW (point);
    point^ := item;
    temp^.toPoint := point;  (* set old last one to point to new one *)
    point^.toPoint := NIL;  (* defines new end *)
END Append;

```

Actually, if one were doing a lot of adding to the end of the list, it would make more sense to keep another program variable that pointed to the last item, that is, that kept the address of the last section of memory assigned by the **NEW** command. This kind of variable is usually called a *tailpointer*.



Now the addition of a member to the list (even the first one, where the head and tail pointers would have both been initialized to *NIL*) could go like this:

```

(* in the main program *)
head2 := NIL;

```

```

tail2 := NIL;

PROCEDURE Append (item : ItemData);    (* pass in item to add *)
VAR
    temp : ItemPointerType ;
BEGIN
    NEW (temp);    (* get space *)
    temp^ := item;    (* assign it *)
    temp^.toPoint := NIL;
    IF head2 = NIL
        THEN
            head2 := temp    (* Do this only if first one *)
        ELSE
            tail2^.toPoint := temp    (* otherwise make old tail item *)
        END;    (* point to new tail *)
    tail2 := temp    (* reset tail pointer always *)
END Append;

```

One advantage of working with dynamic data is that once an item is dynamically established via NEW and the assignment of data, it need not be moved around in memory. This contrasts with an earlier situation where if one had, say, 100 Records in an array and wished to insert a new one at position [3], then everything from position [3] to the last active one would have to be moved to the next higher index number to make room. This may involve a very large amount of memory reassignment.

```

FOR count := last TO 3 BY -1    (* up to 97 steps to make one place *)
DO
    list [count + 1] := list [count]
END; list [3] := item;

```

For very large arrays, and records that have several fields, the amount of time consumed in such a loop is unacceptable. One solution to this problem was given earlier in the chapter, and involved employing an array of pointers to save space and assignment time. Another solution involves implementing the collection as a linked list. Here too, the insertion of a new item just before a position k is much easier.

```

insert =
    set traverse pointer point to head
    move along list k-1 steps (point points to item k-1)
    get new dynamic memory for item with pointer temp
    assign the passed item to temp^
    link the new item in
        copy toPoint from item k-1 to temp^.toPoint
        copy temp to the toPoint from item k-1

```

Here is a diagram showing the old link that is discarded and the new links that are filled in:

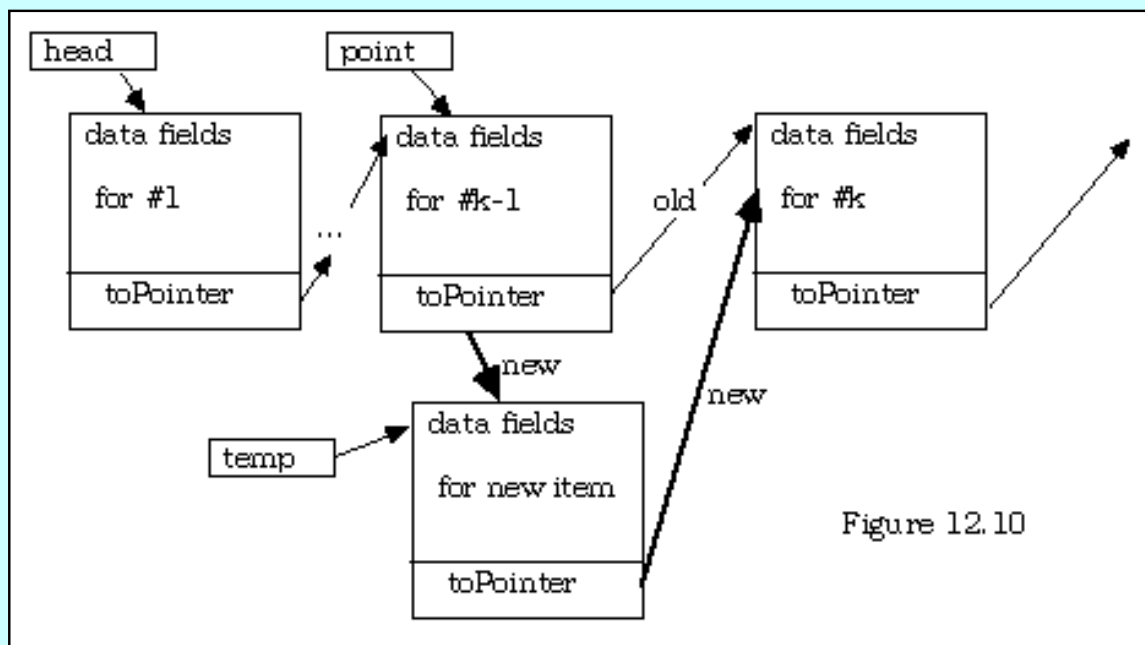


Figure 12.10

Note that the code given below also handles the special cases of inserting before item 0 or 1 (both taken as inserting at the head) and "before" an index that is beyond the tail (interpreted as appending.)

```

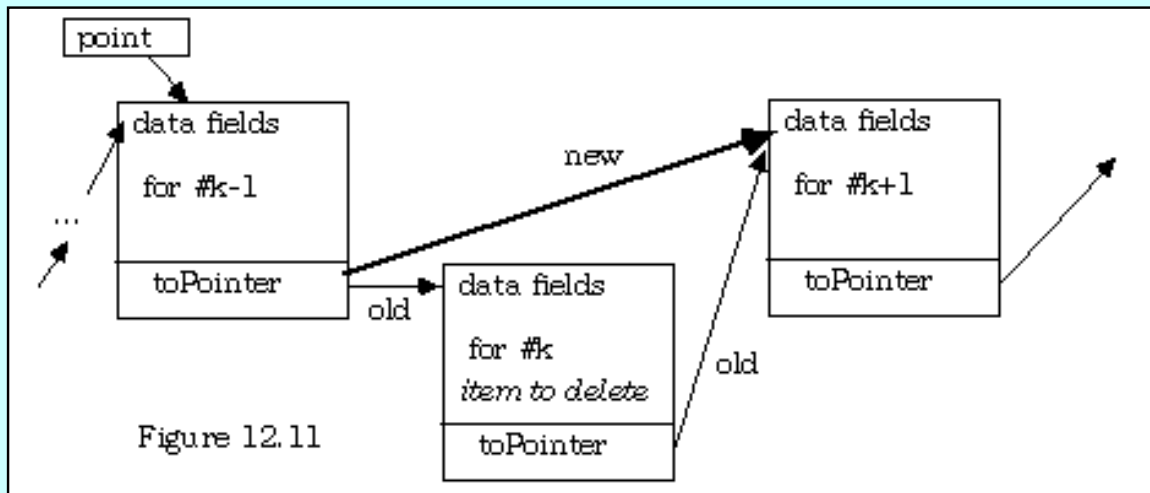
PROCEDURE InsertBeforeIndex (item: ItemData; index: CARDINAL);
  (* pass in item to insert and place to put it
  Post: if index is 0 or 1 insert at head;
         else, insert before given index or at end if given index too high *)
VAR
  point, temp : ItemPointerType;
  count: CARDINAL;
BEGIN
  point := head;
  count := 2;
  WHILE (count < index) AND (point^.toPoint # NIL)
  DO
    point := point^.toPoint;
    INC (count);
  END;      (* point now points to item # index-1 *)
  NEW (temp);      (* get space for item *)
  temp^ := item;    (* assign it *)
  IF point = head   (* i.e. 2 < index THEN (* index past tail *)
    point^.toPoint := temp; (* so do an append *)
    temp^.toPoint := NIL;
  ELSE (* typical insert in between *)
    temp^.toPoint := point^.toPoint; (* new item points to index *)
    point^.toPoint := temp;      (* item# index - 1 points to new one *)
  END;    (* if *)
END InsertBeforeIndex;

```

Rather than move a large amount of data around in memory as would have to be done with array insertion, one just makes item# (index - 1) point to the new item, and the new item point to the old item# index. Changing two

pointer values suffices to link in the new item.

With this code done, it should be easy to see how an item can be removed from the list by breaking a link, establishing a new one, and disposing of the memory associated with the item removed.



```

PROCEDURE Delete (index : CARDINAL);
VAR
    point, temp: ItemPointType;
    (* temp will hold one to dispose of *)
    count: CARDINAL;
BEGIN
    IF head = NIL (* nothing in list *)
    THEN
        RETURN
    END;
    point := head;
    count := 2;
    WHILE (count < index) AND (point^.toPoint # NIL)
    DO
        point := point^.toPoint;
        INC (count);
    END;      (* p now points to item # index-1 *)
    IF point = head
    THEN
        temp := head; (* get one to dispose *)
        head := point^.toPoint (* re-point head *)
    ELSIF count < index THEN (* index past tail *)
        RETURN (* cannot delete what isnt there *)
    ELSE
        temp := point^.toPoint; (* pointer to the one that goes *)
        point^.toPoint := temp^.toPoint (* re-link to next one *)
    END;      (* if *)
    DISPOSE (temp);      (* reclaim its memory *)
END Delete;

```

With these simple procedures, it is now possible to maintain a linked list and to have it contain as many items as the memory will allow.

Of course, there are also disadvantages to using linked lists. Although it may be a simple matter to put items into a list in a particular order (do comparisons first before inserting, or include code to maintain the list sorted as each item is inserted) it would be very tedious indeed to rearrange the list after it was in place. Just swapping two list items would not be a trivial exercise, and moving items around in the manner needed for a sort is not to be thought of. Arrays are much better for the kind of data access that requires frequent reference to items in the same amount of time by their index number in the structure (random access). Note, on the other hand that:

A linked list is by its very nature a sequentially accessed structure.

Nor is it possible to get around the sequential nature of the access by calculating an address based on the starting point and the size and index of an item as was done earlier in this chapter for a dynamic array. The latter was assigned its memory in a single block, and the computation of an address in order within this block made sense. The linked list is not assigned memory in a continuous block at all. Indeed, the assumption is that the locations of successive items in the list could be anywhere in memory, and that the only glue that holds the whole structure together is the pointers that chain one to the next.

The student is invited to build a test harness for the routines above, or, better yet, to incorporate them (with some minor changes) into a library module encapsulating a list ADT for a favourite data item and then test this with an application module. Such a library module would have to contain an implementation of a list type, perhaps

TYPE

```
ItemPoint = POINTER TO Item;
Item =
  RECORD
    (* all the data fields *)
    toPoint : ItemPoint
  END;
List = POINTER TO ListData; (* opaque redefinition *)
ListData =
  RECORD
    head, tail : ItemPoint;
  END;
```

Procedures would have to be included to create one of these lists (set up a data record for it,) dispose of one (delete all items and dispose of the list data record,) and to fetch and update data at some node.

REMINDER: Do not forget to import both ALLOCATE and DEALLOCATE from the module Storage before making use of NEW and DISPOSE.

[Contents](#)

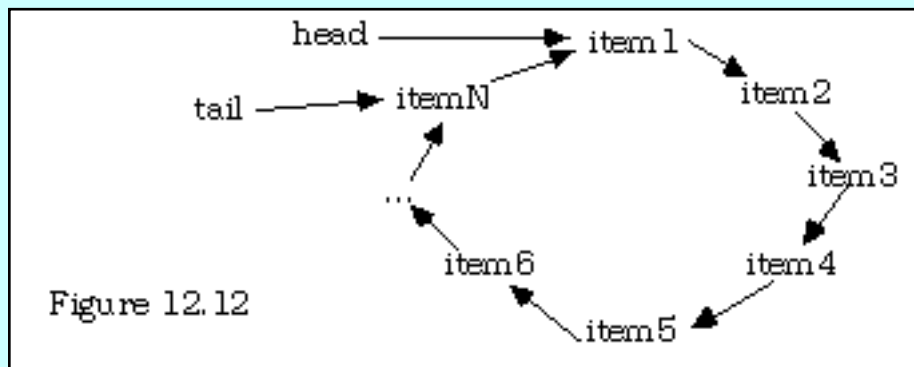
12.11 Variations on the List Theme

One of the difficulties in working with the type of list used so far is that a search for one record in the list must begin with the head pointer and then proceed pointer by pointer, item by item, until the desired object is reached (a strict sequential search.) Keeping a tail pointer around (and updated properly) may help, especially if one frequently appends, but there are other minor improvements that can sometimes gain efficiency.

12.11.1 Circular Lists

A list can "swallow its tail" by having the pointer in the tail item contain the location of the first item, instead of NIL. Now, the position of the head is rather arbitrary, for it is possible to get from any item to any other one. If items were frequently added near each other, it might make sense to change the head pointer so that it always pointed to the most recent insertion.

A list in which the last item points to the first item is said to be circular.

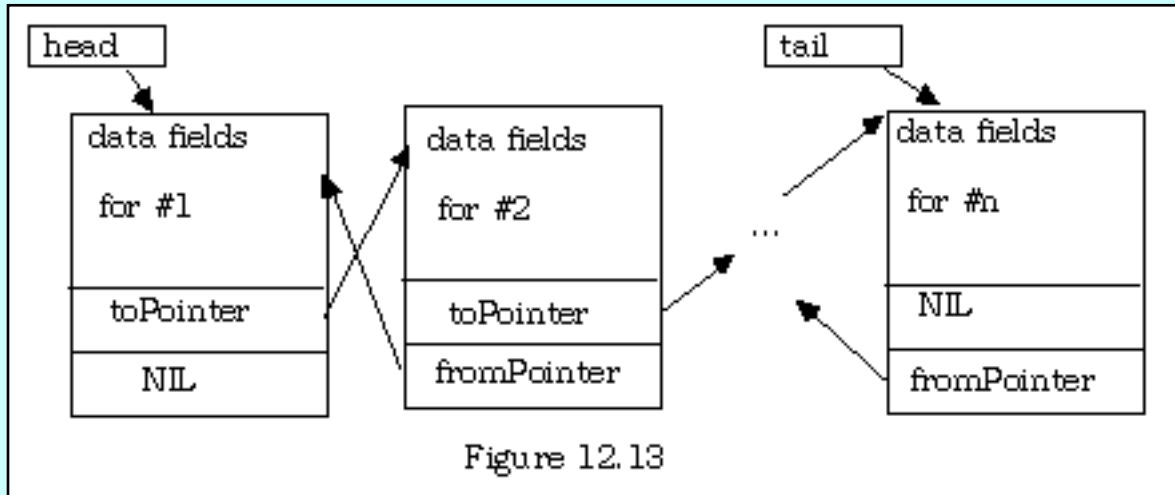


In a circular list, there is in a sense no such thing as a last item, but it is still useful to keep a tail pointer to find the last logical item--after all, it can no longer be found by searching for a NIL pointer. It might also be useful to maintain a variable that stores the number of active items in the list, so that a traverse with a high index number does not waste time by going around the circle and examining items more than once. In such a list, inserting before the head changes the head pointer to point to the new item, and inserting after the tail changes the tail pointer to point to the new item, but both place a new item after the old tail and before the old head in the circle.

12.11.2 Two-Way Lists

Still another scheme would have each data item contain two pointers--one to the previous record in the list and a second to the subsequent one. Naturally, this idea could be combined with the one above and

the forward pointer of the tail item could point to the head item and vice-versa, although this additional variation to produce a circular two way list is not shown in figure 12.13.



The following example sketches the declarations and provides the procedure for inserting data items into a two way linked list. The list is a collection of records, consisting of student names in alphabetical order by last name and also containing their grades. The test harness does a few inserts and prints out only the first character of each last name so as to verify the inserting.

```
MODULE TwoWayList;
```

```
(* a very simple test harness
   to demonstrate inserting in
   a two-way linked list
   by R. Sutcliffe 1995 05 23 *)
```

```
FROM Strings IMPORT
```

```
    Compare, CompareResults;
```

```
FROM Storage IMPORT
```

```
    ALLOCATE, DEALLOCATE;
```

```
FROM STextIO IMPORT
```

```
    ReadString, WriteString, WriteChar, WriteLn, SkipLine;
```

```
TYPE
```

```
    Name = ARRAY [0..10] OF CHAR;
```

```
    StudentRecPtr = POINTER TO StudentRec;
```

```
    StudentRec =
```

```
        RECORD
```

```
            last, first : Name;
```

```
            mark1, mark2 : REAL;
```

```
            toPoint, fromPoint : StudentRecPtr;
```

```
        END;
```

```
VAR
```

```
head : StudentRecPtr;  
student : StudentRec;
```

```
PROCEDURE EnterData (VAR student: StudentRec);
```

```
BEGIN
```

```
    WITH student
```

```
        DO
```

```
            WriteString ("Enter the last name ");
```

```
            ReadString (last);
```

```
        SkipLine;
```

```
        (* and continue in this vein *)
```

```
        END;
```

```
END EnterData;
```

```
PROCEDURE WriteData (student: StudentRec);
```

```
(* just writing one letter - this is only a test *)
```

```
BEGIN
```

```
    WITH student
```

```
        DO
```

```
            WriteChar (last[0]);
```

```
            (* and continue in this vein *)
```

```
        END;
```

```
END WriteData;
```

```
PROCEDURE WriteList (p : StudentRecPtr);
```

```
BEGIN
```

```
    WHILE p # NIL
```

```
        DO
```

```
            WriteData (p^);
```

```
            p := p^.toPoint; (* traverse to next item *)
```

```
        END;
```

```
    WriteLn;
```

```
END WriteList;
```

```
PROCEDURE Insert (classMember : StudentRec);
```

```
VAR
```

```
    temp, point : StudentRecPtr;
```

```
BEGIN
```

```
    (* handle case of first one to do *)
```

```
    IF head = NIL
```

```
        THEN
```

```
            NEW (head);
```

```
            head^ := classMember;
```

```

    head^.toPoint := NIL;
    head^.fromPoint := NIL;
    RETURN
END; (* if *)

(* all subsequent additions come here *)
point := head;
WHILE (point^.toPoint # NIL)
    AND (Compare (classMember.last, point^.last) = greater)
DO
    point := point^.toPoint;
END;
(* found place at point^ *)
NEW (temp);      (* so get memory for it *)
temp^ := classMember; (* put it there *)
    (* and set up the pointers *)

IF (point^.toPoint = NIL) (* at end so check last one *)
    AND (Compare (classMember.last, point^.last) = greater)
THEN (* must append *)
    temp^.fromPoint := point;
    point^.toPoint := temp;
    temp^.toPoint := NIL;
ELSE (* goes before point ^ *)
    IF point = head
        THEN (* reset head *)
            head := temp
        ELSE
            point^.fromPoint^.toPoint := temp;
        END;
    temp^.fromPoint := point^.fromPoint;
    point^.fromPoint := temp;
    temp^.toPoint := point;
END;

END Insert;

(* other procedures go here *)

VAR
    count : CARDINAL;

BEGIN (* main program does a few inserts *)
    head := NIL;

```

```
FOR count := 1 TO 7
DO
    EnterData (student);
    Insert (student);
    WriteList (head);
END;

END TwoWayList.
```

One run log for this test program looked like this:

```
Enter the last name f
f
Enter the last name l
fl
Enter the last name h
fhl
Enter the last name b
bfhl
Enter the last name b
bbfhl
Enter the last name a
abbfhl
Enter the last name m
abbfhlm
```

Notice that the program did not actually *use* the two-way feature of the list in the one procedure presented. One possible use, especially if the list became lengthy would be initiate searches for an item in the last half of the list by starting at the tail item (the pointer would have to be stored, and modifications made even to the one procedure presented here) and working backwards. This would shorten the average search time. A circular list with two index pointers, one for the head and one for the "middle" item (halfway along the list), would require even less search time, but one would have to design the code to search forward or backwards starting at the appropriate one of the three indicated points.

[Contents](#)

12.12 Variant Dynamic Records

If one has a variant record (see [Section 11.7](#)) that contains one or more tag fields and their variants call for differing amounts of memory, then the statements

```
SIZE (thePoint);  
NEW (thePoint);
```

where *thePoint* is a pointer to such a record will necessarily report and reserve (respectively) the maximum number of memory locations possible for the variant.

However, if there are large differences in memory requirements for the variants, it might be useful to reserve only the amount necessary. There is a variant (sic) version of NEW to do just this; one places after the pointer parameter the values of each tag field in turn. For instance, given the declarations:

```
TYPE  
  Item = POINTER TO ItemData;  
  ItemData =  
    RECORD  
      number : CARDINAL;  
      CASE tag : BOOLEAN OF  
        TRUE :  
          married : BOOLEAN |  
        FALSE :  
          num1, num2, num3 : CARDINAL  
      END;  
  END;  
VAR  
  point1, point2, point3: Item;
```

a call to NEW (point1) reserves the maximum possible number of locations (for number, tag, num1, num2, and num3,) but a call to

```
NEW (point2, TRUE);
```

reserves only the number of locations needed for number, tag, and married. A call to

```
NEW (point2, FALSE);
```

reserves the number of locations needed for number, tag, num1, num2, and num3.

If there are more variant parts, the value of each tag field in turn is listed. Exactly the same considerations apply to DISPOSE. If the user wishes to know the number of locations used by a variant, the procedure SIZE will not, as indicated above, report the correct value. In order to make the computation, it is necessary to import the procedure SYSTEM.TSIZE. This procedure has the same syntax as the variant forms of NEW and DISPOSE except that its first parameter is always a type rather than a variable.

To illustrate further, suppose one has:

```
TYPE  
  Item = POINTER TO ItemData;  
  ItemData =  
    RECORD  
      int : INTEGER;  
      CASE b : BOOLEAN OF
```

```

    TRUE :
        sex : BOOLEAN |
    FALSE :
        num1, num2 : INTEGER
END;
CASE descriptor : CARDINAL OF
    1 :
        flag : BOOLEAN|
    2..5 :
        int2 : INTEGER
    ELSE
        num3, num4, num5 : INTEGER
END
END;

```

```

VAR
    point : Item;

```

Now, one may write things like:

```

NEW (point1);
WriteString ("size of point1^ is ");
WriteCard (SIZE (point1^), 6);
WriteLn;

NEW (point2, FALSE);
WriteString ("tsize of point2^ is ");
WriteCard (TSIZE (ItemData, FALSE), 6);
WriteString (" note that size of point2^ is ");
WriteCard (SIZE (point2^), 6);
WriteLn;

NEW (point3, TRUE);
WriteString ("tsize of point3^ is ");
WriteCard (TSIZE (ItemData, TRUE), 6);
WriteLn;

```

When this code was run on one machine, the results were as follows:

```

size of point1^ is      18
tsize of point2^ is    18 note that size of point2^ is      18
tsize of point3^ is     6 note that size of point3^ is      18

```

Observe that SIZE always gives the maximum number of locations whereas TSIZE gives the number appropriate to the particular variant. The ISO standard says that calls to

NEW (p, <expression list<expression list"magic" (varying number of parameters) is in TSIZE and that is why it is located in SYSTEM.

NOTES: 1. For this variation on memory allocation to work correctly, it is essential that a variable's tag fields be correctly initialized and remain the same once it is allocated, so that when DISPOSE is called, the list of constants is the same as when memory was allocated by NEW.

2. Some Modula-2 compilers can only assign the maximum amount of space to variant records, so much of this discussion may not apply to them. They would always

give the same value when the above code was run.

3. There can be difficulties with writing such records out to the disk and later reading back in, as there must be a way of determining the variants before the NEW statement is executed. Moreover, the random access file scheme depends on all records having the same size, and could not be used with variants of different sizes.

In addition, when using TSIZE, if there is more than one parameter, the first must be a record. That is, TSIZE will work on any type, but it makes no sense to supply a type that is not a variant record, and then supply additional parameters that can only be tag field values. The compiler will flag this error. The values for the fields of variant records may also be included as parameters following the record type. TSIZE then returns the size of the specified variant.

For any use of TSIZE, whether translated form a call to NEW or not, if there are two or more variant fields, the values of the various tag fields are listed in the same order they are declared.

NOTES: 1. In some non-ISO versions, this facility is limited to one *main* variant, which must be at the end of the RECORD. Additional variant tags can be listed only if they are *nested* within the first.

2. In some non-ISO versions, SIZE and TSIZE both return the number of words. This must be doubled or possibly quadrupled to obtain the number of storage locations (often called *bytes* on such systems).

3. The tag fields whose values are given must be fields of the main record type itself, and not fields of a sub-record type.

4. The time at which these functions operate is also worth noting. TSIZE operates strictly on entities whose size is known to the compiler, and so does SIZE most of the time. Therefore, these calculations are done by the compiler, not while the finished program is running. There is one exception, but only in non-ISO versions. When SIZE operates on an open array, the size is not known until the assignment has been made during the program's execution. So in such cases instance, SIZE is a run-time procedure. The ISO standard simply does not allow one to take the SIZE of an open array.

The restrictions on SIZE and TSIZE in the ISO version ensure that both can be used in constant expressions and can be evaluated at compile time.

[Contents](#)

12.13 Chapter Summary

This chapter covered these topics:

- pointers
- static and dynamic memory management
- how to obtain and return dynamic memory
- how to create dynamic structures and ADTs
- what opaque data types really are
- an introduction to linked lists
- variant dynamic records

It included discussion of the following Modula-2 items:

Symbols:

^ (dereferencing operator)

Imports:

- From Storage:
 ALLOCATE, DEALLOCATE
- From SYSTEM:
 TSIZE (Two forms)

[Contents](#)

Reserved Words

POINTER

TO (new context)

Standard Identifiers

DISPOSE (Two forms)

NEW (Two forms)

NIL

12.14 Assignments

Questions

1. What is a pointer and how is it declared?
2. What is the Modula-2 symbol \wedge called?
3. What is the difference between a reference to *point* and a reference to *point* \wedge ?
4. With what two types are all pointer variables assignment compatible?
5. How can pointers be used to make a program more efficient even without using dynamic memory?
6. What does it mean to say that some pointer expressions must be *guarded*? What is the error being guarded against?
7. What procedures can be employed to do pointer arithmetic?
8. What is a dangling pointer and how does it get that way?
9. Explain how pointers can be used to simulate the action of a variable parameter.
10. Explain the operation of the machine stack. What is a stack pointer?
11. What is an activation record?
12. Explain how the use of activation records on a stack allows recursion.
13. Why would one not call the use of activation records on a stack a true dynamic activity?
14. What is the heap?
15. What two pervasive (built-in) procedures does Modula-2 employ to manage the heap?
16. What two unusual (magical) properties do these two procedures have?
17. The ISO standard says that the two procedures mentioned in the last question are *often* translated into two library procedures. What two library procedures are these, and why does the standard say *often*?
18. What restriction on the use of function procedure identifiers was highlighted in this chapter?
19. A program needs a pointer to access every dynamic item, yet the goal in using dynamic memory is to be free of the limitations on static memory. How is this problem avoided?
20. What are two applications of dynamic memory?
21. What is a linked list?
22. Name and explain two extensions to the linked list ADT.
23. Write a definition module for a linked list ADT. You may employ any base data type you wish.
24. What is an opaque data type and what does the ISO standard require them to be re-declared as?
25. Why was this limitation on the nature of opaque data types made?
26. How does the handling of variant dynamic records differ from that of ordinary dynamic records?
27. Why is it that a dynamic variant record might occupy different amounts of memory depending on the value of its tag field(s)?
28. A static variant record has the same amount of memory regardless of the value of its tag fields. Explain why this is the case.
29. A record type with a variant is found as the type of a field in another record. Do the sub-record tags affect memory allocation for the entire record?

30. How would you refer to the integer item ultimately pointed to through the variable *handle* below?

TYPE

```
HandleType = POINTER TO POINTER TO INTEGER;
```

VAR

```
handle : HandleType;
```

31. The use of SIZE (point[^]) could, in theory, cause a run-time error if the pointer had a value of NIL. In actual fact, this will never take place in the ISO version of Modula-2 because of the way in which SIZE is evaluated. Explain.

Exercises

32. Write a small program to check on the way your implementation handles Storage.ALLOCATE and Storage.DEALLOCATE. Does a failure to allocate memory result in a NIL value being assigned to the pointer? Does DEALLOCATE assign NIL to the pointer? What does your system actually do when a reference to *point*[^] is made while *point* has the value NIL?

33. Consider the following declarations. Devise a short test harness and determine the amount of memory allocated for the variants on your machine.

TYPE

```
Date =
```

```
  RECORD
```

```
    day: [0 .. 31];
```

```
    month: [0 .. 11];
```

```
    year: CARDINAL;
```

```
  END;
```

```
Index = CARDINAL;
```

```
Frac = REAL;
```

```
STRING = ARRAY [0 .. 80] OF CHAR;
```

```
Person =
```

```
  RECORD
```

```
    name, birthPlace: STRING;
```

```
    birthDate: Date;
```

```
    CASE tag OF
```

```
      student:
```

```
        (* Nothing--null record *)
```

```
      | faculty:
```

```
        rank: STRING
```

```
      | staff:
```

```
        department, position: STRING
```

```
    END
```

END ;

34. Declare your own variant record type with one or more nested variants and test the possible tag field values for the amount of memory allocation on your machine.
35. Declare your own variant record type with two or more non-nested variants and one or more nested variant and test the possible tag field values for the amount of memory allocation on your machine.
36. By finding and printing the address of value parameters and/or a local variable, verify that your system operates a stack for activation records in the manner described in this chapter. Are the first stack addresses greater or less than the addresses of main program memory variables? Does the stack grow up or down in memory (or neither?)
37. Declare variables of two different dynamic types of different sizes. Allocate memory to one, get and print (interesting task) the address, then deallocate the memory and reallocate it, first to the same pointer, then to one of the other type, printing out the addresses each time. What conclusion do you come to about the location of the heap on your machine relative to the main program variables and the stack? Does the memory given up on a deallocate get used when a new allocation is requested?
38. Declare and fill with values a dynamic array of reals (dimension chosen at run time) and then print the values out to verify you have done it correctly.
39. Implement and test for yourself the sorting of a collection of three records with auxiliary pointers as suggested in [section 12.2](#).
40. Implement and test more complete and bullet-proof versions of the modules [MakeRecords](#) and [GetPrintRecords](#) in section 12.6. Your improvements should be substantial, not just cosmetic.
41. Test the module [OpaquePoints](#) in section 12.9 with an appropriate test harness.
42. Assemble the procedures of [section 12.10](#) into a test harness, and add procedures to find and to delete a specific item.
43. Complete the module [TwoWayList](#) in section 12.11 by adding a procedure to delete a specific item by index number and another procedure to delete one by the data it contains.
44. Modify the module [TwoWayList](#) in section 12.11 to have another data field, say, a student number, and two more forward and backward pointers. New items should be added in such a way that they are in order by one set of pointers for the names (as shown in the example) and by the other set of pointers for the numbers. Deletion should be by name or number (student number, not index number.) The procedures to write the data need to be modified as well for testing purposes.

Projects

45. Implement and test the linked list ADT you defined in question 23 above.
 46. Define, implement and test a dynamic string ADT.
 47. Define, implement and test a dynamic array of reals ADT whose dimensions can be established at run time. Include and test in a client program the code to add and to multiply two of these.
 48. Devise, implement, and test a means of storing and retrieving (in random access fashion) variant records of different lengths from a disk file.
-

Contents

Chapter 12

Pointers and Dynamic Data

[12.0 Chapter Goals](#)

[12.1.1 Pointer Variables](#)

[12.1.2 Pointer References](#)

[12.1.3 The value NIL](#)

[12.2 Applications of Pointers](#)

[12.2.1 Pointers and Parameters](#)

[12.2.2 Pointers and Sorting](#)

[12.3 Pointer Arithmetic](#)

[12.4 Dynamic and Static Memory](#)

[12.4.1 Static Memory Use](#)

[12.4.2 Procedures and the Stack--Automatic Dynamics](#)

[12.4.3 Dynamic Memory and the Heap--Program Controlled Dynamics](#)

[12.5 Managing Dynamic Memory in Modula-2](#)

[12.6 An Example--Dynamic Records and Files](#)

[12.7 Towards A Dynamic Array ADT](#)

[12.8 Pointers and Return Types](#)

[12.9 Opaque Types Revealed](#)

[12.10 Pointers and Lists](#)

[12.10.1 Declaring The Linked List Apparatus](#)

[12.10.2 Maintaining The Linked List](#)

[12.11 Variations on the List Theme](#)

[12.11.1 Circular Lists](#)

[12.11.2 Two-Way Lists](#)

[12.12 Variant Dynamic Records](#)

[12.13 Chapter Summary](#)

[12.14 Assignments](#)

13.0 Chapter Goals

The purpose of this chapter is to provide an introduction to some of the methods commonly used to search or sort data. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

No new data types are taken up in chapter 13.

Data Manipulation Abstractions

General:

searching and sorting algorithms and their performance, simple sorts and advanced sorts

Realized in the Modula-2 notation:

linear and binary searches, bubble sort, insert sort, selection sort, Shell sort, comb sort, quicksort, and merge sort

Programming Abstractions

General:

No new programming abstractions are taken up in this chapter; but the ability to analyze the performance of loops is important to this material.

Realized in the Modula-2 notation:

13.1 Searching

The type ARRAY has been used extensively in several chapters thus far, to store and manipulate itemized information of the same type. Now suppose one goes beyond mere storage and asks questions about retrieval systems for data that has been organized this way. Specifically, suppose one has an array of some type of data and wishes to discover whether or not the value of some separately held item of that same type is already in the array, and if so, at what position. (i.e. What is its index?)

13.1.1 Linear (Sequential) Searches

The most straightforward solution to this problem is to start with the lowest array index, and compare the value of the separately held item in question systematically to every item in the array to see if it is equal to one of them.

The search of a list of items in consecutive order by examining each item in the list in turn is called a linear or, sequential search.

A linear search does not depend for its success on the items being searched already being in any particular order. The value being sought will either be equal to one of the ones already in the array, or it will not. If it is, the index number of the position at which it is found can be returned. Otherwise, some indication that the search has failed must be passed back. Here is a procedure to do just that:

```
PROCEDURE Find (item: ItemType;
  offset, num : CARDINAL;
  source: ARRAY OF ItemType;
  VAR found: BOOLEAN; VAR pos : CARDINAL);
```

```
(* searches for the item in the array source from offset to end
Pre: offset is the number of positions after the first at which the search is to
begin; num is the number of positions after the offset that are to be examined.
offset + num <= HIGH ( source)
Post: if the value is found in the array, the cardinal returned is the number of
positions after start where it is found and found is set to true.  If the value is
not found, the cardinal returned is undefined and found is set to false *)
```

```
BEGIN
  pos := offset;
  found := FALSE;
  WHILE (num > 0) AND NOT found
  DO
    IF item = source [pos]
    THEN
      found := TRUE;
    ELSE
      INC (pos); (* check next one *)
      DEC (num); (* decrease number remaining *)
    END (* if *)
  END; (* while *)
```



```
END Find;
```

Because an open array parameter is employed in this procedure, it can be used for any size array. However, since not all the positions of the array may be in use, and to allow some to be skipped at the beginning, the starting point relative to the first item, and the number of items to examine are both passed. If the item value being searched for is found, its relative index from *offset* will be returned, along with a boolean flag set to *true*. If the entire specified range of the array is searched without success, the flag is set to *false* and the value in *pos* will be invalid. (The logic implemented here will actually result in *pos* being set to *offset* + *num*, but the specifications could have been met in some other manner.)

Notice that this procedure cannot be used as is to find an element of any type whatever in an array of that type, just by changing a few names. It will work as it is presented here only for values of those types for which the relation "=" is meaningful. That is, it works for integer, cardinal, real, character, and boolean types, but not for arrays of these.

However, if the line where the comparison is done is replaced by:

```
IF Compare (item, source [count]) = equal;
```

where

```
TYPE
```

```
    CompareResult = (less, equal, greater);
```

and

```
PROCEDURE Compare (item1, item2 : ItemType) : CompareResult;
```

are suitable (possibly imported) procedure for making such comparisons on the data type in question, then this procedure will approach universality. It will need only the specific type name inserted to create a specific instance of it to work in any situation.

To cast this specifically as searching an array of strings to find a specific string, and assuming that the enclosing module contains the lines:

```
FROM Strings IMPORT
```

```
    CompareResult, Compare;
```

```
TYPE
```

```
    String : ARRAY [0 .. maxSize] OF CHAR;
```

yields (with the comments removed to avoid repetition:)

```
PROCEDURE Find (item: String;
```

```
    offset, num : CARDINAL;
```

```
    source: ARRAY OF String;
```

```
    VAR found: BOOLEAN; VAR pos : CARDINAL);
```

```
BEGIN
```

```
    pos := offset;
```

```
    found := FALSE;
```

```
    WHILE (num < source [search position] then    (* ignore top half *)
```

```
        top = search position - 1;
```

```
        (* by setting new top just below middle *)
```

```
    elsif value > source [search position] then
```

```
        (* ignore bottom half *)
```

```

        bottom := search position + 1
        (* set new bottom just above middle *)
    else    (* must be equal *)
        found = true;
    end (* if *)
until found or (top < bottom);

```

A numeric example using this algorithm is stepped through below:

Example:

Given the following sorted list of numbers
11, 21, 46, 52, 55, 67, 79, 81, 86, 97, 99
(i) search for the value 81.

Solution:

The algorithm proceeds as follows:

1. The terms are numbered from 0 .. 10. The first search position is 5, at which is found the number 67. The value 81 is compared to it and 81 is greater.
2. Next the items from position 6 through 10 are considered. These are 79, 81, 86, 97, 99. Here, the search position is 8 at which is the number 86, and 81 is less than this.
3. Thus the list is reduced to items 6 through 7. The remaining list is 79, 81. The search position item is the first one holding 79, and 81 is greater than this.
4. All that remains is the list with item 6 or 81 in it. The one wanted is equal to this, and so has been found. If it were not, then at the next step the bottom counter would be greater than the top counter; there would be no list left, and the search would fail.

(ii) search for 50 in the list.

The successive item numbers at the beginning of each step, the lists to be checked, and the search positions and results for each step are:

range	list	pos	result	new range
1) [0 .. 10]	the whole list;	5 (67)	less	[0 .. 4]
2) [0 .. 4]	11, 21, 46, 52, 55;	2 (46)	greater	[3 .. 4]
3) [3 .. 4]	52, 55;	3 (52)	less	[4 .. 3] fails

(ii) search for 10 in the list.

range	list	pos	result	new range
1) [0 .. 10]	the whole list;	5 (67)	less	[0 .. 4]
2) [0 .. 4]	11, 21, 46, 52, 55;	2 (46)	less	[0 .. 1]
3) [0 .. 1]	11, 21;	0 (11)	less	[0 .. -1] fails

Having now hand traced the algorithm to make sure that it works, it is time to translate this into a working procedure. As in the linear search, the binary one returns both the index, and a flag to indicate the validity of the data. To maintain compatibility with the last search procedure, the parameters are set out in the same manner, with an offset at which to begin within the array, and a maximum number of positions to search beyond that point. In addition, the algorithm has been reformulated in a *while* loop, and to ensure that the condition *top < bottom* for exit does not mean either *top < 0* (integer type needed) or *bottom <= HIGH* (*source*) *Post: if the value is found in the array, the cardinal returned is the number of positions after start where it is found and found is set to true. If the value is not found, the cardinal returned is undefined and*

found is set to false*) **VAR** bottom, top, max : **CARDINAL**; **BEGIN** bottom := offset; max := bottom + num; top := max; found := **FALSE**; **WHILE** (bottom < top) **AND NOT** found **DO** pos := (top + bottom) **DIV** 2; **IF** (item < source [pos]) **THEN** (* ignore top half *) top := pos - 1; **ELSIF** (item > source [pos]) **THEN** bottom := pos + 1 (* ignore bottom half *) **ELSE** (* item = source [pos] *) found := **TRUE**; **END**; (* if *) **IF NOT** found (* but bottom = top now *) **THEN** (* have not yet probed at bottom/top *) pos := bottom; **IF** item = source [pos] **THEN** found := **TRUE**; **END**; **END**; **END**; (* while *) **END** FindB;

Just how much better is the binary search than the linear one? In the list of eleven items in the range [0 .. 10] a maximum of four comparisons were required, though the actual number could be less if the value is found. In the linear case, the average number of comparisons would be 5.5 and the maximum number would be eleven. The number of linear comparisons depends on n, the number of items, (it is $O(n)$) but the number of binary method comparisons depends on the number of times t that one could take half of n before reaching a list length of one. This number is the same as the number of times one would have to double the number one to get n. Writing $2^t = n$, and taking logarithms with base two on both sides yields $t = \log_2(n)$.

A binary search is $O(\log_2(n))$.

For an array of 1000 items, one could be unlucky enough to require 1000 comparisons in a linear search (though the average would be 500), whereas the maximum number of comparisons to search the same data in binary fashion is ten. The savings become truly enormous with large lists. Note however, that the array must be sorted before being searched in binary fashion, and that the sorting will also take time.

However, for smaller arrays, the extra overhead involved in the binary search will be more important than the savings in making fewer comparisons. When the two methods are both used on the same arrays and the results compared, it turns out that a linear search is often more efficient below about 100 items. Since the time ratio between the extra arithmetic on the one hand and the extra comparisons on the other will vary depending on the architecture of the computer, it is not possible to state that this boundary is accurate for all cases. (The student is invited to find a more accurate limit as one of the exercises.) It is also worth noting that the binary search has two comparisons in the IF statement in the inner loop, and succeeds only when both of those comparisons fail. Moreover, an individual probe at a position pos fails to find the item sought more often than it succeeds. One might therefore try to improve the algorithm by having only one comparison inside the inner loop, ensuring that the search converges to a single item and then exits. The test for equality on that single item can then be done once, outside the loop. These ideas can be expressed as follows:

```
PROCEDURE FindB1 (item: ItemType;
    offset, num : CARDINAL;
    source: ARRAY OF ItemType;
    VAR found: BOOLEAN; VAR pos : CARDINAL);

(* same predicates *)
VAR
    bottom, top : CARDINAL;

BEGIN
    bottom := offset;
    top := offset + num;
    found := FALSE;

    WHILE (bottom < top)
    DO
        pos := (top + bottom + 1) DIV 2;
        IF item < source [pos]
        THEN    (* ignore top half *)
            top := pos - 1;
        ELSE
            bottom := pos    (* ignore bottom half *)
```

```

        END; (* if *)
    END; (* while *)
pos := bottom;
IF item = source [pos]
    THEN
        found := TRUE;
    END;
END FindB1;

```

The improvement in the inner loop that results from having only one comparison instead of two is partially offset by having to use the expression $(top + bottom + 1)$ instead of just $(top + bottom)$. It is also reduced by the fact that the loop will now not exit just because it happens to probe the right place. It only exits when the probe range has been reduced to one item. One might expect that any gains achieved are likely to be less with short lists, where the probability of hitting the item with a probe is higher (if it is there). On the other hand, for unsuccessful searches, the second method is likely to have some speed advantage.

For the sake of completeness, the binary search is also presented below in a more generic form that assumes a Compare procedure is visible in the same scope.

```

PROCEDURE FindBg (item: ItemType;
    offset, num : CARDINAL;
    source: ARRAY OF ItemType;
    VAR found: BOOLEAN; VAR pos : CARDINAL);

(* same predicates *)
VAR
    bottom, top : CARDINAL;

BEGIN
    bottom := offset;
    top := offset + num;
    found := FALSE;

    WHILE (bottom < top)
        DO
            pos := (top + bottom + 1) DIV 2;
            IF Compare (item, source [pos]) = less
                THEN (* ignore top half *)
                    top := pos - 1;
                ELSE
                    bottom := pos (* ignore bottom half *)
                END; (* if *)
            END; (* while *)
        pos := bottom;
        IF item = source [pos]
            THEN
                found := TRUE;
            END;
        END FindBg;

```

Several variations on the theme are possible. If the user does not mind using integer variables for the indexing and has an extra unused array position available, the indices can be allowed to run past one another without harm being done. In that event, and since a Compare procedure is ternary-valued, a CASE statement in the inner loop could be quite efficient.

Contents

13.2 Introduction to Sorting

In the last section, the efficient routine for finding data in an array was based on the premise that the data being searched was already sorted. Indeed, computers are used so extensively to process data collections that in many installations, a great deal of their time is spent maintaining that data in sorted order in the first place. It turns out that methods of searching a sorted list have much in common with methods of achieving that sorted condition.

In order to concentrate on sorting abstractions without having to be concerned with the type of data being sorted, most of the code presented in the rest of this chapter will be designed to sort only a single kind of data, namely arrays of cardinals. Only minor modifications in a few places are necessary to use the code presented in the next few sections for other kinds of data. The sorts presented here can be made generic enough to sort any kind of data, but that step requires some programming abstractions that will not be taken until chapter twelve, so the topic will be revisited in chapter thirteen with that aspect in view. The basic premise behind sorting an array is that its elements start out in some (presumably) random order and need to be arranged from lowest to highest. If the number of items to be sorted is small, a human reader may be able to tell at a glance what the correct order ought to be. If there are a large number of items, a more systematic approach is required. To do this, it is necessary to think about what it means for an array to be sorted. It is easy to see that the list

1, 5, 6, 19, 23, 45, 67, 98, 124, 401

is sorted, whereas the list

4, 1, 90, 34, 100, 45, 23, 82, 11, 0, 600, 345

is not. The property that makes the second one "not sorted" is that there are adjacent elements that are out of order. The first item is greater than the second instead of less, and likewise the third is greater than the fourth and so on.

Once this observation is made, it is not very hard to devise a sort that proceeds by examining adjacent elements to see if they are in order, and swapping them if they are not.

13.2.1 The Bubble Sort

This most elementary sorting method proceeds by scanning through the elements a pair at a time, and swapping any adjacent pairs it finds to be out of order. Thus, for the list in the example above, the first two items are swapped, then the (new) second item is compared to the third (and not swapped,) the third is compared to the fourth, and so on to the end. The list would be altered as follows (comparisons are emphasized.)

```
4, 1, 90, 34, 100, 45, 23, 82, 11, 0, 600, 345
1, 4, 90, 34, 100, 45, 23, 82, 11, 0, 600, 345
1, 4, 90, 34, 100, 45, 23, 82, 11, 0, 600, 345
1, 4, 34, 90, 100, 45, 23, 82, 11, 0, 600, 345
1, 4, 34, 90, 100, 45, 23, 82, 11, 0, 600, 345
1, 4, 34, 90, 45, 100, 23, 82, 11, 0, 600, 345
1, 4, 34, 90, 45, 23, 100, 82, 11, 0, 600, 345
1, 4, 34, 90, 45, 23, 82, 100, 11, 0, 600, 345
1, 4, 34, 90, 45, 23, 82, 11, 100, 0, 600, 345
1, 4, 34, 90, 45, 23, 82, 11, 0, 100, 600, 345
1, 4, 34, 90, 45, 23, 82, 11, 0, 100, 600, 345
1, 4, 34, 90, 45, 23, 82, 11, 0, 100, 345, 600
```

Unfortunately, the list is not yet sorted, as there are still several places where adjacent items are out of order. The number 0, for instance, which should be first, is in the ninth slot. Notice, however, that the largest item worked its way to the top position, and indeed, this algorithm will always force this to happen. Thus, if at this point the same strategy is continued, it is only the first n-1 items that need to be scanned. On the second pass, the second largest item will move to its correct position, and on the third pass (stopping at item n-3) the third largest will be in place. It is this gradual percolation, or *bubbling* of the larger items

to the top end that gives this sorting technique its name.

There are two ways in which the sort can terminate with everything in the right order. It could complete by reaching the $n-1$ st pass and placing the second smallest item in its correct position. Alternately, it could find on some earlier pass that nothing needs to be swapped. That is, all adjacent pairs are already in the correct order. In this case, there is no need to go on to subsequent passes, for the sort is complete already. If the list started in sorted order, this would happen on the very first pass. If it started in reverse order, it would not happen until the last one.

The code that follows embodies one way of expressing this algorithm. It requires a separate swap procedure, which is included here for the sake of completeness, and for a reminder of how swapping is done. Here, the type cardinal is used, but other types could also be employed. Note that the array may not be full, and unused items at the end are not to be included in the sort, so a cardinal parameter must be passed to inform the bubble sort how many items are active. This procedure does no error checking on the passed parameter; it is a precondition that the number passed be valid. Also note that the array being sorted is numbered $[1 .. n]$, $[0 .. n - 1]$, or in general $[a .. a + (n - 1)]$ exterior to the bubble sort procedure, but only $[0 .. n - 1]$ inside it. The procedure will therefore be informed where to begin sorting and where to stop, both as an offset from the first index. These will be passed in parameters *lBound* and *uBound* respectively, and there will be the least confusion, of course, if the arrays are numbered by the caller as $[0 .. \text{last}]$.

```
PROCEDURE Swap (VAR first, second : CARDINAL);
```

```
  (* Exchange values in first and second. *)
```

```
VAR
```

```
  temp : CARDINAL;
```

```
BEGIN
```

```
  temp := first;
```

```
  first := second;
```

```
  second := temp;
```

```
END Swap;
```

```
PROCEDURE BubbleSort (VAR source : ARRAY OF CARDINAL;
```

```
  lBound, uBound : CARDINAL);
```

```
  (* sorts beginning at item # lBound and ending at item #uBound, as numbered inside  
the procedure  $[0..HIGH]$ , not as on the outside. *)
```

```
(* Pre: uBound <= HIGH (source)
```

```
  Post : source is bubble sorted *)
```

```
VAR
```

```
  switch, count : CARDINAL;
```

```
BEGIN
```

```
  IF uBound > source [count + 1]
```

```
    (* compare adjacent pairs *)
```

```
    THEN
```

```
      Swap (source [count], source [count + 1]);
```

```
      INC (switch);
```

```
    END; (* if *)
```

```
  END; (* for *)
```

```
  DEC (uBound); (* for the next pass, examine one less item *)
```

```
  UNTIL (switch = 0) OR (uBound = 0);
```

```
  (* till no pairs swapped, or at first *)
```

```
  END; (* if *)
```

```
END BubbleSort;
```

When this sort was tested, statements were included to print the array after each comparison, and to print the number of swaps on each pass. The original data is given first, and the items that are compared are emphasized.

234	77	0	113	404	94	900	113	15	300	13	135
Pass number 1											
<u>77</u>	<u>234</u>	0	113	404	94	900	113	15	300	13	135
77	<u>0</u>	<u>234</u>	113	404	94	900	113	15	300	13	135
77	0	<u>113</u>	<u>234</u>	404	94	900	113	15	300	13	135
77	0	113	<u>234</u>	<u>404</u>	94	900	113	15	300	13	135
77	0	113	234	<u>94</u>	<u>404</u>	900	113	15	300	13	135
77	0	113	234	94	<u>404</u>	<u>900</u>	113	15	300	13	135
77	0	113	234	94	404	<u>113</u>	<u>900</u>	15	300	13	135
77	0	113	234	94	404	113	<u>15</u>	<u>900</u>	300	13	135
77	0	113	234	94	404	113	15	<u>300</u>	<u>900</u>	13	135
77	0	113	234	94	404	113	15	300	<u>13</u>	<u>900</u>	135
77	0	113	234	94	404	113	15	300	13	<u>135</u>	<u>900</u>
Swaps on this pass = 9											

Pass number 2											
<u>0</u>	<u>77</u>	113	234	94	404	113	15	300	13	135	900
0	<u>77</u>	<u>113</u>	234	94	404	113	15	300	13	135	900
0	77	<u>113</u>	<u>234</u>	94	404	113	15	300	13	135	900
0	77	113	<u>94</u>	<u>234</u>	404	113	15	300	13	135	900
0	77	113	94	<u>234</u>	<u>404</u>	113	15	300	13	135	900
0	77	113	94	234	<u>113</u>	<u>404</u>	15	300	13	135	900
0	77	113	94	234	113	<u>15</u>	<u>404</u>	300	13	135	900
0	77	113	94	234	113	15	<u>300</u>	<u>404</u>	13	135	900
0	77	113	94	234	113	15	300	<u>13</u>	<u>404</u>	135	900
0	77	113	94	234	113	15	300	13	<u>135</u>	<u>404</u>	900
Swaps on this pass = 7											

Pass number 3											
<u>0</u>	<u>77</u>	113	94	234	113	15	300	13	135	404	900
0	<u>77</u>	<u>113</u>	94	234	113	15	300	13	135	404	900
0	77	<u>94</u>	<u>113</u>	234	113	15	300	13	135	404	900
0	77	94	<u>113</u>	<u>234</u>	113	15	300	13	135	404	900
0	77	94	113	<u>113</u>	<u>234</u>	15	300	13	135	404	900
0	77	94	113	113	<u>15</u>	<u>234</u>	300	13	135	404	900
0	77	94	113	113	15	<u>234</u>	<u>300</u>	13	135	404	900
0	77	94	113	113	15	234	<u>13</u>	<u>300</u>	135	404	900
0	77	94	113	113	15	234	13	<u>135</u>	<u>300</u>	404	900
Swaps on this pass = 5											

Pass number 4											
<u>0</u>	<u>77</u>	94	113	113	15	234	13	135	300	404	900
0	<u>77</u>	<u>94</u>	113	113	15	234	13	135	300	404	900
0	77	<u>94</u>	<u>113</u>	113	15	234	13	135	300	404	900
0	77	94	<u>113</u>	<u>113</u>	15	234	13	135	300	404	900
0	77	94	113	<u>15</u>	<u>113</u>	234	13	135	300	404	900
0	77	94	113	15	<u>113</u>	<u>234</u>	13	135	300	404	900
0	77	94	113	15	113	<u>13</u>	<u>234</u>	135	300	404	900

0	77	94	113	15	113	13	<u>135</u>	<u>234</u>	300	404	900
---	----	----	-----	----	-----	----	------------	------------	-----	-----	-----

Swaps on this pass = 3

Pass number 5

<u>0</u>	<u>77</u>	94	113	15	113	13	135	234	300	404	900
0	<u>77</u>	<u>94</u>	113	15	113	13	135	234	300	404	900
0	77	<u>94</u>	<u>113</u>	15	113	13	135	234	300	404	900
0	77	94	<u>15</u>	<u>113</u>	113	13	135	234	300	404	900
0	77	94	15	<u>113</u>	<u>113</u>	13	135	234	300	404	900
0	77	94	15	113	<u>13</u>	<u>113</u>	135	234	300	404	900
0	77	94	15	113	13	<u>113</u>	<u>135</u>	234	300	404	900

Swaps on this pass = 2

Pass number 6

<u>0</u>	<u>77</u>	94	15	113	13	113	135	234	300	404	900
0	<u>77</u>	<u>94</u>	15	113	13	113	135	234	300	404	900
0	77	<u>15</u>	<u>94</u>	113	13	113	135	234	300	404	900
0	77	15	<u>94</u>	<u>113</u>	13	113	135	234	300	404	900
0	77	15	94	<u>13</u>	<u>113</u>	113	135	234	300	404	900
0	77	15	94	13	<u>113</u>	<u>113</u>	135	234	300	404	900

Swaps on this pass = 2

Pass number 7

<u>0</u>	<u>77</u>	15	94	13	113	113	135	234	300	404	900
0	<u>15</u>	<u>77</u>	94	13	113	113	135	234	300	404	900
0	15	<u>77</u>	<u>94</u>	13	113	113	135	234	300	404	900
0	15	77	<u>13</u>	<u>94</u>	113	113	135	234	300	404	900
0	15	77	13	<u>94</u>	<u>113</u>	113	135	234	300	404	900

Swaps on this pass = 2

Pass number 8

<u>0</u>	<u>15</u>	77	13	94	113	113	135	234	300	404	900
0	<u>15</u>	<u>77</u>	13	94	113	113	135	234	300	404	900
0	15	<u>13</u>	<u>77</u>	94	113	113	135	234	300	404	900
0	15	13	<u>77</u>	<u>94</u>	113	113	135	234	300	404	900

Swaps on this pass = 1

Pass number 9

<u>0</u>	<u>15</u>	13	77	94	113	113	135	234	300	404	900
0	<u>13</u>	<u>15</u>	77	94	113	113	135	234	300	404	900
0	13	<u>15</u>	<u>77</u>	94	113	113	135	234	300	404	900

Swaps on this pass = 1

Pass number 10

<u>0</u>	<u>13</u>	15	77	94	113	113	135	234	300	404	900
0	<u>13</u>	<u>15</u>	77	94	113	113	135	234	300	404	900

Swaps on this pass = 0

13.2.2 The Selection Sort

It is not difficult to see that some additional efficiency can be obtained for the bubble sort. It uses many swaps to get the largest item into its correct position on each pass, and some of these are wasted. If the scan is modified so that it simply finds the largest item in the range being scanned and no interchanges are done until the scan is finished, all the intermediate swaps can

be eliminated. Then, when the pass is complete, the largest item can be swapped into the top position for that pass. For instance, starting with the list
 4, 1, 90, 34, 100, 45, 23, 82, 11, 0, 600, 345
 one would find the number 600 to be largest on the first pass (as with the bubble sort,) but do only the swap from its present position to the last spot for the pass. Successive passes would produce the lists:

```

4, 1, 90, 34, 100, 45, 23, 82, 11, 0, 345, 600
4, 1, 90, 34, 100, 45, 23, 82, 11, 0, 345, 600
4, 1, 90, 34, 0, 45, 23, 82, 11, 100, 345, 600
4, 1, 11, 34, 0, 45, 23, 82, 90, 100, 345, 600
4, 1, 11, 34, 0, 45, 23, 82, 90, 100, 345, 600
4, 1, 11, 34, 0, 23, 45, 82, 90, 100, 345, 600
4, 1, 11, 23, 0, 34, 45, 82, 90, 100, 345, 600
4, 1, 11, 0, 23, 34, 45, 82, 90, 100, 345, 600
4, 1, 0, 11, 23, 34, 45, 82, 90, 100, 345, 600
0, 1, 4, 11, 23, 34, 45, 82, 90, 100, 345, 600
0, 1, 4, 11, 23, 34, 45, 82, 90, 100, 345, 600
  
```

In each pass, the number successfully placed is underlined. Here is the code to express these ideas:

```

PROCEDURE SelectSort (VAR source : ARRAY OF CARDINAL;
    lBound, uBound : CARDINAL);
(* Pre: uBound <= HIGH (source) and uBound > lBound (* otherwise, there is nothing to
do *)
THEN
    REPEAT (* start a new pass here *)
        iLarge := uBound;
        (* assume last index is of the largest one *)
        FOR count := uBound TO lBound BY -1
            (* scan from current end of list *)
            DO
                IF source [count] emphasized.
  
```

	234	77	0	113	404	94	900	113	15	300	13	135
Pass number	1											
	234	77	0	113	404	94	135	113	15	300	13	<u>900</u>
Pass number	2											
	234	77	0	113	13	94	135	113	15	300	<u>404</u>	900
Pass number	3											
	234	77	0	113	13	94	135	113	15	<u>300</u>	404	900
Pass number	4											
	15	77	0	113	13	94	135	113	<u>234</u>	300	404	900
Pass number	5											
	15	77	0	113	13	94	113	<u>135</u>	234	300	404	900
Pass number	6											
	15	77	0	113	13	94	<u>113</u>	135	234	300	404	900
Pass number	7											
	15	77	0	94	13	<u>113</u>	113	135	234	300	404	900
Pass number	8											
	15	77	0	13	<u>94</u>	113	113	135	234	300	404	900

Pass number	9											
	15	13	0	<u>77</u>	94	113	113	135	234	300	404	900
Pass number	10											
	0	13	<u>15</u>	77	94	113	113	135	234	300	404	900
Pass number	11											
	0	<u>13</u>	15	77	94	113	113	135	234	300	404	900

[Contents](#)

13.3 Analyzing Sorting Routines

Before going on to other methods of sorting, it is time to consider the performance of the methods used so far. For the sake of simplicity, continue to suppose that the data to be sorted consists of an array of cardinals $[1 \dots n]$. Each of these cardinals in turn must be compared with the others in order to find its correct position in the sorted array. The data must also be moved about in the array, and this can (as in the last section) be achieved by swapping items that are out of place with respect to one another. At each comparison, there may or may not be a swap, so in general, the time taken by a sorting algorithm will depend on the total number of comparisons.

For the bubble sort and selection sort, on the first pass there are $n - 1$, comparisons; on the second, there are $n - 2$; on the third $n - 3$, and so on, until the last pass, where there is one comparison. Adding these up in reverse, the total number of comparisons will be:

$$1 + 2 + 3 + \dots + (n-1)$$

an arithmetic sequence of $n - 1$ terms with common difference 1, for which the sum is (see section 3.8)

$$\text{Sum} = \frac{n(a_1 + a_n)}{2}$$

and substituting $n-1$ for the number of terms yields

$$\text{Sum} = \frac{(n-1)(1+n-1)}{2}$$
$$\frac{n^2 - n}{2}$$

which simplifies to

Since the n^2 term grows much faster than n , it will dominate this expression. It is reasonable to conclude, therefore, that in the two cases considered so far, the sorting algorithms are $O(n^2)$ with respect to comparisons. Indeed, this is the worst possible performance, for no more comparisons than $1 + 2 + 3 + \dots + (n-1)$ should have to be made for any sorting algorithm. The bubble sort may make a swap for every comparison, so in the worst case, its performance is also $O(n^2)$ for the swapping. The selection sort makes at most one swap per pass, so it is $O(n)$, with respect to swapping.

From previous analysis of search routines, $O(n^2)$ behaviour seems to relate to the linear nature in which the comparisons are done. It seems reasonable to expect that some method ought to be available to do the comparisons in a binary fashion so that for each of n items there are only $\log_2 n$ comparisons (and possibly swaps). If so, such an algorithm ought to be $O(n \log_2 n)$ but may be expected to have more complex code.

Sorting algorithms that are $O(n^2)$ are said to be simple sorts. Those that are $O(n \log_2 n)$ are said to be advanced sorts.

In the sections that follow, additional examples of both types of sorting algorithm will be explored. Simple sorts (especially the two in the last section) have the easiest code to understand, work well on

small amounts of data, and do not take long to code, but even they can be fine-tuned to increase performance in many cases. The advanced sorts are sometimes more difficult to understand and code, but for very large data sets, it is better to take few times through a complicated loop than far more times through a simple one.

It is also worth noting before going on that the intention of most array sorting is to sort the array items *in place*, that is, to compare and trade (or move) them about while only one (or a small number) are held in temporary variables. A few methods use two arrays, one called the *source*, and the other the *destination*. However, the latter methods require almost twice as much storage space as the *in place* methods, and are not as useful for some applications. It seems reasonable to wonder, however, whether sacrificing storage efficiency might, in some cases, allow a more efficient sorting algorithm to be written. Perhaps there are even circumstances where the very nature of the data requires additional storage space to sort it at all. While this aspect of sorting will not be taken up further in this chapter, a few such sorts will be examined later in the book.

[Contents](#)

13.4 Inserting Methods

The sorting methods in this section work in a different manner from those described in [section 13.2](#). Rather than compare adjacent items, or search for the largest one in some range, they operate on the premise that some portion of the array is already sorted, and some additional item must be inserted into the correct place in that portion. The methods of finding the proper place for an item therefore become important in this discussion. The intention of this section is to push the ideas of finding and inserting to their limits, analyze them fully, and explore every possible avenue for clever improvements. Some of the resulting code may seem to the reader to be too complex to be worth the trouble, and if so, invitations are extended in the exercises at the end of the chapter to check on the actual performance of the code produced.

To start with, consider the problem of sorting data into an array as it arrives from some source (the source could be another part of the same array, but this is not important at the moment.) At any given step, the portion of the array into which the data is being placed is already sorted, and it is necessary to find the correct place for the new item and put it there. If the existing array is sorted from lowest to highest, it is necessary to find the first position in the array with a value greater than that of the item being inserted. If there are none, the correct position is after the last item being examined. The new item will go at this place, so it is necessary to move everything from that index and beyond to the next greater index position and then copy the new value into its correct spot.

Example:

Suppose that at some point in the process the array contained:

A [0] = 0, A [1] = 5, A [2] = 8, A [3] = 12, A [4] = 15

0	1	2	3	4	5	6	7	8	9
0	5	8	12	15					

7

and 7 is the value of the new item to be inserted. The suggested algorithm searches through the items A [0 .. i] until it finds the first one with a value greater than 7 (which in this case is the value 8). The index of that item ([2]) is the place to put the new item. First, though, A [4], A [3], and A [2] must be moved along to the next higher indexed position, and in that order. Do you see why?

0	1	2	3	4	5	6	7	8	9
0	5	8	12	15					

7

↑ insert location

When these steps are complete, the array looks like this:

0	1	2	3	4	5	6	7	8	9
0	5	7	8	12	15				

If the number 17 were now to be inserted, an examination of the items already active would reveal that all were less than the new one, so it would go in the last array position (six) and no moves would have to be made.

This informal examination of the situation then leads to a more formal expression of the problem and its solution:

Problem:

To sort a cardinal *new value* into an existing array [0 .. n].

Pseudocode:

```
Sort To Array
  Find Position
    set pos to 0
```

```

    while (pos < number in use) and (array item [pos] <= new value)
        increment pos
    end while
Make Room
    if pos < number in use then
        for counter = last position used to pos by -1
            set array item [counter + 1] to array item [counter]
        end for
    end if
Insert New Item
    set array item [pos] to value

```

Discussion:

The insertion procedure requires a variable parameter to hold the array being altered and a value parameter to pass it the new number to insert. It also requires the index of the highest assigned element, so that it can both stop at the right place when doing comparisons, and start at the right one when doing moves. Because it is important to test such code carefully, the insert procedure has been enclosed in a module that serves as a test harness.

Code:

```

MODULE TestInsert;

(* By R. Sutcliffe  Last Modified 1995 04 19 *)

FROM STextIO IMPORT
    WriteLn ;
FROM SWholeIO IMPORT
    WriteCard;

PROCEDURE Insert (item : CARDINAL; (* # to insert *)
    VAR source : ARRAY OF CARDINAL; (* array to put it in *)
    lBound, (* position from zero to start using places *)
    uBound : CARDINAL (* position from zero of last useable place *) );

(* inserts the item in the sorted array from internal position lBound to uBound
Pre: the actual array sorted for the first uBound-1 positions and is large enough to
insert an item and do whatever moves are required
Post: the item is inserted at the first position where the old array item is greater,
or at internal position uBound if it is larger than them all  Then, items from the
insert position through to position uBound-1 are moved along along by one *)

VAR
    pos,      (* will contain where to put it *)
    count : CARDINAL;      (* loop counter *)

BEGIN
    (* Find Position *)
    pos := lBound;
    WHILE (pos < uBound) AND (source [pos] <= item )
    DO
        INC (pos)

```

```

END;      (* while *)

(* Make Room *)
IF pos < uBound (* otherwise goes at the end; no moves needed *)
  THEN (* this loop also skipped if howMany is zero *)
    FOR count := uBound - 1 TO pos BY -1 (* start at top end *)
      DO
        source [count + 1] := source [count];
      END;      (* for *)
    END; (* if *)

(* Insert New Item *)
source [pos] := item;

END Insert;

PROCEDURE PrintIt (theArray : ARRAY OF CARDINAL; numToPrint : CARDINAL); (* print the
whole array *)

VAR
  count : CARDINAL;

BEGIN
FOR count := 0 TO numToPrint - 1
  DO
    WriteCard (theArray [count], 5);
  END;
  WriteLn;
END PrintIt;

VAR (* for main program *)
  theStuff : ARRAY [1 .. 10] OF CARDINAL;

BEGIN
  theStuff [1] := 34; theStuff [2] := 37; theStuff [3] := 44;
  theStuff [4] := 134; theStuff [5] := 744; theStuff [6] := 824;
  theStuff [7] := 937; theStuff [8] := 984; theStuff [9] := 1039;
  PrintIt (theStuff, 9); (* the original *)
  Insert (5, theStuff, 0, 0); (* kills first item; no moves *)
  PrintIt (theStuff, 9);
  Insert (3500, theStuff, 7, 9); (* appends new item at the end *)
  PrintIt (theStuff, 10); (* so print one more to see it *)
  Insert (0, theStuff, 0, 1); (* insert before first, move one *)
  PrintIt (theStuff, 10);
  Insert (400, theStuff, 0, 9); (* find place in overall list *)
  PrintIt (theStuff, 10);
  Insert (7, theStuff, 0, 1); (* in after first, none moved *)
  PrintIt (theStuff, 10);
END TestInsert.

```

Output:

```

34    37    44   134   744   824   937   984 1039

```


5	37	44	134	744	824	937	984	1039	
5	37	44	134	744	824	937	984	1039	3500
0	5	44	134	744	824	937	984	1039	3500
0	5	44	134	400	744	824	937	984	1039
0	7	44	134	400	744	824	937	984	1039

Observe that this procedure can be used even to insert the first item in the array, because if it starts with no active items, then the value of *lastPosNum* is zero and neither loop is entered, so the only statement that would be executed is *source [pos] := item* with *pos* equalling zero, which is exactly what is desired.

It should also be clear by now that the searching method presented in the above code is not the most efficient, for it is linear, requiring the examination of each element in turn until the correct location is discovered. The insert procedure can be re-cast so that it too uses the more efficient binary search discussed in the last section.

```

PROCEDURE InsertB (item : CARDINAL; (* # to insert *)
    VAR source : ARRAY OF CARDINAL; (* array to put it in *)
    lBound, (* position from zero to start using places *)
    uBound : CARDINAL (* position from zero of last useable place *) );

(* inserts the item in the sorted array from internal position lBound to uBound
Pre: the actual array sorted for the first uBound-1 positions and is large enough to
insert an item and do whatever moves are required
Post: the item is inserted at the first position where the old array item is greater,
or at internal position uBound if it is larger than them all Then, items from the
insert position through to position uBound-1 are moved along along by one *)

VAR
    pos, (* will contain where to put it *)
    count : CARDINAL; (* loop counter *)
    bottom, top : CARDINAL;

BEGIN
    (* Find Position *)
    bottom := lBound;
    IF uBound # lBound
    THEN (* else using only one item & also must avoid negatives *)
        top := uBound - 1;
        WHILE (bottom < top)
        DO
            pos := (top + bottom + 1) DIV 2;
            IF item < source [pos]
            THEN (* ignore top half *)
                top := pos - 1;
            ELSE
                bottom := pos (* ignore bottom half *)
            END; (* if *)
        END (* while *);
    END;
    pos := bottom;
    (* the insert position is just before, or just after this position and one
comparison has yet to be made *)
    IF (item < uBound)
    THEN (* goes after *)
        INC (pos);

```

```

    END; (* if *)

(* Make Room *)
IF pos < uBound (* otherwise it goes at the end; no moves needed *)
    THEN (* this loop also skipped if uBound is zero *)
        FOR count := uBound - 1 TO pos BY -1 (* start at high end *)
            DO
                source [count + 1] := source [count];
            END; (* for *)
        END; (* if *)

(* Insert New Item *)
source [pos] := item;

END InsertB;

```

This code has the advantage of being much faster at execution time; it has the disadvantage of requiring somewhat more work to plan, write, and debug. Notice the differences between this and the earlier binary find routine. Here, the concern is to locate the position to do the insert. There may already be items in the array with the same value as the one being inserted. If so, this routine places the new item after the last of these. When run as an enclosure to the same program as used to test the previous insert routine, the results were identical.

Note that, as before, the procedure *Strings.Compare*, or an equivalent user-written one, could be used in conjunction with the ideas presented in this section to input and sort, say, a teacher's class list by last name. However, since fewer than a hundred students would likely fall to a single teacher for marking in a given class, it is possible that a linear search and insert would be adequate.

13.4.1 The Insert Sort

So far, all that has accomplished is the insertion of single items into an already sorted array. While this is a little more than just finding out if it was already there, it is short of the goal of arriving at that sorted condition in the first place--or is it?

In fact, the insert method can be used to sort an array that starts out initially in random order. The sort is planned like this:

1. Assume that the first element of the source array constitutes a sorted list. This list one has one thing in it, so it is sorted in a trivial sense.
2. Start with the next unsorted item *source [count]* (initially *count* would equal one.)
3. Use the procedure above to insert this next item into the sorted part of the array (which ends at the position *count - 1*) at the correct point.
4. The sorted portion of the array now has two items. Repeat the above process with a sorted part that grows by one item at each step until the sorted section is the entire array.

Here is what the procedure to sort the array would then look like:

```

PROCEDURE InsertSort (VAR source : ARRAY OF CARDINAL;  lBound, uBound : CARDINAL);

VAR
    count : CARDINAL;
BEGIN
    FOR count := lBound TO uBound
        (* count on indices from second to last for insertions *)
        DO
            InsertB (source [count], source, lBound, count);
        END;
    END InsertSort;

```

This sort was tested with a similar test harness to that in the previous module and the data:

```
theStuff [1] := 113; theStuff [2] := 77; theStuff [3] := 0;
theStuff [4] := 50; theStuff [5] := 113; theStuff [6] := 114;
theStuff [7] := 900; theStuff [8] := 113; theStuff [9] := 15;
theStuff [10] := 300; theStuff [11] := 13; theStuff [12] := 135;
theStuff [13] := 1;
```

The procedure was invoked by:

```
theSort (theStuff,0,13)
```

was made, and a line

```
PrintIt (theStuff,13); WriteLn;
```

was placed on the last line of the FOR loop in the procedure *InsertSort* to follow the progress. The output is given below with the portion of the list that is sorted (within itself) underlined. The identical output was obtained using *Insert* in *InsertSort* rather than *InsertB*.

<u>113</u>	77	0	50	113	114	900	113	15	300	13	135	1
<u>77</u>	<u>113</u>	0	50	113	114	900	113	15	300	13	135	1
<u>0</u>	<u>77</u>	<u>113</u>	50	113	114	900	113	15	300	13	135	1
<u>0</u>	<u>50</u>	<u>77</u>	<u>113</u>	113	114	900	113	15	300	13	135	1
<u>0</u>	<u>50</u>	<u>77</u>	<u>113</u>	<u>113</u>	114	900	113	15	300	13	135	1
<u>0</u>	<u>50</u>	<u>77</u>	<u>113</u>	<u>113</u>	<u>114</u>	900	113	15	300	13	135	1
<u>0</u>	<u>50</u>	<u>77</u>	<u>113</u>	<u>113</u>	<u>114</u>	<u>900</u>	113	15	300	13	135	1
<u>0</u>	<u>50</u>	<u>77</u>	<u>113</u>	<u>113</u>	<u>113</u>	<u>114</u>	<u>900</u>	15	300	13	135	1
<u>0</u>	<u>15</u>	<u>50</u>	<u>77</u>	<u>113</u>	<u>113</u>	<u>113</u>	<u>114</u>	<u>900</u>	300	13	135	1
<u>0</u>	<u>15</u>	<u>50</u>	<u>77</u>	<u>113</u>	<u>113</u>	<u>113</u>	<u>114</u>	<u>300</u>	<u>900</u>	13	135	1
<u>0</u>	<u>13</u>	<u>15</u>	<u>50</u>	<u>77</u>	<u>113</u>	<u>113</u>	<u>113</u>	<u>114</u>	<u>300</u>	<u>900</u>	135	1
<u>0</u>	<u>13</u>	<u>15</u>	<u>50</u>	<u>77</u>	<u>113</u>	<u>113</u>	<u>113</u>	<u>114</u>	<u>135</u>	<u>300</u>	<u>900</u>	1
<u>0</u>	<u>1</u>	<u>13</u>	<u>15</u>	<u>50</u>	<u>77</u>	<u>113</u>	<u>113</u>	<u>113</u>	<u>114</u>	<u>135</u>	<u>300</u>	<u>900</u>

If one wished instead to do the entire job in a single procedure, (cutting down on the overhead of procedure calls,) the plan could be modified as follows:

1. Assign the value of the item to be sorted to a temporary variable *temp*. Its current position in the array is now expendible.
2. Proceed through the sorted part of list, comparing *temp* to the current item at each step. (This is linear.)
3. As this is done, move items that are greater than *temp* one position closer to the high index item, filling the space made available at the last step.
4. Stop when the correct position for *temp* is reached.
5. Put *temp* in this position.
6. Continue on to the next index (the next unsorted item).
7. Repeat until the last indexed item is processed.

Here is the code for this version:

```
PROCEDURE InsertSort (VAR source : ARRAY OF CARDINAL; lBound, uBound : CARDINAL);
VAR
    count, moveCount, temp : CARDINAL;
BEGIN
    IF uBound > lBound
```


10	99	151	121	2	18	96	753	13	472	111
58	390									
26	32	100	58	21	190	56	345	67	88	32
102										

	The array 13-sorted:									
10	1	47	58	2	18	35	342	13	3	32
27	15									
23	32	100	121	21	76	56	345	67	88	82
58	390									
26	99	151	134	89	190	96	753	987	472	111
102										

Notice that "k-sorted" is the same as "sorted" if and only if $k = 1$. However, if one k-sorts an array for a large value of k (moving elements long distances first) and then for progressively smaller values of k (where the sort will become easier as fewer moves are required) and finally ending with a one-sort (which is a straight insertion pass), then fewer moves in total need to be made.

A sequence of k-sorts ending in $k = 1$ is called a Shell sort.

	The 13- sorted array arranged for 4-sorting:			
10	1	47	58	
2	18	35	342	
13	3	32	27	
15	23	32	100	
121	21	76	56	
345	67	88	82	
58	390	26	99	
151	134	89	190	
96	753	987	472	
111	102			

	The array also 4-sorted:			
2	1	26	27	
10	3	32	56	
13	18	32	58	
15	21	35	82	
58	23	47	99	
96	67	76	100	
111	102	88	190	
121	134	89	342	
151	390	987	472	
345	753			

Observe how the smaller items are now clustered toward the beginning, and the larger ones toward the end. An insertion sort (a one-sort) will now have far less work to do to complete the task than if the array had not been 13-sorted and 4-sorted first. The following table illustrates how a collection of characters is sorted, also using the k-sequence 13, 4, 1. This time, the array has not been written in rows of 13 and in rows of four, so it may be a little more difficult to verify the sort by sight.

#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	H	E	R	E	A	R	E	S	O	M	E	L	E	T	T	E	R	S	T	O	S	O	R	T

```

H E E E A R E S O M E L E T T R R S T O S O R T
A E E E E M E L H O E O O R R R R S T S S T T T
A E E E E E E H L M O O O R R R R S S S T T T T

```

The first pass organizes the sub-lists at positions 1 and 14, 2 and 15, 3 and 16, etc. The second organizes the sub-lists at positions 1, 5, 9, 13, 17, and 21; at 2, 6, 10, 14, 18, and 22, and so on. The plain insertion sort conducted at the last step does not have to move any one object very far, so there are fewer reassignments overall.

Various k-sequences have been examined and used for the Shell sort, and, as might be expected, some work better than others. For simplicity, some people generate the k-sequence by starting at *lastPosNum DIV 2* and thereafter taking $k = k \text{ DIV } 2$.

This is fairly satisfactory if *lastPosNum* equals, say, 38. Then, the sequence of values for k is 19, 9, 4, 2, and 1. "So what?" one might say. Observe that a list that is 2-sorted is necessarily also 4-sorted as well, for if a sub-sequence of every second item is in order, the sequence of every fourth one certainly is. Some of the comparisons are therefore wasted, because there is no need to do a 4-sort if there is later going to be a 2-sort as well. In the worst case, if the position of the last item in the source array happens to be a power of 2, so will every number in the k-sequence, and the sort will rapidly lose efficiency. In general, if a j-sort is to be conducted at a later stage of a Shell sort, there is no point in doing an $n*j$ -sort for any n at an earlier stage.

To maximize the effectiveness of a Shell sort, it is even better if there are no factors at all common among the numbers used for the values of k in the k-sequence. That is, the terms of the k-sequence should be relatively prime. (The greatest common divisor of any pair should equal one.)

A sequence that works fairly well is the one generated by the formula $k(i) = 3 * k(i-1) + 1$, namely 1093, 364, 121, 40, 13, 4, 1, although it does contain a subsequence of multiples of 4. The code would now look like this:

```

PROCEDURE ShellSort1 (VAR source: ARRAY OF CARDINAL; lBound, uBound : CARDINAL);
VAR
    k, listCount, count, moveCount, temp : CARDINAL;

BEGIN
    k := 1;
    REPEAT    (* get first "k" in sequence *)
        k := 3 * k + 1
    UNTIL k > uBound - lBound;

    REPEAT
        k := k DIV 3;    (* work backwards on same sequence *)
        FOR listCount := lBound TO lBound + k - 1
            (* Start as many k-lists as possible. *)
            DO
                count := listCount + k;
                WHILE count <= uBound
                    DO (* each k-sort starts here *)
                        temp := source [count];    (* set aside unsorted one *)
                        moveCount := count;
                        WHILE (moveCount >= lBound + k) AND (source [moveCount - k] > temp)
                            DO
                                source [moveCount] := source [moveCount - k];
                                (* move along to make room *)
                                DEC (moveCount, k)    (* count backward in k-list *)
                            END;
                        source [moveCount] := temp;    (* insert new item *)
                        INC (count, k);
                    END;    (* while count *)
            END;    (* for listCount *)
        UNTIL k = 1
    END ShellSort1;

```

There are two additional observations that can be made at this point, some important observations and some further refinements based on these.

First, it is useful to keep the k-sequence on hand instead of generating it with a formula. One must ensure that the sequence runs high enough to sort the largest array, but this method allows for some testing that will enable one to find an optimum k-sequence by trial and error.

Second, the k-sorts in this section were accomplished by a linear search method through each of the k lists. That can be modified to utilize a binary search, even though this is more difficult to write. The key is formulating the search correctly. It is easy to examine only every kth element of a list, but the computation of the middle index takes a little thought. Suppose one has a sequence of indices in the array starting at a₀ and separated by k positions. These indices themselves form an arithmetic sequence, a₁, a₂, a₃, a₄, ... a_i ... whose terms are:

$$a_1, (a_1 + k), (a_1 + 2k), (a_1 + 3k), \dots (a_1 + (i - 1) k)$$

Note the formula for the ith term

$$a_i = (a_1 + (i - 1) k)$$

To get the index of the middle one of these, first note that the number of k's added is

$$i - 1 = (a_i - a_1) \text{ DIV } k, \text{ or } i = 1 + (a_i - a_1) \text{ DIV } k$$

Half of this number is

$$m = i \text{ DIV } 2 = (1 + (a_i - a_1) \text{ DIV } k) \text{ DIV } 2$$

The actual term in the sequence of indices when a total of m k's have been added to the first index is:

$$t_{m+1} = a_1 + mk = a_1 + ((1 + (a_i - a_1) \text{ DIV } k) \text{ DIV } 2) k$$

To illustrate, suppose one was examining the 3-list of terms starting at index 4, that is, the terms t₄, t₇, t₁₀, t₁₃, t₁₆, t₁₉, t₂₂, This list may or may not end at an index that can be expressed in the form 4 + 3j. Looking at the sequence of indices 4, 7, 10, 13, 16, 19, 22, ... and various ending points, the above formula gives for the middle index:

top index	middle
22	13
16	10
21	13

all of which are satisfactory. In the event that k = 1 (the last k-sort done,) the formula above simplifies to:

$$t_{m+1} = a_1 + ((1 + (a_i + a_1) \text{ DIV } 1) \text{ DIV } 2) k$$

or

$$t_{m+1} = a_1 + 1 + (a_i - a_1) \text{ DIV } 2 = a_1 + (1 + (a_i - a_1)) \text{ DIV } 2$$

so that

$$2t_{m+1} = 2a_1 + 1 + a_i + a_1$$

or

$$t_{m+1} = (a_1 + a_i + 1) \text{ DIV } 2$$

which gives the same result as (a₁ + a_i) DIV 2 for an odd number of terms. If there are an even number of terms, there is no true middle, and either of two terms can be selected with equal effectiveness. (a₁ + a_i + 1) DIV 2 gives the one of these two with the higher index, and (a₁ + a_i) DIV 2 gives the one with the lower. Since the latter also has one less calculation, using it will save time.

With suitable changes in notation, these formulas can be used in the find portion of the sorting routine.

With these points in mind, here is another version of the Shell sort. This one conducts the binary search in a slightly different manner than did previous versions.

```
VAR
  kSeq : ARRAY [1 .. 10] OF CARDINAL;
  (* Permits use of a predefined sequence for k-step values. *)
```

```
PROCEDURE NewPos (low, high, step : CARDINAL) : CARDINAL;
```

```
BEGIN  
  IF step = 1  
    THEN  
      RETURN (low + high + 1) DIV 2  
    ELSIF high = low THEN  
      RETURN high  
    ELSE  
      RETURN low + ((1 + (high - low) DIV step) DIV 2) * step  
      (* calculate "middle position" as described above *)  
    END;  
END NewPos;
```

```
PROCEDURE KFindB (item : CARDINAL;  
  VAR source : ARRAY OF CARDINAL;  
  first, last, kStep : CARDINAL) : CARDINAL;
```

(* This procedure is a modification of the earlier binary search. It searches through a sub list of the supplied array considering as consecutive those positions separated by "kStep" using a binary method and returning the index at which the new item must be inserted. *)

```
VAR  
  bottom, top, pos : CARDINAL;
```

```
BEGIN  
  bottom := first;  
  top := last;  
  IF kStep = 1  
    THEN  
      DEC (top)  
    END;  
  WHILE (bottom < top)  
    DO  
      pos := NewPos (bottom, top, kStep);  
      IF item < source [pos]  
        THEN (* discard top half *)  
          top := pos - kStep;  
        ELSE  
          bottom := pos (* discard bottom half *)  
        END; (* if *)  
      END; (* while *)  
  pos := bottom;  
  (* the insert position is just before, or just after this position and one comparison  
  has yet to be made *)  
  
  IF (item >= source [pos]) AND (pos < last)  
    THEN (* goes after *)  
      RETURN (pos + kStep);  
    ELSE  
      RETURN (pos)  
    END; (* if *)
```



```
END KFindB;
```

```
PROCEDURE KInsert (VAR source: ARRAY OF CARDINAL;  
    toIndex, fromIndex, kStep : CARDINAL);
```

```
(* This procedure takes source [fromIndex] and inserts it at source [atIndex] moving  
things along in kStep increments to fill the gap. *)
```

```
VAR
```

```
    temp, count : CARDINAL;  
    swap : BOOLEAN;
```

```
BEGIN
```

```
    temp := source [fromIndex];  
    count := fromIndex;
```

```
    WHILE count >= (toIndex + kStep)  
        DO  
            source [count] := source [count - kStep];  
            DEC (count, kStep)  
        END;  
    source [toIndex] := temp;    (* actual insert *)  
END KInsert;
```

```
(* finally, here is the actual sort procedure that the outside world uses *)
```

```
PROCEDURE ShellSort (VAR source: ARRAY OF CARDINAL;  
    lBound, uBound : CARDINAL);
```

```
VAR
```

```
    kIndex, k, listCount, count, putIndex, temp : CARDINAL;
```

```
(* compute an appropriate value of k from the sequence initialized below *)
```

```
BEGIN
```

```
    kIndex := 0;  
    REPEAT  
        INC (kIndex)  
    UNTIL kSeq [kIndex] > (uBound DIV 2);
```

```
    REPEAT
```

```
        DEC (kIndex);  
        k := kSeq [kIndex];  
        FOR listCount := lBound TO lBound + k - 1  
            (* Start as many k-lists as possible. *)  
            DO  
                count := listCount + k;  
                WHILE count <= uBound  
                    DO    (* each k-sort starts here *)  
                        putIndex := KFindB (source [count], source, listCount, count, k);  
                        IF putIndex # count  
                            THEN  
                                KInsert (source, putIndex, count, k);  
                            END;  
                        INC (count, k)  
                    END;  
            END;    (* while count *)
```

```

    END  (* for listCount *)
UNTIL kIndex = 1
END ShellSort;

```

This was encapsulated in a library module, with the following initialization of the k-sequence values:

```

BEGIN  (* initialization section *)
  kSeq [1] := 1;
  kSeq [2] := 4;
  kSeq [3] := 11;
  kSeq [4] := 23;
  kSeq [5] := 53;
  kSeq [6] := 111;
  kSeq [7] := 223;
  kSeq [8] := 451;
  kSeq [9] := 1003;
  kSeq [10] := 2029;

END Sort.

```

and the following modified version of *PrintIt*:

```

PROCEDURE PrintIt (theArray : ARRAY OF CARDINAL; numToPrint, numPerLine, flen :
CARDINAL);

VAR
  count : CARDINAL;

BEGIN
  FOR count := 0 TO numToPrint - 1
    DO
      WriteCard (theArray [count], flen);
      IF (count + 1) MOD numPerLine = 0
        THEN
          WriteLn;
        END;
      END;
    END PrintIt;

```

A simple test harness to check the library module follows:

```

MODULE TestShellSort;

(*  By R. Sutcliffe
   To illustrate shell sorting
   last modified 1995 05 02 *)

FROM STextIO IMPORT
  WriteChar, WriteString, WriteLn;

FROM Sorts IMPORT
  PrintIt, ShellSort;

```

```

VAR
  theStuff : ARRAY [1 .. 13] OF CARDINAL;

BEGIN
  theStuff [1] := 113; theStuff [2] := 77; theStuff [3] := 0;
  theStuff [4] := 50; theStuff [5] := 113; theStuff [6] := 114;
  theStuff [7] := 900; theStuff [8] := 113; theStuff [9] := 15;
  theStuff [10] := 300; theStuff [11] := 13; theStuff [12] := 135;
  theStuff [13] := 1;

  PrintIt (theStuff,13,13,5); WriteLn;
  ShellSort (theStuff,2,2); (* sort one; does nothing *)
  PrintIt (theStuff,13,13,5); WriteLn;
  ShellSort (theStuff,1,2); (* sort two *)
  PrintIt (theStuff,13,13,5); WriteLn;
  ShellSort (theStuff,2,9); (* all but first two, last three *)
  PrintIt (theStuff,13,13,5); WriteLn;
  ShellSort (theStuff,0,12); (* whole thing *)
  PrintIt (theStuff,13,13,5); WriteLn;

END TestShellSort.

```

This produced the following output:

113	77	0	50	113	114	900	113	15	300	13	135	1
113	77	0	50	113	114	900	113	15	300	13	135	1
113	0	77	50	113	114	900	113	15	300	13	135	1
113	0	15	50	77	113	113	114	300	900	13	135	1
0	1	13	15	50	77	113	113	113	114	135	300	900

At this point, the reader should at least be suspicious that the extensive modifications to the routine to achieve a binary search on the k-sequences may not be worth the trouble. Indeed, they may introduce so much arithmetic that they actually degrade the performance of the sort. Whether this is so, and to what extent, has been left to the exercises.

The performance of a Shell sort in comparison with the other simple sorts is difficult to analyze. It depends on the k-sequence employed, and no general solution to the problem of finding the optimum k-sequence, or analyzing performance for a given k-sequence is known. For the k-sequence 1, 4, 13, 40, 121, ... a Shell sort never does more than $N^{3/2}$ comparisons, and this limit also holds for other k-sequences. One conjecture is that a Shell sort is $O(n (\log_2 n)^2)$. Various sequences can be tested by having them stored in a separate file, and making the Shell sort obtain them from there. The one employed above was obtained with a little trial and error and found to be reasonably effective. Some of the literature on the Shell sort indicates that starting with the number of items and dividing by a *shrink factor* of about 1.7 for the first and each subsequent pass produces optimum results in most cases. For instance, if there were 100 items in the array to be sorted, k would first be initialized to 100, and using

```
k := TRUNC (FLOAT (k) / 1.7);
```

or, better yet (to avoid floating point operations)

```
k := 10 * k / 17
```

thereafter, at the beginning of the sorting loop, the following k-sequence (in order of use) is obtained:

58, 34, 20, 12, 7, 4, 2, 1.

However, in view of the comments above, there are clearly inefficiencies in this sequence, and some further fine tuning perhaps ought to be done. Again, the student is invited to experiment.

13.4.3 The Combsort

The Shell sort rearranges all the k-lists that it can form with a given k. Another approach to the comparison of elements that are relatively far apart in the list is to sort only the k-list starting at position zero for each k. One might immediately suspect that this approach would require somewhat more steps in the sequence for optimum performance, and this turns out to be the case.

A sequence of sorts of only the k-lists starting at position zero for k decreasing by a constant factor is called a comb sort.

The comb sort was first published by Richard Box and Stephen Lacey in the *April 1991* issue of Byte magazine. They found that using a sequence for the gaps that decreased by a shrink factor of 1.3 gave the best results. Trial and error produced the refinement that an eventual gap size of 11, (rather than 9 or 10) gave a more efficient sequence of gaps below that point. Their published sort was based on a modification of a bubble sort routine; here it is presented using an insert technique:

```
PROCEDURE CombSort (VAR source: ARRAY OF CARDINAL; lBound, uBound : CARDINAL);
```

```
VAR  
    gap, count, moveCount, temp : CARDINAL;
```

```
BEGIN  
    gap := uBound - lBound;  
    IF gap = 0 (* check hostile cases outside loop *)  
        (* note: can't hit zero any other way *)  
    THEN  
        RETURN (* nothing to sort *)  
    ELSIF gap = 1 THEN (* special case sorting two *)  
        IF source [lBound] > source [uBound]  
            THEN  
                Swap (source [lBound], source [uBound])  
            END; (* if source *)  
        RETURN (* in either case *)  
    END; (* if gap *)
```

```
REPEAT  
    gap := gap * 10 / 13;  
    IF (gap = 9) OR (gap = 10)  
        THEN  
            gap := 11  
        END;  
  
    count := lBound + gap;  
    WHILE count <= uBound  
        DO  
            temp := source [count]; (* set aside unsorted one *)  
            moveCount := count;  
            WHILE (moveCount >= gap) AND (source [moveCount - gap] > temp)  
                DO  
                    source [moveCount] := source [moveCount - gap];  
                    (* move along to make room *)
```

```

        DEC (moveCount, gap)    (* count backward in list *)
    END;
    source [moveCount] := temp;  (* insert new item *)
    INC (count, gap);
    END;    (* while count *)
UNTIL gap = 1
END CombSort;

```

The difference between operation of the Shell sort and the comb sort can be illustrated by examining the following table, constructed from successive passes of the two sorts with the array printed out at each pass. The underlined items are the ones compared and sorted.

The Combsort												
Position	0	1	2	3	4	5	6	7	8	9	10	11
Initial	234	77	0	113	404	94	900	113	15	300	13	135
gap=11:	<u>135</u>	77	0	113	404	94	900	113	15	300	13	<u>234</u>
gap= 8:	<u>15</u>	77	0	113	404	94	900	113	135	<u>300</u>	13	234
gap= 6:	<u>15</u>	77	0	113	404	94	900	<u>113</u>	135	300	13	234
gap= 4:	<u>15</u>	77	0	113	135	<u>94</u>	900	113	404	<u>300</u>	13	234
gap= 3:	<u>15</u>	77	0	113	<u>135</u>	94	300	<u>113</u>	404	900	<u>13</u>	234
gap= 2:	<u>0</u>	77	<u>13</u>	113	<u>15</u>	94	<u>135</u>	113	<u>300</u>	900	<u>404</u>	234
gap= 1:	<u>0</u>	<u>13</u>	<u>15</u>	<u>77</u>	<u>94</u>	<u>113</u>	<u>113</u>	<u>135</u>	<u>234</u>	<u>300</u>	<u>404</u>	<u>900</u>

The Shell sort												
Position	0	1	2	3	4	5	6	7	8	9	10	11
Initial	234	77	0	113	404	94	900	113	15	300	13	135
k = 4:	<u>15</u>	77	0	113	<u>234</u>	94	900	113	<u>404</u>	300	13	135
k = 4:	15	<u>77</u>	0	113	234	<u>94</u>	900	113	404	<u>300</u>	13	135
k = 4:	15	77	<u>0</u>	113	234	94	<u>13</u>	113	404	300	<u>900</u>	135
k = 4:	15	77	0	<u>113</u>	234	94	13	<u>113</u>	404	300	900	<u>135</u>
k = 1:	<u>0</u>	<u>13</u>	<u>15</u>	<u>77</u>	<u>94</u>	<u>113</u>	<u>113</u>	<u>135</u>	<u>234</u>	<u>300</u>	<u>404</u>	<u>900</u>

The Shell sort makes more sub-lists and does more comparisons in each one, while the comb sort makes more passes, but only constructs a single sub-list at each pass. Box and Lacey claim that their comb sort achieved the best balance of the two between short times and predictability, as it sorts random lists and previously sorted lists with about the same efficiency. They also claim that actual trials show the comb sort to approximate $O(n \log_2 n)$ as n gets large.

13.5 Advanced Sorting of Arrays

It is now time to step back and consider what has been accomplished in the last section. It began with a simple insert sort and, in an effort to reduce the number of swaps, the complexity of the code was increased considerably. In fact, the last version of the Shell sort has so many computations just to decide what to do the comparisons on that it surely loses efficiency on one hand, even as it purportedly gains it on the other.

If one abandon the insert idea altogether, and looks for some entirely different approach, can one achieve both simplicity in coding and also minimize the number of comparisons and exchanges? The answer is "yes", but the cost paid is in subtlety--that is, the algorithms become harder to understand.

13.5.1 The QuickSort

The method presented here is sometimes called the *position-and-exchange*, or *divide-and-conquer* sort, because it depends on carving the original collection of objects into two portions, with the item separating the two parts being, in fact, in its correct final position. This contrasts with the bubble sort and selection sort, which both place the largest item into its place on a given pass, and with all types of insert source, which do not necessarily place any item into its final position on a given pass. One proceeds something like this:

1. Call the first item in the portion list required to be sorted the pivot.
2. Find the correct final position for the pivot and put it there.
3. In doing so, ensure that the items placed before the pivot are all less than it and the ones placed after the pivot are all greater than it.
4. This partitions the list into a left sub-list and a right sub-list.
5. Repeat steps 1-6 recursively for the left sub-list provided it has more than one element.
6. Repeat steps 1-6 recursively for the right sub-list provided it has more than one element.

Each time a sub-list is considered, exactly one item is placed in its correct final position. According to the earlier analysis of the binary search pattern, this leads one to conclude that $\log_2 n$ lists will be examined in all. This conclusion may be somewhat optimistic, however, for it assumes that each step will partition the list exactly in half--an outcome that, though desirable, will not always be achieved. Still, this partitioning into two equal portions should approximate the behaviour of this algorithm on average, so it is worthwhile to proceed. An attempt will be made to fine tune on this point later.

The method of sorting by partitioning a list into two sub-lists around some pivot, (where the items in the left sub-list are all less than the pivot, and those in the right sub-list are all greater than the pivot), and then recursively sorting the left and right sub-lists in the same manner is called a quicksort.

The key to realizing steps two and three in the sequence above is to start with the next item after the pivot, and counting forward, discover the first one that is greater than the pivot. (i.e. it really belongs to the right sub-list). One then commences examining items from the last in the current list backwards to find the first one that is less than the pivot. (i.e. it really belongs to the left sub-list). Since these two items are both now in the wrong sub-list, swap them. Then pick up the count again where it left off and continue, gradually working toward a collision between the two counters. To see this in operation, suppose one begins with the list:

16, 7, 9, 44, 2, 18, 8, 53, 1, 5, 17,

Isolating item #1 (16) as the pivot, scan from #2 and discover that item #4 (44) is the first one greater than 16.

Suspending the count here, begin with item #11 (17) and work backwards. Item #10 (5) is the first one less than 16, so swap it with item #4 to get:

16 7, 9, 5, 2, 18, 8, 53, 1, 44, 17,

The underscore marks the two that were swapped and also the places where counting was suspended in the forward and backward scans. Continuing, the next two that must be swapped because they are in the wrong sub-list are item #6 (18) and item #9 (1) so the list becomes:

16 7, 9, 5, 2, 18, 8, 53, 1, 44, 17,

The next item found that is greater than 16 is item #8 (53), but at this point, the count from the right passes with the one from the left (at the arrow) without finding another item less than the pivot value, so there are no more swaps to make in this pass, and the scan has now found the correct place to put the pivot item (at position #7.)

16 7, 9, 5, 2, 1, 8, 53, 18, 44, 17,



Rather than move everything down and insert it there, observe that the lower items are out of order anyway and can be re-positioned later, so it will do to simply swap the pivot item with item #7. This produces:

8, 7, 9, 5, 2, 1, 16, 53, 18, 44, 17,

The list is written with the 16 isolated, inverted and separating two sub-lists so as to emphasize the fact of the partition.

The item 16 is now in the correct position and need never be touched again. Because of the way in which the correct position for the item 16 was located, everything in the left sub-list is now less than 16, and everything in the right sub-list is greater than 16 (hence the name *pivot*). Thinking recursively, it is now possible to observe that when the left and right sub-lists are sorted, the entire list will be sorted.

Repeating for the left sub-list yields:

8 7, 9, 5, 2, 1, 53, 18, 44, 17,

8 7, 1, 5, 2, 9, 16 53, 18, 44, 17, no more out of place



2, 7, 1, 5, 8, 9, 16 53, 18, 44, 17,

Repeat for the sub-list left of the 8, obtaining

2 7, 1, 5, 8, 9, 16 53, 18, 44, 17,

2 1, 7, 5, 8, 9, 16 53, 18, 44, 17,



1, 2, 7, 5, 8, 9, 16 53, 18, 44, 17,

The left sub-list of the pivot 2 is sorted. (one item) When the right sub-list of 2 (two items) is sorted, one has:

1, 2, 5, 7, 8, 9, 16 53, 18, 44, 17,

Now the left sub-list of the last pivot (7) has only one item and so is sorted. Backing up the recursion chain, the right sub-list of the pivot 8 also has only one item (9) so it is already sorted, too. Backing up farther, the next list to sort is the right sub-list of the original pivot 16.

1, 2, 5, 7, 8, 9, 16 53, 18, 44, 17,



1, 2, 5, 7, 8, 9, 16 17, 18, 44, 53,

1, 2, 5, 7, 8, 9, 16 17, 18, 44, 53,



1, 2, 5, 7, 8, 9, 16 17, 18, 44, 53,



1, 2, 5, 7, 8, 9, 16 17, 18, 44, 53,

The last right sub-list of the pivot 18 has only one element and is sorted, so the entire sort is finished. Note that there

were even places where one or the other of the two sub-lists was empty and also did not need to be considered. Having carefully hand-stepped through this sorting method, it is now possible to produce the code for the quicksort. This code will depart a little from the previous conventions for sorts in this chapter, in that it will assume that it is being called by a client that is already numbering the array [0 .. n-1]. This assumption is made because the quicksort, being recursive, is its own major client, and already has the array numbered in that manner when it calls itself. Notice also the treatment of numbers scanned in either direction that are equal to the pivot. They are left where they are for that scan. Eventually they will be shuffled adjacent to the this pivot when another sub-list is sorted.

```

PROCEDURE QuickSort1 (VAR source : ARRAY OF CARDINAL; lBound, uBound : CARDINAL);
VAR
    lCount, uCount, pivot : CARDINAL;

BEGIN

    IF lBound <= uCount
    DO
        WHILE (lCount <= uBound) AND (source [lCount] <= pivot)
        DO (* scan the list from the front until *)
            INC (lCount) (* the first number greater than the *)
        END; (* pivot is found or the count gets past the end *)
        WHILE (uCount >= pivot)
        DO (* scan the list from the end until *)
            DEC (uCount) (* the first number less than the *)
        END; (* pivot is found or the count gets to the start *)
        IF lCount < uCount (* counts have not passed each other yet *)
        THEN (* so trade between upper and lower partitions *)
            Swap (source [lCount], source [uCount]);
        END;
        END; (* while lCount <= uCount *)

        (* now swap the pivot into the correct position *)
        source [lBound] := source [uCount];
        source [uCount] := pivot;

        (* and do the left and right sub-lists *)
        IF uCount # lBound (* otherwise no need to swap & no left list*)
        THEN
            QuickSort (source, lBound, uCount - 1);
            (* sort the remaining left list *)
        END;

        IF uCount < uBound (* otherwise there is no right list *)
        THEN
            QuickSort (source, uCount + 1, uBound)
        END;

    END QuickSort1;

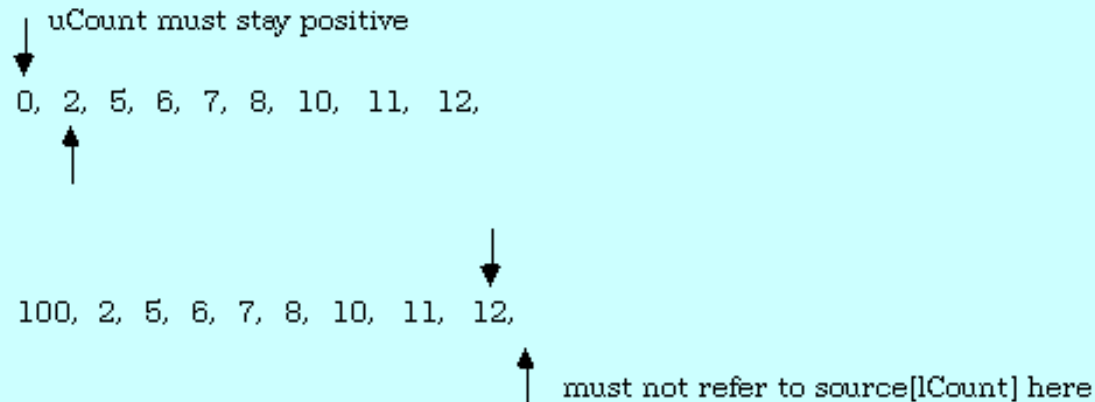
```

Notice that the inner loop, while it consists only of increments and decrements, involves two comparisons in the conditions for each of the while statements. Most of the time, these comparisons, designed to prevent the indices from becoming invalid, will not be done, for most of the work is done away from the endpoints of the range. Thus, the

majority of these tests are wasted. The extra comparisons can be eliminated, but this is achieved at the cost of doing a few more swaps and making some other changes.

At the top end, the *lCount* must be prevented from going past *uBound* and causing a reference to an item not in the range being sorted. In the code above, *lCount* could become one greater than *uBound*, but on the subsequent entry to the loop, no invalid reference can be made to *source [lCount]* because of the short-circuited boolean evaluation on the condition. This can also be done by ensuring that the item at *uBound* is greater than or equal to the pivot to begin with (check and swap if needed,) and by not ignoring items equal to the pivot on the scan, but including them as swappable. This latter suggestion will take care of the low end too, where the main concern is that the index must not underflow the cardinal type.

Situations to watch:



In the variation on the code that follows, all these ideas have been implemented. The *lCount* starts after the pivot item, and because the item at *uBound* was previously guaranteed to be not less than the pivot, and *lCount* skips only the items less than the pivot, it is guaranteed to stop on a valid index. Likewise, the *uCount* starts before the item at *uBound* (because it is known not to be less than the pivot) and is guaranteed not to pass the pivot item. Any items equal to the pivot that happen to be in the range between the two ends will get swapped to the other partition by this version (they were skipped by the last one), but this is a small price to pay to eliminate one comparison from every iteration of the inner loop.

```

PROCEDURE QuickSort2 (VAR source : ARRAY OF CARDINAL; lBound, uBound : CARDINAL);
VAR
    lCount, uCount, pivot : CARDINAL;

BEGIN

    IF lBound < source [lBound]
    THEN
        Swap (source [uBound], source [lBound])
    END;

    pivot := source [lBound];    (* comparisons made with this one *)
    lCount := lBound; (* forward count starts at the pivot *)
    uCount := uBound; (* backward count starts at end *)

    WHILE lCount <= uCount
    DO
        REPEAT (* scan the list from the front + 1 until *)
            INC (lCount) (* the first number greater than *)
        UNTIL (source [lCount] <= pivot); (* or equal pivot is found *)
    END

```

```

    IF lCount < uCount (* counts have not passed each other yet *)
        THEN (* so trade between upper and lower partitions *)
            Swap (source [lCount], source [uCount]);
        END;
    END; (* while lCount <= uCount *)

(* now swap the pivot into the correct position *)
source [lBound] := source [uCount];
source [uCount] := pivot;

(* and do the left and right sub-lists *)
IF uCount # lBound (* otherwise no need to swap & no left list*)
    THEN
        QuickSort (source, lBound, uCount - 1);
        (* sort the remaining left list *)
    END;

IF uCount < uBound (* otherwise there is no right list *)
    THEN
        QuickSort (source, uCount + 1, uBound)
    END;

END QuickSort2;

```

Earlier, it was noted that if the partitions did not divide the list into roughly equal parts, this sort would lose efficiency. In fact, the worst possible case is a list that starts out already sorted, for each partition will have an empty left sub-list and all but the one element in the right. (Starting out reverse sorted is just as bad). Quicksort degenerates into a *sort* in such cases--indeed the insert method is faster. There are several ways to help ensure that this degeneration does not take place. Perhaps the easiest is to begin by using the middle item as the pivot. One would be extremely unlucky if this produced the worse case each time. The middle item could be swapped with the low index item in the range before doing anything else, or it could be used as is. If the latter is done, however, it would have to be compared to ensure that it is not less than the low index item so as to prevent the possibility of underflowing the cardinal type if the index becomes negative. Here is the first part of the modified code, with the first suggestion implemented. Note that there is no need to do a swap if the list has only two items.

```

IF lBound > 2
    THEN
        Swap (source [(uBound + lBound) DIV 2], source [lBound])
    END;

IF source [uBound] < source [lBound]
    THEN
        Swap (source [uBound], source [lBound])
    END;

```

An even more sophisticated modification is to select the median of the first, middle, and last items as the pivot in each case. These three could be sorted in place, and the middle one used as the pivot. The sort would be unlikely to degenerate, and the first and last items would be arranged so as to ensure the appropriate stopping points for the two counts.

Yet another modification could be done based on the observation that the computations involved here have become

relatively large in number if only a small list is to be sorted. At some point, the insert sort is actually faster, so one calls it to finish things off.

The first portion of the quicksort would then be expressed as:

```
PROCEDURE QuickSort (VAR source : ARRAY OF CARDINAL; lBound, uBound : CARDINAL);  
  
VAR  
    lCount, uCount, pivot : CARDINAL;  
  
BEGIN  
  
    IF lBound < 10 THEN (* insert sort if list small *)  
        InsertSort (source, lBound, uBound);  
    ELSE (* staying here, so use middle as pivot *)  
        Swap (source [(uBound + lBound) DIV 2], source [lBound])  
    END;  
  
    IF source [uBound] < source [lBound]  
    THEN  
        Swap (source [uBound], source [lBound])  
    END;  
  
    (* etc....*)
```

with the rest of the code identical to the last version. The number ten as the cutoff point for the minimum list length to be considered for a quicksort is arbitrary. The reader is invited to time the two versions and find a better boundary for a particular system. Note, however, that it is important to ensure that the quicksort algorithm was operating correctly even for small lists before making this modification. This approach should always be used when implementing algorithms that are new to the programmer--first ensure that it works in all situations; then fine tune it and adapt it for the special cases. The quicksort is regarded as the all-round best sorting algorithm for rearranging an array in place, but it does need to be handled carefully, and its performance is quite sensitive to this fine tuning.

Other slight alterations could also be considered to the quicksort, but the reader is invited to consult a text on algorithms to that end.

[Contents](#)

13.6 Sorting With Auxiliary Storage

All the sorting methods considered thus far used a single array as both the source and the destination of the data. In most cases, a single temporary variable to hold a pivot or a value for swapping was all the extra storage required. Sometimes, however, it may not be practical to do things this way, and additional so data being sorted may be required. Consider, for instance, the problem of combining two already sorted lists of items into a single sorted list.

Combining two sorted lists into a single sorted list is called merging.

The original lists will not do as a destination for the data (they may well be too small for one thing). Even if each were actually large enough to hold all the data from both, an auxiliary store might still be needed to hold data from one while that from the other was being inserted into its proper place.

Example:

Merge list 2 with list 1.

list 1:

1,4,5,23,78,90,340,1190,3456,...

list 2:

150,234,250,300,340,456,784,987,1121,...

Because of this, merging is usually done from the two source lists into a third destination one. A counter is set up for each of the three lists, and the smallest of the two items at the current counter positions in the source lists is placed at the counter position in the destination. When one of the two sources is exhausted, the other source is copied into the destination. Here is some pseudocode to express this algorithm:

Merge:

```
set countS1, countS2 and countD all to zero
while countS1 < lengthS1 and countS2 < lengthS2
  if source1 [countS1] < source2 [countS2] then
    set dest [countD] to source1 [countS1]
    increment countS1
  else
    set dest [countD] to source2 [countS2]
    increment countS2
  end while
  increment countD
if countS1 = lengthS1 then
  while countS2 < lengthS2
    set dest [countD] to source2 [countS2]
    increment countS2
    increment countD
  end while
else
  while countS1 < lengthS1
```

```

        set dest [countD] to source1 [countS1]
        increment countS1
        increment countD
    end while
end if

```

This code could be simplified somewhat by introducing sentinels after the end of the two source ranges, provided those positions are available for use, and one is not concerned about a few extra comparisons. In this version, when one list is exhausted, the comparisons are still made, but to the sentinel, and the other list is appended without having to write separate code to do so.

```

Merge1:
set countS1, and countS2 to zero
set source1 [lengthS1] and source1 [lengthS2] to maxcard
for countD = 0 to lengthS1 + lengthS2
    if source1 [countS1] < source2 [countS2] then
        set dest [countD] to source1 [countS1]
        increment countS1
    else
        set dest [countD] to source2 [countS2]
        increment countS2
    end while
    increment countD

```

Merging is a useful technique for combining two sorted disk files into one. If the cost of the extra space can be borne, however, the method can also be used to sort an array.

13.6.1 The Merge Sort

One of the difficulties with the quicksort was the fact that it could not be guaranteed to partition the initial list into two (roughly) equal sub-lists, even when the middle item in the initial list is chosen as the pivot. This partitioning could be *forced* to take place, though it would be done at the expense of not positioning a particular item in the right position when the partition is done. Some other method must then be found to get the items in the right positions. Suppose upon doing a partition that there is only one item in the sub-list created. Then, the sub-list is already sorted, and the routine can return. If there are two items, they can be compared and swapped if necessary. On a small scale, this gives a left and right sub-list (but not around a placed pivot) that are both sorted in themselves. At this point, the two lists need to be combined into a single sorted entity before returning to the next higher level of the recursive calls. It is at this point that a merge is performed and some extra storage is required. Since all the sorting routines thus far have sorted open arrays, where the number of elements has not been specified ahead of time, it may not be immediately obvious where the extra storage space is going to come from. One solution is to enclose the working code in a nested procedure with a value parameter to which the array can be copied. The internal procedure can then copy from portions of this copy to the actual array which can remain global to it. Because the two lists being merged are adjacent in the original array, it is not possible to use the trick with the sentinels mentioned above, so the first of the two algorithms is employed here.

```

PROCEDURE MergeSort (VAR source: ARRAY OF CARDINAL; lBound, uBound : CARDINAL);

PROCEDURE Merge (temp : ARRAY OF CARDINAL;

```

```
left, mid, right : CARDINAL);
```

```
(* merges lists [left .. mid] and [mid + 1 .. right] to the  
list [left .. right] copying from the temporary copy to the main  
array global to this procedure *)
```

```
VAR
```

```
countS1, countS2, countD : CARDINAL;
```

```
BEGIN
```

```
countS1 := left;  
countS2 := mid + 1;  
countD := left;
```

```
WHILE (countS1 <= mid) AND (countS2 <= right)  
DO
```

```
IF (temp [countS1] < temp [countS2])  
THEN  
source [countD] := temp [countS1];  
INC (countS1)  
ELSE  
source [countD] := temp [countS2];  
INC (countS2)  
END; (* if *)
```

```
INC (countD);
```

```
END; (* while *)
```

```
IF countS1 <= right
```

```
DO  
source [countD] := temp [countS2];  
INC (countS2);  
INC (countD);  
END; (* while *)
```

```
ELSE
```

```
WHILE countS1 <= mid
```

```
DO  
source [countD] := temp [countS1];  
INC (countS1);  
INC (countD);  
END; (* while *)
```

```
END; (* if *)
```

```
END Merge;
```

```
VAR
```

```
middle : CARDINAL;
```

```
BEGIN (* main mergesort procedure *)
```

```
IF lBound <= mid (* will go through left list *)
```

```
DO (* set up counters on both sides *)
```

```

countS1 := startS1;
countS2 := mid + 1;

WHILE (countS2 <= right)
DO
    IF source [countS1] <= mid (* quit when first list exhausted *)
DO
    IF source [countS1] <= right) AND (temp < second.studentNumber
    (* alter code to suit data type *)
THEN
    RETURN less
ELIF first.studentNumber "k" in sequence *)
k := 3 * k + 1
UNTIL k <= uBound
DO (* each k-sort starts here *)
    temp := source [count];
    (* set aside pointer to unsorted one *)
    moveCount := count;
    WHILE (moveCount > temp) *) (* original *)
    (CompareData (source [moveCount - k]^, temp^) = greater)
    DO
        source [moveCount] := source [moveCount - k];
        (* move along to make room *)
        DEC (moveCount, k) (* count backward in k-list *)
    END;
    source [moveCount] := temp; (* insert new item *)
    INC (count, k);
END; (* while count *)
END; (* for listCount *)
UNTIL k = 1
END ShellSort;

```

In the main program code, two copies were made of the arrays of pointers to this data:

```

FOR count := 1 TO 13 (* set up addresses *)
DO
    theStuffNumAdrs [count] := ADR (theStuff [count])
END;
theStuffNameAdrs := theStuffNumAdrs; (* start both arrays same *)

```

Finally, the sorting procedures were checked with invocations (among other) including:

```

CompareData := CompareDataNum;
PrintIt (theStuffNumAdrs,13, 4, 5); WriteLn; (* before num sort *)
ShellSort (theStuffNumAdrs,0,12); (* whole thing *)
PrintIt (theStuffNumAdrs,13, 4, 5); WriteLn; (* after num sort *)

CompareData := CompareDataName;

```

```
PrintIt (theStuffNameAdrs,13, 4, 5); WriteLn; (* before name sort *)
ShellSort (theStuffNameAdrs,0,12); (* whole thing *)
PrintIt (theStuffNameAdrs,13, 4, 5); WriteLn; (* after name sort *)
```

This version of the procedure `PrintIt` is not included here, but the parameters indicate the number of items to print, and some formatting information. The results were:

113 = joe	77 = bob	0 = alice	50 = gerda
113 = richard	114 = allan	900 = fred	113 = freda
15 = donna	300 = rod	13 = don	135 = joe
1 = Alouyicious			
0 = alice	1 = Alouyicious	13 = don	15 = donna
50 = gerda	77 = bob	113 = freda	113 = joe
113 = richard	114 = allan	135 = joe	300 = rod
900 = fred			
113 = joe	77 = bob	0 = alice	50 = gerda
113 = richard	114 = allan	900 = fred	113 = freda
15 = donna	300 = rod	13 = don	135 = joe
1 = Alouyicious			
1 = Alouyicious	0 = alice	114 = allan	77 = bob
13 = don	15 = donna	900 = fred	113 = freda
50 = gerda	113 = joe	135 = joe	113 = richard
300 = rod			

Notice that because the actual array is accessed via the pointers (including in the procedure *PrintIt*) the sorting by numbers has no affect on the sorting by name--the pointers are in different arrays, and the original data is never moved. Of course, something similar could be done with other sorting algorithms, and with much larger data collections.

[Contents](#)

13.7 An Extended Example (Searching in Text)

One of the most commonly used functions in word processing (and other text manipulation) is that of searching some body of text (the target) for a particular pattern of letters (the pattern.) For example, the ISO module *Strings* has

```
PROCEDURE FindNext(pattern, stringToSearch : ARRAY OF CHAR;  
    startIndex : CARDINAL;  
    VAR patternFound : BOOLEAN;  
    VAR posOfPattern : CARDINAL);
```

The simplest and most natural algorithm for implementing such a search is to count through the target until the first letter of the pattern is matched, then count on both strings to see if the match continues. If it does for the entire pattern, the whole string matches. If a mismatch is found, one then continues to try to match the first character of the pattern with characters in the target.

Example:

Find the pattern "sip" in the target "Mississippi."

Progress:

Compare "s" to "M" and "i" and, finding no match, advance.
Finding a match at "s" look at "i" which fails to match.

```
Mississippi  
  sip
```

The next try letter matches "s," again and then also "i", but not "p."

```
Mississippi  
  sip
```

At the next position there is a mismatch, and then there is another match at "s," but the next letter fails. On the next advance, one has

```
Mississippi  
  sip
```

and here the desired match is obtained. If m and n denote the lengths of the two strings, this (brute force) algorithm may require up to mn comparisons, that is, it is $O(mn)$. Here is an implementation of this procedure :

```
PROCEDURE FindNext(pattern, stringToSearch : ARRAY OF CHAR;  
    startIndex : CARDINAL;  
    VAR patternFound : BOOLEAN;  
    VAR posOfPattern : CARDINAL);  
  
VAR
```

```

countSi : CARDINAL; (* initial stringToSearch position count *)
countP : CARDINAL; (* pattern count *)
countSr : CARDINAL; (* count to the right of stringToSearch start *)
endP : CARDINAL; (* last pattern position *)
lenS : CARDINAL; (* length of stringToSearch *)
ch : CHAR; (* temp holder to reduce referencing *)

```

BEGIN

```

patternFound := FALSE;
endP := LENGTH (pattern);
IF endP << lenS AND (ch << lenS
    THEN (* we exited the loop before end of string *)
        patternFound := TRUE; (* so we got that one match *)
        posOfPattern := countSi;
        RETURN; (* tell the world so *)
    END; (* if countSi < lenS *)

```

```

ELSE (* more than one character to be matched *)
    ch := pattern[endP]; (* start at back end *)
    WHILE countSi + endP < lenS

```

```

    DO

```

```

        IF ch = stringToSearch[countSi+endP]
            THEN (* got a first character match *)
                countP := 0; (* do look for more *)
                countSr := countSi;

```

```

        LOOP

```

IF pattern[countP] <<"si" and there is then a mismatch, one already knows that the initial "s" cannot match the second letter "i" and so the search can be advanced by two places rather than one, and one comparison can be saved. Sometimes the potential savings are more dramatic:

Example:

Find the pattern "gead" in the target "geaageabgeacgead"

Progress:

```

geaageabgeacgead
gead

```

The first three letters match, but when the fourth letter fails to match, they are known not to match the first letter either, so the pattern can be moved over to:

```

geaageabgeacgead
    gead

```

then to

```
geaageabgeacgead
      gead
```

and finally to

```
geaageabgeacgead
      gead
```

where a match on all four characters is finally obtained. Notice that this particular search turns out to be $O(n)$, where n is the length of the target string. However, the ability to slide the matching process over by three characters depends on the fact that there are no repetitions in the pattern string. If there are repetitions, the amount that the search can be shifted depends on the place where the repetition is found in the pattern.

Example:

Find the pattern "papa" in the target "papuapapyruspapa"

Progress:

```
papuapapyruspapa
papa
```

This time, when fails to match, the pattern can be moved over only by two places because of the repetition of the first letter.

```
papuapapyruspapa
  papa
```

Similar considerations apply to a later match of "pap" followed by a mismatch.

In order to implement this strategy, therefore, it is necessary to search the pattern within itself looking for repetitions in order to determine how far the pattern can be moved after a failure at each position. Since this means examining the pattern, and then following up by examining the target, this algorithm is $O(m + n)$. It is called the Knuth-Morris-Pratt algorithm, after the three who originated and refined

it.

```
Knuth-Morris-Pratt search =
  scan pattern and construct "backup" vector (table)
  scan the target
    while characters match, advance in both pattern and target
    on mismatch, shift by amount in "backup" vector for that position
```

Example:

Find the pattern "cashcar" in the target "xcucatcastcashewcashcucashcatcashcart"

Step 1

(construct backup vector): (T = target and P = pattern)

Matched	Mismatch@		backup	meaning	action
	pos	letter			
none	0	c	-1	[0] # c	next T pos, resume P at 0
c	1	a	0	[1] # a maybe c	keep T pos, resume P at 0
ca	2	s	0	[2] # s maybe c	keep T pos, resume P at 0
cas	3	h	0	[3] # h maybe c	keep T pos, resume P at 0
cash	4	c	-1	[4] # c	next T pos, resume P at 0
cashc	5	a	0	[5] # a maybe c	keep T pos, resume P at 0
cashca	6	r	4	[6] # r, got "ca"	next T pos, resume P at 2

The term "backup" refers to the position in the pattern that the search can be backed up to in order to continue examining the target without wasting comparisons. The value -1 will serve to flag that the position in the target is to be incremented for the next comparison. When the scan "backs up" to the position -1, both the pattern and target counters will be incremented; when it backs up to the position 0 (or to any other positive number), only the target position will be incremented before continuing comparisons. Here is the KNP (Knuth-Morris-Pratt) version of the FindNext procedure:

```
PROCEDURE FindNext (pattern, stringToSearch : ARRAY OF CHAR;
  startIndex : CARDINAL;
  VAR patternFound : BOOLEAN;
  VAR posOfPattern : CARDINAL);
```

(* Knuth Morris Pratt version
 adapted by R. Sutcliffe

last modified 1995 05 04 *)

VAR

count1, count2, sCount, pCount, patternLen, targetLength: **INTEGER**;
backup : Vector;

BEGIN

targetLength := **LENGTH** (stringToSearch);
patternLen := **LENGTH** (pattern);

(* construct backup vector *)

count1 := 0;
count2 := -1;
backup [0] := -1;

REPEAT (* look for match in pattern with itself *)

IF (count2 = -1) (* starting (over) *)

OR (pattern [count1] = pattern [count2]) (* (still) matching *)

THEN

INC(count1); (* keep going *)

INC(count2);

IF pattern [count1] # pattern [count2] (* no match there *)

THEN (* keep same count *)

backup [count1] := count2;

ELSE (* keep same backup *)

backup [count1] := backup [count2];

END; (* if pattern *)

ELSE

count2 := backup[count2];

END; (* if count2 *)

UNTIL count1 >= patternLen) **OR** (sCount >= patternLen

THEN (* got it so report back *)

posOfPattern := sCount - patternLen;

patternFound := **TRUE**;

ELSE (* failed *)

posOfPattern := targetLength;

patternFound := **FALSE**;

END; (* if pCount *)

END FindNext;

Unless the pattern being searched for has considerable repetition, this method may not yield much better results than the brute force method in practice. Yet another method that provides much faster searching capability is the Boyer-Moore algorithm; however, it will not be discussed here as it is rather more complicated.

[Contents](#)

13.8 Chapter Summary

This chapter covered these topics:

- linear and binary searches
- the bubble sort, selection and insert (simple) sorts
- the Shell sort
- the quick and merge (advanced) sorts
- performance analysis of searching and sorting algorithms, including the O notation
- the brute force and Knuth-Morris-Pratt string searching algorithms

No new reserved words, standard identifiers, or library routines were taken up in this chapter.

[Contents](#)

13.9 Exercises

Questions:

1. What is the purpose of writing the more complicated code to do binary searches, when a linear one will not only find the item if it is there, but does not even require the data to be sorted in the first place?
2. What is the difference between a simple sort and an advanced sort?
3. What does the **O** notation (big-O notation) signify?
4. What is meant by the term *in-place* sorting?
5. (To look up in a suitable reference:) Why is only one of the sorts (Shell) spelled with a capital letter?
6. (To look up in a suitable reference:) List the names of the inventors (with approximate dates) of each of the sorting methods discussed in the chapter for which you can find the information.
7. From your own research and reading, list at least ten data processing situations in which it is necessary to sort data or to search for data in a list.
8. The second version of the in-place merge contained the line

WHILE (countS2 <= right) **AND** (temp "notnothingnothingsham" and then use the code given in this chapter to check your work.

12. Look up the Boyer-Moore algorithm and write out an explanation of it together with a trace of an actual search as was done in this chapter for the KNP algorithm.

Problems:

NOTE: Several of the problems below ask for tests of performance. To answer these, you ought to have on hand a file of, say, 10 000 or more randomly generated cardinals to answer these questions. Make tests for arrays of various sizes drawn from these numbers before drawing any conclusions. You ought also in some cases to consider whether it makes any difference if the arrays to be sorted start out in random, sorted, or reverse sorted order. In order to ensure that the times are properly torture tested, the arrays should contain several items each of which equals zero and several items each of which equals MAX(CARDINAL).

13. Is there a break-even point in the number of items in a list below which a linear sort is faster than a binary search?
14. Rewrite the bubble sort and the merge sort to produce the list in reverse order (largest to smallest.)
15. Implement a Shell sort and a quicksort for the type cardinal (as in the text) and the type real. By what factor are the performances different for the type real? What conclusions can you draw?
16. Write a program module that will read an array of strings from a disk file, sort it, and write it back. To demonstrate that you have done this, you will need to dump the file to the printer before and after sorting. Of course, you will need to create the file in the first place. You may use the module *Strings* if it is available, or

if not, write such procedures as you happen to need.

17. Test and time a sort of the array generated above with the insertion and Shell sorts as they are presented in the text. You might want to try both linear and binary searching methods in each. Be sure to time only the sorting part of the routine, not the time spent reading it in from a file. You may wish to have the program write the original and sorted arrays sent to the printer.

18. How much performance difference does it make to modify the insert sort to use a binary search?

19. How much performance difference does it make to modify the Shell sort to use a binary search?

20. Modify the Shell sort as finally shown in the text to use a stored array for the k-sequence. To begin with, have the program ask you for the sequence from the keyboard. Test various sequences and try to find an optimum one. Once you have this, organize your work in modules as with your k-sequence safely tucked away in the implementation part.

21. Write a sort routine for the type REAL. Use this to add to your statistics module above a procedure to compute the Median (middle number) in an array of REAL.

22. On your system, for what list length is a binary search more time efficient than a linear search?

23. Encapsulate in a library module and test the first (linear search) style of Shell sort. Compare it with the second style by sorting lists of various lengths and timing the performance. Is the extra arithmetic needed to do a binary search on the k-lists worth it or not?

24. Test a comb sort with a variety of data, (sorted, random, and reverse sorted) and various sizes of data sets. Are the claims made for performance of this sort justified? Are there sizes for the data set for which a straight insert sort is better?

25. Test the quicksort with and without the modification that changes it into an insert sort below some cutoff point. Find the cutoff point that gives the best overall performance, if possible.

26. Suppose instead of using the middle item for the pivot in a quick sort, one used the median of the left, middle, and right items instead. Write and test this modification, determining whether or not there is a performance increase or decrease for random and already sorted arrays.

27. Implement a merge sort to combine the contents of two previously created files into a single file, without holding the entire sorted array in memory. (Note that three files must be open at once).

28. Implement the merge sort as presented in the text, (either version) place suitable print statements at strategic places, and print out the kind of detailed traced that the text presents for other sorts.

29. Actually test the two methods of merging two lists in an array in place, and determine which gives the better performance of the two. How do both compare with an ordinary insert sort, and with a quicksort?

30. Modify one simple sort and one advanced sort to sort your favourite type of

record and test the results to ensure you have not introduced any errors.

31. Modify one simple sort and one advanced sort to sort strings and test the results to ensure you have not introduced any errors.

32. Implement and test a version of the KNP algorithm that searches a file for a pattern.

33. Implement and test for yourself the sorting of an array with auxiliary pointers shown in [section 13.6.2](#). In order to make things more interesting, use a Quicksort routine.

Project:

34. Implement and test a version of the Boyer-Moore string search algorithm.

35. Devise a routine to graph the time per item sorted (some scaling will be needed). Run Quicksort successively on one through 4000 active items and graph the results. Comment on the stability of Quicksort. Does it take dramatically more time for certain sizes of data collections?

[Contents](#)

Chapter 13

Searching and Sorting

[13.0 Chapter Goals](#)

[13.1 Searching](#)

[13.1.1 Linear \(Sequential\) Searches](#)

[13.1.2 Binary Searches](#)

[13.2 Introduction to Sorting](#)

[13.2.1 The Bubble Sort](#)

[13.2.2 The Selection Sort](#)

[13.3 Analyzing Sorting Routines](#)

[13.4 Inserting Methods](#)

[13.4.1 The Insert Sort](#)

[13.4.2 The Shell Sort](#)

[13.4.3 The Combsort](#)

[13.5 Advanced Sorting of Arrays](#)

[13.5.1 The QuickSort](#)

[13.6 Sorting With Auxiliary Storage](#)

[13.6.1 The Merge Sort](#)

[13.6.2 Pointers and Sorting](#)

[13.7 An Extended Example \(Searching in Text\)](#)

[13.8 Chapter Summary](#)

[13.9 Exercises](#)

[Contents](#)

14.0 Chapter Goals

The purpose of this chapter is to discuss the use of a variety of common Abstract Data Types using Modula-2. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

data aggregates in a variety of forms

Realized in the Modula-2 notation:

one or more implementations of each of lists, queues, stacks, tables, and trees in a semi-generic fashion

Data Manipulation Abstractions

General:

a number of techniques for working with pointers and ADTs

Realized in the Modula-2 notation:

insertion and deletion in lists, queues, stacks, tables, and trees

Programming Abstractions

General:

traversing a structure with a procedure to act on data in the nodes

Realized in the Modula-2 notation:

linear traversals in lists, queues and stacks, and in, pre, and post order traversals of trees

14.1 Introduction to Intermediate Data Structures

Simple linked lists were introduced in the last chapter. The utility of these can be extended considerably by adding a few new procedures and by allowing for the data type used to be easily changed.

Besides the various kinds of lists, certain other structures for data are also widely used. Sometimes these are just variations on the linked list theme, and sometimes they are not linear at all. While the subject of data structures in general is usually thought of as providing the material for an entire text, enough of the principles of ADTs have already been discussed here to allow these structures to be dealt with in a more succinct fashion than in the traditional course, and yet not lose any generality. Taken together with the discussions on algorithms in the previous chapter and the extensions in the next two, this material can itself provide the basis for an adequate replacement for the traditional data structures course.

[Contents](#)

14.2 Lists Revisited

The lists of chapter twelve were rather simple in that only a few operations were sketched and the natural step of creating a library module to encapsulate the ADT List was not taken. A number of issues arise when this next step is undertaken:

- what kind(s) of data is (are) to be listed

If there is only one type, a customized linked list package that takes the characteristics of the data fields into consideration in some way may be best. One way of doing this is to provide more than one set of links to sort data by more than one key field. If more than one kind of list (different types of enlisted data) are envisioned, then a more generic library package capable of easy modification to a variety of data is indicated. In the example of this section, genericity is ensured by having the definition module import the data type from some other module and then rename it to a local type name. This ensures that the module need only be changed in its first few lines to generate a new linked list for a different type of data.

- whether single or double links will be used

In this section, doubly linked lists will be employed, though in some applications single or circular links are indicated.

- whether insertion and deletion are allowed by index number
- whether data fetching and updating are allowed by index number

In the example of this section, they will be. However, for very large lists, it might be more useful to provide procedures to search for specific items by one of their fields and then update, fetch, insert, or delete the one found.

- whether a "current" pointer will be kept for some operations.

In the example of this section, such pointers will be kept separately for insertion, deletion, and fetch/update operations. Moreover, an enumerated type is defined so that setting these pointers can be done operation by operation at a specific index number.

With these ideas in mind, here is the definition module for a relatively robust linked list type. It depends on a library module called *ItemADT* but this name can be changed as desired. The only thing required from that module for the definition module *lists* to work is the definition of the ADT itself. Observe the careful specification of predicates for each operation.

DEFINITION MODULE Lists;

```
(* semi- generic implementation of lists  *)
(* copyright © 1995 by R. Sutcliffe *)
(* last modification 1995 05 29 *)
```

```
(* To use, change ItemADT to the module name where the ADT to be listed is defined
and ItemType in the following two lines to the correct name found there. Also
required: an assignment procedures for the ADT. *)
```

FROM ItemADT **IMPORT**

 ItemType;

TYPE

 DataType (* local name *) = ItemType;

TYPE

```
List;  
Operation = (insert, delete, fetchup);
```

PROCEDURE Create (**VAR** list : List);

(* Pre: none

Post: a new list of ItemType structure is initialized with length zero. Insert, delete and fetch/update start out at the head of the list. *)

PROCEDURE Discard (**VAR** list : List);

(* Pre: list is a validly created list

Post: list is undefined *)

PROCEDURE Length (list : List) : **CARDINAL**;

(* Pre: list is a validly created list

Post: The number of items in the list is returned. *)

PROCEDURE SetAtHead (**VAR** list : List; op : Operation);

(* Pre: list is a validly created list

Post: The position for the given insert, delete, or fetch/update operation is the first item. *)

PROCEDURE SetAtTail (**VAR** list : List; op : Operation);

(* Pre: list is a validly created list

Post: The position for the given insert, delete, or fetch/update operation is the last item. *)

PROCEDURE SetAtPos (**VAR** list : List; op : Operation; itemNum : **CARDINAL**);

(* Pre: list is a validly created

Post: The position for the given insert, delete, or fetch/update operation is the itemNum item. If ItemNum > 0

DO

Delete (list);

END;

DISPOSE (list);

END Discard;

PROCEDURE Length (list : List) : **CARDINAL**;

BEGIN

RETURN list^.numItems;

END Length;

PROCEDURE SetAtHead (**VAR** list : List; op : Operation);

BEGIN

CASE op OF

insert:

list^.curInsert := list^.head |

delete:

list^.curDelete := list^.head;

list^.delAtHead := **TRUE**; |

fetchup:

list^.curFetchup := list^.head;

```

    END;
END SetAtHead;

PROCEDURE SetAtTail (VAR list : List; op : Operation);

BEGIN
    CASE op OF
        insert:
            list^.curInsert := list^.tail |
        delete:
            list^.curDelete := list^.tail;
            list^.delAtHead := FALSE; |
        fetchup:
            list^.curFetchup := list^.tail;
    END;
END SetAtTail;

PROCEDURE SetAtPos (VAR list : List; op : Operation; itemNum : CARDINAL);

VAR
    count : CARDINAL;
    tempPoint : NodePoint;

BEGIN
    IF itemNum = 0
    THEN
        SetAtHead (list, op);
    ELSIF itemNum > (list^.numItems DIV 2) THEN (* past middle? *)
        count := list^.numItems;
        tempPoint := list^.tail; (* start at the back *)
        WHILE count < itemNum    (* go forward  if necessary *)
        DO
            tempPoint := tempPoint^.next;
            INC (count);
        END;
    END;
    CASE op OF
        insert:
            list^.curInsert := tempPoint |
        delete:
            list^.curDelete := tempPoint;
            list^.delAtHead := FALSE; |
        fetchup:
            list^.curFetchup := tempPoint;
    END;
END;

END SetAtPos;

PROCEDURE Insert (VAR list : List; item : DataType);

VAR
    local : NodePoint;

```

```

BEGIN
    NEW (local);
    local^.data := item;
    local^.next := list^.curInsert;
    IF list^.curInsert # list^.head (*inserting at head? *)
    THEN (* no, so chain in new item *)
        local^.last := list^.curInsert^.last; (* point back to previous node *)
        list^.curInsert^.last^.next := local; (* and make it point to new one *)
    ELSE
        local^.last := NIL; (* yes, so back pointer is NIL *)
        IF (list^.curDelete = list^.head) AND list^.delAtHead
            (* if delete is at head too, keep it there *)
        THEN
            list^.curDelete := local;
        END;
        IF list^.curFetchup = list^.head (* if fetchUp is at head too, keep it there *)
        THEN
            list^.curFetchup := local;
        END;
        IF list^.tail = NIL (* if this is the first item in *)
        THEN
            list^.tail := local;
        END;
        list^.head := local; (* revise the head *)
    END;
    IF list^.curInsert # NIL
    THEN
        list^.curInsert^.last := local;
    END;
    list^.curInsert := local; (* insert point becomes new item *)
    INC (list^.numItems);

```

END Insert;

```

PROCEDURE Append (VAR list : List; item : DataType);

```

```

VAR
    local : NodePoint;

```

```

BEGIN
    NEW (local);
    local^.data := item;
    local^.last := list^.tail;
    local^.next := NIL;
    IF list^.tail = NIL (* list currently empty *)
    THEN
        WITH list^
        DO
            head := local;
            curInsert := head;
            curDelete := head;
            curFetchup := head;
        END;
    END;

```



```

ELSE
    list^.tail^.next := local;
END;
list^.tail := local;
INC (list^.numItems);

```

```

END Append;

```

```

PROCEDURE Update (VAR list : List; item : DataType);

```

```

BEGIN
    Assign (item, list^.curFetchup^.data);
END Update;

```

```

PROCEDURE Fetch (list : List; VAR item : DataType);

```

```

BEGIN
    Assign (list^.curFetchup^.data, item);
END Fetch;

```

```

PROCEDURE Delete (VAR list : List);

```

```

VAR
    newCurDel, temp : NodePoint;

```

```

BEGIN
    IF list^.numItems = 0
    THEN
        RETURN
    END;
    temp := list^.curDelete;
    IF list^.curDelete^.last # NIL (* if not at #1 *)
    THEN
        list^.curDelete^.last^.next := list^.curDelete^.next;
    ELSE
        list^.head := list^.curDelete^.next;
    END;
    IF list^.curDelete^.next # NIL (* if not at last item *)
    THEN
        list^.curDelete^.next^.last := list^.curDelete^.last;
        newCurDel := list^.curDelete^.next;
    ELSE
        list^.tail := list^.curDelete^.last;
        newCurDel := list^.curDelete^.last;
    END;
    IF list^.curDelete = list^.curInsert (* hammered off insert item? *)
    THEN
        list^.curInsert := newCurDel;
    END;
    IF list^.curFetchup = list^.curInsert (* hammered off fetchup item? *)
    THEN
        list^.curFetchup := newCurDel;
    END;
    DEC (list^.numItems);

```

```

    list^.curDelete := newCurDel;
    DISPOSE (temp);
END Delete;

END Lists.

```

In order to complete the task, the ADT to be listed must be defined and implemented in a suitable module, along with the procedure *Assign*. The name of this module and the name of the ADT are then substituted in the top lines of the definition and implementation modules above, and (possibly) new copies saved with unique names (*ListOfMyADT*). All these must then be compiled before a client program can be written and linked.

If a second list type using another ADT is required (perhaps even for the same client) it is not difficult to define that other ADT and make some minor editorial changes in the first few lines (only) of the modules above, then compile again. This can be done as many times as desired without making any changes to the body of the implementation.

Here is a simple version of a typical supplier of the information. It actually uses the name *ItemADT* so that no changes at all are necessary to proceed.

```

DEFINITION MODULE ItemADT;

TYPE
    ItemType = CARDINAL;

PROCEDURE Assign (a : ItemType; VAR b : ItemType);

END ItemADT.

IMPLEMENTATION MODULE ItemADT;

PROCEDURE Assign ( a : ItemType; VAR b : ItemType);
BEGIN
    b:= a;
END Assign;

END ItemADT.

```

[Contents](#)

14.3 Queues

When data is organized in a list (or an array), one can get access to any one of the items in the structure. However, if data structures are to mirror real-life situations, this kind of access may not always be desirable.

Consider for instance the problem of simulating something like a cashier's lineup. Here, new data items are added at one end of the line, but they are only served at the other, in order of their addition to the line. This is termed a first-in-first-out data structure.

Example:

Action	Resulting Queue
A arrives	A
B arrives	AB
C arrives	ABC
service	BC
D arrives	BCD
service	CD

A first-in-first-out data structure is called a queue.

One would probably implement a queue in such a way that the only operations on it were, say, *Init*, *Destroy*, *Full*, *Empty*, *Enqueue* (add to queue), and *Serve*. This would prevent access to any but the end of the queue for *adding* and the beginning for *serving* (removing an item). *Full* would be used to check the status of the queue before attempting to add, and *Empty* to check before attempting to serve.

Init would create an empty queue, and *Destroy* would terminate it (giving back all the memory it used in the dynamic allocation).

Here, for example is a module to define and implement a queue for text items:

```
DEFINITION MODULE TextQueues;
```

```
(* a simple text queue of specified size  
  by   R. Sutcliffe  
  last modified 1995 05 29   *)
```

TYPE

```
Queue;    (* details in implementation *)  
ActionProc = PROCEDURE (CHAR);
```

```
PROCEDURE Init (maxSize : CARDINAL) : Queue;
```

```
(* Pre: none  
   Post: Establish a text queue that can enqueue at most maxSize characters.   *)
```

```
PROCEDURE Destroy (q : Queue);
```

```
(* Pre: q is a previously established queue.  
   Post: any memory previously allocated to q is now returned.   *)
```

```
PROCEDURE Full (q : Queue) : BOOLEAN;
```

```
(* Pre: q is a previously established queue.
```

Post: if the number of active items in the queue equals the `maxSize` used when `Init` was called, returns **TRUE**; otherwise returns **FALSE**. *)

PROCEDURE Empty (q : Queue) : **BOOLEAN**;

(* Pre: q is a previously established queue.

Post: if the number of active items in the queue is zero, returns **TRUE**; otherwise returns **FALSE**. *)

PROCEDURE Enqueue (q : Queue; item : **CHAR**);

(* Pre: q is a previously established queue.

Post: if the number of active items in the queue is less than the `maxSize` used when `Init` was called, the given character is added to the queue; otherwise no action is taken. *)

PROCEDURE Serve (q : Queue; **VAR** item : **CHAR**);

(* Pre: q is a previously established queue.

Post: if the number of active items in the queue is greater than the zero, a character is removed from the queue and returned on a first-in-first-out basis; otherwise no action is taken. *)

PROCEDURE Traverse (q : Queue; Proc : ActionProc);

(* Pre: q is a previously established queue.

Post: The action specified by `Proc` is taken on each item in the queue from the head to the tail without affecting the queue itself. *)

END TextQueues.

This particular specification envisions a maximum size for a queue defined when the queue is first created. It also has a procedure that allows a client application to "walk" or "traverse" down the queue and do something with each item without affecting the queue itself. The "something" that is done depends on the procedure that is passed.

In the implementation, *Queue* must be redefined as a pointer to a data structure containing information as to the size of the queue, a pointer to the actual data, the number of active items, and a pointer to indicate the position in the queue for enqueueing (head) and for serving (tail.) *Init* must create and initialize one of these structures and set aside enough memory for *maxSize* characters. *Enqueue* must enter the supplied character into the queue and adjust the counter and enqueueing position. *Serve* must remove and return the character at the serving position, adjust the serving position and the count. One possible implementation of this queue uses a dynamic array to hold the characters. If in adjusting the count, one goes past the end of the structure, the number must be wrapped to the beginning (if possible.) If the two counters collide, the queue is either full or empty, depending on which catches up with which. Here, though, these conditions are kept track of by the number of items that are currently in the queue.

Because it would be somewhat inefficient to implement this by dynamically allocating a separate piece of memory for each character to be enqueued, the *Init* procedure has been specified to take sufficient memory for the maximum number of items that can be enqueued. Here is the implementation:

IMPLEMENTATION MODULE TextQueues;

(* a simple text queue of specified size

by R. Sutcliffe

last modified 1995 05 29 *)

FROM SYSTEM **IMPORT**

ADDRESS, ADDADR;

FROM Storage **IMPORT**

ALLOCATE, DEALLOCATE;

```

TYPE
Queue = POINTER TO QueueData;
QueueData =
    RECORD
        size : CARDINAL;
        numberActive: CARDINAL;
        head, tail : CARDINAL;
        dataPtr : ADDRESS;
    END;
CharAdr = POINTER TO CHAR;

PROCEDURE Init (maxSize : CARDINAL) : Queue;
VAR
    q : Queue;
BEGIN
    NEW (q); (* set up a queue record *)
    q^.size := maxSize;
    q^.numberActive := 0;
    q^.head := 0;
    q^.tail := 0;
    ALLOCATE (q^.dataPtr, SIZE (CHAR) * maxSize);
    IF q^.dataPtr = NIL (* couldn't get memory for data *)
    THEN
        DISPOSE (q);
    END;
    RETURN q;
END Init;

PROCEDURE Destroy (q : Queue);
BEGIN
    DEALLOCATE (q^.dataPtr, SIZE (CHAR) * q^.size); (* data *)
    DISPOSE (q); (* and queue info *)
END Destroy;

PROCEDURE Full (q : Queue) : BOOLEAN;
BEGIN
    IF q^.numberActive = q^.size
    THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    END;
END Full;

PROCEDURE Empty (q : Queue) : BOOLEAN;
BEGIN
    IF q^.numberActive = 0
    THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    END;

```

```

END Empty;

PROCEDURE Enqueue (q : Queue; item : CHAR);
VAR
    ptr : CharAdr;
BEGIN
    IF NOT Full (q)
    THEN
        ptr := ADDADR (q^.dataPtr, q^.tail * SIZE (CHAR));
        ptr^ := item;
        INC (q^.numberActive);
        q^.tail := (q^.tail + 1) MOD q^.size;
    END; (* does nothing if full *)
END Enqueue;

PROCEDURE Serve (q : Queue; VAR item : CHAR);
VAR
    ptr : CharAdr;
BEGIN
    IF NOT Empty (q)
    THEN
        ptr := ADDADR (q^.dataPtr, q^.head * SIZE (CHAR));
        item := ptr^;
        DEC (q^.numberActive);
        q^.head := (q^.head + 1) MOD q^.size;
    END; (* does nothing if empty *)
END Serve;

PROCEDURE Traverse (q : Queue; Proc : ActionProc);
VAR
    ptr : CharAdr;
    pos : CARDINAL;
BEGIN
    IF NOT Empty (q)
    THEN
        pos := q^.head; (* start at head of queue *)
        REPEAT
            ptr := ADDADR (q^.dataPtr, pos * SIZE (CHAR));
            Proc (ptr^); (* do whatever we were told to *)
            INC (pos);
            IF pos = q^.size
            THEN (* wrap around if necessary *)
                pos := 0;
            END;
        UNTIL pos = q^.tail;
    END;
END Traverse;

END TextQueues.

```

If some other abstract data type is to be enqueued using this approach, another library module must be written. The example below has the predicates removed to make it more compact. It is written in the semi-generic style of the list module in the previous section.

```

DEFINITION MODULE anAdtQueues;

(* a simple semi-generic queue
   by R. Sutcliffe
   last modified 1995 05 29   *)

FROM ItemADT IMPORT
    ItemType;
TYPE
    DataType (* local name *) = ItemType;
    ActionProc = PROCEDURE (DataType);
    Queue;

PROCEDURE Init (VAR q : Queue);
PROCEDURE Destroy (q : Queue);
PROCEDURE Full (q : Queue) : BOOLEAN;
PROCEDURE Empty (q : Queue) : BOOLEAN;
PROCEDURE Enqueue (q : Queue; item : DataType);
PROCEDURE Serve (q : Queue; VAR item : DataType);
PROCEDURE Traverse (q : Queue; Proc : ActionProc);

END anAdtQueues.

```

This time, a singly linked approach is used to the implementation. Here, a queue is only full if no more memory can be obtained for the data node. Since it is necessary to be able to check that fact ahead of time by calling *Full*, the space for the next data node is obtained after the current one has been linked in. The very first one is obtained in the body of the module. If the last obtained data node pointer is NIL, then all queues established by this module are full (so, the parameter on *Full* could have been left out).

```

IMPLEMENTATION MODULE anAdtQueues;

(* a simple semi-generic queue
   by R. Sutcliffe
   last modified 1995 05 29   *)

FROM Storage IMPORT
    ALLOCATE, DEALLOCATE;

TYPE
    DataPtr = POINTER TO DataNode;
    DataNode =
        RECORD
            data : DataType;
            toPoint : DataPtr; (* single node linking *)
        END;
    Queue = POINTER TO QueueData;
    QueueData = (* each queue has this data *)
        RECORD
            head, tail : DataPtr;
        END;
VAR
    nextItem : DataPtr;
    (* a global for reserving space ahead of time *)

```

```
PROCEDURE GetNextItemSpace;  
BEGIN
```

```
    NEW (nextItem);  
END GetNextItemSpace;
```

```
PROCEDURE Init (VAR q : Queue);
```

```
BEGIN  
    NEW (q);  (* set up a queue record *)  
    IF q # NIL  
        THEN  
            q^.head := NIL;  
            q^.tail := NIL;  
        END;  
END Init;
```

```
PROCEDURE Destroy (q : Queue);
```

```
VAR  
    dummy : DataType;  (* not going anywhere, just killing them *)  
BEGIN  
    IF NOT Empty (q)  
        THEN  
            REPEAT  
                Serve (q, dummy);  
            UNTIL Empty (q);  
        END;  
    DISPOSE (q);  
END Destroy;
```

```
PROCEDURE Full (q : Queue) : BOOLEAN;
```

```
(* whenever the last call to GetNextItemSpace returned NIL, all lists are full. *)
```

```
BEGIN  
    RETURN (nextItem = NIL);  
END Full;
```

```
PROCEDURE Empty (q : Queue) : BOOLEAN;
```

```
BEGIN  
    IF q^.head = NIL  
        THEN  
            RETURN TRUE  
        ELSE  
            RETURN FALSE  
        END;  
END Empty;
```

```
PROCEDURE Enqueue (q : Queue; item : DataType);
```

```
BEGIN  
    IF NOT Full (q)  
        THEN  
            (* use global nextItem already obtained *)  
            nextItem^.data := item;  
            nextItem^.toPoint := NIL;
```



```

    IF Empty (q) (* i.e. before this *)
    THEN (* change head *)
        q^.head := nextItem
    ELSE (* chain old tail to new one *)
        q^.tail^.toPoint := nextItem;
    END;
    q^.tail := nextItem; (* always *)
    GetNextItemSpace; (* for next time; if fails, will show full *)
END; (* does nothing if full *)
END Enqueue;

```

```

PROCEDURE Serve (q : Queue; VAR item : DataType);
VAR

```

```

    temp : DataPtr;
BEGIN
    IF NOT Empty (q)
    THEN
        temp := q^.head;
        q^.head := q^.head^.toPoint;
        IF NOT Full (q)
        THEN
            DISPOSE (temp)
        ELSE
            nextItem := temp;
        END;
    END; (* does nothing if empty *)
    IF Empty (q) (* i.e. now empty *)
    THEN
        q^.tail := NIL;
    END;
END Serve;

```

```

PROCEDURE Traverse (q : Queue; Proc : ActionProc);
VAR

```

```

    ptr, next : DataPtr;
BEGIN
    IF NOT Empty (q)
    THEN
        ptr := q^.head; (* start at head of queue *)
        REPEAT
            next := ptr^.toPoint;
            Proc (ptr^.data); (* do whatever we were told to *)
            ptr := next;
        UNTIL ptr = NIL;
    END;
END Traverse;

```

```

BEGIN
    GetNextItemSpace; (* do one ahead of time *)
END anAdtQueues.

```

The very simple test harness below was used to check the operation of this library. Note that the library itself would be renamed according to the data type it was importing. The procedure passed to traverse would also depend on the data being

used. Perhaps that procedure too could be placed in the ADT library and imported to the queue library.

```
MODULE QueueTest;
(* crude test program to test the queue library with the cardinal ADT as imported to
anAdtQueues from ItemADT
by R. Sutcliffe  modified 1995 05 29 *)

FROM STextIO IMPORT
    WriteString, WriteLn, ReadChar, WriteChar, SkipLine;
FROM SWholeIO IMPORT
    WriteCard;
FROM anAdtQueues IMPORT
    Queue, Init, Destroy, Enqueue, Full, Empty, Traverse, Serve;

VAR
    q : Queue;
    n : CARDINAL;
    ch : CHAR;

PROCEDURE WriteItem (card: CARDINAL);
(* a procedure to pass as a parameter in the next procedure *)
BEGIN
    WriteCard (card, 0);
    WriteChar (",")
END WriteItem;

PROCEDURE WriteCardQueue (q : Queue);
BEGIN
    Traverse (q, WriteItem);
    WriteLn;
END WriteCardQueue;

BEGIN (* main *)
    Init (q);
    n := 1;
    Enqueue (q, n);
    WriteCardQueue (q);
    n := 2;
    Enqueue (q, n);
    WriteCardQueue (q);
    n := 1000;
    Enqueue (q, n);
    WriteCardQueue (q);
    Serve (q, n);
    WriteCardQueue (q);
    n := 2000;
    Enqueue (q, n);
    WriteCardQueue (q);
    Serve (q, n);
    WriteCardQueue (q);
    n := 30000;
    Enqueue (q, n);
    WriteCardQueue (q);
```

END QueueTest.

The results of this test are shown below:

**** Run log starts here ****

1,

1,2,

1,2,1000,

2,1000,

2,1000,2000,

1000,2000,

1000,2000,30000,

In the next two chapters, more advanced techniques will be discussed to allow the concept of the *queue* to be abstracted without reference to any particular data type that is to be enqueued and served.

[Contents](#)

14.4 Stacks

A similar type of data structure is used in situations where the last item entered will be the next item served. Consider the example of the previous section, but with another column added to contrast the action of a stack with that of a queue.

Example:

Action	Resulting Queue	Resulting Stack
A arrives	A	A
B arrives	AB	AB
C arrives	ABC	ABC
service BC	AB	
D arrives	BCD	ABD
service CD	AB	

Item A will not be served until it is the only remaining item on the stack.

A last-in-first-out data structure is called a stack.

The classic everyday instance of a stack is the pile of dinner plates in the university cafeteria. Plates are added to the top of the stack, and they are also removed from the top--the opposite end to queue serving. Here, the two basic operations are called *Push* (add something to the stack) and *Pull* (take something off the stack).

What follows is the definition of a module designed to stack cardinals:

```
DEFINITION MODULE anADTStacks;
```

```
(* a simple semi-generic stack  
  by R. Sutcliffe  
  last modified 1995 05 29  *)
```

```
FROM ItemADT IMPORT
```

```
  ItemType;
```

```
TYPE
```

```
  DataType (* local name *) = ItemType;  
  ActionProc = PROCEDURE (DataType);  
  Stack;    (* details in implementation *)
```

```
PROCEDURE Init (VAR s : Stack) ;
```

```
(* Pre: none  
   Post: Establish an empty stack *)
```

```
PROCEDURE Destroy (s : Stack);
```

```
(* Pre: s is a previously established stack.  
   Post: any memory previously allocated to q is now returned. *)
```

```
PROCEDURE Full (s : Stack) : BOOLEAN;
```

```
(* Pre: s is a previously established stack.  
   Post: if no more items can be stacked, returns TRUE; otherwise returns FALSE. *)
```

```
PROCEDURE Empty (s : Stack) : BOOLEAN;
```

```
(* Pre: s is a previously established stack.
```

```

    Post: if the number of active items in the stack is zero, returns TRUE; otherwise
returns FALSE. *)
PROCEDURE Push (s : Stack; item : DataType);
(* Pre: s is a previously established stack.
    Post: if the stack is not full the given item is added to the stack; otherwise no
action is taken. *)
PROCEDURE Pull (s : Stack; VAR item : DataType);
(* Pre: s is a previously established stack.
    Post: if the number of active items in the stack is greater than zero, an item is
removed from the stack and returned on a last-in-first-out basis; otherwise no action
is taken. *)
PROCEDURE Traverse (s : Stack; Proc : ActionProc);
(* Pre: s is a previously established stack.
    Post: The action specified by Proc is taken on each item in the stack from the
head to the tail without affecting the stack itself. *)

END anADTStacks.

```

The implementation can be with an array as in the example for *TextQueues* or it can be as a linked structure in the same manner as that for *anADTQueues*. In the latter case, the code can be identical, with *Push* substituted for *Enqueue* and *Pull* written like *Serve* but removing things at the other end. There is no need for a tail pointer, only for a head pointer.

[Contents](#)

14.5 Tables

Data is commonly displayed in the form of a table with headings for the columns. The first column can be thought of as an index into the table.

Example 1 (National Data)

<u>Country</u>	<u>Gross National Product</u>
Samovia	\$13 000 000
Xanadu	\$3 000
Lundy	\$42 000
Pompey	\$13 000

Here the names of the countries are used as an index to the table in order to find the gross national products of each.

Example 2 (Numerical function)

x	f(x)
1	8.9
2	4.5
3	6.7
4	5.1

A lookup table or, for short, a table is a finite set of ordered pairs $\{(x, f(x))\}$, that is, it is a function on a finite domain (the first column) to some range (the second column).

Lookup tables are very commonly used in a variety of applications in computing. For instance:

- The WITH statement is usually implemented as a table. The values of the enumeration type are the domain, and the addresses of the statement sequence associated with each form the range of the function.
- A compiler usually has a table with domain the various statements in the language, and range the code to output in order to execute that statement.
- Mail servers and other accounting programs on network machines keep tables of user account names and their passwords. They also have tables that relate the names of machines to their network addresses and to routes by which data packets can be sent to other machines.

There is no particular reason to limit tables to two columns. They could be n-tuples, and be thought of as functions from a finite domain to an (n-1)-tuple. However, to keep the illustrations in this section simple, attention will be confined to the easiest case. In a fashion similar to the one already in use throughout this chapter, first define the data type to be entabled. Here, example 1 above is used.

DEFINITION MODULE Countries;

```
(* define one data type in a generic way
for use in semi-generic structures
```

by R. Sutcliffe last modified 1995 06 07 *)

```
FROM Strings IMPORT
    CompareResults;
CONST
    nameLength = 30;
TYPE
    Country;
    KeyType = ARRAY [0..nameLength] OF CHAR;
    FieldType = CARDINAL;
    ActionProc = PROCEDURE (Country);

PROCEDURE New (VAR c : Country);
PROCEDURE Valid (c : Country) : BOOLEAN;
    (* if new was never called for this country, this call is meaningless; if it was,
    returns true if variable is ok, false if either new failed or dispose has been called
    *)
PROCEDURE Dispose (VAR c : Country);
PROCEDURE Assign (source : Country; VAR dest : Country);
    (* assignment with value semantics *)
PROCEDURE WriteCountryData (c : Country);
PROCEDURE Compare (key1, key2: KeyType) : CompareResults;
PROCEDURE SetKey (c : Country; name : KeyType);
PROCEDURE GetKey (c : Country): KeyType;
PROCEDURE SetField (c : Country; gnp : FieldType);
PROCEDURE GetField (c : Country) : FieldType;
END Countries.
```

Observe that an *Assign* procedure has been defined. One could just write

```
destCountry := sourceCountry;
```

but that kind of assignment is an assignment of the opaque item itself, pointer to the data, so that *destCountry* now points to the same location as *sourceCountry* does. The *Assign* in the module above is an assignment of the value of *source*[^] to the dereferenced location of the destination, so that the value of *dest*[^] is changed, but not the pointer itself. Which kind of assignment is used in a program depends on whether the programmer wishes to move data or change the pointer.

Assignment by changing the pointer is said to have reference semantics (meaning), and assignment by moving the values pointed to without changing the pointers is said to have value semantics.

These ideas can be examined closely in the implementation below. Observe also the definition of the procedure *WriteCountryData* to print out one data item. This will be useful when it comes time to print out the entire table, for this procedure will do one line of such a printout.

```
IMPLEMENTATION MODULE Countries;

(* define one data type in a generic way
for use in semi-generic structures
by R. Sutcliffe    last modified 1995 06 07 *)
```

```
FROM Storage IMPORT
    ALLOCATE, DEALLOCATE;
```

```

FROM STextIO IMPORT
    WriteString, WriteLn, WriteChar;
FROM SWholeIO IMPORT
    WriteCard;
IMPORT Strings;
FROM Strings IMPORT
    CompareResults;

TYPE
    Country = POINTER TO CountryDataNode;
    CountryDataNode =
        RECORD
            name : KeyType;
            gnp : FieldType;
        END;

PROCEDURE New (VAR c : Country);
BEGIN
    NEW (c);
    c^.name := "";
    c^.gnp := 0;
END New;

PROCEDURE Valid (c : Country) : BOOLEAN;
BEGIN
    RETURN (c # NIL);
END Valid;

PROCEDURE Dispose (VAR c : Country);
BEGIN
    DISPOSE (c);
END Dispose;

PROCEDURE Assign (source : Country; VAR dest : Country);
    (* assignment with value semantics *)
BEGIN
    dest^ := source^;
END Assign;

PROCEDURE WriteCountryData (c : Country);
VAR
    count : CARDINAL;
BEGIN
    WriteString (c^.name);
    FOR count := 1 TO 2 + nameLength - LENGTH (c^.name)
        DO (* pad to next field *)
            WriteChar (" ");
        END;
    WriteCard (c^.gnp, 12);
    WriteLn;
END WriteCountryData;

PROCEDURE Compare (key1, key2: KeyType) : CompareResults;
BEGIN

```



```

    RETURN Strings.Compare (key1, key2);
END Compare;

PROCEDURE SetKey (c : Country; name : KeyType);
BEGIN
    c^.name := name;
END SetKey;

PROCEDURE GetKey (c : Country): KeyType;
BEGIN
    RETURN c^.name;
END GetKey;

PROCEDURE SetField (c : Country; gnp : FieldType);
BEGIN
    c^.gnp := gnp;
END SetField;

PROCEDURE GetField (c : Country) : FieldType;
BEGIN
    RETURN c^.gnp;
END GetField;

END Countries.

```

The next step is to define the table type. Here, a module name of *CountryTable* is employed, but the same semi-generic style as before is used. If some other data type is to be entabled in this way, one need only define the type as above, then

- change the definition below to have the name <myADTs
- change the import line to

```

FROM <myADTs><myADT>

```

- change the renaming line to

```

TYPE
    DataType = <myADT> len
    THEN
        len := len2  (* take the maximum of the two *)
    END;
IF t^.title1 [0] # ""
    THEN
        WriteString (t^.title1);
        WriteLn;
    END;
IF t^.title2 [0] # ""
    THEN
        WriteString (t^.title2);
        WriteLn;
    END;
IF len # 0
    THEN
        FOR count := 1 TO len

```

```

        DO (* write a bar below the title *)
            WriteChar ("-");
        END;
        WriteLn;
    END;
END WriteTitle;

PROCEDURE Traverse (t : Table; Proc : ActionProc);
VAR
    point : NodePointer;
BEGIN
    point := t^.top;
    WHILE point # NIL
        DO
            Proc ( point^.item);
            point := point^.toPoint;
        END;
    END Traverse;

END CountryTable.

```

In the test harness below, note that various aspects of the initial ADT module *Countries* are tested, and then the table module is checked extensively to see if the entabling, searching, and removal of items works correctly. A data type such as *Table* should always be checked in this way, with simple data, so as to ensure that the logic is correct, before employing the library for larger types that are more difficult to test. While the author is not so foolish as to guarantee that the libraries above are error free, it is notable that the testing of this carefully designed pair of library modules turned up only one minor logical error--all other changes made during the testing process were of a purely cosmetic nature.

```

MODULE TestCountryTable;

(* program to test the logic of the table library with countries and their GNP
by R. Sutcliffe    modified 1995 06 07 *)

IMPORT
    Countries, CountryTable, STextIO, SWholeIO;

VAR
    table : CountryTable.Table;
    country : Countries.Country;
    str : Countries.KeyType;
    num : Countries.FieldType;
    gotIt : BOOLEAN;

PROCEDURE WriteTable;
BEGIN
    CountryTable.WriteTitle (table);
    CountryTable.Traverse (table, Countries.WriteCountryData);
END WriteTable;

```

BEGIN

```
CountryTable.Create (table,
    "      Table of countries and their GNP",
    "Country Name                               GNP      ");
Countries.New (country); (* just do it once *)
Countries.SetKey (country, "Samovia");
Countries.SetField (country, 13000000);

    (* test some stuff in Countries *)
str := Countries.GetKey (country);
STextIO.WriteString (str);
num := Countries.GetField (country);
SWholeIO.WriteCard (num,10);
STextIO.WriteLine; STextIO.WriteLine;

    (* now get the table filled up *)
CountryTable.Entable (table, country);
WriteTable;
STextIO.WriteLine;
Countries.SetKey (country, "Xanadu");
Countries.SetField (country, 3000);
CountryTable.Entable (table, country);
WriteTable;
STextIO.WriteLine;
Countries.SetKey (country, "Lundy");
Countries.SetField (country, 42000);
CountryTable.Entable (table, country);
WriteTable;
STextIO.WriteLine;
Countries.SetKey (country, "Pompey");
Countries.SetField (country, 13000);
CountryTable.Entable (table, country);
WriteTable;
STextIO.WriteLine; STextIO.WriteLine;

    (* test finds *)
CountryTable.Find(table, "Pompey", country, gotIt);
IF gotIt
    THEN
        str := Countries.GetKey (country);
        STextIO.WriteString ("Got  ");
        STextIO.WriteString (str);
    ELSE
        STextIO.WriteString ("no got Pompey");
    END;
STextIO.WriteLine; STextIO.WriteLine;

CountryTable.Find(table, "Canada", country, gotIt);
IF gotIt
    THEN
        str := Countries.GetKey (country);
        STextIO.WriteString ("Got  ");
        STextIO.WriteString (str);
```

```

ELSE
    STextIO.WriteString ("no got Canada");
END;
STextIO.WriteLine; STextIO.WriteLine;

(* test removes *)
CountryTable.Remove(table, "Pompey", country);
IF CountryTable.TableStatus (table) = CountryTable.allRight
THEN
    STextIO.WriteString ("removed  Pompey");
ELSE
    STextIO.WriteString ("could not remove Pompey");
END;
STextIO.WriteLine; STextIO.WriteLine;

(* now check to ensure its gone *)
CountryTable.Find(table, "Pompey", country, gotIt);
IF gotIt
THEN
    str := Countries.GetKey (country);
    STextIO.WriteString ("Got  ");
    STextIO.WriteString (str);
ELSE
    STextIO.WriteString ("no got Pompey");
END;
STextIO.WriteLine; STextIO.WriteLine;
WriteTable;
STextIO.WriteLine; STextIO.WriteLine;

(* now try to remove something not there *)
CountryTable.Remove(table, "Canada", country);
IF CountryTable.TableStatus (table) = CountryTable.allRight
THEN
    STextIO.WriteString ("removed  Canada");
ELSE
    STextIO.WriteString ("could not remove Canada");
END;
STextIO.WriteLine; STextIO.WriteLine;

(* now remove one not at the start *)
CountryTable.Remove(table, "Xanadu", country);
IF CountryTable.TableStatus (table) = CountryTable.allRight
THEN
    STextIO.WriteString ("removed  Xanadu");
ELSE
    STextIO.WriteString ("could not remove Xanadu");
END;
STextIO.WriteLine; STextIO.WriteLine;
WriteTable;
STextIO.WriteLine; STextIO.WriteLine;

(* now see if destroy seems to work *)
CountryTable.Destroy (table);

```

```

IF CountryTable.TableStatus (table) = CountryTable.bad
THEN
    STextIO.WriteString ("table deleted");
ELSE
    STextIO.WriteString ("could not destroy");
END;
STextIO.WriteLine; STextIO.WriteLine;

END TestCountryTable.

```

When this test was run, the following results were obtained:

**** Run log starts here ****

Samovia 13000000

Table of countries and their GNP

Country Name	GNP
Samovia	13000000

Table of countries and their GNP

Country Name	GNP
Xanadu	3000
Samovia	13000000

Table of countries and their GNP

Country Name	GNP
Lundy	42000
Xanadu	3000
Samovia	13000000

Table of countries and their GNP

Country Name	GNP
Pompey	13000
Lundy	42000
Xanadu	3000
Samovia	13000000

Got Pompey

no got Canada

removed Pompey

no got Pompey

Table of countries and their GNP

Country Name	GNP
--------------	-----

Lundy	42000
Xanadu	3000
Samovia	13000000

could not remove Canada

removed Xanadu

Table of countries and their GNP	
Country Name	GNP

Lundy	42000
Samovia	13000000

table deleted

As the output from this test illustrates, this implementation of a table had the semantics of set membership--a data item either was in the table or it was not. No sorting was done. If sorted tables are desired, the entabbling routine would have to be written accordingly. This is left as an exercise for the reader.

14.6 An Introduction to Table Indexing Methods

A number of other topics related to tables are somewhat more advanced, and will not be considered here in any more detail than will assist the reader in consulting further references. It should be noted, however, that as a table grows very large, the sequential access method implied by the linked structure becomes a hindrance. If the table is sorted, possible solutions include:

- If the maximum size of the table is known ahead of time, it could be implemented as an array to speed access.
- In either case, the table could be indexed and a separate array or linked list contain the indices into the table. For instance a table ordered alphabetically could have a separate index table by the first letter of the key so that a search for "Xanadu" would start at the beginning of the "X" entries.
- One could go further than this and have a separate linked list for each letter of the alphabet. Conceptually, one could regard the data structure as a linked list of linked lists (or of queues or stacks, if appropriate). This could be done for any appropriate index. For instance the employees of an organization could be organized by department. Each department name provides an index to a linked list of the people in that department.

The maintenance of a table of starting points of separate linked lists is called the indexed sequential access method (ISAM), although this term is most commonly used specifically for the indexing of sequential files.

These linked lists could either be kept separate, or they could be chained together into one long list, with the indices being starting points for a logical grouping. In either case, the entire collection is the table.

- Sometimes the indexing into a table is done dynamically by performing some computation based on the key. In the case of alphabetically sorted data, this could be an extension of the last two ideas. For instance, if d is the distance in letters of a letter from "A," then the formula $26 * d(\text{first letter}) + d(\text{second letter})$ which, if applied to "Canada" yields the value 52, could be used to compute a finer index into a table than just using the first letter alone. Of course, "Cambodia" and "Cameroun" yield the same result, so the answer does not uniquely determine the position of an item in the table; it just provides a starting point to begin the search.

Any method of computing directly from the search key at run time an index into a table of data is called hashing; a table indexed in this way is called a hash table; and the function employed for the calculation is called a hash function.

Alphabetically ordered items are not spread evenly through the letters; they are clustered heavily in some parts. For instance, a telephone book may contain many last names "Kim," "Smith," "Wong," and "Friesen," but not very many "Sutcliffe," "Hacker," "Zinzelmeyer," "Szczepanczyk," or "Herod." Moreover, the nature of the clustering would not be the same in every country (not many named "Smith" in China; mostly "Kim" in Korea, and lots of "Sutcliffe" in Yorkshire). Thus, the development of effective and efficient hash functions to index large tables depends very much on the data at hand, and may change over time as the data is maintained. The reader is invited to consult an advanced work on these subjects.

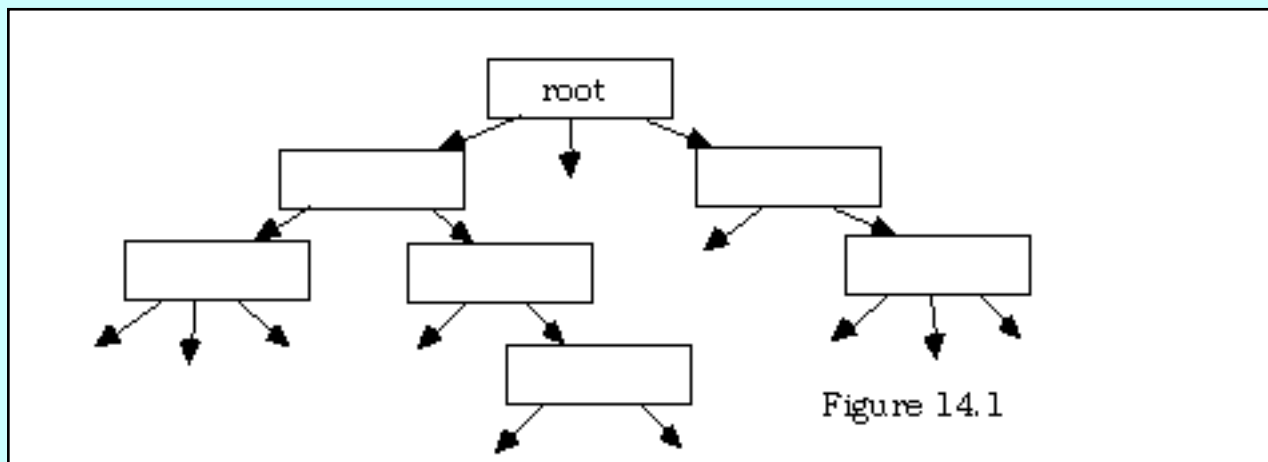
[Contents](#)

14.7 Trees

As with the treatment of sorting, it is natural to ask whether the problems of linear access to data that become evident with linked lists (whether viewed as lists, stacks, queues, or tables) can be overcome by organizing the data in some manner that is not strictly sequential. There is another data structure that can be employed to cut down on the amount of search time considerably; it involves attaching the connecting strands in such a way that a given item (node) in the structure can point to more than one other successor item.

As before, there is a unique first item that has no predecessor in the structure, and a pointer (initially NIL) is kept to this item, but this time the first node is called the *root* rather than the *head*.

A data structure in which all nodes may have more than one successor and all nodes except one (the root) have one predecessor is called a tree.



In figure 14.1, each box represents a node that encapsulates some data elements and some pointers to subsequent nodes. Pointers without nodes attached would be given the value NIL to mark the end of a chain of descent through the tree. In this diagram, some of the boxes have two pointers and some have three, which is all right in an abstract tree in mathematics, but will not do for the data structures here, as one would want all the nodes to be of the same *Node Type*.

So, usually, one works with a small but fixed number of pointers. In the simplest possible case, this number is two. The root item has two pointers, called a *left* and *right* pointer, and the items pointed to in turn each point to at most two more, (also via *left* and *right* pointers) and so on, through possibly several levels. As with other structures, one eventually gets to items with no successors; their pointers are set to *NIL*.

A tree whose nodes each have two pointers to potential subsequent nodes is called a binary tree.

Here are some commonly used terms that apply to all trees, whether binary or not.

The number of pointer links required to move from a given node to the root node is called the level of that node.

The highest level number of any node in a tree is its depth.

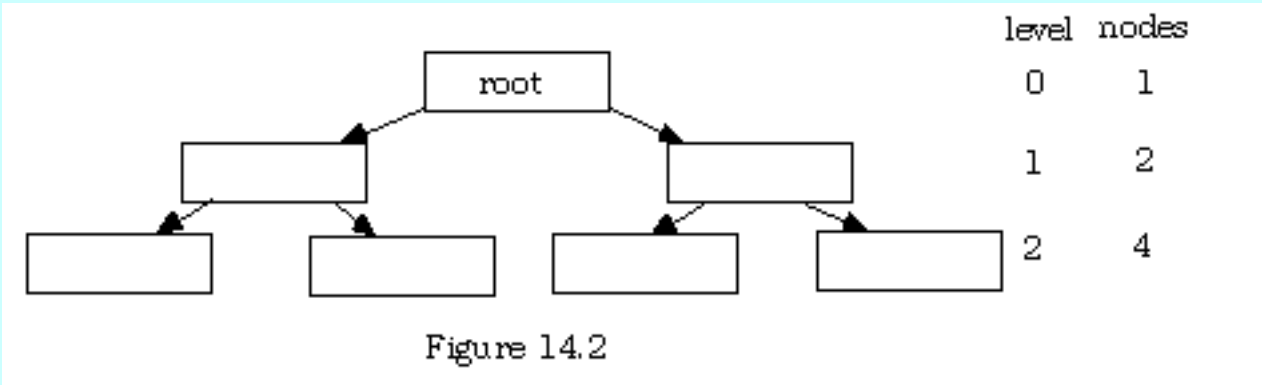
A node with no successors (both pointer values are NIL) is called a leaf.

An immediate successor node is called a child of its predecessor.

An immediate predecessor node is called the parent of its children.

Nodes with a parent-child path of length one or more between them are called ancestors and descendents. The root has no ancestors, and leaves have no children.

The maximum number of children a node can have is called the degree of the node (and hence, by extension, the degree of the tree.)



At level i there are potentially 2^i items, and if there are n levels, then there are at most $(2^{(n+1)}) - 1$ items in the tree altogether. At the last level, there are many NIL pointers, and for simplicity, these have not been shown on this diagram. Of course, the tree may have some positions not filled on one or more levels, but it is possible to characterize a tree that has the maximum number of items actually present by the following:

If all the NIL pointers on level n are NIL (no children) then the tree is full.

Notice that this use of the term *full* is different than earlier uses of the same word. Here, it does not mean that no more items can be added (just start on the next level). Rather it means that all potential positions up to the given level are actively being used. Of course, if the attempt to allocate memory for a new node fails, the tree could be *full* full, not just mathematically or abstractly full.

What is the advantage of using a tree to store data nodes? Observe that a 15-item collection has only four levels when organized this way and a pointer-by-pointer search for data starting at the root will take at

most four pointer assignments, including the first, whereas the same search on strictly linear (un-indexed) data could take fifteen steps. In other words,

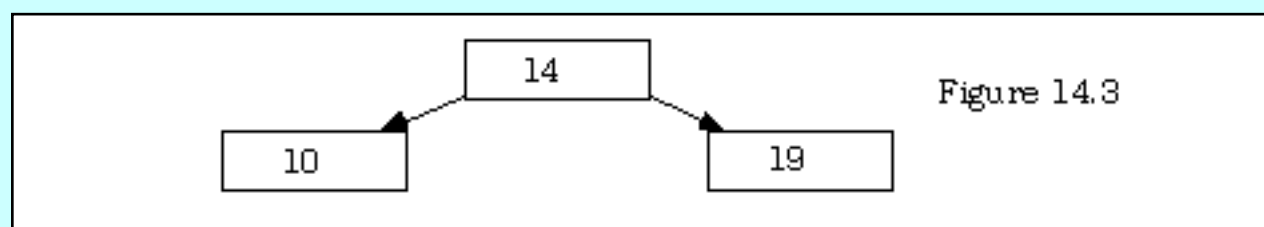
Linear searches are $O(n)$ but searches in binary trees are $O(\log_2 n)$.

All that remains is to develop a rule whereby new data is sorted into a tree. The same rule can be adapted for the traversal of a tree, whether it is to search for an item or to process them all. The most commonly used rule for binary trees is:

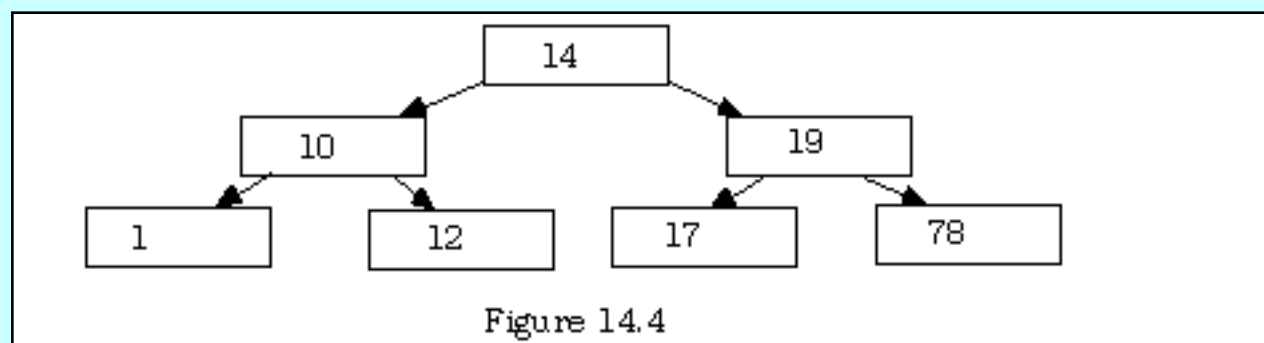
If an item is less, it goes on the left; if it is greater, it goes on the right.

Example:

Data whose key items are the cardinals 4, 10, 19, 1, 78, 17, and 12 arrive to be entered into the binary tree above. The first item is placed in the root node, the second is greater, so it goes into the left child node of the root. The third is greater than the root and goes into its right child, producing figure 14.3 at this stage.



Now the number 1 arrives. It is less than 14 so move left, and less than 10, so move left again. Here there is no node yet, so make one. Likewise, the number 78 goes at the far right, 17 to the right of 14 and then the left of 19, and finally 12 goes to the left of 14 and the right of 10, neatly filling the tree as shown in figure 14.4.



When the tree is traversed for searching or processing, there are three possible orders in which the traversal can be done (recursively).

In-order traversal: left child, parent, right child.

Pre-order traversal: parent, left child, right child.

Post-order traversal: left child, right child, parent.

For instance in the above tree, if the data were processed in-order starting at the root, the processor would postpone the root and go left, postpone that node as a parent and go left again, and so on until reaching a leaf. This leaf would be processed, then its parent, then anything to the right of that parent in the same fashion. Next, processing would back up to the parent of that parent and process to its right (the left is done.) The rightmost child is processed last. This yields (for the tree above)

1, 10, 12, 14, 17, 19, 78

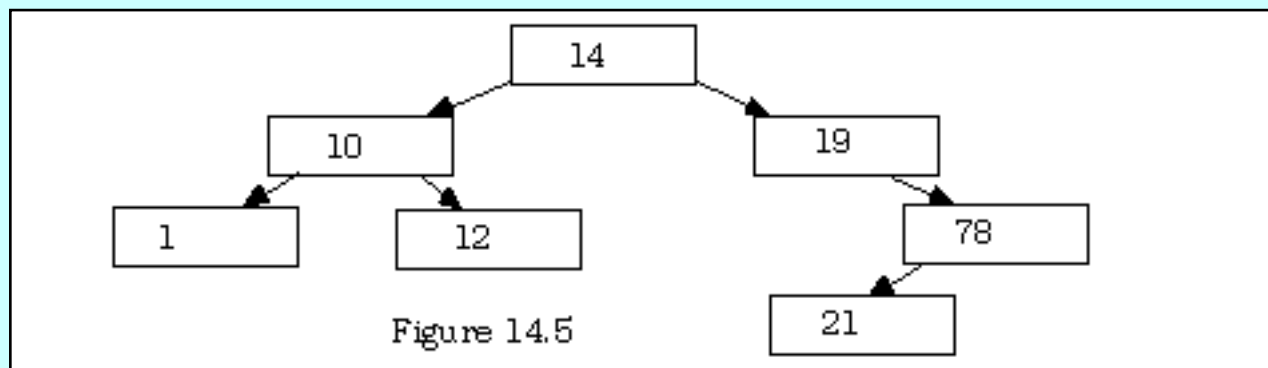
The same tree traversed pre-order would result in processing the nodes in the sequence:

14, 10, 1, 12, 19, 17, 78

The same tree traversed post-order would result in processing the nodes in the sequence:

1, 12, 10, 17, 78, 19, 14

Notice that a slight change to the data results in a rather different tree. For instance suppose the number 17 were changed to the number 21. It would have gone right of the 19 and then left of the 78, producing the tree in figure 14.5.



An in traversal of this tree processes the nodes in the sequence:

1, 10, 12, 14, 19, 21, 78

which is, as above, the correct order from lowest to highest. Indeed, a binary tree filled with the rule "left is less and right is more" can always be traversed in-order to process the elements in their sorted order. There may be times when one of the other two processing orders are more useful, but for sorted binary trees, it seems that only one of the methods is of immediate use. It is useful for the purposes of this work to think of a parent and all of its descendents as a *sub-tree* of the original tree. In so doing, such processing provides a interesting illustration of recursion.

For instance, suppose one had:

TYPE

```
ShortString = ARRAY [0..10] OF CHAR;
```

```
NodePointer = POINTER TO Node;
```

```
Node =
```

RECORD

```
  name : ShortString;
```

```
  (* more data fields here *)
```

```
  leftPoint, rightPoint : NodePointer;
```

END;

```
ActionProc = PROCEDURE (NodePointer);
```

VAR

```
root : Point;
```

Then, assuming the list to have been correctly initialized, the following procedure will take the pointer to any node and process the data at all following nodes (descendents) from the given parent:

```
PROCEDURE InTraverse (parent : NodePointer, Proc :ActionProc);  
BEGIN  
  IF root # NIL  
    THEN  
      InTraverse (parent^.leftPoint, Proc);  
      ActionProc (parent);  
      InTraverse (parent^.rightPoint, Proc);  
    END;  
END PrintTree;
```

If the desired processing involved printing out data elements, a procedure such as :

```
PROCEDURE WriteNode (point : NodePointer);  
BEGIN  
  WITH point^  
    DO  
      WriteString (name);  
      (* print other fields here *)  
    END;  
END WriteNode;
```

could be executed on the entire tree with the call

```
InTraverse (root, WriteNode);
```

and, if necessary, corresponding *PreTraverse* and *PostTraverse* procedures could easily be constructed from *InTraverse* by reordering the three lines. In a more complex situation, one would modify this to print out as much of the data as required at each node before printing the left and right sub-trees. If all this were to be encapsulated in a library, some additional definitions would be necessary to account for the opacity of the type *Tree* and the traverse procedure available to the outside world would be written with a tree parameter rather than with a node pointer parameter.

14.8 An Extended Example--A Binary Search Tree

As in previous examples, the concern here will not be with the data to be placed in the tree, but with the mechanism for implementing the tree structure itself. This can be done in the same semi-generic fashion as in other examples in the text. Since there is already on hand (in the form of the module *Countries*) a suitable data type, the module here is called *CountryBinaryTree*. The usual minor renaming is needed to use any other ADT in the place of *Country*.

In this version, the three kinds of traverse are distinguished via an enumeration, a parameter of which type must be passed to determine which traverse to perform with the procedure acting on the data items.

DEFINITION MODULE CountryBinaryTree;

```
(* semi-generic tree type
by R. Sutcliffe    last modified 1995 06 07 *)
```

FROM Countries **IMPORT**
Country, KeyType, ActionProc;

TYPE
DataType = Country; (* change this line and import as needed *)
TreeState = (allRight, empty, entreeFailed, notFound, bad);
BinaryTree; (* opaque type *)
TraverseOrder = (in, pre, post);

PROCEDURE TreeStatus (t : BinaryTree) : TreeState;
(* Pre : t is a valid initialized table
Post : The State of the tree is returned *)

PROCEDURE Create (**VAR** t : BinaryTree);
(* Pre : none
Post : t is a newly created empty tree. *)

PROCEDURE Insert (t : BinaryTree; data : DataType);
(* Pre : t is a valid initialized tree
Post : memory is obtained and data has been entreed in the proper place for a
binary tree using the **ADT** compare procedure and the state of the tree is allRight or
the entreeing failed and the state is entreeFailed. *)

PROCEDURE Fetch (t : BinaryTree; key : KeyType; **VAR** data : DataType);
(* Pre : t is a valid initialized tree
Post : data matching key is returned in data and the state of the tree is allRight
or the fetch failed and the state is notFound. *)

PROCEDURE Update (t : BinaryTree; data : DataType);
(* Pre : t is a valid initialized tree
Post : data matching the key of the data is updated in the tree and the state of
the tree is allRight or the update failed and the state is notFound. *)

PROCEDURE Remove (t : BinaryTree; key : KeyType; **VAR** data : DataType);
(* Pre : t is a valid initialized tree
Post : data matching key has been removed and returned in data (not disposed of)

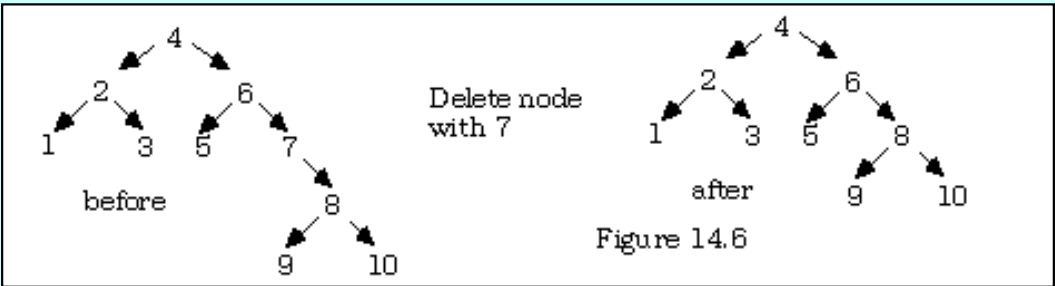
and the state of the tree is allRight or the removal failed and the state is notFound. *)

```
PROCEDURE Destroy (VAR t : BinaryTree);
(* Pre : t is a valid initialized tree
   Post : the tree memory is returned and the variable is invalid
   and the memory associated with the items in the tree is removed by calling the ADT
module dispose procedure. *)

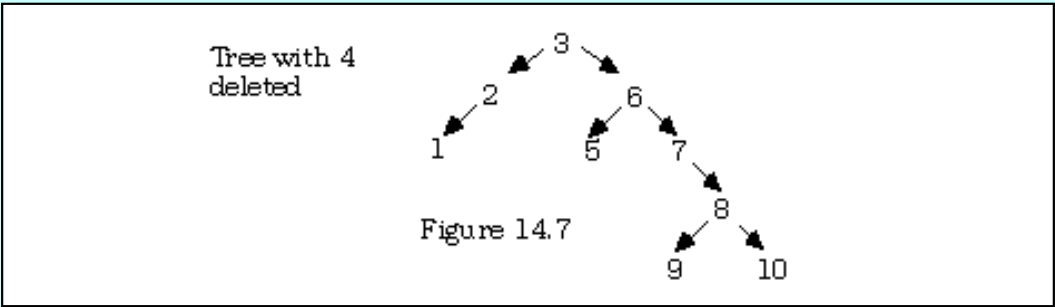
PROCEDURE Traverse (t : BinaryTree; Proc : ActionProc; order : TraverseOrder);
(* Pre : t is a valid initialized tree
   Post : the table items are traversed in the order given and Proc is performed on
each one. *)

END CountryBinaryTree.
```

When this was implemented, a number of local procedures were developed, most of which act on nodes rather than on the data in the nodes. Such actions need to be hidden from the outside world, and separating them even from the abstract (and somewhat generic) procedures that handle data is recommended. The reader should note that insertion of new items always takes place at a leaf, but deletion at other than a leaf position is rather complex. If the node to be deleted has only one subtree hanging from it, then that subtree can be drawn up to take its place, as shown in figure 14.6.



If the deletion is at an interior node (a position with two children), one first finds the predecessor node (in the in order sense, not in the tree structure sense). This node will always be a leaf (why?), and can be found by looking at the left child of the starting node, then going right as far as possible. This predecessor node has its data swapped with the target node, and then the predecessor leaf node is deleted. (This is not the only possible strategy.) For instance, if in the tree on the left of figure 14.6 the node with the data 4 (happens to be the root) were to be deleted, its in order predecessor (go left, then take as many rights as possible) is 3, which is swapped with the 4, and its node deleted, resulting in the structure shown in figure 14.7.



Here is the implementation of the binary tree, with the specific imports for this same data type. To change the implementation, just change the name of this module and of the ADT module being imported from (write it first) appropriately; nothing else needs to be altered.

```
IMPLEMENTATION MODULE CountryBinaryTree;

(* semi-generic tree type
```

by R. Sutcliffe last modified 1995 06 07 *)

FROM Countries **IMPORT**

Country, KeyType, ActionProc, Compare, GetKey, Assign, New, Valid, Dispose;

FROM Storage **IMPORT**

ALLOCATE, DEALLOCATE;

FROM STextIO **IMPORT**

WriteString, WriteLn, WriteChar;

FROM Strings **IMPORT**

CompareResults;

TYPE

NodePointer = **POINTER TO** TreeNode;

TreeNode =

RECORD

item : DataType;

leftPoint, rightPoint, parent : NodePointer;

END;

BinaryTree = **POINTER TO** TreeData;

TreeData =

RECORD

root : NodePointer;

state : TreeState;

END;

NodeProc = **PROCEDURE** (NodePointer);

(* TreeState = (allRight, empty, entableFailed, notFound, bad); *)

(* Here is a collection of local procs used in this module *)

PROCEDURE MakeNode () : NodePointer;

VAR

temp : NodePointer;

BEGIN

NEW (temp); (* get node memory *)

IF temp # **NIL**

THEN

New (temp^.item); (* node OK so get data value memory *)

IF NOT Valid (temp^.item)

THEN (* failed so return **NIL** *)

DISPOSE (temp);

END;

END;

RETURN temp;

END MakeNode;

PROCEDURE InsertNode (**VAR** root : NodePointer; newNode : NodePointer);

VAR

point : NodePointer;

done : **BOOLEAN**;

BEGIN

IF root = **NIL**

THEN (* first item *)

root := newNode;


```

    newNode^.parent := NIL;
ELSE
    point := root;
    done := FALSE;
    REPEAT
        IF Compare(GetKey(newNode^.item), GetKey (point^.item)) = greater
            THEN
                IF point^.rightPoint = NIL (* at end *)
                    THEN
                        point^.rightPoint := newNode;
                        done := TRUE;
                    ELSE
                        point := point^.rightPoint
                    END; (* if point *)
                ELSE (* less or equal *)
                    IF point^.leftPoint = NIL (* at end *)
                        THEN
                            point^.leftPoint := newNode;
                            done := TRUE;
                        ELSE
                            point := point^.leftPoint
                        END; (* if point *)
                    END; (* if compare *)
            UNTIL done;
        newNode^.parent := point;
    END; (* if root *)
END InsertNode;

PROCEDURE Find (root : NodePointer; key : KeyType; VAR point : NodePointer);
(* get a pointer to the node belonging to the key. Returns NIL if not found *)
BEGIN
    IF (root = NIL) (* recursion trapdoor *)
        OR (Compare (key, GetKey (root^.item)) = equal) (* got it *)
        THEN
            point := root;
            RETURN;
        END; (* if root *)
    Find (root^.leftPoint, key, point);
    (* if we get it, we don't want to look to the right at all *)
    IF point = NIL (* not found yet *)
        THEN
            Find (root^.rightPoint, key, point);
        END; (* if point *)
    END Find;

PROCEDURE InOrderPredPoint (node: NodePointer) : NodePointer;
(* Find pointer to Inorder predecessor, i.e. to the rightmost node in left subtree
Pre: the node has a left child
Post: a pointer to its in order predecessor leaf is returned *)
VAR
    pred : NodePointer;
BEGIN
    pred := node^.leftPoint; (* one left *)

```

```

WHILE pred^.rightPoint # NIL
    DO (* go as far right as possible *)
        pred := pred^.rightPoint;
    END; (* while *)
    RETURN pred;
END InOrderPredPoint;

PROCEDURE SwapNodeVal (VAR a, b : NodePointer);
VAR
    temp : DataType;
BEGIN
    temp := a^.item;
    a^.item := b^.item;
    b^.item := temp
END SwapNodeVal;

PROCEDURE Delete (t : BinaryTree; node : NodePointer);
(* delete a node *)
VAR
    temp : NodePointer;
BEGIN
    temp := node;
    IF temp^.leftPoint = NIL
    THEN (* empty left branch *)
        IF temp^.rightPoint = NIL (* I am a leaf *)
        THEN
            IF temp^.parent = NIL (* I am root too *)
            THEN
                t^.root := NIL;
            ELSE (* just leaf *)
                IF temp^.parent^.leftPoint = temp
                THEN
                    temp^.parent^.leftPoint := NIL
                ELSE
                    temp^.parent^.rightPoint := NIL
                END;
            END;
            KillNode (temp);
            RETURN;
        ELSE (* not a leaf so pull up right subtree *)
            node := node^.rightPoint;
            KillNode (temp);
            RETURN;
        END;
    ELSIF temp^.rightPoint = NIL THEN (* empty right branch *)
        node := node^.leftPoint; (* so pull up left subtree *)
        KillNode (temp);
        RETURN;
    ELSE (* no branch empty, find inorder predecessor *)
        temp := InOrderPredPoint (node);
        SwapNodeVal (node, temp);
        Delete (t, temp); (* recursively remove node swapped *)
    END; (* if *)

```

```
END Delete;
```

```
PROCEDURE KillNode (VAR node : NodePointer);  
(* give back all memory associated with node *)  
BEGIN  
  IF node # NIL  
    THEN  
      Dispose (node^.item);  
      DISPOSE (node);  
    END;  
END KillNode;
```

```
PROCEDURE Erase (VAR r : NodePointer);  
(* Pre: r is the root of a subtree  
  Post: recursive post traverse killing all nodes *)  
BEGIN  
  IF r # NIL  
    THEN  
      Erase (r^.leftPoint);  
      Erase (r^.rightPoint);  
      KillNode (r);  
    END;  
END Erase;
```

```
(* end local procs *)
```

```
PROCEDURE TreeStatus (t : BinaryTree) : TreeState;  
BEGIN  
  IF t # NIL  
    THEN  
      RETURN t^.state;  
    ELSE  
      RETURN bad;  
    END;  
END TreeStatus;
```

```
PROCEDURE Create (VAR t : BinaryTree);  
BEGIN  
  NEW (t);  
  t^.root := NIL;  
  t^.state := empty;  
END Create;
```

```
PROCEDURE Insert (t : BinaryTree; data : DataType);  
VAR  
  temp : NodePointer;  
  state : TreeState;  
BEGIN  
  state := TreeStatus (t);  
  IF (state = bad) OR (state = entreeFailed)  
    THEN  
      t^.state := entreeFailed;  
      RETURN  
    END;
```

```
temp := MakeNode (); (* status ok so get node memory *)
```

```
IF temp = NIL
```

```
THEN
```

```
    t^.state := entreeFailed;
```

```
    RETURN
```

```
END;
```

```
    (* all OK so put it together *)
```

```
Assign (data, temp^.item); (* move data value in *)
```

```
temp^.leftPoint := NIL; (* always adding a leaf *)
```

```
temp^.rightPoint := NIL;
```

```
InsertNode (t^.root, temp);
```

```
t^.state := allRight;
```

```
END Insert;
```

```
PROCEDURE Fetch (t : BinaryTree; key : KeyType; VAR data : DataType);
```

```
VAR
```

```
    point : NodePointer;
```

```
BEGIN
```

```
    IF t = NIL
```

```
    THEN
```

```
        t^.state := bad;
```

```
        RETURN
```

```
    ELSE
```

```
        Find (t^.root, key, point);
```

```
        IF point = NIL
```

```
        THEN
```

```
            t^.state := notFound;
```

```
        ELSE
```

```
            t^.state := allRight;
```

```
            data := point^.item;
```

```
        END; (* if point *)
```

```
    END; (* if t *)
```

```
END Fetch;
```

```
PROCEDURE Update (t : BinaryTree; data : DataType);
```

```
VAR
```

```
    point : NodePointer;
```

```
BEGIN
```

```
    IF t = NIL
```

```
    THEN
```

```
        t^.state := notFound;
```

```
        RETURN
```

```
    ELSE
```

```
        Find (t^.root, GetKey (data), point);
```

```
        IF point # NIL
```

```
        THEN
```

```
            t^.state := allRight;
```

```
            point^.item := data;
```

```
        END;
```

```
    END;
```

```
END Update;
```

```
PROCEDURE Remove (t : BinaryTree; key : KeyType; VAR data : DataType);
```

```

VAR
    point : NodePointer;
BEGIN
    IF t = NIL
    THEN
        t^.state := bad;
        RETURN
    ELSE
        Find (t^.root, key, point);
        IF point = NIL
        THEN
            t^.state := notFound;
        ELSE
            t^.state := allRight;
            data := point^.item;
            Delete (t, point);
        END;
    END;
END Remove;

PROCEDURE Destroy (VAR t : BinaryTree);
BEGIN
    Erase (t^.root); (* all nodes *)
    DISPOSE (t); (* tree data *)
END Destroy;

(* local procs: three ways to traverse a sub-tree *)

PROCEDURE InTraverse (r : NodePointer; Proc : ActionProc);
BEGIN
    IF r = NIL (* recursion trap door *)
    THEN
        RETURN
    END;
    InTraverse (r^.leftPoint, Proc);
    Proc (r^.item);
    InTraverse (r^.rightPoint, Proc);
END InTraverse;

PROCEDURE PreTraverse (r : NodePointer; Proc : ActionProc);
BEGIN
    IF r = NIL (* recursion trap door *)
    THEN
        RETURN
    END;
    Proc (r^.item);
    PreTraverse (r^.leftPoint, Proc);
    PreTraverse (r^.rightPoint, Proc);
END PreTraverse;

PROCEDURE PostTraverse (r : NodePointer; Proc : ActionProc);
BEGIN
    IF r = NIL (* recursion trap door *)

```

```

    THEN
        RETURN
    END;
PostTraverse (r^.leftPoint, Proc);
PostTraverse (r^.rightPoint, Proc);
Proc (r^.item);
END PostTraverse;

(* end local procs *)

PROCEDURE Traverse (t : BinaryTree; Proc : ActionProc; order : TraverseOrder);
BEGIN
    IF t = NIL
    THEN
        RETURN
    END;
    CASE order OF
        in:
            InTraverse (t^.root, Proc) |
        pre:
            PreTraverse (t^.root, Proc) |
        post:
            PostTraverse (t^.root, Proc)
    END;
END Traverse;

END CountryBinaryTree.

```

As before, a simple test harness is provided. In order to ensure that all aspects of the library were tested, it contains (in, pre, and post) procedures to traverse the tree and to write out enough of the data from the items being entreed to ensure that the structure is correctly maintained.

This module is in the style favoured by some that employs only unqualified import. As can readily be seen, such a style tends to become cumbersome as the module names grow.

```

MODULE TestCountryBinaryTree;

```

```

(* program to test the logic of the Tree library with countries and their gnp
by R. Sutcliffe    modified 1995 06 01 *)

```

```

IMPORT
    Countries, CountryBinaryTree, STextIO, SWholeIO;

```

```

VAR
    Tree : CountryBinaryTree.BinaryTree;
    country, fetched : Countries.Country;
    str : Countries.KeyType;
    num : Countries.FieldType;
    gotIt : BOOLEAN;

```

```

PROCEDURE WriteCountryName (c : Countries.Country);
BEGIN
    STextIO.WriteString (Countries.GetKey (c));
    STextIO.WriteChar (" ");

```

```

END WriteCountryName;

PROCEDURE WriteTree;  (* all data *)
BEGIN
    CountryBinaryTree.Traverse (Tree, Countries.WriteCountryData,
CountryBinaryTree.in);
END WriteTree;

(* these three just write the names *)

PROCEDURE WriteTreeIn;
BEGIN
    CountryBinaryTree.Traverse (Tree, WriteCountryName, CountryBinaryTree.in);
END WriteTreeIn;

PROCEDURE WriteTreePre;
BEGIN
    CountryBinaryTree.Traverse (Tree, WriteCountryName, CountryBinaryTree.pre);
END WriteTreePre;

PROCEDURE WriteTreePost;
BEGIN
    CountryBinaryTree.Traverse (Tree, WriteCountryName, CountryBinaryTree.post);
END WriteTreePost;

PROCEDURE WriteTreeAll;
BEGIN
    STextIO.WriteString ("InOrder  : ");WriteTreeIn; STextIO.WriteLine;
    STextIO.WriteString ("PreOrder : ");WriteTreePre; STextIO.WriteLine;
    STextIO.WriteString ("PostOrder: ");WriteTreePost; STextIO.WriteLine;
END WriteTreeAll;

PROCEDURE TestFetch (name : Countries.KeyType);
BEGIN
    CountryBinaryTree.Fetch (Tree, name, fetched);
    gotIt := (CountryBinaryTree.TreeStatus (Tree) # CountryBinaryTree.notFound);
    IF gotIt
        THEN
            str := Countries.GetKey (fetched);
            STextIO.WriteString ("Got  ");
            STextIO.WriteString (str);
        ELSE
            STextIO.WriteString ("no got ");
            STextIO.WriteString (name);
        END;
    STextIO.WriteLine; STextIO.WriteLine;
END TestFetch;

BEGIN
    Countries.New (country);  (* do only once *)
    CountryBinaryTree.Create (Tree);

    (* test Fetch; should fail *)
    CountryBinaryTree.Fetch (Tree, "Xanadu", fetched);

```

```

gotIt :=
  (CountryBinaryTree.TreeStatus (Tree) # CountryBinaryTree.notFound);
IF gotIt
  THEN
    str := Countries.GetKey (fetched);
    STextIO.WriteString ("Got  ");
    STextIO.WriteString (str);
  ELSE
    STextIO.WriteString ("no got Xanadu");
  END;
STextIO.WriteLine; STextIO.WriteLine;

  (* now get the Tree filled up *)
Countries.SetKey (country, "Samovia");
Countries.SetField (country, 13000000);
CountryBinaryTree.Insert (Tree, country);
Countries.SetKey (country, "Xanadu");
Countries.SetField (country, 3000);
CountryBinaryTree.Insert (Tree, country);
Countries.SetKey (country, "Lundy");
Countries.SetField (country, 42000);
CountryBinaryTree.Insert (Tree, country);
Countries.SetKey (country, "Pompey");
Countries.SetField (country, 13000);
CountryBinaryTree.Insert (Tree, country);
Countries.SetKey (country, "Alberta");
Countries.SetField (country, 43000);
CountryBinaryTree.Insert (Tree, country);
Countries.SetKey (country, "Yesterday");
Countries.SetField (country, 11000);
CountryBinaryTree.Insert (Tree, country);
Countries.SetKey (country, "Yahk");
Countries.SetField (country, 3000);
CountryBinaryTree.Insert (Tree, country);
Countries.SetKey (country, "Toronto");
Countries.SetField (country, 0);
CountryBinaryTree.Insert (Tree, country);
WriteTreeAll;

  (* test Fetchs *)
TestFetch ("Xanadu"); (* should be ok *)
TestFetch ("Pompey"); (* should be ok *)
TestFetch ("Canada"); (* should Not be ok *)
TestFetch ("Toronto"); (* should be ok *)

  (* test update *)
STextIO.WriteString ("Before Update");
STextIO.WriteLine;
WriteTree;
STextIO.WriteLine;
Countries.SetField (country, 10); (* should still be on Toronto *)
CountryBinaryTree.Update (Tree, country);
STextIO.WriteString ("After Update");

```



```

STextIO.WriteLine;
WriteTree;
STextIO.WriteLine;

    (* test removes *)
CountryBinaryTree.Remove (Tree, "Pompey", fetched);
IF CountryBinaryTree.TreeStatus (Tree) = CountryBinaryTree.allRight
THEN
    STextIO.WriteString ("removed  Pompey");
ELSE
    STextIO.WriteString ("could not remove Pompey");
END;
STextIO.WriteLine; STextIO.WriteLine;
    (* now check to ensure its really gone *)
TestFetch ("Pompey"); (* should Not be ok *)
STextIO.WriteString ("after Pompey removal:");
STextIO.WriteLine;
WriteTreeIn;
STextIO.WriteLine; STextIO.WriteLine;

    (* now try to remove something not there *)
CountryBinaryTree.Remove (Tree, "Canada", fetched);
IF CountryBinaryTree.TreeStatus (Tree) = CountryBinaryTree.allRight
THEN
    STextIO.WriteString ("removed  Canada");
ELSE
    STextIO.WriteString ("could not remove Canada");
END;
STextIO.WriteLine; STextIO.WriteLine;

    (* now remove one at an interior node *)
CountryBinaryTree.Remove (Tree, "Xanadu", fetched);
IF CountryBinaryTree.TreeStatus (Tree) = CountryBinaryTree.allRight
THEN
    STextIO.WriteString ("removed  Xanadu");
ELSE
    STextIO.WriteString ("could not remove Xanadu");
END;
STextIO.WriteLine; STextIO.WriteLine;
STextIO.WriteString ("after Xanadu removal ");
STextIO.WriteLine;
WriteTreeIn;
STextIO.WriteLine; STextIO.WriteLine;

    (* now see if destroy seems to work *)
CountryBinaryTree.Destroy (Tree);
IF CountryBinaryTree.TreeStatus (Tree) = CountryBinaryTree.bad
THEN
    STextIO.WriteString ("Tree deleted");
ELSE
    STextIO.WriteString ("could not destroy");
END;
STextIO.WriteLine; STextIO.WriteLine;

```

END TestCountryBinaryTree.

A run of the above test harness yielded the following results. The reader should check these results against the expected ones.

** Run log starts here **
no got Xanadu

InOrder : Alberta Lundy Pompey Samovia Toronto Xanadu Yahk Yesterday
PreOrder : Samovia Lundy Alberta Pompey Xanadu Toronto Yesterday Yahk
PostOrder: Alberta Pompey Lundy Toronto Yahk Yesterday Xanadu Samovia
Got Xanadu

Got Pompey

no got Canada

Got Toronto

Before Update	
Alberta	43000
Lundy	42000
Pompey	13000
Samovia	13000000
Toronto	0
Xanadu	3000
Yahk	3000
Yesterday	11000

After Update	
Alberta	43000
Lundy	42000
Pompey	13000
Samovia	13000000
Toronto	10
Xanadu	3000
Yahk	3000
Yesterday	11000

removed Pompey

no got Pompey

after Pompey removal:
Alberta Lundy Samovia Toronto Xanadu Yahk Yesterday

could not remove Canada

removed Xanadu

after Xanadu removal
Alberta Lundy Samovia Toronto Yahk Yesterday

Tree deleted

Contents

14.9 Chapter Summary

This chapter covered these topics:

- issues in the construction of lists
- the use of semi-generic methods in structures
- software queues and stacks
- tables
- trees
- binary search trees
- recursive pre, in, and post traversal of trees
- the testing of software structures

No new reserved words, standard identifiers, symbols or imports were employed in this chapter.

[Contents](#)

14.10 Assignments

Questions

1. What does the term *semi-generic* mean?
2. What are the principal issues that need to be resolved before implementing a list ADT?
3. What is the difference between a queue and a stack?
4. What is a table?
5. The table was implemented using a linked list structure. What is it not correct to assert that a table is a linked list?
6. What is a tree?
7. What is a binary tree?
8. What does a root lack that every other node has?
9. What do leaves lack that other nodes have?
10. What does an interior node always have?
11. What is the degree of a node in a binary tree?
12. A binary tree with eight levels is full. What is the number of occupied nodes?
13. Explain the two different senses of the word *full* when applied to a binary tree.
14. Why is a binary search tree a more efficient data structure than a linked list when it comes to searching through large data aggregates?
15. Both a two way linked list and a binary tree have two node pointers. What is the difference between these structures?
16. What does the term ISAM mean?
17. What is "hashing?"
18. Describe the three different forms of tree traversal.
19. Suppose a tree were *ternary* (degree three). How many different ways would there be of traversing it?
20. In the entable procedure of the module *CountryTable*, suppose the line *Assign (data, temp^.item);* is replaced with the line *temp^.item := data;* What difference will this make? Predict what will happen, then try it.
21. In the insert procedure of the module *CountryBinaryTree*, suppose the line *Assign (data, temp^.item);* is replaced with the line *temp^.item := data;* What difference will this make? Predict what will happen, then try it.
22. Carefully examine and complete a detailed commenting of the implementation module [Lists](#) in section 14.2.
23. Carefully examine and complete a detailed commenting of the implementation modules [TextQueues](#) and [AnAdtQueues](#) in section 14.3.
24. Carefully examine and complete a detailed commenting of the implementation module [CountryTable](#) in section 14.5.

25. Carefully examine and complete a detailed commenting of the implementation module [*CountryBinaryTree*](#) in section 14.8.

Problems

26. You have a stamp collection to keep track of. Implement an abstract data type *Stamp* with fields for the country name, year of issue, name of stamp, condition, and current value. Now implement a list structure of these using the simple modifications to the list type discussed in this chapter.
27. Implement and test the module [*anADTStacks*](#) in section 14.4.
28. Revise (and test) the *Table* data type as a *SortedTable* type. Invent your own data type to entable.
29. A *priority queue* or *sorted queue* incorporates new items into the queue in a sorted fashion according to the value of some field. Serving is still done at the head of the queue as before. Modify the semi-generic queue in this chapter to be a priority queue for some key field.
30. Re-implement the text queue using a linked list of characters.
31. Re-implement *anAdtQueues* without the need to obtain a data node ahead of time, finding another way to report the error and determine the full condition. Hint: look at the other ADTs in this chapter.
32. A *dequeue* or double-ended queue allows items to be entered and served at either end. Thus it has *headEnqueue*, *tailEnqueue*, *headServe*, and *tailServe* routines. Implement and test this structure.
33. A *stack pair* is a pair of stacks in a single structure. Implement this as an array with one stack pointer crawling forward from zero as items are pushed and back as they are pulled and the other stack pointer moving back from the maximum index as items are pushed and forward as they are pulled. Thus, it has *push1*, *pull1*, *push2*, and *pull2* routines. Both stacks are full (only one procedure needed) if the two pointers collide. Test your implementation.
34. Adapt the example in section 12.2 of the last chapter to the semi-generic methods of this chapter and create a linked list with two sets of links to enlist data consisting of people's names and their ages in two different sorted orders.
35. Another way of solving the last problem is to have only one set of links in the linked lists but two keys for sorting and two of the lists. Solve the problem this way.
36. Develop an airline flight reservation list. The passenger records should have last name, first name, smoker or non-smoker, and final destination. Set it up to read in any passenger's information and print out flight lists in alphabetical order and by destination. For each flight you should be able to select from a menu that allows you to add a passenger, delete a passenger, or print a list. Each flight is a list, so a main menu will have to control starting and ending flights.
37. Represent a polynomial as a linked list where each term is of the following type:

TYPE

Term =

RECORD

toPoint : Point;
coefficient : INTEGER;
exponent : CARDINAL

END

Each linked list is one polynomial and has as many records in the chain as there are terms in the polynomial. Write and test the routines to add, subtract, and multiply such polynomials, and to display the result.

38. Write a program module that will take data input from the keyboard and sort it into a tree. Each record tree consists of a person's name, age, sex, and whether they eat raspberry ice cream or not. Test your module by having your application program write it all out in alphabetical order by last name.

39. Add to the traversals in the semi-generic binary tree *reverse* in, pre, and post order traversals. Test all three.

40. Write a module to implement a data base to keep track of your library. The important information includes the publisher, author, title, subject, and ISBN number. Your data structure(s) must keep track of each of these in alphanumeric order, sorting items as they are read in from the keyboard or from the disk file where they are stored. Routines must be available to add to the list, store and retrieve via the disk and print it out on the screen or printer in order by selected category. Remember, sorting is to be done at input time, not at the printing stage. All five fields will be printed in any case; only the order is to be different.

41. Write a program similar in structure to the above, but designed to keep an inventory--say of chemicals, equipment, or merchandise for sale. Appropriate data items might include price paid, stock number, name of item, sale price, and date of acquisition. The print routines should output the data in a neat table ordered on one of four (or more) of the fields.

Projects

42. Look up and implement AVL balanced trees as an ADT.

43. Develop an ADT *FamilyTree* that allows you to express family relationships. Include the capability to express successive marriages for one person.

44. Define and implement an ADT *Table* for data that has two key types and three other data fields.

[Contents](#)

Chapter 14

Intermediate Data Structures

- [14.0 Chapter Goals](#)
 - [14.1 Introduction to Intermediate Data Structures](#)
 - [14.2 Lists Revisited](#)
 - [14.3 Queues](#)
 - [14.4 Stacks](#)
 - [14.5 Tables](#)
 - [14.6 An Introduction to Table Indexing Methods](#)
 - [14.7 Trees](#)
 - [14.8 An Extended Example--A Binary Search Tree](#)
 - [14.9 Chapter Summary](#)
 - [14.10 Assignments](#)
-

[Contents](#)

15.0 Chapter Goals

The purpose of this chapter is to develop some more advanced data structure ideas, to explore some of the problems and pitfalls of using pointers, and to provoke the reader to think about the issues raised by attempts to make software as generic as possible. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

General:

Handles, B-trees and Heaps are defined in Chapter 15.

Realized in the Modula-2 notation:

B-trees and Heaps are implemented, and a probable (implementation defined) implementation of handles is discussed.

Data Manipulation Abstractions

General:

The problem of generic handling of data and algorithms is revisited and pointers are discussed again at some length.

Realized in the Modula-2 notation:

the structure of B-Trees and Heaps and the heapsort

Programming Abstractions

General:

Memory fragmentation and garbage are discussed, and some possible solutions to these problems are explored. Programming for genericity is emphasized.

Realized in the Modula-2 notation:

Some additional genericity is achieved using ISO Standard Modula-2.

15.1 B-trees

It is important to note that the "B" in the name *B-tree* does NOT stand for "Binary" but for "*Balanced*." Confusing the two is a common beginners' mistake. The reader should also be warned that there are other kinds of "balanced" trees than B-trees.

15.1.1 B-trees Defined

The trees considered so far have always had one (potential) data items in each node, and each node had the same number of (potential) descendents. By contrast, B-trees may have more than one data item in a node, and the number of descendents of a node depends on whether the node is a root or not and how large the tree happens to be. B-trees are wider than binary trees, and not as deep. They are automatically kept balanced by the use of carefully constructed insertion and deletion algorithms so that the height of all the leaf elements is the same. This assists in such tasks as searching for data. Each B-tree has a number associated with it, called its order, that determines the maximum amount of data in a node.

A B-tree of order n has - a root node containing from 1 to $2n$ elements - from n to $2n$ elements in all other nodes - from 0 to $k+1$ descendents for a node having k elements - all leaves at the same level.

Conceptually, each node contains a number k of data items as well as the linkage to $k+1$ other nodes. It could be thought of as a sequence

$p_0, d_1, p_1, d_2, p_2, d_3, \dots, d_n, p_n, \dots, d_{2n}, p_{2n}$

where each p_i is pointer to a successor node and each d_i is a data item. Only the first k data positions and $k+1$ pointer positions are in use at a given time. Observe that if all $2n$ data positions are occupied, there are $2n+1$ descendent nodes. Several implementations are possible. One could have, for instance:

```
CONST
    min = order;
    max = 2*order;

TYPE
    range = [1 .. max];
    BTree = POINTER TO BNode;
    NodeItem =
        RECORD
            data : ItemType;
            rPoint : BTree;
        END;
    BTreeNode =
        RECORD
            lPoint : BTree; (* the zeroth pointer; left of first item *)
            data : ARRAY Range OF NodeItem;
            lastUsed : Range;
        END;
```

or, still using arrays:

```
TYPE
    dRange = [1 .. max];
    pRange = [1 .. max+1];
```

```

BTree = POINTER TO BNode;
BTreeNode =
RECORD
    successor : ARRAY Range OF BTree;
    data : ARRAY Range OF ItemType;
    lastUsed : Range;
END;

```

Alternately, each node could be constructed as a linked list of pointers and data.

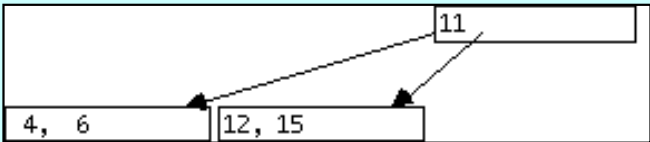
In order to enter the data into a B-tree, one uses the same "left-is-less and right-is-greater" rule as for binary search trees. Moreover, the data in each node is kept sorted from lowest to highest. The B-tree is kept balanced by splitting nodes when they become full. To illustrate, suppose a B-tree of order 2 (maximum data items per node is four) has cardinal data arriving in the order:

15, 4, 6, 12, 11, 17, 20, 30, 31, 5, 16, 37

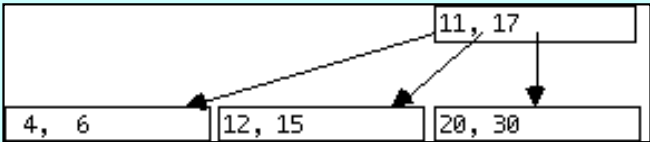
The first four items are sorted into the root node until it looks like:

4, 6, 12, 15

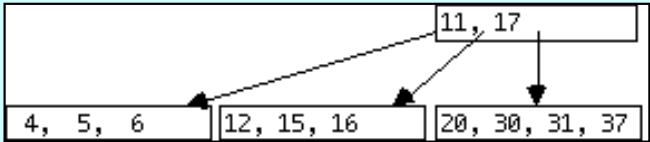
No successor pointers are shown yet, because none are needed. The arrival of the next item causes the middle one of the five to be passed up a level (occupying a new root) and the node with the other four to be split in two. Everything is arranged with the "left-is-less and right-is-greater" rule and the tree now looks like this:



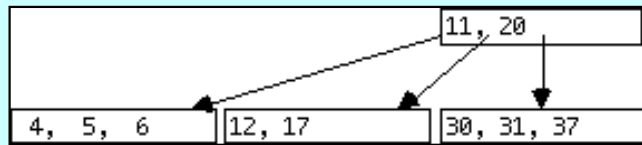
The next three items (17, 20, and 30) all go into the second node, and on the arrival of the number 30, it splits as well, promoting the middle item up to its parent. Since there is room for it there, a new root need not be created. If there is not room for the promoted item, a split may take place on that level and an item (not necessarily the same one) promoted up one more level. Such splits and promotions either stop at a level where there is room, or propagate all the way to the root, and are either entered there or cause a new root to be created.



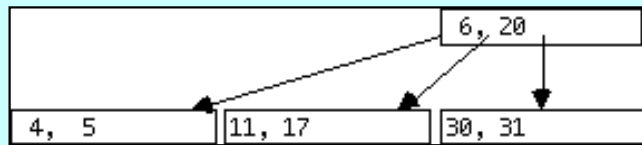
The last three items (31, 5, 16, and 37) contribute to the filling of the nodes on the second level, but do not cause any more splits.



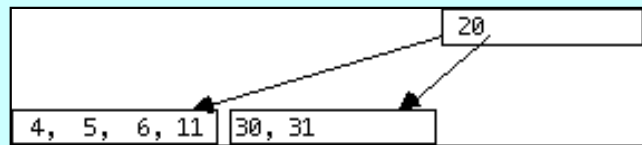
To maintain the shallow structure of the B-tree, deletion of data items must proceed conversely. If a node with more than n items has one deleted, the structure remains the same. However, if a deletion would cause the number of data items in a node to become fewer than the minimum n (the order) then the node is combined with the node on its left or right and the parent data item separating them is pulled down. If the combined number items is exactly $2n$, the deletion is finished. If it is more, the node is split, causing a different item than before to be promoted back up. For instance, assuming that both the 15 and 16 were deleted, and the rule is "combine with the node containing the larger data if you can", the result would be:



On a further deletion of the items 37 and then 12, its node finds there are only n items in the node with greater data, so it borrows from the one with the lesser data instead, producing:



If 17 is deleted, its node cannot borrow from the node with the greater data or the lesser data (in that order), so it combines with the one with the lesser data and the result is:



If the last item in the root node is ever deleted, the two successor nodes are combined (and then split if necessary). If the result is a single node of $2n$ elements, it becomes the root.

15.1.2 Defining A Semi-Generic B-tree

Since in "real life" the data type to be entered will not likely be as simple as a cardinal, provision is made in the following for the easy replacement of the type to be entered. The illustration above seems to indicate that inorder traversal is the only one worth using, so no provision is made for preorder and postorder traversals.

DEFINITION MODULE BTrees;

```
( *****
  Design by R. Sutcliffe copyright 1995-1996
  Initial Code by Gordon Tisher
  June 9, 1995; last modified June 23, 1995
  Modified by R. Sutcliffe 1996 10 11
  This module provides a B-Tree ADT.
  ***** )
```

FROM DataADT **IMPORT**

KeyFieldType, DataType, ActionProc;

(* rename the imported types to the local ones. This makes things more generic and easier to change. *)

TYPE

ItemType = DataType;

KeyType = KeyFieldType;

CONST

order = 2;

TYPE

BTree;

BTreeState = (allRight, empty, entreeFailed);

```

PROCEDURE Status (tree : BTree) : BTreeState;
(* Pre : t is a valid initialized BTree
   Post : The State of the tree is returned *)

PROCEDURE Init (VAR tree : BTree);
(* Allocates memory for a new tree sets state to allRight *)

PROCEDURE Destroy (VAR tree : BTree);
(* disposes the whole tree *)

PROCEDURE Add (VAR tree : BTree; data : ItemType);
(* Adds a new item to the tree. If successful sets state to allRight, else to
entreeFailed *)

PROCEDURE Delete (VAR tree : BTree; key : KeyType);
(* deletes an item whose key is defined equal to _key_ from the tree. If successful
sets state to allRight; if tree was empty sets state to empty *)

PROCEDURE Depth (tree : BTree) : CARDINAL;
(* returns the number of levels in the tree *)

PROCEDURE Search (tree : BTree; key : KeyType; VAR data : ItemType) : BOOLEAN;
(* if found, returns TRUE and _data_ returns item at that point  *)

PROCEDURE WriteTree (tree : BTree);
(* writes out the tree. This is for testing purposes and likely would not be
included in a finished product. *)

PROCEDURE Traverse (tree : BTree; Proc : ActionProc);
(* Pre : tree is a valid initialized BTree
   Post : the nodes are traversed inorder and Proc is performed on each data item. *)

END BTrees.

```

[Contents](#)

15.2 Implementing and Testing a Semi-Generic B-tree

Rather than offer pseudocode for the algorithms, the actual code is presented with heavy commenting. The reader should carefully match what is said below with the description given above. Note that inserting and deleting procedures have to proceed recursively as whatever affects a given level may affect the one above it. Thus, several of the procedures are recursive and return a flag to the level above to inform the calling version of the same (or another) procedure about the necessity of splitting or consolidating. Also observe that subsidiary procedures are included inside their main procedures rather than being allowed to be visible in the main module.

```
IMPLEMENTATION MODULE BTrees;
```

```
( *****
  Design by R. Sutcliffe copyright 1995-1996
  Initial Code by Gordon Tisher
  June 9, 1995; last modified 1999 08 30
  Modified and rewritten by R. Sutcliffe 1996 10 10
    Bug in rearrange removed 1999 08 30; weren't checking case of no data to right
    big thanks to Florenz Plassmann, Münster, Deutschland, who had the wit to try
    the test code with an order three tree and uncovered a huge flaw.
    also added more and better comments in that section
    deanonymized main data structure
    and removed a redundant count variable in DelSpecial

  This module provides a B-Tree ADT.
  ***** )
```

```
FROM Storage IMPORT
  ALLOCATE, DEALLOCATE;
FROM STextIO IMPORT
  WriteString, WriteLn, SkipLine;
FROM SWholeIO IMPORT
  WriteCard;
FROM DataADT IMPORT
  Assign, GetKey, WriteData, ActionProc, Compare, CompareResults;
```

```
CONST
  maxIndex = 2 * order;
  maxIndexPlus = maxIndex + 1;
```

```
TYPE
  BTree = POINTER TO BNode;
    ItemArray = ARRAY[1..maxIndexPlus] OF ItemType;
    PointerArray = ARRAY[0..maxIndexPlus] OF BTree;
  BNode =
    RECORD
      numItems : CARDINAL;
      state    : BTreeState;
      Items    : ItemArray;
      Pointers : PointerArray;
```

```

    END; (* record *)
(* There is one extra position in the arrays to aid sorting a new item in. This
saves a lot of copying to and from temporaries. *)

(* BTreeState = (allRight, empty, entreeFailed) *)

(*****
Utility Procedures
*****)

PROCEDURE FindPos (keyToSearchFor : KeyType; node : BTree; VAR pos : CARDINAL);
(* find the position within a node of a data item by search key before or at which
the item would go
Pre: node is a valid tree
Post: the position returned is that at or before keyToSearchFor belongs. If the
keyToSearchFor is smaller than any of the data, returns 1; if keyToSearchFor is
larger than any of the data, returns node^.numItems+1 *)
BEGIN
    pos := 1;
    WHILE (pos <= node^.numItems)
        AND (Compare(keyToSearchFor, GetKey (node^.Items[pos])) # less)
    DO
        INC (pos);
    END; (* while *)
END FindPos;

(*****
Exported Procedures
*****)

PROCEDURE Status (tree : BTree) : BTreeState;
(* Pre : t is a valid initialized BTree
Post : The State of the tree is returned *)
BEGIN
    RETURN tree^.state;
END Status;

PROCEDURE Init (VAR tree : BTree);
(* Allocates memory for a new node and initializes it
Pre: none
Post: if memory is found, sets up the tree structure with NIL pointers and zero items
*)
VAR
    count: CARDINAL;
    temp : BTree;
BEGIN
    NEW (temp);
    IF (temp # NIL)
    THEN
        temp^.numItems := 0;
        FOR count := 0 TO maxIndexPlus
        DO
            temp^.Pointers[count] := NIL;
        END; (* for *)
    END;

```

```

        temp^.state := allRight;
        tree := temp;
    END; (* if *)
END Init;

PROCEDURE Destroy (VAR tree : BTree);
(* disposes the whole tree *)

    PROCEDURE disp (node : BTree; level : CARDINAL);
    (* recursively remove a node *)
    VAR
        count : CARDINAL;
    BEGIN
        IF (node # NIL)
            THEN
                FOR count := 0 TO node^.numItems
                    DO
                        disp (node^.Pointers[count], level+1);
                    END; (* for *)
                DISPOSE (node);
            END; (* if *)
        END disp;
    BEGIN
        disp (tree, 0);
    END Destroy;

PROCEDURE Add (VAR tree : BTree; data : ItemType);
(* Adds a new item to the tree. *)

    PROCEDURE Insert (VAR data : ItemType; pos : CARDINAL; node : BTree;
        VAR newNode : BTree) : BOOLEAN;
    (* put the data item into the specified node position and the newNode pointer after
    it if one is generated. *)
    (* returns true if inserting is finished, false as a flag to the next higher level
    to insert the new centre item up there after a split. *)

    VAR
        count      : CARDINAL;
        temp       : BTree;
    BEGIN
        (* first move things over to make room for new item *)
        FOR count := node^.numItems TO pos BY -1
            DO
                Assign (node^.Items[count], node^.Items[count+1]);
                node^.Pointers[count+1] := node^.Pointers[count];
            END; (* for *)

        (* pop new one into place *)
        Assign (data, node^.Items[pos]);
        node^.Pointers[pos] := newNode;

        (* check to see if the resulting node needs to be split *)
        IF (node^.numItems = maxIndex) (* was at max before so split it! *)

```


THEN

Init (temp); (* make a new node *)

IF (temp # **NIL**)

THEN (* split node *)

(* reset number of items in each to order *)

node^.numItems := order;

temp^.numItems := order;

(* store new middle for upstairs to insert recursively *)

Assign (node^.Items[order+1], data);

(* copy new zeroth pointer *)

temp^.Pointers[0] := node^.Pointers[order+1];

(* copy data and rest of pointers into both *)

FOR count := 1 **TO** order

DO (* clean up the rest of the node *)

node^.Pointers[count+order] := **NIL**;

Assign (node^.Items[count+order+1], temp^.Items[count]);

temp^.Pointers[count] := node^.Pointers[count+order+1];

END; (* for *)

node^.Pointers[maxIndexPlus] := **NIL**; (* fix last pointer too; not in loop

tree^.state := allRight

ELSE (* did not work *)

tree^.state := entreeFailed;

END; (* if temp *)

newNode := temp;

RETURN FALSE; (* flag for split *)

ELSE (* don't split it just adjust count *)

INC (node^.numItems);

newNode := **NIL**;

RETURN TRUE; (* flag for just did an insert *)

END; (* if node*)

END Insert;

PROCEDURE AddItem (**VAR** data : ItemType; node : BTree;

VAR newNode : BTree): **BOOLEAN**;

(* returns true if added with no split, false if added with split *)

VAR

pos : **CARDINAL**;

BEGIN

(* look for correct position *)

FindPos (GetKey (data), node, pos);

(* insert the item at or to the left of the position found *)

IF (node^.Pointers[pos-1] # **NIL**) (* there is a node below *)

THEN (* go down there *)

IF ~AddItem (data, node^.Pointers[pos-1], newNode) (* node below was split

THEN (* so find spot for dividing item that was passed up here *)

FindPos (GetKey (data), node, pos);

RETURN Insert (data, pos, node, newNode);

```

    ELSE
        RETURN TRUE; (* tell level up that no split *)
    END; (* if ~AddItem *)
ELSE (* add the item to the current node *)
    (* and send back the result of doing that *)
    RETURN Insert (data, pos, node, newNode);
END; (* if node *)
END AddItem;

```

```

VAR
    temp, newNode : BTree;
BEGIN (* main for Add *)
    newNode := NIL;
    IF ~AddItem (data, tree, newNode) (* got a split on first level below *)
    THEN (* so, must turn temp into a new root *)
        Init (temp); (* set up a new node for it *)
        IF (temp # NIL)
        THEN
            temp^.numItems := 1; (* it has only this item for now *)
            temp^.Pointers[0] := tree;
            Assign (data, temp^.Items[1]);
            temp^.Pointers[1] := newNode;
            tree := temp;
        END; (* if temp *)
    END; (* if ~AddItem *)
END Add;

```

```

PROCEDURE Delete (VAR tree : BTree; key : KeyType);
(* deletes an item whose key is equal to _key_ from the tree
If the item isn't there, nothing happens, but if the tree was empty and we tried to
delete the state is set to empty. *)

```

```

PROCEDURE Rearrange (node : BTree; pos : CARDINAL) : BOOLEAN;
(* this sub-procedure is called to rearrange a parent node with respect to its
children when a deletion causes the number of items to become too small *)
(* pos is that of the data item 1..order in the parent node separating the two
child nodes that need combining *)
(* returns true if node needs no further rearrangement above, false if it does *)

```

```

VAR
    child, sibling : BTree;
    count, num : CARDINAL;
BEGIN
    child := node^.Pointers[pos-1]; (* look at the child node that's short on data *)

    (* check to see if we must combine with sibling having lesser data to its left *)
    IF (* first case *) (node^.Pointers[pos] = NIL) (* no node available to the right
*)
    OR (* second case *) ((pos > 1) (* there is a sibling to the left *)
        AND (node^.Pointers[pos]^numItems = order))
        (* no surplus among the greater to the right *)
    (* The 'default' situation is that there is a node to the right and it can be
used,

```

```

else we stay here *)
THEN (* must stay here and combine with node of the lesser data *)
  DEC (pos);
  sibling := node^.Pointers[pos-1]; (* node from the lesser data position *)

  (* calculate num to move sibling ==> child *)
  num := (sibling^.numItems + 1 - order) DIV 2;
  IF (num > 0)
    THEN (* move num items from sibling ==> child *)
      FOR count := (order-1) TO 1 BY -1 (* shift right to make room *)
        DO
          Assign (child^.Items[count], child^.Items[count+num]);
          child^.Pointers[count+num] := child^.Pointers[count];
        END; (* for count *)

      (* move old separator down to child *)
      Assign (node^.Items[pos], child^.Items[num]);
      child^.Pointers[num] := child^.Pointers[0];

      (* now move stuff over *)
      FOR count := (num-1) TO 1 BY -1 (* copy from adjacent node *)
        DO
          Assign (sibling^.Items[count+(sibling^.numItems+1-num)],
                  child^.Items[count]);
          child^.Pointers[count] :=
            sibling^.Pointers[count+(sibling^.numItems+1-num)];
          sibling^.Pointers[count+(sibling^.numItems+1-num)] := NIL;
        END; (* for count *)
      child^.Pointers[0] := sibling^.Pointers[sibling^.numItems+1-num];

      (* move last item in node with extras up to be new separator *)
      Assign (sibling^.Items[sibling^.numItems+1-num], node^.Items[pos]);
      node^.Pointers[pos] := child;

      (* adjust item numbers in both nodes *)
      sibling^.numItems := sibling^.numItems - num;
      child^.numItems := order - 1 + num;

      (* adjust last pointer in sibling node *)
      sibling^.Pointers[sibling^.numItems+1] := NIL;
      RETURN TRUE; (* tell upstairs all OK *)
    ELSE (* none avail to move, so just merge nodes *)
      Assign (node^.Items[pos], sibling^.Items[sibling^.numItems + 1]);
      sibling^.Pointers[sibling^.numItems + 1] := child^.Pointers[0];
      FOR count := 1 TO (order-1) (* copy child to sibling *)
        DO
          Assign (child^.Items[count],
                  sibling^.Items[count+sibling^.numItems+1]);
          sibling^.Pointers[count+sibling^.numItems+1] :=
            child^.Pointers[count];
        END; (* for count *)

      (* shift left node items *)
      FOR count := pos TO (node^.numItems - 1)

```

```

DO
    Assign (node^.Items[count+1], node^.Items[count]);
    node^.Pointers[count] := node^.Pointers[count+1];
END; (* for count *)
sibling^.numItems := maxIndex; (* adjust num of items to max *)
node^.Pointers[node^.numItems] := NIL; (* get rid of pointer *)
DEC (node^.numItems);
DISPOSE (child);
RETURN (node^.numItems >= order); (* flag up if consolidation needed *)
END; (* if num *)

```

(* this section combines our child node with a sibling on its right *)
 (* We get here always if the child is the first node, or, in the default, if

there

is a node to the right of our problem child and it has a surplus available *)

ELSE

```

sibling := node^.Pointers[pos];
(* the one after or having greater data, that is, to the child's right *)
num := (sibling^.numItems - order + 1) DIV 2; (* calculate num to move *)

```

(* copy item from parent to orderth position first *)

```

Assign (node^.Items[pos], child^.Items[order]);
child^.Pointers[order] := sibling^.Pointers[0];

```

IF (num > 0)

THEN (* move items from sibling to this child in next positions *)

FOR count := 1 **TO** (num-1)

DO

```

    Assign (sibling^.Items[count], child^.Items[count+order]);
    child^.Pointers[count+order] := sibling^.Pointers[count]

```

END; (* for count *)

(* new separator move from sibling *)

```

Assign (sibling^.Items[num], node^.Items[pos]);

```

(* now, fix sibling up *)

```

sibling^.Pointers[0] := sibling^.Pointers[num];
DEC (sibling^.numItems, num);

```

(* shift left remaining elements of that sibling *)

FOR count := 1 **TO** sibling^.numItems

DO

```

    Assign (sibling^.Items[count+num], sibling^.Items[count]);
    sibling^.Pointers[count] := sibling^.Pointers[count+num];
    sibling^.Pointers[count+num] := NIL;

```

END; (* for count *)

```

child^.numItems := order - 1 + num;

```

RETURN TRUE;

ELSE (* not enough on that side so merge nodes *)

FOR count := 1 **TO** order

DO

```

    Assign (sibling^.Items[count], child^.Items[count+order]);
    child^.Pointers[count+order] := sibling^.Pointers[count];

```

END; (* for count *)

```

    (* shift left node items *)
    FOR count := pos TO (node^.numItems - 1)
        DO
            Assign (node^.Items[count+1], node^.Items[count]);
            node^.Pointers[count] := node^.Pointers[count+1];
        END; (* for *)
    child^.numItems := maxIndex; (* adjust num of items to max *)
    node^.Pointers[node^.numItems] := NIL; (* get rid of pointer *)
    DEC (node^.numItems);
    DISPOSE (sibling);
    RETURN (node^.numItems >= order);
END; (* if num *)
END; (* if pos *)
END Rearrange;

PROCEDURE DelSpecial (node : BTree; VAR data : ItemType) : BOOLEAN;
(* after deleting from an interior node, find largest item less than it to remove
from a node below and pass it up in "data" to become the new divisor *)
(* returns true if node is OK, false if too small & needs work from above *)
BEGIN
    (* check for more nodes and do it recursively to the greater side to get the
biggest *)
    IF (node^.Pointers[node^.numItems] # NIL)
        THEN
            (* see if level below says rearrange needed *)
            IF ~DelSpecial (node^.Pointers[node^.numItems], data)
                THEN (* so, do it *)
                    RETURN Rearrange (node, node^.numItems+1);
                ELSE
                    RETURN TRUE;
            END; (* if *)
        ELSE (* remove the item *)
            Assign (node^.Items[node^.numItems], data); (* save in data to send up top *)
            DEC (node^.numItems); (* decrease this node size *)
            RETURN (node^.numItems >= order); (* and return fla to next level up *)
        END; (* if *)
    END DelSpecial;

PROCEDURE Del (node : BTree; key : KeyType) : BOOLEAN;
(* finds and delete the item with the key from the node; works recursively down
returns true if node is OK now, false if next higher level must work on it as too
small *)
VAR
    count, pos : CARDINAL;
    data : ItemType;
BEGIN
    FindPos (key, node, pos); (* key is at or left of position *)
    IF (pos > 1) AND (pos <= node^.numItems)
        (* else, left of 1 or right of numItems means not in this node *)
        AND (Compare (GetKey (node^.Items[pos-1]),key) = equal) (* check to see if bang
on *)
        THEN (* item actually found in this node *)

```

```

    DEC (pos); (* now at the item to delete *)
    IF (node^.Pointers[pos-1] # NIL) (* stuff hanging below it & prior to it *)
        THEN (* use prior pointer to fish below for largest item for possible
promotion *)
            (* see if got it with rearranging needed *)
            IF ~DelSpecial (node^.Pointers[pos-1], data)
                THEN (* do a rearrange as node below has too few items *)
                    Assign (data, node^.Items[pos]); (* so put it in place *)
                    RETURN Rearrange (node, pos); (* and do the rearrange on this parent
*)
                ELSE (* no rearrange, so just replace item with one from lower down *)
                    Assign (data, node^.Items[pos]);
                    RETURN TRUE; (* and tell upstairs we're happy *)
                END; (* if *)
            ELSE (* nothing below so node of found item is a leaf *)
                FOR count := pos TO (node^.numItems-1)
                    DO (* move everything over to the left *)
                        Assign (node^.Items[count+1], node^.Items[count]);
                        (* don't need to assign pointers, as the node is a leaf *)
                    END; (* for *)
                DEC (node^.numItems);
                RETURN (node^.numItems >= order); (* tell upstairs if rearrange needed *)
            END; (* if *)
        ELSE
            (* item is not in this node so try down below using pointer left of node
position# *)
            IF (node^.Pointers[pos-1] # NIL) (* there is a "below" before it *)
                AND ~Del (node^.Pointers[pos-1], key) (* and we need to rearrange *)
                THEN
                    RETURN Rearrange (node, pos);
                    (* rearrange this node; provide position in parent to the left of which
lies the
node (pointer one less) below that has declined in population and send
flag up *)
                ELSE
                    RETURN TRUE; (* tell upstairs no rearrange *)
                END; (* if node *)
            END; (* if pos *)
        END Del;

```

```

VAR
    oldTree : BTree;
BEGIN (* main delete proc *)
    IF tree^.numItems = 0 (* nothing to delete *)
        THEN (* this is an error; set flag so client can find out if it checks *)
            tree^.state := empty;
        END;
    IF ~Del (tree, key) AND (tree^.numItems = 0) (* after del *)
        AND (tree^.Pointers[0] # NIL)
        THEN
            oldTree := tree;
            tree := oldTree^.Pointers[0];
            DISPOSE (oldTree);

```

```

        tree^.state := allRight;
    END; (* if *)
END Delete;

PROCEDURE Depth (tree : BTree) : CARDINAL;
(* returns the number of levels in the tree *)
BEGIN
    IF (tree # NIL)
    THEN
        RETURN Depth (tree^.Pointers[0]) + 1;
    ELSE
        RETURN 0;
    END; (* if *)
END Depth;

PROCEDURE Search (tree : BTree; key : KeyType; VAR data : ItemType) : BOOLEAN;
(* if found, returns TRUE and _data_ returns item at that point *)
VAR
    pos : CARDINAL;
BEGIN
    IF (tree = NIL)
    THEN
        RETURN FALSE;
    ELSE
        FindPos (key, tree, pos);
        IF (pos > 1) AND (pos <= maxIndexPlus)
            AND (Compare (GetKey (tree^.Items[pos-1]), key) = equal)
        THEN
            Assign (tree^.Items[pos-1], data);
            RETURN TRUE;
        ELSE
            RETURN Search (tree^.Pointers[pos-1], key, data);
        END; (* if pos *)
    END; (* if tree *)
END Search;

PROCEDURE WriteTree (tree : BTree);
(* writes out the tree *)

PROCEDURE WriteInOrder (node : BTree);
VAR
    count : CARDINAL;
BEGIN
    IF (node = NIL)
    THEN
        RETURN;
    END; (* if *)

    WriteInOrder (node^.Pointers[0]);
    FOR count := 1 TO node^.numItems
    DO
        WriteData (node^.Items[count]);
        INC (gcount);
    
```

```

        WriteString (" ", " ");
        IF (gcount > 10)
            THEN
                WriteLn;
                gcount := 1;
            END; (* if *)
        WriteInOrder (node^.Pointers[count]);
    END; (* for *)
END WriteInOrder;

PROCEDURE WriteNodes (node : BTree; level : CARDINAL);
VAR
    count : CARDINAL;
BEGIN
    IF (node = NIL)
        THEN
            RETURN;
        END; (* if *)

    FOR count := 1 TO (level*2)
        DO
            WriteString (" ");
        END; (* for *)

    IF node^.numItems > 0
        THEN
            WriteData (node^.Items[1]);
        END; (* if *)
    FOR count := 2 TO node^.numItems
        DO
            WriteString (" ", " ");
            WriteData (node^.Items[count]);
        END; (* for *)
    WriteLn;
    INC (gcount);
    IF (gcount > 33)
        THEN
            WriteString ("press return to continiue...");
            SkipLine;
            gcount := 1;
        END; (* if *)

    FOR count := 0 TO maxIndex
        DO
            WriteNodes (node^.Pointers[count], level+1);
        END; (* for *)
    END WriteNodes;

VAR
    gcount : CARDINAL;

BEGIN
    WriteString ("*****"); WriteLn;
    WriteString ("Tree view:"); WriteLn;

```



```

gcount := 1;
WriteNodes (tree, 0);
WriteLn;
IF gcount > 34
    THEN
        WriteString ("press return to continue...");
        SkipLine;
    END; (* if *)
WriteString ("Inorder view:"); WriteLn;
gcount := 1;
WriteInOrder (tree);
WriteLn;
WriteString ("*****"); WriteLn;
END WriteTree;

PROCEDURE Traverse (tree : BTree; Proc : ActionProc);
(* Pre : tree is a valid initialized BTree
   Post : the nodes are traversed inorder and Proc is performed on each data item. *)
VAR
    count : CARDINAL;

BEGIN
    IF (tree = NIL)
        THEN
            RETURN;
        END; (* if *)

    Traverse (tree^.Pointers[0], Proc);
    FOR count := 1 TO tree^.numItems
        DO
            Proc (tree^.Items[count]);
            Traverse (tree^.Pointers[count], Proc);
        END; (* for count *)
END Traverse;

END BTrees.

```

This module was given a brief workout using cardinals as the data type, and their own value as the key. Observe that the data type to be entered could have been a complex record as long as some compare procedure was defined on a key field. First, the data type to be imported and entreed is defined:

```

DEFINITION MODULE DataADT;

TYPE
    CompareResults = (less, equal, greater);
    KeyFieldType = CARDINAL;
    DataType = CARDINAL;
    ActionProc = PROCEDURE (DataType);

PROCEDURE Assign (a : DataType; VAR b : DataType);
PROCEDURE GetKey (a : DataType) : KeyFieldType;
PROCEDURE WriteData (a : DataType);
PROCEDURE Compare (a, b : KeyFieldType) : CompareResults;

```

```

END DataADT.

IMPLEMENTATION MODULE DataADT;
IMPORT SWholeIO;

PROCEDURE Assign (a : DataType; VAR b : DataType);BEGIN
    b:= a;
END Assign;

PROCEDURE GetKey (a : DataType) : KeyFieldType;
BEGIN
    RETURN a;
END GetKey;

PROCEDURE WriteData (a : DataType);
BEGIN
    SWholeIO.WriteCard (a,0);
END WriteData;

PROCEDURE Compare (a, b : KeyFieldType) : CompareResults;
BEGIN
    IF a = b
        THEN
            RETURN equal
        ELSIF a < b THEN
            RETURN less
        ELSE
            RETURN greater
        END
    END Compare;
END DataADT.

```

The actual test uses the same data as in the initial discussion. A few additional insertions (same value as one already there) and deletions (a value not present) are also done. At intervals, the tree is printed out, and at one point, a traverse is done adding all the key values.

```

MODULE TestBTrees;
(* A simple program to test the Binary tree library module.
by R. Sutcliffe
last modified 1996 10 15 *)

IMPORT BTrees, DataADT, SWholeIO, STextIO;

VAR
    theTree : BTrees.BTree;
    sum : CARDINAL;

PROCEDURE Summit (item : DataADT.DataType);
BEGIN
    sum := sum + DataADT.GetKey (item)
END Summit;

BEGIN
    BTrees.Init (theTree);

```

```

BTrees.Add (theTree, 15);
BTrees.Add (theTree, 4);
BTrees.Add (theTree, 6);
BTrees.Add (theTree, 12);
BTrees.WriteTree(theTree);

BTrees.Add (theTree, 11);
BTrees.WriteTree(theTree);

BTrees.Add (theTree, 17);
BTrees.Add (theTree, 20);
BTrees.Add (theTree, 30);
BTrees.Add (theTree, 31);
BTrees.Add (theTree, 5);
BTrees.Add (theTree, 16);
BTrees.Add (theTree, 37);
BTrees.WriteTree(theTree);

sum := 0;
BTrees.Traverse (theTree, Summit);
STextIO.WriteLine;
STextIO.WriteString ("Sum is ");
SWholeIO.WriteCard (sum, 10);
STextIO.WriteLine;

BTrees.Delete (theTree, 15);
BTrees.WriteTree(theTree);
BTrees.Delete (theTree, 16);
BTrees.WriteTree(theTree);

BTrees.Delete (theTree, 37);
BTrees.Delete (theTree, 12);
BTrees.WriteTree(theTree);

BTrees.Delete (theTree, 17);
BTrees.WriteTree(theTree);

BTrees.Delete (theTree, 42);
BTrees.WriteTree(theTree);

BTrees.Add (theTree, 4);
BTrees.WriteTree(theTree);

END TestBTrees.

```

When this program was run, the output looked like this:

```

*****
Tree view:
 4,  6, 12, 15

Inorder view:
 4,  6, 12, 15,
*****

```

Tree view:

```
11
  4,  6
 12, 15
```

Inorder view:

```
4,  6, 11, 12, 15,
```

Tree view:

```
11, 17
  4,  5,  6
 12, 15, 16
 20, 30, 31, 37
```

Inorder view:

```
4,  5,  6, 11, 12, 15, 16, 17, 20, 30, 31, 37,
```

Sum is 204

Tree view:

```
11, 17
  4,  5,  6
 12, 16
 20, 30, 31, 37
```

Inorder view:

```
4,  5,  6, 11, 12, 16, 17, 20, 30, 31, 37,
```

Tree view:

```
11, 20
  4,  5,  6
 12, 17
 30, 31, 37
```

Inorder view:

```
4,  5,  6, 11, 12, 17, 20, 30, 31, 37,
```

Tree view:

```
6, 20
  4,  5
 11, 17
 30, 31
```

Inorder view:

```
4,  5,  6, 11, 17, 20, 30, 31,
```

Tree view:

```
20
  4,  5,  6, 11
 30, 31
```

Inorder view:

```
4,  5,  6, 11, 20, 30, 31,
*****
*****
```

Tree view:

```
20
  4,  5,  6, 11
 30, 31
```

Inorder view:

```
4,  5,  6, 11, 20, 30, 31,
*****
*****
```

Tree view:

```
5, 20
  4,  4
  6, 11
 30, 31
```

Inorder view:

```
4,  4,  5,  6, 11, 20, 30, 31,
*****
```

15.3 Heaps

15.3.1 Heaps Defined

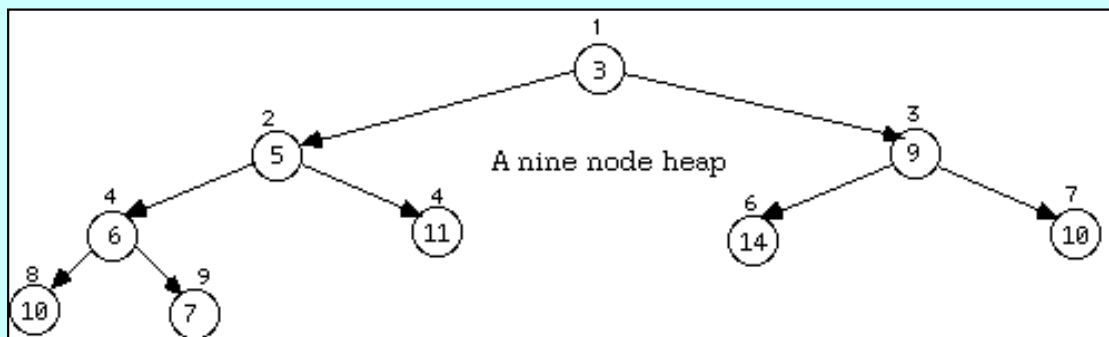
The structure known as a heap is an interesting hybrid. It is most easily conceptualized as a binary tree constructed according to some specific rules on the data being entreed. However, each node of the tree is numbered consecutively, and it is possible to work with a heap by implementing it as an array.

A heap is a sequence of data nodes $n_1, n_2, n_3, n_4, \dots, n_{\max}$ in which $n_i \leq n_{2i}$ and $n_i \leq n_{2i+1}$ for all i with $1 \leq i \leq \max/2$.

In a tree implementation, the n_{2i} and n_{2i+1} are the children of the node n_i , (this makes the tree binary) so the rule in the definition can be expressed as:

In a heap every parent is greater than its child.

A reverse heap (in which the rule is "less than the child" could also be constructed, but this will not be done here. In the heap shown below there are nine nodes. The cardinal data at each node is placed inside the circle and each node is numbered outside the node to illustrate the structural relationships.



15.3.2 Heaps as Binary Trees

In order to build a heap, data is placed in the tree as it arrives in the following manner:

Add Heap Data =

- add a new node at the next leaf position

- use this new node as the current position

- While new data is less than that in the parent of the current node

 - move the parent down to the current node

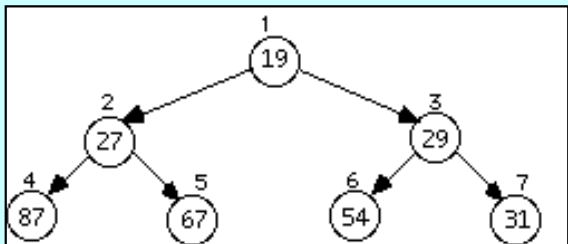
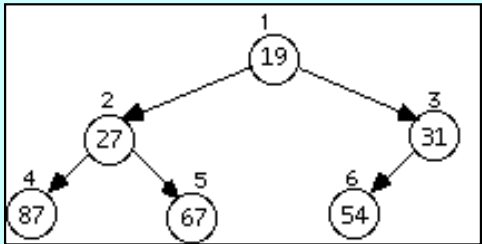
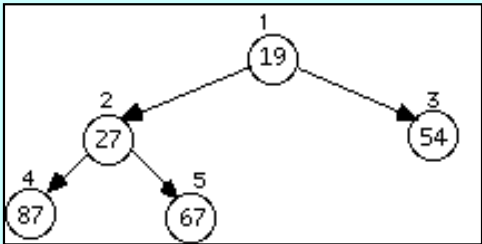
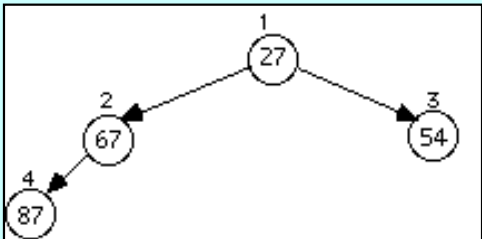
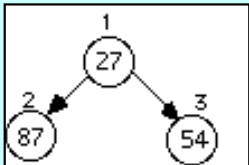
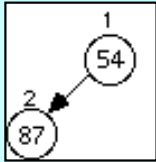
 - make the parent (now vacant) the current node

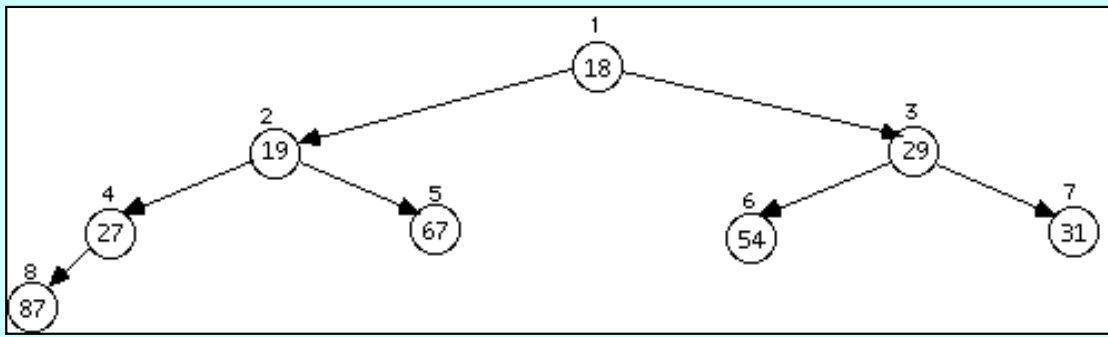
- Place data in current node

Thus, new nodes are always added on the deepest level in an orderly way, and the tree never becomes unbalanced. There is no particular relationship among the data items in the nodes on any given level, even the ones that have the same parent, so certainly a heap is not a sorted structure. It can however, be regarded as partially ordered. One should also note that a given set

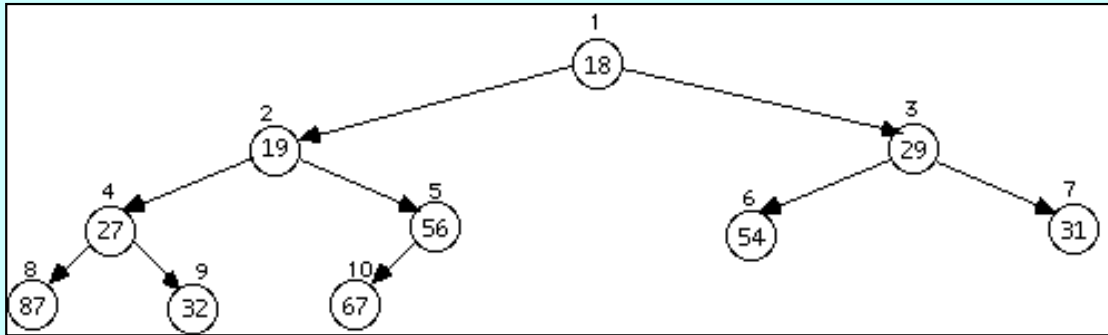
of data can be formed into many different heaps. Which one results from the heapification will depend on the order in which the data arrives.

Example: Data arrives to be heaped in the order 54, 87, 27, 67, 19, 31, 29, 18, 32, 56, 7, 12, 31
The series of diagrams illustrates what the heap looks like as each data item is added.

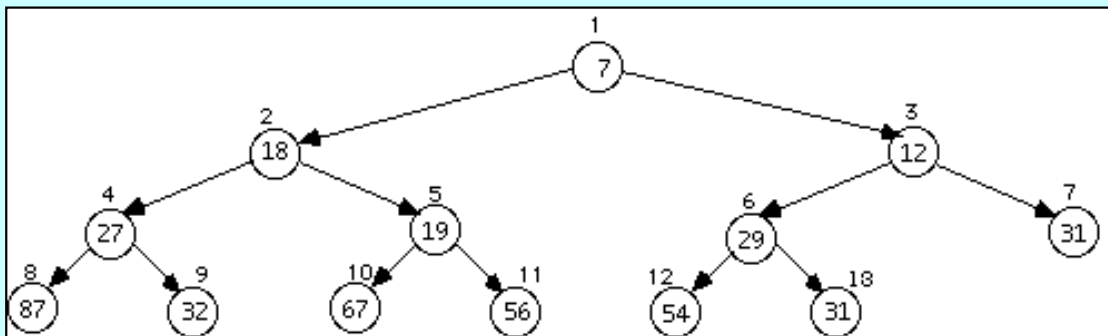




The next two insertions do not cause a rearrangement



The following two both cause a rearrangement, and the one after that does not.



If deletions of data are to be done, some work may be necessary to restore the tree to the properly filled state, because not all leaves are on the same level. Only if the data in the highest numbered node is deleted will the operation be a simple inverse of insertion, for in that case the node can simply be removed without affecting anything else. If data in some other node is deleted, there are two possible choices for what the algorithm could do:

- mark the node as unused and do the next insert there rather than by generating a new node
- shift data around in the tree so that all the nodes are occupied except the highest numbered one, and then delete that.

The first has the advantage of not requiring the tree to be rearranged, but the disadvantage of making insertion more complicated by requiring a list of unused nodes to be maintained. The second has the advantage that insertion need not be changed, and the advantage of being more symmetric with insertion, but the disadvantage of producing a more complicated deletion algorithm:

Delete and restructure=

save data from last node

If node to delete data from is not last node

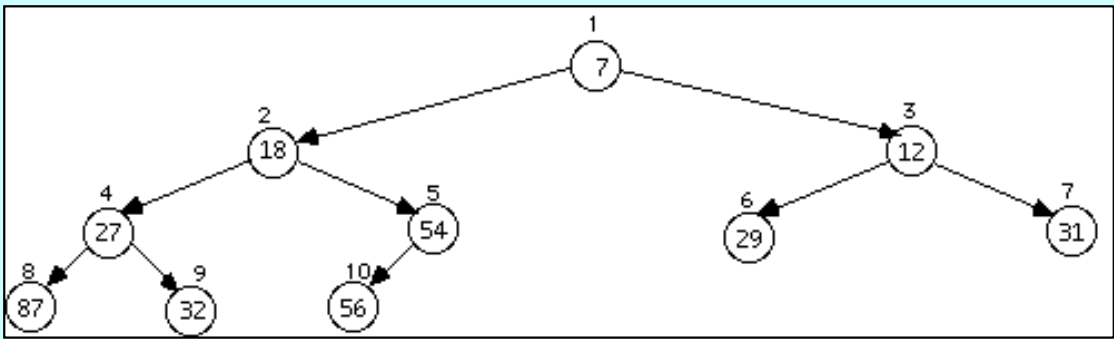
insert data from last node starting in node where data is to be deleted

reheapify working up or down as necessary from this point.

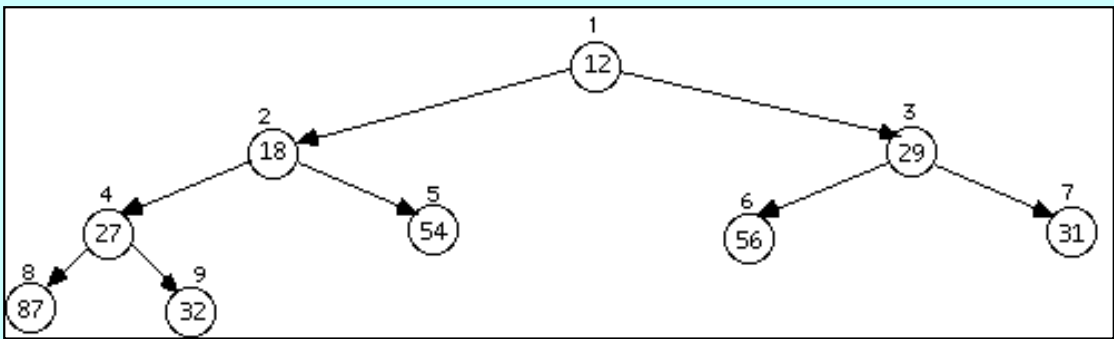
delete last node

If duplicate items are allowed, as in the example above, which one gets deleted depends on the order in which the tree is traversed to find the data. If the item 31 is deleted then any forward traversal to do so will find and delete the one on the bottom row. Other sets of rules are possible of course. Adding an item could mean forbidding duplicates. Deleting could mean

deleting all items with the given key. Here is what the tree looks like after deleting the single item 31 followed by the item 67. Deleting item 19 puts the 56 temporarily on its spot and then sifts it down to trade with item 54, producing:



Deleting the item at the root (7) causes a substantial rearrangement as item 56 is placed in its spot and then sifted down the lesser of the child paths until it is a correct child again, producing:



An attempt to delete something that is not there should not change the tree, but the programmer could require this to set some kind of error condition that could then be checked.

15.3.3 Defining the Heap

With this description complete, the following can be offered as a definition for the Heap ADT. Observe that it is somewhat more elaborate than the definition of the B-tree, as it allows for four kinds of traversals and in either direction.

```
DEFINITION MODULE Heaps;

( *****
  Design by R. Sutcliffe copyright 1996
  Modified 1996 10 16
  This module provides a Heap ADT.
  ***** )

FROM DataADT IMPORT
  KeyFieldType, DataType, ActionProc;

(* rename the imported types to the local ones. This makes things more generic and
easier to change. *)
TYPE
  ItemType = DataType;
  KeyType = KeyFieldType;

TYPE
  Heap;
  HeapState = (allRight, empty, enheapFailed);
```

```

    TraverseKind = (preOrder, inOrder, postOrder, rowOrder);

PROCEDURE Status (heap : Heap) : HeapState;
(* Pre : t is a valid initialized heap
   Post : The State of the heap is returned *)

PROCEDURE Init (VAR heap : Heap);
(* Allocates memory for a new heap sets state to allRight *)

PROCEDURE Destroy (VAR heap : Heap);
(* disposes the whole heap *)

PROCEDURE Add (VAR heap : Heap; data : ItemType);
(* Pre : heap is a valid initialized Heap
   Adds a new item to the heap. If successful sets state to allRight, else to
   enheapFailed *)

PROCEDURE Delete (VAR heap : Heap; key : KeyType);
(* Pre : heap is a valid initialized Heap
   deletes an item whose key is defined equal to _key_ from the heap. If successful
   sets state to allRight; if heap was empty sets state to empty *)

PROCEDURE Search (heap : Heap; key : KeyType; VAR data : ItemType) : BOOLEAN;
(* Pre : heap is a valid initialized Heap
   if found, returns TRUE and _data_ returns item at that point  *)

PROCEDURE SetTraverseKind (heap : Heap; tKind : TraverseKind);
(* Pre : heap is a valid initialized Heap
   The default is inorder *)

PROCEDURE ReverseTraverseDirection (heap : Heap);
(* Pre : heap is a valid initialized Heap
   The default is forward, but this can be changed to and fro.  The user has to keep
   track. *)

PROCEDURE Size (heap : Heap) : CARDINAL;
(* Pre : heap is a valid initialized Heap
   Post: The number of data items in the heap is returned*)

PROCEDURE Traverse (heap : Heap; Proc : ActionProc);
(* Pre : heap is a valid initialized Heap
   Post : the nodes are traversed inorder and Proc is performed on each data item. *)

END Heaps.

```

15.4 Implementing and Testing a Semi-Generic Heap

Enough pseudocode was given in [section 15.3.2](#) to allow the following heavily commented implementation to stand on its own.

```
IMPLEMENTATION MODULE Heaps;
```

```
(*****  
  Design by R. Sutcliffe copyright 1996  
  Modified 1996 10 16  
  This module provides a Heap ADT.  
  *****)
```

```
FROM Storage IMPORT  
  ALLOCATE, DEALLOCATE;  
FROM DataADT IMPORT  
  DataType, Assign, GetKey, ActionProc, Compare, CompareResults;
```

```
TYPE  
  NodePointer = POINTER TO TreeNode;  
  TreeNode =  
    RECORD  
      dataItem : DataType;  
      leftPoint, rightPoint, parent, (* binary tree linkage *)  
      next, prev : NodePointer; (* linear linkage across rows *)  
    END;  
  Heap = POINTER TO TreeData;  
  TreeData =  
    RECORD  
      root, (* first node *)  
      last, (* last node *)  
      lowerLeft (* first node in last row; helps for adding linkage to next row *)  
        : NodePointer;  
      state : HeapState; (* stores error conditions *)  
      travKind : TraverseKind; (* inOrder, preOrder, postOrder or rowOrder *)  
      travDirIsForward : BOOLEAN;  
      room, (* how many could be stored if last row full *)  
      numItems (* how many are actually stored *)  
        : CARDINAL;  
    END;
```

```
PROCEDURE MakeNode () : NodePointer;  
(* set up one new node with all nil pointers and no data; return a pointer to the new  
node. *)  
VAR  
  temp : NodePointer;  
BEGIN  
  NEW (temp); (* get node memory *)  
  IF temp # NIL
```

```

    THEN
        temp^.leftPoint := NIL;
        temp^.rightPoint := NIL;
        temp^.parent := NIL;
        temp^.next := NIL;
        temp^.prev := NIL;
    END;
    RETURN temp;
END MakeNode;

PROCEDURE KillNode (VAR node : NodePointer);
(* give back all memory associated with node *)
BEGIN
    IF node # NIL
    THEN
        DISPOSE (node);
    END;
END KillNode;

PROCEDURE Erase (VAR r : NodePointer);
(* Pre: r is the root of a subtree
   Post: recursive post traverse killing all nodes below as well as the one passed in
   *)
BEGIN
    IF r # NIL
    THEN
        Erase (r^.leftPoint);
        Erase (r^.rightPoint);
        KillNode (r);
    END;
END Erase;

(* It turned out the following was not needed, but who knows; why not leave it. *)
PROCEDURE IsLeaf (VAR node : NodePointer) : BOOLEAN;
BEGIN
    RETURN (node # NIL) AND (node^.leftPoint = NIL); (* don't care about right *)
END IsLeaf;

PROCEDURE FindKey (node : NodePointer; key : KeyType;
    VAR result : NodePointer) : BOOLEAN;
(* start at the given node and go looking for the data with the given key.  If found,
return both a pointer to it and TRUE; if not found, return FALSE.
Recursive preorder traversal *)
BEGIN
    IF node = NIL (* safety measure *)
    THEN
        RETURN FALSE;
    (* look at node data first *)
    ELSEIF Compare (GetKey(node^.dataItem), key) = equal THEN
        result := node;
        RETURN TRUE;
    (* then at the left subtree *)

```

```

ELSIF FindKey (node^.leftPoint, key, result) THEN
    RETURN TRUE;
(* and at the right one *)
ELSE
    RETURN FindKey (node^.rightPoint, key, result)
END;
END FindKey;

PROCEDURE TraverseRows (heap : Heap; Proc : ActionProc);
(* Traverse the tree row by row, that is, using the linear linkage doing the
procedure on each data item *)
VAR
    count : CARDINAL;
    node: NodePointer;
BEGIN
    IF heap^.travDirIsForward
    THEN (* start at the root *)
        count := 0;
        node := heap^.root;
        (* and work consecutively through the noides *)
        WHILE count < heap^.numItems
        DO
            INC (count);
            Proc (node^.dataItem);
            node := node^.next;
        END; (* while *)
    ELSE (* go in reverse order *)
        count := heap^.numItems;
        node := heap^.last;
        WHILE count > heap^.room THEN (* need to make new row *)
            mom := heap^.lowerLeft;
            heap^.lowerLeft := temp;
            heap^.room := 2*heap^.room + 1;
        ELSE (* continue on the same row *)
            (* either the parent can take a new right child *)
            mom := heap^.last^.parent;
            IF mom ^.rightPoint # NIL
            THEN (* or the next one on the row can -- not at end *)
                mom := mom^.next;
            END;
        END; (* if heap *)
        (* now set up all the rest of the linkage *)
        temp^.parent := mom;
        IF mom ^.leftPoint = NIL
        THEN
            mom^.leftPoint := temp;
        ELSE
            mom^.rightPoint := temp;
        END; (* if mom *)
        heap^.last^.next := temp;
        temp^.prev := heap^.last;
        heap^.last := temp;
        (* ensure data goes to right ancestral node *)

```

```

        SiftUp (temp);
        heap^.state := allRight;
    ELSE (* couldn't get node room *)
        heap^.state := enheapFailed;
    END (* if temp *)
ELSE (* heap itself is NIL *)
    heap^.state := enheapFailed;
END; (* if heap *)
END Add;

PROCEDURE Delete (VAR heap : Heap; key : KeyType);
(* deletes an item whose key is defined equal to _key_ from the heap. If successful
sets state to allRight; if heap was empty sets state to empty *)
VAR
    targetNode, temp : NodePointer;
    lastData : DataType;
BEGIN
    (* find the node with the data if it is there *)
    IF heap^.numItems = 0
    THEN (* can't delete from an empty heap so set flag *)
        heap^.state := empty;
        RETURN;
    ELSE (* whether we find an item to delete does not matter *)
        heap^.state := allRight;
    END;
    (* ok so go out and look for it *)
    IF FindKey (heap^.root, key, targetNode)
    THEN
        temp := heap^.last; (* save data from end of heap *)
        (* now fix all the pointers at the end to delete that last node *)
        lastData := temp^.dataItem;
        heap^.last := temp^.prev;
        IF temp^.parent^.leftPoint = temp
        THEN
            temp^.parent^.leftPoint := NIL;
        ELSE
            temp^.parent^.rightPoint := NIL;
        END; (* if temp *)
        DEC (heap^.numItems);

        (* check to see if must shrink number of levels *)
        IF heap^.numItems = heap^.room DIV 2
        THEN (* must have killed first item in row, so *)
            heap^.lowerLeft := temp^.parent;
            heap^.room := heap^.numItems;
        END; (* if heap *)
        IF temp # targetNode (* if it is, we're done *)
        THEN
            (* pop the data item from last into node of data to delete *)
            Assign (lastData, targetNode^.dataItem);
            (* then see if it needs moving up or down *)
            (* only one of the following will do anything *)
            SiftDown (targetNode);

```

```

        SiftUp (targetNode);
    END; (* if temp *)
    (* finally, dump memory from the last node *)
    KillNode (temp);
ELSE (* if FindKey *)
    (* nothing. If data not found we just don't care. *)
    END; (* if FindKey *)
END Delete;

PROCEDURE Search (heap : Heap; key : KeyType; VAR data : ItemType) : BOOLEAN;
(* if found, returns TRUE and _data_ returns item at that point *)
VAR
    temp : NodePointer;
BEGIN
    IF (heap^.root # NIL) AND (heap^.numItems # 0)
        AND (FindKey (heap^.root, key, temp))
    THEN
        data := temp^.dataItem;
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END;
END Search;

PROCEDURE SetTraverseKind (heap : Heap; tKind : TraverseKind);
(* The default is inorder *)
BEGIN
    IF heap # NIL
    THEN
        heap^.travKind := tKind;
    END;
END SetTraverseKind;

PROCEDURE ReverseTraverseDirection (heap : Heap);
(* The default is forward, but this can be changed to and fro. The user has to keep
track. *)
BEGIN
    IF heap # NIL
    THEN
        heap^.travDirIsForward := ~heap^.travDirIsForward;
    END;
END ReverseTraverseDirection;

PROCEDURE Size (heap : Heap) : CARDINAL;
(* Pre : heap is a valid initialized Heap
Post: The number of data items in the heap is returned *)
BEGIN
    RETURN heap^.numItems
END Size;

PROCEDURE Traverse (heap : Heap; Proc : ActionProc);
(* Pre : heap is a valid initialized Heap
Post : the nodes are traversed inorder and Proc is performed on each data item. *)

```

```

VAR
    temp : NodePointer;
BEGIN
    IF (heap^.root # NIL) AND (heap^.numItems # 0)
        THEN
            (* special case the linear, nonrecursive traverses *)
            IF heap^.travKind = rowOrder
                THEN
                    TraverseRows (heap, Proc);
            ELSIF heap^.travDirIsForward THEN
                ForwardTraverseNodes (heap^.root, heap^.travKind, Proc);
            ELSE
                ReverseTraverseNodes (heap^.root, heap^.travKind, Proc);
            END;
        END;
    END Traverse;

END Heaps.

```

The same cardinal ADT was used in the testing of this module as in the testing of the B-tree module. In addition, the following program module was written to check the implementation and ensure that it was correct. It should be studied carefully for completeness. The data used is that shown above in the discussion of heaps.

```

MODULE TestHeaps;
(* A simple program to test the Heaps library module.
by R. Sutcliffe
last modified 1996 10 18 *)

```

```

IMPORT Heaps, DataADT, SWholeIO, STextIO;
FROM Heaps IMPORT
    TraverseKind;

```

```

VAR
    theHeap : Heaps.Heap;
    sum : CARDINAL;
    dataRet: DataADT.DataType;

```

```

PROCEDURE Summit (item : DataADT.DataType);
(* a procedure to use in a test traverse *)
BEGIN
    sum := sum + DataADT.GetKey (item)
END Summit;

```

```

(* The following procedures are used to print out the tree looking a little like a
tree *)

```

```

PROCEDURE WriteSpace (n:CARDINAL);
(* write a specified number of spaces *)

```

```

VAR
    count : CARDINAL;
BEGIN
    FOR count := 1 TO n
        DO

```



```

        STextIO.WriteChar (" ");
    END;
END WriteSpace;

(* these need to be global as both procs manipulate them *)
VAR
    count, rowEnd, space : CARDINAL;

PROCEDURE AltWriteData ( item : DataADT.DataType);
(* write out a heap item followed by some space.
If at row end, start a new row and adjust spacing for that row. *)
BEGIN
    IF count = rowEnd
    THEN
        STextIO.WriteLine;
        space := space DIV 2;
        IF space # 0
        THEN
            WriteSpace (space-1);
        END;
        rowEnd := rowEnd*2 +1;
    END;
    DataADT.WriteData (item);
    INC (count);
    IF (space # 0) AND (count # rowEnd)
    THEN
        WriteSpace (2*space-1);
    END;
END AltWriteData;

PROCEDURE WriteHeap ( heap : Heaps.Heap);
(* Writes a heap in a way that resembles a tree.
Won't work very well except to write a number, say a key. *)
VAR
    size : CARDINAL;

BEGIN
    Heaps.SetTraverseKind (theHeap,rowOrder);
    (* compute spacing parameters based on size of heap *)
    size := Heaps.Size(heap);
    space := 1;
    WHILE space <= size
    DO
        space := 2 * space;
    END;
    (* so, it's empirical.  Experiment. *)
    space := 2 * space - 1;
    count := 0;
    rowEnd := 0;
    Heaps.Traverse (heap, AltWriteData);
    STextIO.WriteLine;
    STextIO.WriteLine;
END WriteHeap;

```

BEGIN

```
Heaps.Init (theHeap);
Heaps.Add (theHeap, 54);WriteHeap (theHeap);
Heaps.Add (theHeap, 87);WriteHeap (theHeap);
Heaps.Add (theHeap, 27);WriteHeap (theHeap);
Heaps.Add (theHeap, 67);WriteHeap (theHeap);
Heaps.Add (theHeap, 19);WriteHeap (theHeap);
Heaps.Add (theHeap, 31);WriteHeap (theHeap);
Heaps.Add (theHeap, 29);WriteHeap (theHeap);
Heaps.Add (theHeap, 18);WriteHeap (theHeap);
Heaps.Add (theHeap, 32);WriteHeap (theHeap);
Heaps.Add (theHeap, 56);WriteHeap (theHeap);
Heaps.Add (theHeap, 7);WriteHeap (theHeap);
Heaps.Add (theHeap, 12);WriteHeap (theHeap);
Heaps.Add (theHeap, 31);WriteHeap (theHeap);

STextIO.WriteString ("*****forward traverses*****"); STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,inOrder);
STextIO.WriteString ("in  :");
Heaps.Traverse (theHeap, DataADT.WriteData);STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,preOrder);
STextIO.WriteString ("pre :");
Heaps.Traverse (theHeap, DataADT.WriteData);STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,postOrder);
STextIO.WriteString ("post:");
Heaps.Traverse (theHeap, DataADT.WriteData);STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,rowOrder);
STextIO.WriteString ("row :");
Heaps.Traverse (theHeap, DataADT.WriteData);STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,inOrder);
STextIO.WriteString ("*****end forward traverses*****");
STextIO.WriteLine;STextIO.WriteLine;

Heaps.ReverseTraverseDirection(theHeap);

STextIO.WriteString ("*****reverse traverses*****"); STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,inOrder);
STextIO.WriteString ("in  :");
Heaps.Traverse (theHeap, DataADT.WriteData);STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,preOrder);
STextIO.WriteString ("pre :");
Heaps.Traverse (theHeap, DataADT.WriteData);STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,postOrder);
STextIO.WriteString ("post:");
Heaps.Traverse (theHeap, DataADT.WriteData);STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,rowOrder);
STextIO.WriteString ("row :");
Heaps.Traverse (theHeap, DataADT.WriteData);STextIO.WriteLine;
Heaps.SetTraverseKind (theHeap,inOrder);
STextIO.WriteString ("*****end reverse traverses*****");
STextIO.WriteLine;STextIO.WriteLine;
```

```

(* look for something that is supposed to be there *)
IF Heaps.Search (theHeap,31,dataRet)
  THEN
    STextIO.WriteString ("data found OK as ");
    DataADT.WriteData (dataRet);
  ELSE
    STextIO.WriteString ("31 not found");
  END;
STextIO.WriteLine; STextIO.WriteLine;

(* and for something that is not *)
IF Heaps.Search (theHeap,100,dataRet)
  THEN
    STextIO.WriteString ("data found OK as ");
    DataADT.WriteData (dataRet);
  ELSE
    STextIO.WriteString ("100 not found");
  END;
STextIO.WriteLine;STextIO.WriteLine;

(* now traverse the heap and add everything up *)
sum := 0;
Heaps.Traverse (theHeap, Summit);
STextIO.WriteLine;
STextIO.WriteString ("Sum is ");
SWholeIO.WriteCard (sum, 10);
STextIO.WriteLine;STextIO.WriteLine;

(* now, try a few deletes *)
Heaps.ReverseTraverseDirection(theHeap);
Heaps.Delete (theHeap, 31);WriteHeap (theHeap);
Heaps.Delete (theHeap, 67);WriteHeap (theHeap);
Heaps.Delete (theHeap, 19);WriteHeap (theHeap);
Heaps.Delete (theHeap, 7);WriteHeap (theHeap);
Heaps.Delete (theHeap, 42);WriteHeap (theHeap);
Heaps.Add (theHeap, 12); WriteHeap (theHeap);

END TestHeaps

```

When this program was run, the following output was collected. The reader should verify that the traverses are in fact all correct.

54

54

87

27

87 54

27
67 54
87

19
27 54
87 67

19
27 31
87 67 54

19
27 29
87 67 54 31

18
19 29
27 67 54 31
87

18
19 29
27 67 54 31
87 32

18
19 29
27 56 54 31
87 32 67

7
18 29
27 19 54 31
87 32 67 56

7
18 12
27 19 29 31
87 32 67 56 54

7
18 12
27 19 29 31
87 32 67 56 54 31

*****forward traverses*****
in : 87 27 32 18 67 19 56 7 54 29 31 12 31
pre : 7 18 27 87 32 19 67 56 12 29 54 31 31
post: 87 32 27 67 56 19 18 54 31 29 31 12 7
row : 7 18 12 27 19 29 31 87 32 67 56 54 31
****end forward traverses*****

*****reverse traverses*****
in : 31 12 31 29 54 7 56 19 67 18 32 27 87
pre : 7 12 31 29 31 54 18 19 56 67 27 32 87
post: 31 31 54 29 12 56 67 19 32 87 27 18 7
row : 31 54 56 67 32 87 31 29 19 27 12 18 7
****end reverse traverses*****

data found OK as 31

100 not found

Sum is 470

7
18 12
27 19 29 31
87 32 67 56 54

7
18 12
27 19 29 31
87 32 54 56

7
18 12
27 54 29 31
87 32 56

12
18 29
27 54 56 31
87 32

12
18 29
27 54 56 31
87 32

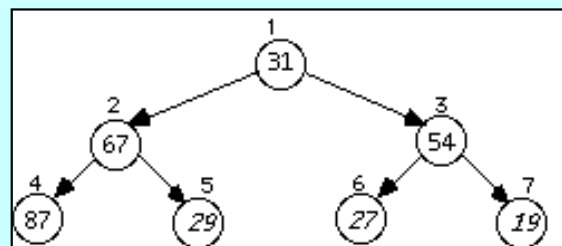
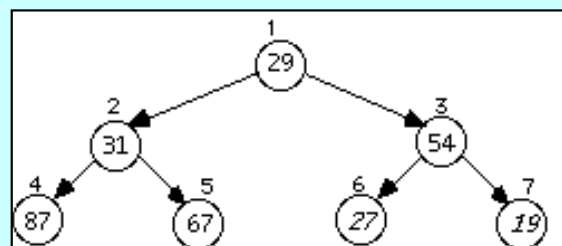
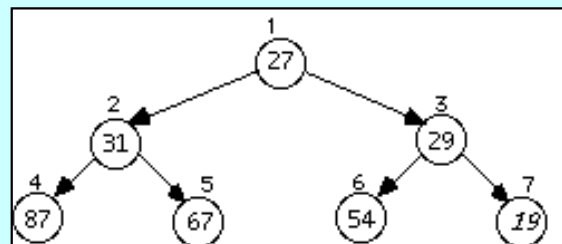
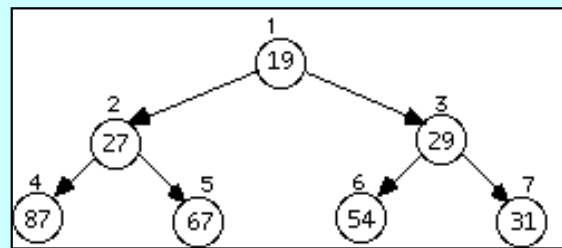
12
12 29
27 18 56 31

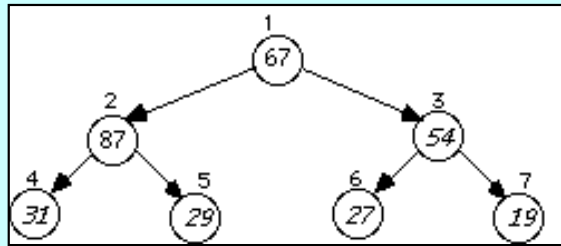
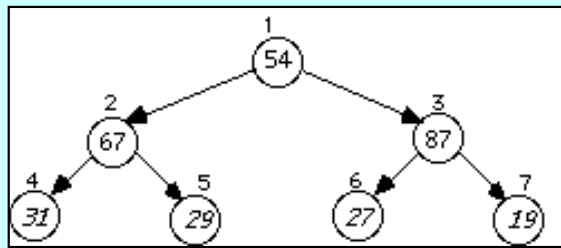
Contents

15.5 Array Implementation and Sorting With Heaps

In the tree implementation of the heap, both binary tree pointers and linear pointers were employed so as to diversify the traversability of the tree. However, if it is noted that when the nodes are numbered in row order, the children of a node_i are node_{2i} and node_{2i+1}, and that conversely, the parent of a node_i is node_{iDIV2}, it becomes apparent that the tree structure is not required for the implementation, and that an array could be used instead. The advantage of using an array is that items can be randomly accessed by their node number, and the necessity for all the linkage is removed. However, the linkage does allow for efficient binary type searches in the tree, so the advantage is not all to the array implementation.

However, there is an exception to this even handed discussion, and it arises from the observation that it is possible to use a heap to generate sorted data. Consider some data that is already in a heap. By virtue of this, the root is the smallest item. Swap it with the last item, mark it as "sorted" and sift the new root down the smaller branch (but not into the sorted part) until it is again a proper child. The root is now the second largest item. Repeat the above process, swapping with the second last item. If we trace this with a seven item heap, and mark the sorted items in bold, the steps are:





The largest item is now at the root, and the tree can now be traversed in reverse row order to visit the items from least to greatest. The structure is now both sorted and is a reverse (inverted) heap. All this could be implemented and added to the tree implementation shown above, or it could be implemented as a sorting routine acting on arrays in the same manner as the ones in Chapter 13.

15.5.1 Heapsort

In the simple array implementation of the heapsort that follows, the parameters of the heapsort procedure are the same as in earlier sorting routines. The array is heapified by starting at the middle element and working back to the first element, sifting down as one goes. It is then sorted by the method outlined above.

```

MODULE HSort;

(* A simple demonstration of heap sorting
   by R. Sutcliffe
   modified 1996 10 22 *)

FROM STextIO IMPORT
    WriteLn, WriteChar;
FROM SWholeIO IMPORT
    WriteCard;

CONST
    max = 18;
VAR
    source : ARRAY [1..max] OF CARDINAL; (* the array to sort *)

PROCEDURE WriteArray;
(* writes out the elements comma delimited *)
VAR
    count : CARDINAL;
BEGIN
    FOR count := 1 TO max
    DO
        WriteCard (source [count], 3);
        IF count # max
        THEN
            WriteChar (",");

```



```

        ELSE
            WriteLn;
        END;
    END;
END WriteArray;

PROCEDURE Swap (VAR one, two : CARDINAL);
    (* Exchange values in one and two. *)
VAR
    temp : CARDINAL;
BEGIN
    temp := one; one := two; two := temp;
END Swap;

(* Since arrays to sort are open, they are numbered from 0.. n-1, so the child
functions have to be altered from the way in which they are defined to add one to the
index *)

PROCEDURE LeftChild (n : CARDINAL) : CARDINAL;
BEGIN
    RETURN 2 * n + 1;
END LeftChild;

PROCEDURE RightChild (n : CARDINAL) : CARDINAL;
BEGIN
    RETURN 2 * n + 2;
END RightChild;

PROCEDURE SiftDown (VAR source : ARRAY OF CARDINAL; node, end : CARDINAL);
(* Sift data item down through heap until it is a proper child.  If it is already in
the right place, nothing happens. *)
VAR
    data : CARDINAL;
BEGIN
    (* set data item from node aside *)
    data := source [node];
    (* see if it needs to go down the tree *)
    WHILE ((LeftChild (node) <= end) AND
        ( data <= end) AND (data > end)
            OR (source [LeftChild (node)] <= source [RightChild (node)]))
        THEN
            source [node] := source [LeftChild (node)];
            node := LeftChild (node);
        ELSE
            source [node] := source [RightChild (node)];
            node := RightChild (node);
        END;
    END;
    (* put data into proper place *)
    source [node] := data;
END SiftDown;

PROCEDURE HeapSort (VAR source: ARRAY OF CARDINAL;

```

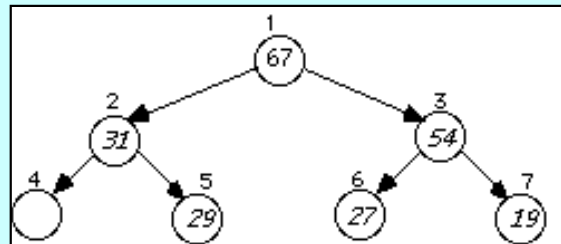
```

    lBound, uBound : CARDINAL);
(* The heapsort is a two step process.  First, the data are put in a heap, and then
the data is de-heaped in reverse sorted order. *)
VAR
    count : CARDINAL;

BEGIN
    (* Construct the initial heap. *)
    (* We do this by starting at the last item that has a child and sifting down, then
backing up toward the top one step at a time and repeating. *)
    IF (uBound <= HIGH (source)) AND (uBound > lBound
        DO
            DEC (count);
            SiftDown (source, count, uBound);
        END; (* while count *)

    (* Now de-heap smallest item to largest *)
    count := uBound;
    WHILE count

```



As this is a reverse sorted heap, the node with priority 19 does not move when sift up is called.

A complete implementation is not shown here, but is left as an exercise for the reader.

[Contents](#)

15.6 Toward More Generic Structures

Throughout this book, the principles of code reusability have been heavily promoted. Where possible, routines have been made multi-purpose, and the structures employed so far (such as the list, stack, trees, etc.) have been capable of modification by using a different data module to import from and recompiling.

In order to write these more generic routines than these, we could try to construct them in such a way that they can be used without resorting to recompiling for every new data type. That is, a list is just that, and nothing more. It is conceived of as a list, not as a list of something. One very general way to work with such abstractions is to ensure that the items that are to be listed are of the type `ARRAY OF LOC`. If this is done, any type of data is compatible. When a new structure, such as a list, is created, the size of the items to be placed in the structure must be supplied, and then whenever space for a data item is required in the implementation module, `ALLOCATE` can be called directly on a pointer variable to fetch some dynamic memory. The declaration of the list and node types could look like:

TYPE

```
NodePoint = POINTER TO Node;
```

```
Node =
```

RECORD

```
    dataLoc : ADDRESS;
```

```
    next,
```

```
    last : NodePoint;
```

END;

```
List = POINTER TO ListInfo;
```

```
ListInfo =
```

RECORD

```
    dataSize,
```

```
    numItems : CARDINAL;
```

```
    head,
```

```
    tail,
```

```
    curInsert,
```

```
    curDelete,
```

```
    curFetchup : NodePoint;
```

```
    delAtHead : BOOLEAN;
```

END;

When memory is needed for a new node and its associated data, one could call:

```
ALLOCATE (local, SIZE(Node) ); (* get a new node *)
```

```
    ALLOCATE (local^.dataLoc, list^.dataSize); (* get space for actual data *)
```

However, a difficulty that must still be overcome is that of handling assignments. One cannot simply write (in the implementation module)

```
theNode.dataLoc^ := itemPassedIn;
```

where *itemPassedIn* is of type `ARRAY OF LOC`, as there will be a type conflict error. The relevant field is an address, not a pointer to anything in particular. To get around this problem, it is necessary to first build two routines to copy data—one from

an address into the contents of a variable represented as an ARRAY OF LOC, and the other to do the reverse. This can be done as follows:

15.6.1 Low Level Assignment Routines

DEFINITION MODULE CopyLocs;

```
(* Low level copying routines to move data around by locs *)
(* copyright © 1995 by R. Sutcliffe *)
(* last modification 1996 01 03 *)
```

FROM SYSTEM **IMPORT**

ADDRESS, LOC;

PROCEDURE CopyToAdr (source: **ARRAY OF** LOC; adr: ADDRESS);

```
(* Pre: the user has SIZE (source) LOCs available to use at adr
   Post: the source item is copied to the locations starting at the given address *)
```

PROCEDURE CopyFromAdr (adr: ADDRESS; **VAR** dest: **ARRAY OF** LOC);

```
(* Pre: none, though the items at adr ought to be meaningful to the program
   Post: the SIZE (dest) LOCs strting at adr are copied into dest. *)
```

END CopyLocs.

IMPLEMENTATION MODULE CopyLocs;

```
(* Low level copying routines to move data around by locs *)
(* copyright © 1995 by R. Sutcliffe *)
(* last modification 1995 03 31 *)
```

FROM SYSTEM **IMPORT**

ADDRESS, LOC, ADDADR;

PROCEDURE CopyToAdr (source: **ARRAY OF** LOC; adr: ADDRESS);

VAR

count: **CARDINAL**;

BEGIN

```
FOR count := 0 TO HIGH (source) (* can't do SIZE on open array *)
DO
  adr^ := source [count];
  adr := ADDADR (adr, 1);
END;
```

END CopyToAdr;

PROCEDURE CopyFromAdr (adr: ADDRESS; **VAR** dest: **ARRAY OF** LOC);

VAR

count: **CARDINAL**;

BEGIN

```
FOR count := 0 TO HIGH (dest)
```

```
DO
    dest [count] := adr^;
    adr := ADDADR (adr, 1);;
END;
END CopyFromAdr;

END CopyLocs.
```

[Contents](#)

15.7 A Generic List Type With ARRAY OF LOC

With the question of assignment out of the way, the Lists module of chapter 12 can be modified to be entirely generic, using ARRAY OF LOC for its parameters, and the above procedures for assignments. However, there is a down side to the increased flexibility, and that is that ARRAY OF LOC parameters have no type checking whatsoever. This means that anything could be passed to the procedure that adds an item to the list, and there would be no way of checking to ensure the data type is correct. However, the approach shown here is a typical one in ISO Standard Modula-2.

DEFINITION MODULE Lists;

```
(* Generic implementation of lists (not safely type checked) *)
(* copyright © 1995 by R. Sutcliffe *)
(* last modification 1995 03 31 *)
```

FROM SYSTEM IMPORT

LOC;

TYPE

```
List;
Operation = (insert, delete, fetchup);
```

PROCEDURE Create (itemSize : **CARDINAL**) : List;

```
(* Pre: itemSize is the size in storage units of the items to be listed
Post: a new list structure is initialized with length zero.
Insert, delete and fetch/update start out at the head of the list. *)
```

PROCEDURE Discard (VAR list : List);

```
(* Pre: list is a validly created list
Post: list is undefined *)
```

PROCEDURE Length (list : List) : **CARDINAL**;

```
(* Pre: list is a validly created list
Post: The number of items in the list is returned. *)
```

PROCEDURE SetAtHead (VAR list : List; op : Operation);

```
(* Pre: list is a validly created list
Post: The position for the given insert, delete, or fetch/update
operation is the first item. *)
```

PROCEDURE SetAtTail (VAR list : List; op : Operation);

```
(* Pre: list is a validly created list
Post: The position for the given insert, delete, or fetch/update
operation is the last item. *)
```

PROCEDURE SetAtPos (VAR list : List; op : Operation; itemNum : **CARDINAL**);

```
(* Pre: list is a validly created
Post: The position for the given insert, delete, or fetch/update
operation is the itemNum item. If ItemNum > 0
```

DO

```
Delete (list);
```

```

    END;
DEALLOCATE (list, SIZE(ListInfo) );
list := NIL;
END Discard;

PROCEDURE Length (list : List) : CARDINAL;

BEGIN
    RETURN list^.numItems;
END Length;

PROCEDURE SetAtHead (VAR list : List; op : Operation);

BEGIN
    CASE op OF
        insert:
            list^.curInsert := list^.head |
        delete:
            list^.curDelete := list^.head;
            list^.delAtHead := TRUE; |
        fetchup:
            list^.curFetchup := list^.head;
    END;
END SetAtHead;

PROCEDURE SetAtTail (VAR list : List; op : Operation);

BEGIN
    CASE op OF
        insert:
            list^.curInsert := list^.tail |
        delete:
            list^.curDelete := list^.tail;
            list^.delAtHead := FALSE; |
        fetchup:
            list^.curFetchup := list^.tail;
    END;
END SetAtTail;

PROCEDURE SetAtPos (VAR list : List; op : Operation; itemNum : CARDINAL);

VAR
    count : CARDINAL;
    tempPoint : NodePoint;

BEGIN
    IF itemNum = 0
    THEN
        SetAtHead (list, op);
    ELIF itemNum > (list^.numItems DIV 2) THEN (* past middle? *)
        count := list^.numItems;
        tempPoint := list^.tail; (* start at the back *)
        WHILE count < itemNum    (* go forward if necessary *)
        DO

```

```

        tempPoint := tempPoint^.next;
        INC (count);
    END;
END;
CASE op OF
    insert:
        list^.curInsert := tempPoint |
    delete:
        list^.curDelete := tempPoint;
        list^.delAtHead := FALSE; |
    fetchup:
        list^.curFetchup := tempPoint;
    END;
END;

```

```

END SetAtPos;

```

```

PROCEDURE GetNode (list : List; item :ARRAY OF LOC) : NodePoint;

```

```

(* This is a local procedure *)

```

```

VAR

```

```

    local : NodePoint;

```

```

BEGIN

```

```

    ALLOCATE (local, SIZE(Node) ); (* get a new node *)

```

```

    ALLOCATE (local^.dataLoc, list^.dataSize); (* get space for actual data *)

```

```

    CopyToAdr (item, local^.dataLoc); (* put data there *)

```

```

    RETURN local;

```

```

END GetNode;

```

```

PROCEDURE Insert (VAR list : List; item : ARRAY OF LOC);

```

```

VAR

```

```

    local : NodePoint;

```

```

BEGIN

```

```

    local := GetNode (list, item);

```

```

    local^.next := list^.curInsert;

```

```

    IF list^.curInsert # list^.head (*inserting at head? *)

```

```

    THEN (* no, so chain in new item *)

```

```

        local^.last := list^.curInsert^.last; (* point back to previous node *)

```

```

        list^.curInsert^.last^.next := local; (* and make it point to new one *)

```

```

    ELSE

```

```

        local^.last := NIL; (* yes, so back pointer is NIL *)

```

```

        IF (list^.curDelete = list^.head) AND list^.delAtHead

```

```

        (* if delete is at head too, keep it there *)

```

```

            THEN

```

```

                list^.curDelete := local;

```

```

            END;

```

```

        IF list^.curFetchup = list^.head (* if fetchUp is at head too, keep it there *)

```

```

            THEN

```

```

                list^.curFetchup := local;

```

```

            END;

```



```

    IF list^.tail = NIL (* if this is the first item in *)
    THEN
        list^.tail := local;
    END;
    list^.head := local; (* revise the head *)
END;
IF list^.curInsert # NIL
THEN
    list^.curInsert^.last := local;
    END;
    list^.curInsert := local; (* insert point becomes new item *)
INC (list^.numItems);

```

END Insert;

PROCEDURE Append (VAR list : List; item : ARRAY OF LOC);

```

VAR
    local : NodePoint;

BEGIN
    local := GetNode (list, item);
    local^.last := list^.tail;
    local^.next := NIL;
    IF list^.tail = NIL (* list currently empty *)
    THEN
        WITH list^
        DO
            head := local;
            curInsert := head;
            curDelete := head;
            curFetchup := head;
        END;
    ELSE
        list^.tail^.next := local;
    END;
    list^.tail := local;
    INC (list^.numItems);

```

END Append;

PROCEDURE Update (VAR list : List; item : ARRAY OF LOC);

```

BEGIN
    CopyToAdr (item, list^.curFetchup^.dataLoc);
END Update;

```

PROCEDURE Fetch (list : List; VAR item : ARRAY OF LOC);

```

BEGIN
    CopyFromAdr (list^.curFetchup^.dataLoc, item);
END Fetch;

```

PROCEDURE Delete (VAR list : List);

```

VAR
    newCurDel : NodePoint;

BEGIN
    IF list^.numItems = 0
    THEN
        RETURN
    END;
DEALLOCATE (list^.curDelete^.dataLoc, list^.dataSize);
    IF list^.curDelete^.last # NIL (* if not at #1 *)
    THEN
        list^.curDelete^.last^.next := list^.curDelete^.next;
    ELSE
        list^.head := list^.curDelete^.next;
    END;
    IF list^.curDelete^.next # NIL (* if not at last item *)
    THEN
        list^.curDelete^.next^.last := list^.curDelete^.last;
        newCurDel := list^.curDelete^.next;
    ELSE
        list^.tail := list^.curDelete^.last;
        newCurDel := list^.curDelete^.last;
    END;
    IF list^.curDelete = list^.curInsert (* hammered off insert item? *)
    THEN
        list^.curInsert := newCurDel;
    END;
    IF list^.curFetchup = list^.curInsert (* hammered off fetchup item? *)
    THEN
        list^.curFetchup := newCurDel;
    END;
    DEC (list^.numItems);
    list^.curDelete := newCurDel;
END Delete;

END Lists.

```

Observe that this version of *Create* tells the list module only the size of the items that are going to be enlisted. Not only does the list module have no means to check the type of the items listed, it does not even have code to check that the correct size is indeed used. The latter oversight however, is simple for the student to correct, and is left as an exercise. The absence of type checking on the items being listed is more serious, however, and is a serious drawback to this method. The very strength of Modula-2 (strong type checking) must be sacrificed here to achieve genericity. Because of this drawback, other methods will also be explored later in chapter 16.

15.8 Pointers and Memory Management Revisited

The implementations of most of the data structures thus far depended heavily on the use of pointers and dynamic memory allocation and deallocation. Such techniques have the advantage that one does not have to say ahead of time how large the structure is to be (as with static types such as arrays). However, dynamic use of memory in and of itself is not without its problems.

15.8.1 Orphans

Consider a piece of code that does, in effect, the following:

```
NEW (p) ;  
q := p ;  
DISPOSE (p) ;
```

In practice, the offending lines will probably be too far away from each other physically or logically for the programmer to notice. The pointer q now points to an area of memory that the program has given up ownership on, and is therefore logically invalid. It is not *NIL*, however, so references to it will appear to work. However, the memory in question can be allocated for another purpose meanwhile, and a value assigned there via yet another pointer is now in direct conflict with the use as the invalid q^{\wedge} . The pointer q has been logically orphaned, and the program that does this is headed for disaster. Such errors are extremely difficult to find, and programs that use a lot of pointers are therefore very prone to errors and very hard to debug. As in most situations where the error is logical in nature, no solution can be offered except:

- plan program logic carefully
- avoid pointers except where necessary
- encapsulate dangerous code in a separate module to make errors easier to find
- test routines one at a time to narrow down pointer errors

15.8.2 Garbage

Now, consider a slightly different piece of code. Again, in real life, the offending lines are likely to be far apart rather than contiguous.

```
NEW (p) ;  
p := q ;
```

That is, the value of a pointer is reused for something else (perhaps instead of base assignment another NEW is performed on it) without first calling DISPOSE to give back the memory that was allocated. The chunk of memory on the heap that p pointed to after the call to NEW (p) remains allocated, but the program has now lost the means to refer to it, because it no longer has a pointer variable with the correct value.

Memory that a program allocates and then loses its reference to without first deallocating is called garbage.

Once again, little can be offered as a solution in such cases except to appeal to the programmer to take more care with program logic. Some debugging environments also offer the use of tools to monitor memory allocation and determine whether or not the program is "leaking" memory and generating a garbage heap.

15.8.3 Fragmentation

Suppose that a program has been working for some time, allocating and deallocating memory in the heap area reserved for the program. The sizes of the allocation blocks that have been used may vary considerably, and the memory map becomes a patchwork of allocated and free blocks of varying sizes as time goes on. It is not difficult to get to the point where one asks for a new block of, say 100 LOCs and the memory manager discovers that the largest available block is only, say 70 LOCs. Now, there may be dozens of such blocks, for a total many times what is needed, but because the memory is not all in one place, the allocation fails.

If not all the memory available to a program is in a single contiguous block, the memory is said to be fragmented.

The small free blocks are sometimes loosely called *garbage*, but this is not quite correct, because the program does have access to them individually if it asks only for a small amount of memory. Slight memory fragmentation probably has little or no effect on program performance. However, heavy use by a program of dynamic memory in blocks of varying sizes will eventually cause severe fragmentation and may mean that the program can only operate for a certain length of time, and then invariably fails.

15.8.4 Defragmentation and Garbage Collection

Some operating systems have a means to defragment and/or do garbage collection on the dynamic memory automatically. The two tasks are sometimes grouped under a single heading as "garbage collection" though this is not quite accurate. A defragmenter might pause when asked for memory and try to move things around so that a chunk becomes available, then allocate from that. It may try to determine what memory is no longer pointed to by a program pointer variable and mark it as available. However, such a behaviour creates a serious problem for a program that depends on knowing the values of pointer variables. There is no way for the memory manager to know how the pointers have been used

in the logic of the program. Unless it could guarantee that *every* pointer variable that pointed to a specific location was known by the memory manager and changed at the same time, things could go seriously awry. Consider the code:

```
heap^.root := temp;  
heap^.lowerLeft := temp;  
heap^.last := temp;
```

As this code concludes, the pointer to a specific piece of memory is held in four places altogether. What if this value changes while the assignments are being done? What if the memory manager changes one of the pointers but not all of them? What if the pointer has been CAST into a variable of some other type to do some low level work? What if the code calls for the pointer to be incremented using SYSTEM.ADDADR and the memory consolidator changes the value of the pointer in the middle of the loop that is doing this?

The problems with garbage collection are similar. Unless the collector routines know about all the pointer variables that point to a location and can therefore deallocate memory in a way that is guaranteed to be error free, the integrity of a program that uses pointers cannot be maintained.

Dynamically allocated memory, all the references to which are guaranteed to be known to the memory manager is said to be traced. Other dynamically allocated memory is said to be untraced.

NOTE: In common with most procedural languages such as Pascal and C, base-standard Modula-2 does not guarantee tracing of memory allocated by NEW and referred to by pointer variables.

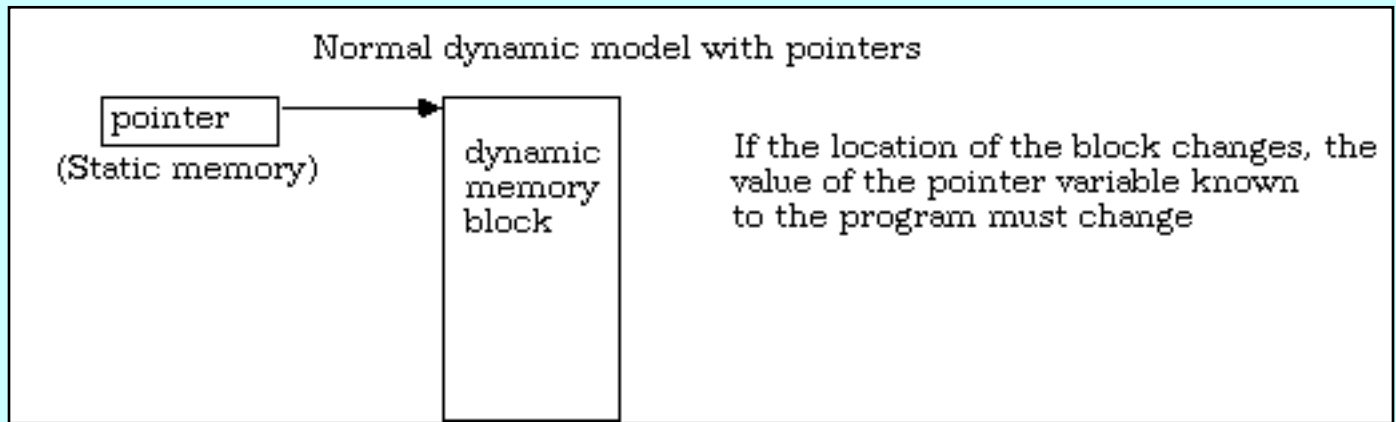
Thus, at least with respect to garbage, Modula-2 offers no solutions to the programmer who creates it with ill-thought-out code. If a programmer decides to use pointers and therefore takes on the task of memory management, then the job must be done completely. The base-standard Modula-2 rule is:

I'm not your mother. Clean up your own garbage.

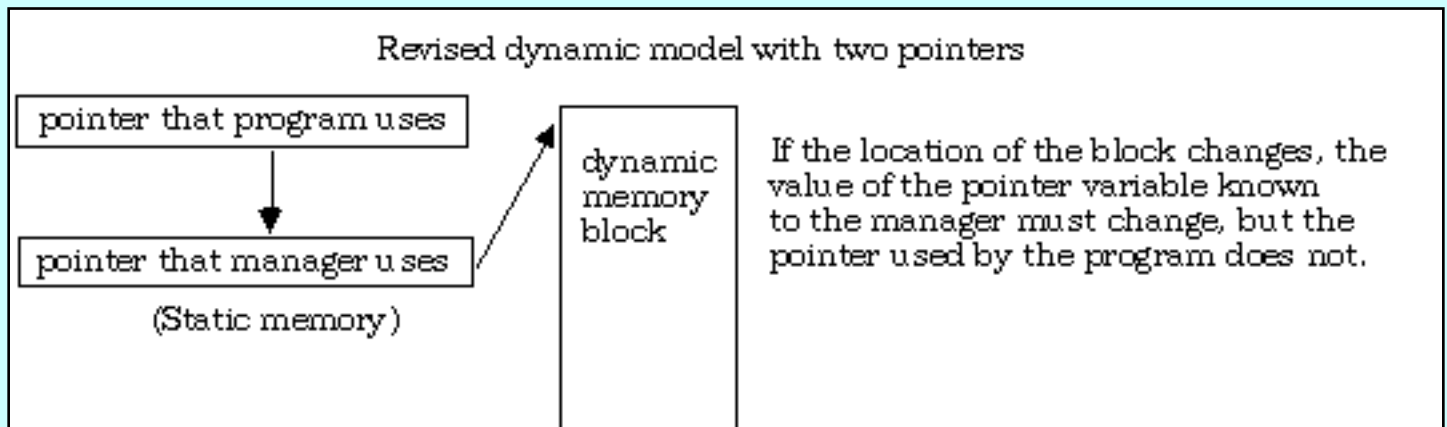
NOTE: If the programmer has the ISO standard object oriented extensions to the base language and uses them appropriately, some automatic garbage collection becomes available. (See a later chapter for details.)

15.8.5 Handles

There may in some cases be a way around the problem of memory fragmentation, however. Provided that the system actually does have a defragmenter in the memory manager, and provided that the programmer cooperates with it, the programmer may be able to designate parts of the heap as "moveable" and allow the memory manager to defragment those sections other parts are "locked" and no defragmentation is allowed in those sections. The details of how to do this are, of course, system dependent. However, the way shown here for the program to maintain its integrity in the face of possible memory moves is fairly common.



Instead of the normal scheme, a program cooperates in integrity with a defragmenting memory manager by having two pointers, one in static memory, and the other either in static memory, or at least in locked (nonrelocatable) memory. The first pointer is actually used by the program, but only points to the second pointer. The second pointer is never referenced directly by the program, and points to the dynamic memory. The defragmenter is then free to move the memory about, changing the second pointer, and the value of the first will remain untouched, because the location of the second does not change just because its value does.



A pointer to a pointer to data is called a handle.

The declarations could look like this:

TYPE

```

MyDataHandle = POINTER TO POINTER TO Node;
Node =
    RECORD
        fields
    END;

```

VAR

```

theHandle : MyDataHandle;

```

```
thePointer : POINTER TO Node; (* not needed if handle is a type *)  
temp : Node;
```

In the code, references to the data are done like this:

```
theHandle^^ := temp;
```

References via handles are said to be doubly dereferenced or indirectly referenced.

That is, if *hand* is a handle, then *hand*[^] is a pointer and *hand*^{^^} is the data.

In standard Modula-2 the variables would have to be set up in the first place, and this might be thought of as:

```
theHandle := CAST (MyDataHandle, SYSTEM.ADR (thePointer));  
NEW (thePointer);
```

However, in order for the memory manager to actually trace the variables it needs to, there are usually some non-standard reserved words added to the language, and the code looks like this:

```
NEWHANDLE (theHandle);
```

If it is done this way, the handle is the only variable used in the program. It neither declares nor directly knows about the pointer variable.

WARNING: The details will vary from one system to another. There may even be a non-standard pervasive **HANDLE** that is used in place of **POINTER TO POINTER**, so that one writes **HANDLE TO**. References to the data are still done with double indirection, however.

WARNING: The programmer must cooperate with this arrangement by carefully avoiding any single references such as *theHandle*[^], as the value stored there is subject to change at any time by the defragmenter.

While the information here is necessarily incomplete, the ideas and vocabulary ought to be sufficient to allow the programmer to find the additional material in the system documentation that is needed to make all this work.

Examples: Since the size and resolution of monitors varies greatly from one computer to another, the record that comprises the information shown on the screen is usually held in a relocatable block of memory, so that it can be re-sized dynamically. Any program referring to this section of memory must do so through a handle rather than a pointer. The same is often true of other blocks of system information such as those allocated to file records.

15.9 Pointers and Generic Structures

With all the caveats of the previous section ringing in ones ears, it scarcely seems appropriate to propose a scheme for more generic data types that uses even more pointers than thus far. However, as seen in [section 15.7](#), standard Modula-2 has some limitations that need to be worked around if one is going to use structures like the list, stack, queue, and trees in the most generic way possible, that is, where the code can be written and compiled without reference to the type itself.

Yet another solution to the problem of making code is not to have any actual data in the data node, only its address. If the code can be constructed to work this way, there can be some efficiencies as well, because there will be a lot of copying that can be avoided.

With this in mind, here is yet another attempt to make the creation and management of lists generic. This time, only a stripped down definition module is presented, and the implementation is left to the student. Even the comments have been removed, as they are the same as in the version found in [section 15.6](#).

```
DEFINITION MODULE Lists;
```

```
(* Generic implementation of lists (not safely type checked) *)
(* copyright © 1995 by R. Sutcliffe *)
(* last modification 1995 03 31 *)
```

```
FROM SYSTEM IMPORT
  ADDRESS;
```

```
TYPE
  List;
  Operation = (insert, delete, fetchup);
```

```
PROCEDURE Create () : List;
```

```
PROCEDURE Discard (VAR list : List);
```

```
PROCEDURE Length (list : List) : CARDINAL;
```

```
PROCEDURE SetAtHead (VAR list : List; op : Operation);
```

```
PROCEDURE SetAtTail (VAR list : List; op : Operation);
```

```
PROCEDURE SetAtPos (VAR list : List; op : Operation; itemNum : CARDINAL);
```

```
PROCEDURE Insert (VAR list : List; item : ADDRESS);
```

```
PROCEDURE Append (VAR list : List; item : ADDRESS);
```



```
PROCEDURE Update (VAR list : List; item : ADDRESS);  
  
PROCEDURE Fetch (list : List; VAR item : ADDRESS);  
  
PROCEDURE Delete (VAR list : List);  
  
END Lists.
```

The implementation of this module is left as an exercise for the reader.

[Contents](#)

15.10 Addresses and Generic Sorting Techniques

One of the techniques that made the generic approach to data structures work in the first place was taking care to have all operations such as comparisons located in the module that defined the ADT. This works fairly well in the case of lists, where not much comparison work needs to be done. However, the very warp and woof of inserting and deleting in some structures is tied up with the making of comparisons. In the case of binary trees, B-trees, and heaps, this was accomplished by importing a compare procedure from the module defining the ADT. However, if truly generic software is required, this will not quite do.

It will also not quite do if the generic routine being considered is a sort. Suppose one wanted to write a sort routine that could sort any kind of data. It has already been shown that the only two routines that need to know about the data type are *Swap* and *Compare*. As shown in section 8.4.3, *Swap* can be made absolutely generic, because all that is involved is moving around a specified number of memory locations, and this can be achieved without the procedure having to know anything about the type, though at the usual cost that the *Swap* is not able to check for type. One simply writes:

```
PROCEDURE Swap (VAR a, b : ARRAY OF CHAR);
```

The question that remains is, can the comparison procedure be made generic in the same way, and then passed as a parameter to the module doing sorting. One might like to have:

TYPE

```
CompareResults = (less, equal, greater);  
CompareProcs = PROCEDURE (ARRAY OF LOC, ARRAY OF LOC) : CompareResults;
```

and in the sorting module, a procedure like

```
PROCEDURE SetSortingProc (theProc : CompareProcs);
```

One might be tempted to suppose that one could then write such a procedure as

```
PROCEDURE CompareCards (a, b : CARDINAL) : CompareResults;
```

and issue a call

```
SetSortingProc (CompareCards);
```

Alas, an attempt to do this produces an error, because the parameters of a procedure assigned to a procedure variable must exactly match those in the type of the procedure variable, not just be assignment compatible as are **CARDINAL** and **ARRAY OF LOC**.

What to do? Suppose, instead of the above, one defined:

TYPE

```
CompareProcs = PROCEDURE (ADDRESS, ADDRESS) : CompareResults;
```

and

```
PROCEDURE CompareCards (a, b : ADDRESS) : CompareResults;
```



```
PROCEDURE QuickSort (VAR source: ARRAY OF ADDRESS;  
    lBound, uBound : CARDINAL);  
  
PROCEDURE SetCompareProc (theProc : compareProc);  
(* must be set by caller *)  
  
END pSorting.
```

In principle this kind of definition suffices to develop generic sorting routines. The same objections to this kind of genericity are present as for those of the generic list routines discussed above. There is no longer any type checking whatsoever. If the programmer is happy with that loss, such routines may suffice. If all that is available is standard Modula-2, such routines *must* suffice. However, there is at least one other way of approaching the question of writing generic routines and data structures, and it will be presented in Chapter 16 in a discussion of Standard Generic Modula-2.

[Contents](#)

15.11 Chapter Summary

This chapter covered these topics:

- B-trees, their definition and implementation
- Heaps, their definition and implementation
- Heapsort
- A discussion of generic structures and techniques
- Two approaches to generic lists
- Generic sorting methods using pointers
- Advanced memory management topics
- Fragmentation and garbage

It included discussion of the following Modula-2 built-ins:

Non-Standard Pervasive Identifiers

HANDLE

NEWHANDLE

[Contents](#)

15.12 Assignments

Questions

1. What does the "B" in B-tree stand for?
2. Explain all differences between B-trees and binary trees.
3. Devise a formula that takes as input the order of a B-tree and the number of levels it has, and produces the maximum storage capacity of the tree.
4. If a B-tree of order 2 has been employed to contain n items, what is the maximum number of levels (depth) that need to be searched to find an item?
5. Suppose the B-tree is changed to order three. Trace by hand what the data provided for testing should produce with each insertion and deletion, then recompile and relink run the test program with this change made and check your predictions
6. Define a heap. Define a reverse heap.
7. What does the word "generic" mean in the contest of this book so far, and why should software be written generically?
8. Outline the three methods detailed thus far for making software generic.
9. What are the major advantages and disadvantages of using ARRAY OF LOC or ADDRESS instead of the actual data type in a structure?
10. The cardinal data 23, 54, 2, 98, 34, 17, 9, 27, 31, 3, 78, 9, 54, 12 arrives at a B-tree of order two. Show how the data is entreed.
11. The same data as in question 10 arrives at a B-tree of order three. Show how the data is entreed.
12. The same data as in question 10 arrives at a heap and is enheaped as it arrives. Show the result after each step.
13. The same data as in question 10 is in an array and is then heapsorted using the two step process detailed in the chapter. Show the transformations of the array one step at a time as the heap is first constructed and then sorted.
14. Write a generic swap routine that uses only the addresses of the items to be swapped. What additional parameter will be needed?
15. Look up and then detail here how your implementation deals with handles, if at all.
16. Some languages have built in garbage collections. Find at least three that do.
17. What is the difference between garbage collection and defragmentation?
18. Explain the comment (* don't care about right *) in the unused procedure [IsLeaf](#) found in the Heap implementation in section 15.4.

Problems

19. Add and test routines to traverse the supplied B-tree in a variety of ways, in the same manner as provided for in the Heap structure.
20. Revise the code for inserting an item in a B-tree so that it does not insert in the heap when an identical item is found there.
21. Alternately (it makes no sense to do both) revise the code for deletion from a B-tree so that it deletes

all items with the same key.

22. Create a searchable database to store information on a collection. (Stamps, coins or sports cards will do.) Store the information for searching in the B-tree as it was implemented in this chapter.

23. Implement a B-tree ADT with linked lists of pointers and data nodes. Do the *Init* routine so that it takes a parameter that is the order of the B-tree.

24. Implement a B-tree that entrees a type `ARRAY OF LOC`, rather than a specific imported data type.

25. Implement a B-tree that entrees only the addresses of the data, not the data itself.

26. In the [section on testing the Heap](#), a small routine to print out a heap in a manner resembling an actual tree was set up. Do the same for a B-tree and test it. (The tree view routine supplied in the chapter is rather primitive.)

27. Revise the code for inserting an item in a Heap so that it does not insert in the heap when an identical item is found there.

28. Alternately (it makes no sense to do both) revise the code for deletion from a Heap so that it deletes all items with the same key.

29. Implement a Heap that entrees an `ARRAY OF LOC`, rather than a specific imported data type.

30. Implement a Heap that entrees only the addresses of the data, not the data itself.

31. Implement a priority queue based on a heap as suggested in [section 15.5.2](#).

32. Implement the same heap definition module given in [section 15.3.1](#) as an array rather than as a tree.

33. Re-implement the binary tree of [Chapter 14](#) with all the traverse methods shown in this chapter for the Heap.

34. Implement a structure such as the queue or the stack that operates on `ARRAY OF LOC` rather than on a specific imported data type.

35. Implement a forward (increasing) heapsort. Keep the same parameters for the heapsort itself, but add a procedure to set the direction of sorting as increasing or decreasing. Encapsulate the whole thing in a library module, making it reasonably generic in the process (use one of the three methods in the chapter) and then test it with two different data types.

36. Heapsort was implemented with a sift down procedure only. Re-do this with either a sift up procedure only or both a sift up and sift down procedure.

37. Test Heapsort against Quicksort on data sets of various sizes and determine which is better for random data and for already sorted data.

38. One might hypothesize that a simple sort is more efficient than a Heapsort for small data sets (as seemed to be the case for Quicksort.) Test Heapsort against Insert Sort on data sets of various sizes and determine at what point (if any) it makes sense to switch between the two.

39. Test the supplied generic list type using the `ARRAY OF LOC` using a data type that is a record and has a key field.

40. Implement the generic list type so that only addresses are enlisted, using the definition module in [section 15.9](#).

41. Implement and test the module *pSorting* in [section 15.10.1](#).

42. Conduct a test of sorting records based on a key field and determine whether it really is better to hold only the addresses in an array as opposed to the entire data structure.

Contents

Chapter 15

Advanced Data Types and Techniques

[15.0 Chapter Goals](#)

[15.1.1 B-trees Defined](#)

[15.2 Implementing and Testing a Semi-Generic B-tree](#)

[15.3 Heaps](#)

[15.3.1 Heaps Defined](#)

[15.3.2 Heaps as Binary Trees](#)

[15.3.3 Defining the Heap](#)

[15.4 Implementing and Testing a Semi-Generic Heap](#)

[15.5 Array Implementation and Sorting With Heaps](#)

[15.5.1 Heapsort](#)

[15.6 Toward More Generic Structures](#)

[15.6.1 Low Level Assignment Routines](#)

[15.8 Pointers and Memory Management Revisited](#)

[15.8.1 Orphans](#)

[15.8.2 Garbage](#)

[15.8.3 Fragmentation](#)

[15.8.4 Defragmentation and Garbage Collection](#)

[15.8.5 Handles](#)

[15.9 Pointers and Generic Structures](#)

[15.10.1 Generic Sorts Defined](#)

[15.12 Assignments](#)

16.0 Chapter Goals

The purpose of this chapter is to elaborate on the concepts of developing generic software. On completing the chapter, the student should understand and be able to use the following:

Data Representation Abstractions

No new data types are taken up in this chapter.

Data Manipulation Abstractions

General:

routines that handle data expressed as generic items

Realized in the Modula-2 notation:

formally defined generic routines and their refinements

Programming Abstractions *General:*

expression of abstract data types such as structures in data-independent (generic) ways

Realized in the Modula-2 notation:

formally defined generic modules, refinement as separate modules and as local modules, adding and removing ADT facilities using local refinement

16.1 Generics In the Base Language (Revisited)

Chapter 15 included some discussion of the problems encountered when attempting to make various techniques and abstract data types generic, that is, to make them readily applicable to a variety of underlying data types. Here, these goals and the difficulties with them are reviewed so as to set the stage for the introduction of more powerful techniques from ISO standard Generic Modula-2.

16.1.1 Semi Generic Methods and Structures

Chapters 14 and 15 emphasized the need to create conventional data structures in as generic a fashion as possible. The basic technique was to write the major part of the module in terms of some general data type name such as *TheDataDype* and then to have a renaming line at the top where the module was customized to a particular type. As this line is easily changed, the bulk of the code can readily be re-used. Thus, one would write, for instance

```
DEFINITION MODULE IntSorts;  
FROM IntegerInfo IMPORT  
    Compare;  
FROM Comparisons IMPORT  
    CompareResults;  
TYPE  
    Item = INTEGER;  
CONST  
    GenCompare = Compare;  
  
TYPE  
    CompareProc = PROCEDURE (Item, Item) : CompareResults;  
  
PROCEDURE Quick (VAR data : ARRAY OF Item);  
  
    (* Other procedures and functions could be included as well. *)  
END IntSorts.
```

If the issue were the creation of a data structure, one could proceed as follows:

```
DEFINITION MODULE CardStack;  
  
TYPE  
    Element = CARDINAL;
```

```

CONST
    StackSize = 100;

PROCEDURE Push (item : Element);

PROCEDURE Pop (VAR item : Element);

PROCEDURE Empty () : BOOLEAN;

END CardStack.

```

The implementation module can access the renamed item, but its code has to be written in terms of the general name *Element* rather than the specific one *CARDINAL*. A simple change on one line of a copy the definition part of the module followed by a recompilation of both under a different module name then suffices to re-use the code with another numeric type.

```

IMPLEMENTATION MODULE CardStack;
VAR
    stack      : ARRAY [0..StackSize] OF Element;
    stackPtr   : CARDINAL;
    (* One could also arrange for StackSize to be a parameter. *)

PROCEDURE Push (item : Element);
BEGIN
    INC (stackPtr);
    stack[stackPtr] := item;
END Push;

PROCEDURE Pop (VAR item : Element);
BEGIN
    item := stack[stackPtr];
    DEC (stackPtr)
END Pop;

PROCEDURE Empty () : BOOLEAN;
BEGIN
    RETURN stackPtr = 0
END Empty;

BEGIN (* module body initialization *)
    stackPtr := 0
END CardStack.

```

16.1.2 Limitations of Fully Generic Techniques

As indicated in [section 15.10](#), when one attempts to implement a fully generic technique (such as sorting), one is forced to sacrifice type checking, and this is a serious loss given the otherwise closely type-checked style of Modula-2. Consider, for example a simple swap, which in order to be generic has to be written:

```
PROCEDURE Swap (itemA, itemB : ARRAY OF LOC);
```

which is quite unsatisfactory, both because there is now no proper type checking. Moreover one cannot even use the following --

```
TYPE
```

```
    SwapProc = PROCEDURE (VAR ARRAY OF LOC, VAR ARRAY OF LOC);
```

```
    CompareProc = PROCEDURE (ARRAY OF LOC, ARRAY OF LOC) : CompareResults;
```

because specific typed procedures, such as

```
PROCEDURE SwapReal (VAR x, y : REAL);
```

and

```
PROCEDURE CompareReal (x, y : REAL) : CompareResults;
```

cannot be assigned, respectively, to variables of these two types due to the type compatibility rules for parameters. (An **ARRAY OF LOC** is not the same kind of parameter as a **REAL**.)

16.1.3 Limitations of Fully Generic Structures

Such code can also be made entirely generic, but at the same kind of cost as noted for techniques in [section 16.1.1](#). For instance, an ADT such as a list can only be established if, upon creation of an instance of such a list, information is made available about the size of the items to be listed. This is possible by using the following strategy:

```
DEFINITION MODULE Lists;
```

```
TYPE
```

```
    List;
```

```
PROCEDURE Init (Size : CARDINAL) : List;
```

```
PROCEDURE Append (Item : ARRAY OF LOC; To : List);
```

```
    (* etc. *)
```

```
END Lists.
```

which could then be implemented in a manner similar to the following:

```
IMPLEMENTATION MODULE Lists;

IMPORT Storage;
FROM SYSTEM IMPORT
    LOC;

TYPE
    NodePointer = POINTER TO Node;
    Node =
        RECORD
            to : NodePointer
        END;
    List = POINTER TO ListData;
    ListData =
        RECORD
            item_size, length : CARDINAL;
            head : NodePointer
        END;

PROCEDURE Init (Size : CARDINAL) : List;

VAR
    Res : List;

BEGIN
    NEW (Res);
    Res^.item_size := Size;
    Res^.length := 0;
    Res^.head := NIL;
    RETURN Res
END Init;

PROCEDURE Append (Item : ARRAY OF LOC; To : List);

VAR
    Ptr : ADDRESS;
    Last : NodePointer;

BEGIN
    IF To.size = SIZE (item)
        THEN
```

```

ALLOCATE (Ptr, To.size);
IF To.length = 0
    THEN
        To.head := Ptr
    ELSE
        Last := To.head;
        WHILE Last.to <> NIL
            DO
                Last := Last.to
            END;
        Last.to := Ptr;
        Last.to.to := NIL
    END
ELSE
    (* code to raise a suitable exception *)
END
END Append;

END Lists.

```

Notice in this example that, apart from the awkwardness of this method, no safe check can be conducted on the type of the item being enlisted, or on the instance of the list itself. Only the size of the item one is attempting to enlist can be checked, so that if several lists of items of different types (but of the same size) are generated in a single application program, this technique is all but worthless. The only fix evident would be to generate yet another module that would serve as a kind of type registry to generate unique values of a type *TypeId* that would then have to be produced when generating the instance of the generic type and then supplied on every call to one of the procedures so that its value could be checked. Such a scheme might recover safety, but at the expense of both more awkwardness and re-inventing at run time the type checking that in most circumstances is done automatically by the compiler in a static fashion.

16.1.4 Summary

One is forced to conclude that truly generic software is difficult to produce in the base Modula-2 language, and that doing so has significant cost. That is, in the critical matter of producing re-usable code, Modula-2 fails to deliver on safe type-checked techniques, and so is a clumsy tool at best for generic software.

The question is, can the base language rules be modified in a simple and unobtrusive way to allow the development of generic structures and techniques that permit the translator to do its proper job of type checking on the actual parameters when an instance of the structure or technique is employed?

Such questions were raised by R. Sutcliffe at the 1987 meeting of the Modula-2 standards committee, and again in a discussion paper by K. Hopper and R. Sutcliffe at its 1994 meeting. After two years of considering various proposals, the committee decided that the answer is yes. In order to meet these requirements for modern programming facilities, Modula-2 can be extended by introducing parameterised type abstraction modules as described in the following sections.

NOTE: These facilities will only be available as described here in implementations that have included the generic Modula-2 extensions to the base language.

[Contents](#)

16.2 Generic Separate Library Modules

The first step is the introduction of a new type of library module called a generic separate module. This is a new kind of module having formal parameters that may be either type parameters and/or constant value parameters. A generic separate module serves as a template for constructing specific refinements of itself that have been customized using actual types and/or actual constant expressions.

*A generic separate module consists of a generic definition module and a generic implementation module, each prefixed by the reserved word **GENERIC**. The generic definition module is a template from which a specific definition module may be refined. The generic implementation module is a template from which a specific implementation module may be refined.*

NOTE: The process of refinement will be described in later sections of the chapter.

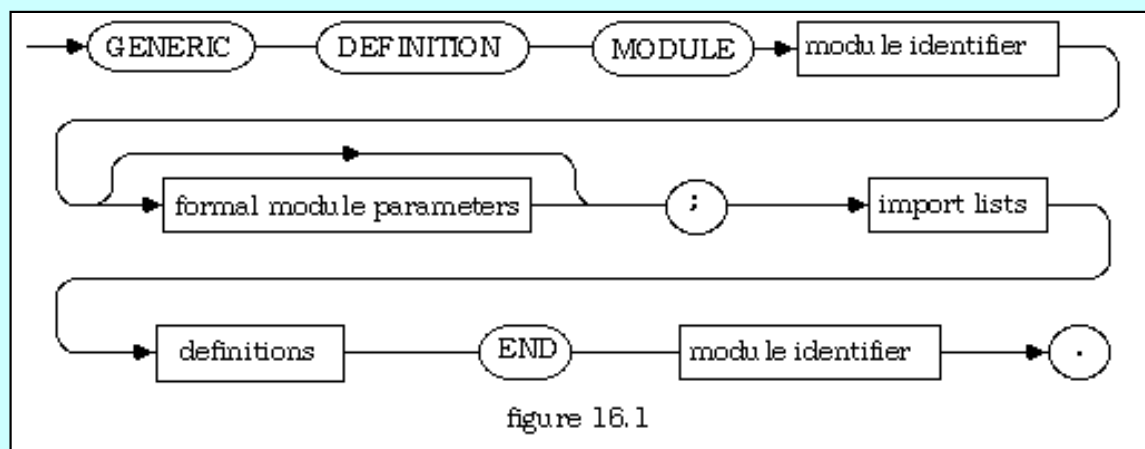
The important thing to note about this is that generic modules have formal parameters, just as do procedure definitions and declarations. The manner in which one substitutes actual parameters will be discussed in the next section. This one will concentrate on defining and declaring the generic modules.

As hinted at by the definition above, a generic module is distinguished from any other module by two things:

1. It starts with the reserved word **GENERIC**. This is how the module is flagged for the benefit of the compiler.
2. It (optionally) has formal parameters. These allow it to be used for a variety of data types. The formal parameters of a generic module are restricted to types and constant values only.

16.2.1 Generic Definition Modules

The definition part of a generic separate module is like any other definition module, except that it begins with the keyword **GENERIC** and it has a formal parameter list. The complete syntax is shown in figure 16.1.



Example 1: Produce a generic definition of a stack

Discussion: In such applications, it is expected that the structure will be manipulated independent of the kind of element it contains, so this data type is provided as a formal type parameter at this stage of things. All the work of programming is in the structure manipulation; the provision of the items to enter into the structure is made when the actual parameters are supplied as shown in the next section.

NOTE: This definition module is not complete; it is only an outline so as to show the syntax of a generic definition module.

```
GENERIC DEFINITION MODULE Stacks (Element: TYPE);
```

```
CONST
```

```
StackSize = 100;
```

```
PROCEDURE Push (item : Element);
```

```
PROCEDURE Pop (VAR item : Element);
```

```
PROCEDURE Empty () : BOOLEAN;
```

```
END Stacks.
```

NOTES: 1. Specification of a constant such as StackSize in the manner of this example constrains all the refinements of this module. If that were not the intention, such an item could instead be made a constant value parameter and then specified by the actual parameter.

2. Such a constant may be used in the corresponding generic implementation module, because as always implementation modules have access to such information in their corresponding definition modules.

3. Observe that the reserved word TYPE is here being used as though it were a type itself. That is, the word TYPE is here employed in the manner a pervasive standard identifier. It is, however, still a reserved word because of the base language definition of it.

Example 2: (A matrix generic as to both size and elements.)

Discussion: In cases like this the data structure itself needs parameterization in order to make the size (of the matrix in this case) generic.

```
GENERIC DEFINITION MODULE Matrix (Rows, Cols : CARDINAL; MatrixElement : TYPE);
```

```
TYPE
```

```
  TMatrix = ARRAY [0 .. Rows-1] OF ARRAY [0 .. Cols-1] OF MatrixElement;
```

```
PROCEDURE Invert (VAR m : TMatrix);
```

```
END Matrix.
```

Example 3: Provide a generic definition of a module that defines a procedure type, and provides a procedure of this type that takes a generic parameter.

Discussion: This illustration defines a generic technique, as opposed to a generic Abstract Data Type (ADT). The definition below has *both* a parameter of the desired type and a procedure of the same type. This is unlikely in practice, as one would normally do one or the other in a given context. However, both approaches are allowed. Note that the type defined in the module can be used in the parameter list. This is a new kind of forward reference and can be employed only in this context.

```
GENERIC DEFINITION MODULE Validate (PType : TYPE; PValidProc : ValidProcType);
```

```
TYPE
```

```
  ValidProcType = PROCEDURE (item : PType) : BOOLEAN;
```

```
PROCEDURE Valid (item : PType) : BOOLEAN;
```

```
END Validate.
```

Example 4: (An outline of a generic sort)

Discussion: This example illustrates the abstraction of a generic technique applied to an existing kind of structure (an array), rather than to the manipulation of a user-defined structure.

```
GENERIC DEFINITION MODULE Sorts (Item : TYPE; GenCompare : CompareProc);
```

```
FROM Comparisons IMPORT
  CompareResults;

TYPE
  CompareProc = PROCEDURE (Item, Item) : CompareResults;

PROCEDURE Quick (VAR data : ARRAY OF Item);

  (* Other procedures and functions would be included as well. *)
END Sorts.
```

Example 5: (Parameters are optional)
Discussion: This example illustrates that although in the majority of cases generic separate modules will be parameterized, this is not required, and non-parameterized instances are also useful. This module defines a *counter*, any number of instances of which may be refined under different names as shown in the next section.

```
GENERIC DEFINITION MODULE Counter;

PROCEDURE Inc;

PROCEDURE Reset;

PROCEDURE Count () : CARDINAL;

END Counter.
```

16.2.2 Generic Implementation Modules

Each of the generic definition parts shown in the previous subsection also needs a corresponding generic implementation. As usual, however, the order in which all this is done depends somewhat on the needs of the project. It is possible to produce generic definition modules and the specific refinements with actual parameters as shown in the next section *before* writing the generic implementation parts (with their formal parameters) and then refining them with actual parameters. The implementation part of a generic separate module is also like any other implementation module, except that it begins with the keyword **GENERIC** and it has a formal parameter list. The syntax is shown in figure 16.2.

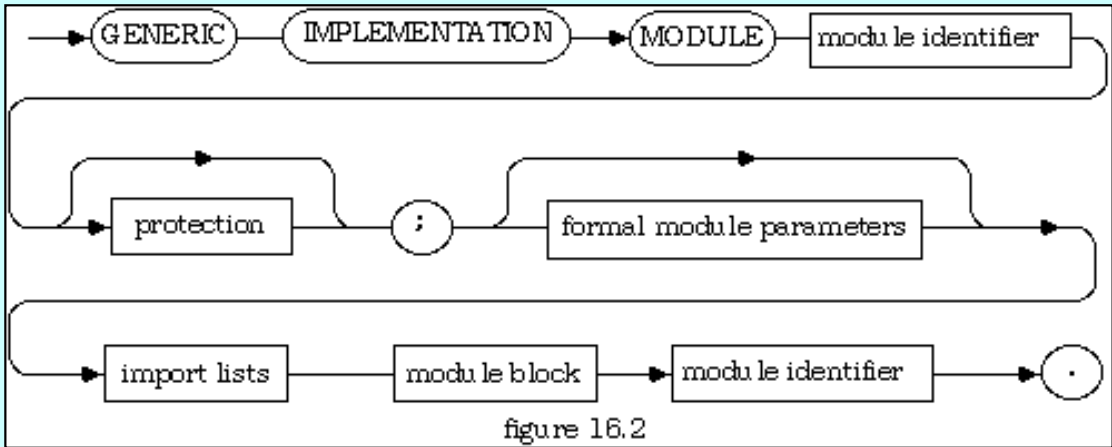


figure 16.2

In order to ensure that the generic definition and generic implementation parts have the same contract with the user that is implied by having parameters in the first place, generic Modula-2 adopts the following rule:

The parameters of the generic implementation module shall be identical to the parameters of the corresponding generic definition module.

This rule is similar to the rule that says that procedures defined as forward must have the same parameters when they are eventually declared and is also similar to the relationship between procedure definitions in a definition module and their declarations in the corresponding implementation module.

In each case, observe how the code sketches (for that is all they are) are done in terms of the formal (generic) parameters. Shown below are the implementation parts that correspond to the definition parts of the generic modules in the last subsection.

Example 1: What follows is a possible generic implementation of the [first example](#) in section 16.2.1

```
GENERIC IMPLEMENTATION MODULE Stacks (Element : TYPE);

VAR
    stack      : ARRAY [0..StackSize] OF Element;
    stackPtr   : CARDINAL;
(* One could also arrange for StackSize to be a parameter. *)

PROCEDURE Push (item : Element);
BEGIN
    INC (stackPtr);
    stack[stackPtr] := item;
END Push;

PROCEDURE Pop (VAR item : Element);
BEGIN
    item := stack[stackPtr];
    DEC (stackPtr)
END Pop;

PROCEDURE Empty () : BOOLEAN;
BEGIN
    RETURN stackPtr = 0
END Empty;

BEGIN (* module body initialization *)
    stackPtr := 0
END Stacks.
```

NOTE: The constant *StackSize* is from the corresponding definition module. It may be used in this generic implementation module. However, the correctness of such use might not be checked until the refinement of that implementation is performed.

Example 2: The next module is a corresponding implementation for the generic definition of [example 2](#) in section 16.2.1.

```
GENERIC IMPLEMENTATION MODULE Matrix (Rows, Cols : CARDINAL; MatrixElement : TYPE);

(* Note that, as in any implementation module, the generic implementation module has
access to the types defined in the corresponding definition. *)

PROCEDURE Invert (VAR m : TMatrix);
BEGIN
    (* your favourite technique *)
END Invert;

END Matrix.
```

Example 3: The next module is a corresponding implementation for the generic definition of [example 3](#) in section 16.2.1.

```

GENERIC IMPLEMENTATION MODULE Validate (PType : TYPE; PValidProc : ValidProcType);

PROCEDURE Valid (item : PType) : BOOLEAN;
BEGIN
    RETURN PValidProc (item)
END Valid;

END Validate.

```

Example 4: Here is a sketch of a generic implementation of the Sorts module defined in [example 4](#) of section 16.2.1.

```

GENERIC IMPLEMENTATION MODULE Sorts (Item : TYPE; GenCompare : CompareProc);

FROM Comparisons IMPORT
    CompareResults;

PROCEDURE Swap (VAR a, b : Item);
VAR
    temp : Item;

BEGIN
    temp := a;
    a := b;
    b := temp;
END Swap;

PROCEDURE Quick (VAR data : ARRAY OF Item);
BEGIN
    (* typical quicksort algorithm, except that compares are done by a procedure Compare
    which returns values of type Comparisons.CompareResults. Swaps are done using the
    refinement of the generic swap above. *)

END Quick;

END Sorts.

```

Example 5: (Parameters are optional) This example implements the [non-parameterized counter](#) found in section 16.2.1.

```

GENERIC IMPLEMENTATION MODULE Counter;
VAR
    CurrentCount : CARDINAL;

PROCEDURE Inc;
BEGIN
    INC (CurrentCount);
END Inc;

PROCEDURE Reset;
BEGIN
    CurrentCount := 0;
END Reset;

PROCEDURE Count ( ) : CARDINAL;
BEGIN

```

```
    RETURN CurrentCount
END Count;

BEGIN (* main *)
    Reset;
END Counter.
```

16.2.3 Formal Module Parameters

As an optional part of the definition and implementation modules of a generic separate module, a formal module parameter provides an explicit interface between the body of the generic separate module and the body of any module refined from it. If the generic separate module has formal parameters (the usual situation), any refiner of it must provide corresponding actual parameters as shown in [section 16.3](#). As indicated above, the code in the implementation part of the module works with the formal names, in the same manner as the code of a procedure.

There are two kinds of formal module parameters: constant value parameters and type parameters. The types of the former are specified by their formal type names; the latter are types, and this is specified by the keyword `TYPE`.

Constant value parameters provide a means of passing a constant to the refinement of a generic separate module. Constant value parameters are value parameters in almost the same sense as the value parameters used with procedures.

The type of a formal constant value parameter may be:

- any pervasive type,
- any type previously named as a formal module type parameter in the same module parameter list,
- any type imported into the generic separate module in its import list, or
- any type defined in the generic separate module itself.

In the latter two cases, the type must be imported into or defined in (respectively) the generic definition module because the parameter lists of the definition and implementation modules are required to be the same.

Type parameters, on the other hand, provide a means of passing a type identifier to the refinement of a generic separate module.

NOTE: The new use of the word *TYPE* as a name is confined to the specific context of generic module parameters and does not extend to procedure parameter lists or any other context.

[Contents](#)

16.3 Refining Separate Library Modules

Having seen how to define generic separate modules, it is time to turn attention to the production of specific instances of these library modules by providing the actual parameters that will determine a specific use. This is done by employing another new kind of module called a refining module.

A refining separate module consists of a refining definition module and a refining implementation module. The refining definition module provides to a generic definition module that it names the actual parameters to substitute for its formal parameters so that a definition module may be refined. The refining implementation module provides to a generic implementation module that it names the actual parameters to substitute for its formal parameters so that an implementation module may be refined.

The syntax of a refining separate module is very simple. It consists of the word *DEFINITION* or *IMPLEMENTATION* (as the case may be) followed by the word *MODULE* followed by the name to be given to the resulting (refined) module, then an equal sign, and then the formal parameter list. To be consistent with the rest of Modula-2, it concludes with an *END* and a repetition of the name of the module.

16.3.1 Refining Definition Modules

Generic definition modules are refined separately from their implementation parts because the designer may wish to test the interface in a client program before writing any of the code, even in generic fashion. A refining definition module is similar to a definition module in the sense of the base language, and the rules for definition modules given in the base standard apply, with two additions or changes.

First, the imports of a refining definition module are the imports of the definition module of the generic separate module of which it is a refiner together with the results of evaluating the actual module parameters (i.e. these values may be treated as imports.)

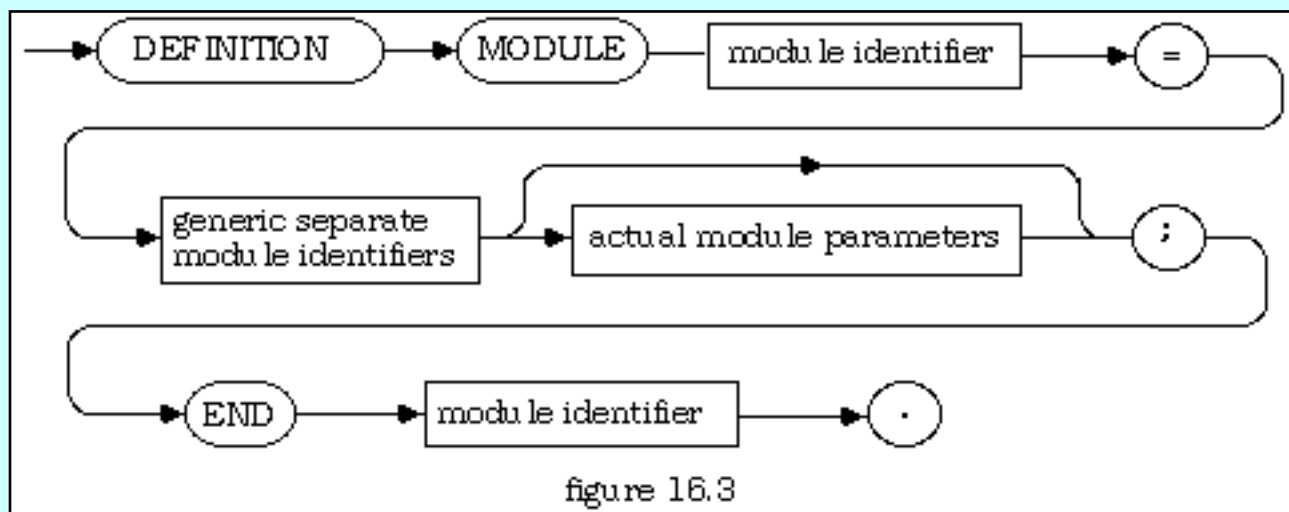
Second, the exports of a refining definition module are the refinements of the items defined in the definition module of the generic separate module of which it is a refiner.

A refining definition module cannot have additional imports or declarations of its own in addition to the ones in the generic separate module from which it refines.

The effect of compiling a refining definition module is the same as constructing and then compiling a definition module (in the sense of the base language) obtained from the definition module of the generic separate module that it refines from but with the results of evaluating the actual parameters of the refining definition module substituted for the formal parameters of the definition module of the generic separate module that is being refined. The resulting refinement that is translated is a definition module in the sense of the base language, and the translation that is done following refinement employs only the

rules of the base language.

Because this refinement is performed by supplying actual parameters, the parameters of the refining definition module must match the parameters of the generic definition module of which it is a refiner. If the definition module of the generic separate module has no parameters, the refining definition module shall also have none, not even an empty parameter list. The syntax of a refining definition module is shown in figure 16.3.



Examples

Example 1: What follows is a possible refiner of the [first example](#) in section 16.2.1

```
DEFINITION MODULE CardStack = Stacks (CARDINAL);
END CardStack.
```

Example 2: In order to refine the module *Sorts* as a separate module, one must first supply the necessary Compare procedure in a separate module. For instance to refine an integer sort, first supply:

```
DEFINITION MODULE IntegerInfo;

FROM Comparisons IMPORT
    CompareResults;

PROCEDURE Compare (a, b : INTEGER) : CompareResults;

END IntegerInfo.
```

When this is done, refining is by a module such as:

```
DEFINITION MODULE IntSorts = Sorts (INTEGER, IntegerInfo.Compare);
END IntSorts.
```

Example 3: Any number of specific instances of the generic counter may be refined, though this may not

be a common use of the facility:
One refiner that illustrates the syntax is:

```
DEFINITION MODULE ACount = Counter;  
END ACount.
```

Example 4: The generic matrix is refined with constants, as in the following:

```
DEFINITION MODULE RealMatrix45 = Matrix (4, 5, REAL);  
END RealMatrix45.
```

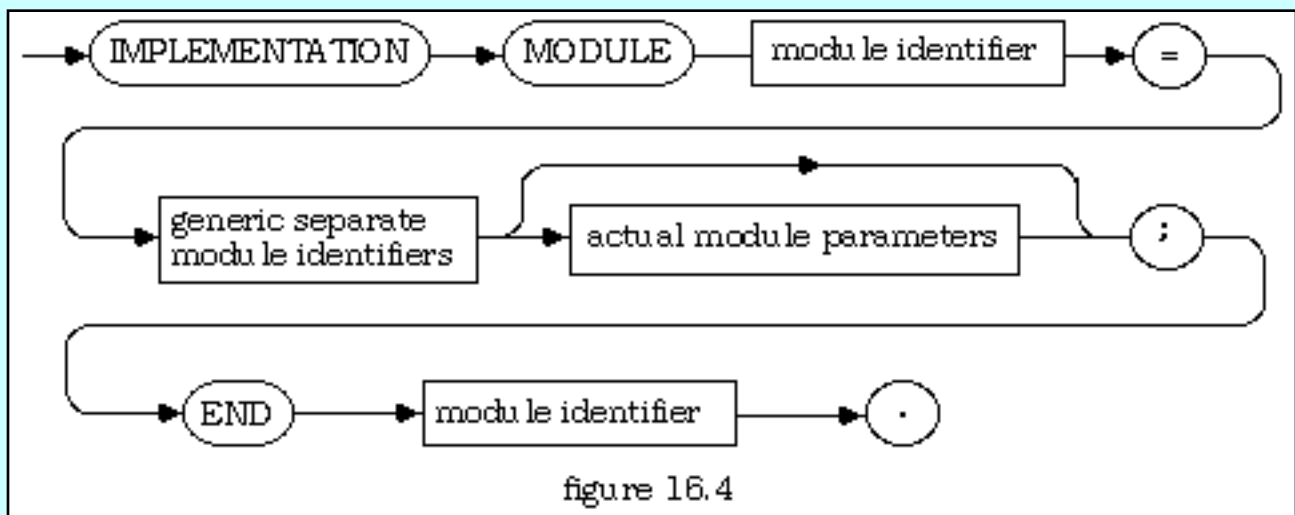
16.3.2 Refining Implementation Modules

At some point before linking a final client program together, the implementation parts of any generic definition modules that have been refined for the project must also be refined from the corresponding generic implementation modules. This is done using exactly the same syntax as for refining definition modules, except of course with the word *IMPLEMENTATION*.

A refining implementation module is similar to an implementation module in the sense of the base standard, and the rules for implementation modules (and their relationships with the corresponding definition modules) given in the base standard apply, with the following addition.

The imports of a refining implementation module are the imports of the implementation module of the generic separate module of which it is a refiner together with the actual parameters of the refining implementation module (i.e. the values may be treated as imports.)

The effect of compiling a refining implementation module is the same as constructing and then translating an implementation module (in the sense of the base standard) obtained from the implementation module of the generic separate module that it refines from, but with the results of evaluating the actual parameters of the refining module substituted for the formal parameters of the generic separate module that is being refined. The resulting refinement that is translated is an implementation module in the sense of the base language, and the translation that is done following refinement employs only the rules of the base language. The syntax of a refining implementation module is shown in figure 16.4.



NOTE: The parameters of the refining implementation module must be the same as the parameters of the corresponding refining definition module.

Example 1: What follows is a possible refining module of the [first example](#) in section 16.2.1

```
IMPLEMENTATION MODULE CardStack = Stacks (CARDINAL);
END CardStack.
```

Example 2: To complete the refiner of the [generic sorting module](#) in 16.2.1, one still needs:

```
IMPLEMENTATION MODULE IntSorts = Sorts (INTEGER, IntegerInfo.Compare);
END IntSorts.
```

A client has:

```
FROM IntSorts IMPORT
    Quick;
```

or, if more than one refinement to separate modules is done:

```
IMPORT IntSorts, RealSorts, RecSorts;
```

Example 3: Here is a refining implementation part of the counter example:

```
IMPLEMENTATION MODULE ACount = Counter;
END ACount.
```

Example 4: To illustrate the necessity of the parameters for both parts of the module being the same, here is the refining implementation for the corresponding refining definition module of [example 4](#) in section 16.3.1

```
IMPLEMENTATION MODULE RealMatrix45 = Matrix (4, 5, REAL);  
END RealMatrix45.
```

16.3.3 Multiple Refinements

Any number of separate refinements of a generic separate module can be performed under a variety of names. Thus the generic module *Matrix* could be refined to hold reals, cardinals, integers, or any other numeric data type, and the module *ACounter* could be refined several times, each with a different name to produce different counters. However, two precautions have to be observed:

1. Different refinements should be done by refiners with different names, so that if two are imported into a single client there will not be a name clash.
2. In a module like *Matrix* there will be many operations on the matrices that use arithmetic operations on the individual elements. For instance, two matrices of the same size are added by adding the elements on corresponding positions. Such operations require that the elements actually have addition defined on them. Thus, while the compiler might allow

```
DEFINITION MODULE BoolMatrix45 = Matrix (4, 5, BOOLEAN);  
END BoolMatrix45.
```

because the parameters are correct, the attempt to use the refiner

```
IMPLEMENTATION MODULE BoolMatrix45 = Matrix (4, 5, BOOLEAN);  
END BoolMatrix45.
```

would fail, not because of the parameters *per se* but because when it then attempted to compile the result after making the parameter substitutions, there would be type incompatibility errors in the arithmetic operations. Similar problems could arise in other contexts as well, so the programmer must ensure that refinements are appropriate to the data types used and the operations employed on them. That is, not all refinements of a given module are logically correct, even if at the parameter substitution level, they may appear to be syntactically correct.

These comments highlight the need to plan ahead of time to take such situations into account. That is why the module *Sorts* was planned from the start to take a *Compare* procedure as a parameter. Had this module simply relied on the *less than* operator for comparisons, only numeric data could have been sorted by it.

16.3.4 Actual Parameters

If the generic separate module has formal parameters, any refiner of it must provide corresponding actual parameters. These are evaluated, and the resulting arguments are accessed in the refined module through the identifiers of the formal parameters. As when using procedure parameters, it is important to ensure that actual module parameters are compatible with the formal ones in the generic modules from which refining is being done.

First, if the formal parameters are constant value parameters, the corresponding actual parameters must be constant expressions of a type compatible to the formal parameter type.

NOTE: This means that actual parameters may not be variables. This differs from the corresponding situation with procedure value parameters, which are allowed to be variables. The reason for this restriction is that the compiler must be able to evaluate the actual parameters and construct the refined module from them, and a compiler cannot evaluate variables because they do not get their values until run time.

Second, for a formal TYPE parameter the identifier of a visible type shall be provided. Any type that is visible at the point of the refining module is potentially usable as an actual parameter, or as the type of an actual parameter. This includes pervasive types, user defined types, or types qualified by a module name.

NOTE : Since a module parameter list may include items of types named previously in the list, such lists shall be evaluated left to right—a restriction not present in the base language for evaluating parameters of procedures.

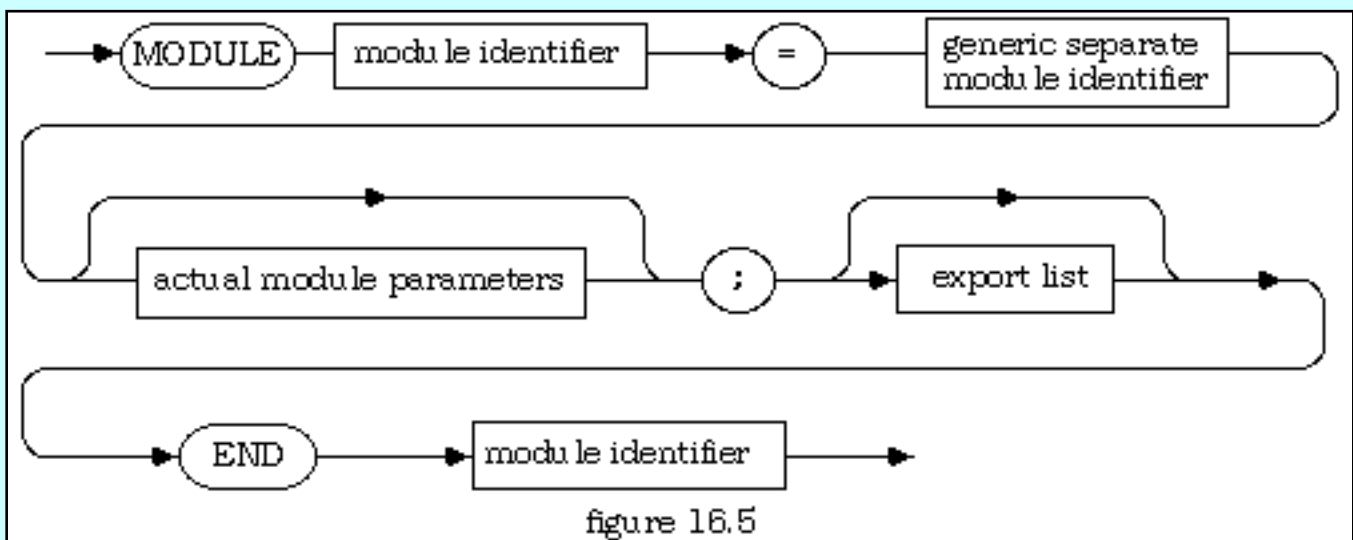
Other than the restrictions on refining module actual parameters (to constant expression parameters and type parameters), and the requirement of left-to-right evaluation, the rules for these parameters (matching, compatibility) are the same as those given in the base standard for actual procedure parameters of these kinds. This means, for instance, that the parameters must be visible at the point in the module where they are being used. One may use a separate library module item by naming it as an identifier qualified by the module name in the manner of *StextIO.WriteLn* because the library module names are all visible to the compiler. The effect is similar to an import.

[Contents](#)

16.4 Refining Within a Program Module

Besides refining a generic separate module as a refined separate module, it is also possible to refine as a local module. A refining local module is similar to a local module in the sense of the base language, and the rules for local modules given in the base standard apply, with the following change.

The syntax of a refining local module is even simpler than that of a refining separate module. It consists of the word *MODULE* followed by the name to be given to the resulting (refined) module, then an equal sign, and then the formal parameter list. To be consistent with the rest of Modula-2, it concludes with an *END* and a repetition of the name of the module. This is shown in figure 16.5.



NOTE : The identifier of the generic separate module named in the refining local module declaration must be visible in the scope of the refining local module. This means that it will have to be named in an import statement in the enclosing module

Refinement of generic separate modules as local modules is similar to refinement as separate modules in that:

- The parameters of the refining local module must match the parameters of the generic separate module of which it is a refiner.
- The export list of a refined local module is the export list specified in the refining local module by which it is refined. The syntax and semantics are those of the base standard.

However, there are some differences. A local module has only one part (not a separate definition and implementation). Thus, the result of refining locally has to be some combination of the results of refining as separate modules. The interpretation of the refinement of a generic separate module as a local module is that:

1. The refinement is the merger of the refinements of the library definition and implementation

module pair of the generic separate module (the name of which must be visible at the place of the refinement.)

2. The list of exports of the refined local module into its surrounding scope is specified in the refining local module. Because these exports may be qualified, two refinements (under different names) of the same generic separate module could both be made in one scope without causing a name clash.
3. Such a merger only makes sense if the parameter lists of the definition and implementation parts of the generic separate module are the same, and this is one of the reasons why the standards committee adopted this rule.
4. If the implementation part of a generic separate module contains a refining local module, that refining local module cannot refine from the same generic separate module in which it is contained whether directly or indirectly. That is, recursive refinement is not permitted.
5. The translator shall be able to detect a new syntax error, for it is not possible to use (apart from refinement) items from a generic separate module directly in another module. For instance, if one were to import for the purpose of local refinement the module `Sorts`, a use of `Sorts.Quick` is invalid.
6. In such a case, the name `Sorts` does have to be imported into the scope of the refinement, however. This import provides only the name of the module, not the names of any items in it. The latter must be refined to be used.

Example 1: What follows is a sketch of a program client containing two local refinements of the [first example](#) in section 16.2.1

```
MODULE StackClient;  
IMPORT Stacks; (* the generic name has to be imported *)
```

```
TYPE  
  RecDef =  
    RECORD  
      c : CHAR;  
      i : INTEGER  
    END (* record *);
```

```
MODULE CardStack = Stacks (CARDINAL);  
EXPORT QUALIFIED StackSize, Push, Pop, Empty;  
END CardStack;
```

```
MODULE RecStack = Stacks (RecDef);  
EXPORT StackSize, Push, Pop, Empty;  
END RecStack;
```

```
VAR  
  c : CARDINAL;
```

```

    r : RecDef;

BEGIN (* main *)
    CardStack.Push (c);
    Push (r);
END StackClient.

```

On the one hand, the refinement of the local modules of this module has to export according to the export list in the refiner, and on the other hand it must consist of the merger of the corresponding definition and implementation parts of the generic separate modules. This means, for instance, that the refiner can only export items that are in the definition module of the generic separate module from which it is refining. However, it can export those items either qualified or unqualified, and it need not export them all.

Example 2: The generic matrices defined in [example 2](#) of section 16.2.1 may be refined locally using literal or constant data established in the main module.

```

MODULE Client;

IMPORT Matrix;

    MODULE Mat10x20 = Matrix (10, 20, CARDINAL);
    EXPORT QUALIFIED TMatrix, Invert;
    END Mat10x20;

CONST
    row = 4;
    col = 6;

    MODULE MatRowXCol = Matrix (row, col, CARDINAL);
    EXPORT QUALIFIED TMatrix, Invert;
    END MatRowXCol;

VAR
    m : Mat10x20.TMatrix;
    n : MatRowXCol.TMatrix;

BEGIN
    (*
    .
    .*)
    Mat10x20.Invert (m);
    MatRowXCol.Invert (n);
END Client.

```

Example 3: Two independent local refinements of the module counter in [example 5](#) of section 16.2.1

may be performed, in effect creating two ADT *counters* both of which are hidden from the program and modifiable only through the refined procedures.

```
MODULE NeedsACounter;  
  
IMPORT Counter;  
  
VAR  
    countingCondition1, countingCondition2 : BOOLEAN;  
  
MODULE Duke = Counter;  
EXPORT QUALIFIED Inc, Reset, Count;  
END Duke;  
  
MODULE Baron = Counter;  
EXPORT QUALIFIED Inc, Reset, Count;  
END Baron;  
  
BEGIN (* main program module *)  
  
IF countingCondition1  
    THEN  
        Duke.Inc;  
    ELSIF countingCondition2 THEN  
        Baron.Inc;  
    END;  
  
END NeedsACounter.
```

[Contents](#)

16.5 Refining Within an Implementation Module

A refining local module may be employed in any context in which the refined local module is permitted by the base language, including a program module (as in [section 16.4](#)), an implementation module of a separate library module, or a local module contained within either.

Example 1: Generic modules to implement structures and those to implement data manipulation can be defined as in the examples in [16.2.1](#) and then combined in a new generic definition module using local modules. First create the generic definition module.

```
GENERIC DEFINITION MODULE ValidStacks (PType : TYPE; PValidProc : ValidProcType);  
  
TYPE  
  ValidProcType = PROCEDURE (item : PType) : BOOLEAN;  
  
PROCEDURE PushValid (item : PType);  
  
END ValidStacks.
```

Then, combine the two by refining both within the implementation via local modules and then employing services from both so that, in this case, only valid items are pushed on the stack. In addition, note that the data items are to be defined in another module and are assumed to have ADT components available for use as follows:

- (i) *PType* that can be mapped both to the *Element* required by *Stacks* and to the *Item* required by *Validate* and
- (ii) *PValidProc* compatible with the *Validation* required by *Validate*.

```
GENERIC IMPLEMENTATION MODULE ValidStacks (PType : TYPE; PValidProc : ValidProcType);  
IMPORT Stacks, Validate;  
  
MODULE MyStacks = Stacks (PType);  
EXPORT QUALIFIED StackSize, Push, Pop, Empty;  
END MyStacks;  
  
MODULE MyValidate = Validate (PType, PValidProc);  
EXPORT QUALIFIED Valid;  
END MyValidate;  
  
PROCEDURE PushValid (item : PType);  
BEGIN  
  IF MyValidate.Valid (item) THEN  
    MyStacks.Push (item)  
  END (* if *)  
END PushValid;  
  
END ValidStacks.
```

These may then be refined further for a specific data type and validation procedure.

16.6 Making New ADTs from Old With Generics

The technique shown in the previous section is worth expanding upon in some additional detail as the most likely use of Generic Modula-2 will be to refine Abstract Data Types as separate library modules that are then used as many times as desired in an application. Along the way, new facilities could be added to a generic separate module to form a new generic separate module that could in turn be refined with a specific data type.

Suppose one begins with a classical data structure such as a list, parameterized for the type to list. A framework could look like:

```
GENERIC DEFINITION MODULE Lists (DataType : TYPE);  
TYPE  
  List;  
PROCEDURE Create (VAR l : List);  
PROCEDURE Destroy (VAR l : List);  
PROCEDURE Add (l : List; item : DataType);  
PROCEDURE Delete (l : List; item : DataType);  
PROCEDURE Find (l : List; item : DataType);  
END Lists.  
  
GENERIC IMPLEMENTATION MODULE Lists (DataType : TYPE);  
TYPE  
  NodePointer = POINTER TO Node;  
  Node =  
    RECORD  
      theItem : DataType;  
      next : NodePointer;  
    END;  
  List = POINTER TO ListRecord;  
  ListRecord =  
    RECORD  
      Head : NodePointer;  
      numberActive : CARDINAL;  
    END;  
  (* all stubbs, testing concept only *)  
PROCEDURE Create (VAR l : List);  
END Create;  
PROCEDURE Destroy (VAR l : List);  
END Destroy;  
PROCEDURE Add (l : List; item : DataType);  
END Add;  
PROCEDURE Delete (l : List; item : DataType);  
END Delete;  
PROCEDURE Find (l : List; item : DataType);  
END Find;  
END Lists.
```

This generic separate module may be refined as it is to produce new separate or local modules implementing lists for a

specific data type, and as many of these lists as desired may then be employed. Alternately, one could refine in a generic module to produce, say, generic sorted lists. One first creates a new generic definition module and then refines the generic separate module above in the implementation, with appropriate exports per the definition:

```
GENERIC DEFINITION MODULE ListsSorted (DataType : TYPE; Compare : CompareProc );  
  
FROM Comparisons IMPORT  
    CompareResults;  
  
TYPE  
    List;  
    CompareProc = PROCEDURE (DataType, DataType) : CompareResults;  
  
PROCEDURE Create (VAR l : List);  
PROCEDURE Destroy (VAR l : List);  
PROCEDURE Delete (l : List; item : DataType);  
PROCEDURE Find (l : List; item : DataType);  
PROCEDURE Insert (theList : List; theItem : DataType);  
END ListsSorted.  
  
GENERIC IMPLEMENTATION MODULE ListsSorted (DataType : TYPE; Compare : CompareProc);  
  
MODULE SLists = Lists (DataType);  
EXPORT List, Create, Destroy, Delete, Find;  
END SLists;  
  
PROCEDURE Insert (theList : List; theItem : DataType);  
END Insert;  
  
END ListsSorted.
```

This new generic separate module may now be refined to produce a separate module for lists of a particular data type; for instance:

```
DEFINITION MODULE IntListsSorted = ListsSorted (INTEGER, IntegerInfo.Compare);  
END IntListsSorted.
```

A program module may be written that imports and uses this separate module, creating as many of the sorted lists of integers as desired.

Here is a second example of a similar kind--this one employing successive refinements of a generic *Queue* data type. One begins, as always, with the definition and implementation parts of the generic separate module.

```
GENERIC DEFINITION MODULE Queues (itemType : TYPE; maxSize : CARDINAL);  
  
TYPE  
    Queue;    (* details in implementation *)  
    ActionProc = PROCEDURE (itemType);  
  
PROCEDURE Init (VAR q : Queue);  
PROCEDURE Destroy (VAR q : Queue);  
PROCEDURE Full (q : Queue) : BOOLEAN;  
PROCEDURE Empty (q : Queue) : BOOLEAN;  
PROCEDURE Enqueue (q : Queue; item : itemType);
```

```

PROCEDURE Serve (q : Queue; VAR item : itemType);
PROCEDURE Traverse (q : Queue; Proc : ActionProc);

END Queues.

GENERIC IMPLEMENTATION MODULE Queues (itemType : TYPE; maxSize : CARDINAL);

TYPE
    Queue      (* details here *);

(* provide code *)
END Queues.

```

In the examples of local refinement taken thus far, no new facilities were added. However, this may also be done. Perhaps the most interesting way is to define new separate modules (whether Generic or not) that use some or all of the functionality developed in existing generic separate modules but with some changes or additions.

Suppose, for instance, one wished to develop a new separate module to define and implement priority queues, using as a partial base the generic separate module *Queues* illustrated above. The definition might be altered as follows:

```

GENERIC DEFINITION MODULE PriorityQueues
    (itemType : TYPE; maxSize : CARDINAL, Compare : CompareProc);

TYPE
    PQueue;      (* details in implementation *)
    ActionProc = PROCEDURE (itemType);
    CompareResults = (less, equal, greater);
    CompareProc = PROCEDURE (itemType, itemType) : CompareResults;

PROCEDURE Init (VAR q : PQueue);
PROCEDURE Destroy (VAR q : PQueue);
PROCEDURE Full (q : PQueue) : BOOLEAN;
PROCEDURE Empty (q : PQueue) : BOOLEAN;
PROCEDURE Enqueue (q : PQueue; item : itemType);
PROCEDURE Serve (q : PQueue; VAR item : itemType);
PROCEDURE Traverse (q : PQueue; Proc : ActionProc);

END PriorityQueues.

```

In this version, refinement requires a procedure parameter that provides the functionality of determining the relative priority of two items of the type to be enqueued (presumably by examining some field of the data), so that a new item can be placed in the proper place in the queue.

When the implementation part is written, most of the procedures in the original generic implementation part can be retained by exporting them in the refinement. The *Enqueue* procedure will, however, have to be changed. Portions of the implementation would look like:

```

GENERIC IMPLEMENTATION MODULE PriorityQueues
    (itemType : TYPE; maxSize : CARDINAL; Compare : CompareProc);

IMPORT Queues; (* this done for refinement purposes *)
TYPE
    PQueue = Queue; (* type exported from local refiner *)
MODULE LocalQueues = Queues (itemType, maxSize); (* pass along parameters. *)

```

```
EXPORT Queue, Init, Destroy, Full, Empty, Serve, Traverse;  
END LocalQueues;
```

```
PROCEDURE Enqueue (q : PQueue; item : itemType);  
(* code to place a data item in position by doing comparisons as needed *)  
END Enqueue;  
END PriorityQueues.
```

NOTES: 1. Since a local refinement is performed on the implementation part of the generic separate module, the details of the data type PQueue itself are available in the new module for its use and therefore it is possible to replace the procedure *Enqueue* in this way. This should not be thought of as a relaxation of the rules for opaque types, but rather as the inclusion of what can be regarded as a copy of the (refined) code (that is, with the appropriate parameter substitutions made) in the implementation module.

2. For clients of refinements of this new module, the type PQueue is of course exported opaquely by the refinements of the definition part of the module.

3. Because the refined implementation can export into the scope of the refinement only those items named in the generic separate definition module, any other items in the implementation can not be made available in the scope of the refinement for use there and remain strictly private.

Should it be also be thought desirable to refine a separate module specified as to the data type and size, it would be a simple matter to define the appropriate type in a separate module, import from this in the new definition part, and then refine fully in the implementation part, thus:

```
DEFINITION MODULE StudentPriorityQueues;
```

```
FROM Students IMPORT  
    Student;
```

```
TYPE  
    PQueue;    (* details in implementation *)  
    ActionProc = PROCEDURE (CHAR);
```

```
PROCEDURE Init (VAR q : PQueue);  
PROCEDURE Destroy (VAR q : PQueue);  
PROCEDURE Full (q : PQueue) : BOOLEAN;  
PROCEDURE Empty (q : PQueue) : BOOLEAN;  
PROCEDURE Enqueue (q : PQueue; item : Student);  
PROCEDURE Serve (q : PQueue; VAR item : Student);  
PROCEDURE Traverse (q : PQueue; Proc : ActionProc);  
  
END StudentPriorityQueues.
```

```
IMPLEMENTATION MODULE StudentPriorityQueues;
```

```
IMPORT Queues;  
FROM Students IMPORT  
    Student, CompareStudent;
```

```
TYPE  
    PQueue = Queues.Queue;  
MODULE LocalQueues = Queues (Student, maxSize );  
EXPORT Queue, Init, Destroy, Full, Empty, Serve, Traverse;  
END LocalQueues;
```

```
PROCEDURE Enqueue (q : PQueue; item : Student);  
(* code to place a data item in position by doing comparisons  
   as needed *)  
END Enqueue;  
END StudentPriorityQueues.
```

In this version, the functionality associated with the data type *Student* and that associated with the generic separate module *Queues* is combined (and extended as above) into a single refined module with a standard client interface.

NOTE: The new definition part of the refined module must be produced in full in such cases, because the base language does not permit a definition module to contain a local module, and hence it cannot itself contain a refining local module acting on the definition part of the original generic separate module. This is one reason why the refinement of generic modules as local modules is defined in standard Generic Modula-2 to be the refinement of the implementation part of the generic separate module.

The partial refinement of a generic separate module allows the programmer to add facilities to or restrict facilities from an existing generic separate module when producing either a new separate module (generic or not) or a refinement in a program module.

[Contents](#)

16.7 Extended ADT Examples

This section contains more complete examples of abstract data types realized in Generic Modula-2. The design work for these modules was done in VDM-SL (Vienna Documentation Method-Specification Language) by Matt Suderman for an undergraduate thesis submitted in partial fulfillment of a B.Sc. degree at Trinity Western University in the Spring of 1997. Only a general discussion and the end result in Generic Modula-2 is give here.

16.7.1 Generic Lists

For this example, the ADT *List* has been chosen. The generic aspects are the type of the data and a procedure to do assignments. In order to make a close fit with the design notation, a type has been introduced to model the *Natural Numbers*, that is, the positive whole numbers (not including zero). This is used to number the list. In addition, take note of the systematic use of preconditions and the provision of procedures to check those preconditions. As usual, various procedures are expressed in terms of the formal parameter names whose values are yet to be supplied.

```
GENERIC DEFINITION MODULE Lists (Data: TYPE; AssignData: AssignProcType);
```

```
TYPE
```

```
  AssignProcType = PROCEDURE (VAR Data: Data);  
  (* ``Nat'' means ``Natural Number'' as in VDM-SL *)  
  Nat1 = [1 .. MAX (CARDINAL)];  
  Nat = CARDINAL;  
  Traversal = PROCEDURE (Data): Data;  
  List;
```

```
PROCEDURE InitList (VAR list: List);  
(* Post: Empty (list) = TRUE *)
```

```
PROCEDURE CanInsert (list: List; position: Nat1): BOOLEAN;  
(* Pre: 'memory available' AND position <= Length (list) + 1 *)
```

```
PROCEDURE Insert (VAR list: List; data: Data; position: Nat1);  
(* Post: GetLength (list') + 1 = GetLength (list)  
  AND GetElement (list', 1..position - 1)  
    = GetElement (list, 1..position - 1)  
  AND GetElement (list', position..GetLength (list'))  
    = GetElement (list, position + 1..GetLength (list))  
  AND GetElement (list, position) = data *)
```

```
PROCEDURE CanUpdate (list: List; position: Nat1): BOOLEAN;  
(* Pre: position <= Length (list) *)
```

```
PROCEDURE Update (VAR list: List; data: Data; position: Nat1);  
(* Post: GetLength (list') = GetLength (list)  
  AND p <> position  
  ==> GetElement (list', p) = GetElement (list, p)  
  AND GetElement (list, position) = data *)
```

```
PROCEDURE CanAppend (list: List; data: Data): BOOLEAN;  
(* Pre: 'memory available' AND NOT Empty (list) *)
```



```

PROCEDURE Append (VAR list: List; data: Data);
(* Post:  GetLength (list') + 1 = GetLength (list)
    AND GetElement (GetLength (list)) = data
    AND GetElement (list', 1..GetLength (list'))
    = GetLength (list, 1..GetLength (list')) *)

PROCEDURE CanDelete (list: List; position: Nat1): BOOLEAN;
(* Pre: position <= GetLength (list) *)
PROCEDURE Delete (VAR list: List; position: Nat1);
(* Post: GetLength (list') = GetLength (list) + 1
    AND GetElement (list', 1..position - 1)
    = GetElement (list, 1..position - 1)
    AND GetElement (list', position + 1..)
    = GetElement (list, position ..) *)

PROCEDURE Traverse (VAR list: List; traversal: Traversal);
(* Post: for all p GetElement (list, p)
    = traversal (GetElement (list', p)) *)

PROCEDURE GetLength (list: List): Nat;
(* Post: the number of elements in the list *)

PROCEDURE Empty (list: List): BOOLEAN;
(* Post: GetLength (list) = 0  Empty (list) = TRUE *)

PROCEDURE CanGetElement (list: List; position: Nat1): BOOLEAN;
(* Pre: position <= GetLength (list) *)
PROCEDURE GetElement (list: List; position: Nat1): Data;
(* Post: GetElement (list, position) = data at position in the list *)

END Lists.

```

The implementation part of this generic separate module supplies the code to make everything happen, again expressed in terms of the formal parameters. The formal design of this module included various preconditions, and implementation of these require that one be able to determine ahead of time whether storage space is available. To meet this requirement, a local module is employed that buffers one storage location. Whenever this is used, another one is requested. The function *StorageAvailable* can examine the buffered location and determine if a valid one is available, and calls to *MYNEW* return the location available and attempt to replenish the buffer for the next call. Matt has formulated his list design as an inherently recursive structure. A list points to another list (sometimes called its *tail*) recursively until there are no more data items.

```

GENERIC IMPLEMENTATION MODULE Lists (Data: TYPE; AssignData: AssignProcType);

FROM Storage IMPORT ALLOCATE, DEALLOCATE;

TYPE
  NodePtr = POINTER TO Node;
  Node = RECORD
    data: Data;
    next: NodePtr;
  END;
  List = NodePtr;

MODULE MyStorage;

```

```
(* Module ``MyStorage'' is necessary to implement the function
``StorageAvailable''. ``StorageAvailable'' returns true if
``MYNEW'' will return a new non-NIL pointer to a node; otherwise,
if ``MYNEW'' will return a NIL pointer ``StorageAvailable'' returns
false. In other words, ``StorageAvailable'' returns true if another
``Node'' can be allocated in the system heap and false otherwise. *)
```

```
IMPORT ALLOCATE, DEALLOCATE, Node, NodePtr;
```

```
EXPORT MYNEW, StorageAvailable;
```

```
VAR temp: NodePtr;
```

```
PROCEDURE StorageAvailable (): BOOLEAN;
```

```
BEGIN
```

```
    RETURN temp <> NIL;
```

```
END StorageAvailable;
```

```
PROCEDURE MYNEW (init: Node): NodePtr;
```

```
(* return the buffered value, and if it was not NIL, get a new one *)
```

```
VAR
```

```
    new: NodePtr;
```

```
BEGIN
```

```
    IF temp = NIL
```

```
        THEN
```

```
            RETURN NIL
```

```
    ELSE
```

```
        new := temp;
```

```
        new^ := init;
```

```
        NEW (temp);
```

```
        RETURN new
```

```
    END
```

```
END MYNEW;
```

```
BEGIN (* initialize one location in the buffer called temp *)
```

```
    NEW (temp);
```

```
END MyStorage; (* end of local module *)
```

```
(* resume declarations in main module *)
```

```
PROCEDURE InitList (VAR list: List);
```

```
BEGIN
```

```
    list := NIL;
```

```
END InitList;
```

```
PROCEDURE MkNode (data: Data; next: NodePtr): Node;
```

```
VAR
```

```
    node: Node;
```

```
BEGIN
```

```
    AssignData (node.data, data);
```

```
    node.next := next;
```

```
    RETURN node;
```

```
END MkNode;
```

```
PROCEDURE IsEmpty (list: List): BOOLEAN;
```

```
BEGIN
```

```
    RETURN list = NIL
```

```
END IsEmpty;
```

```
PROCEDURE Length (list: List): Nat;
```

```
(* observe his formulation of length as a recursive procedure. This may take longer  
to calculate, but he does not have to store and update a value *)
```

```
BEGIN
```

```
    IF NOT IsEmpty (list) THEN
```

```
        RETURN 1 + Length (list^.next);
```

```
    ELSE
```

```
        RETURN 0;
```

```
    END;
```

```
END Length;
```

```
PROCEDURE PtrToNode (list: List; position: Nat1): NodePtr;
```

```
(* work along the list until the correct position is found, then return a pointer to  
the rest of the list *)
```

```
BEGIN
```

```
    IF position > 1
```

```
        THEN
```

```
            RETURN PtrToNode (list^.next, position - 1);
```

```
        ELSE
```

```
            RETURN list;
```

```
        END;
```

```
END PtrToNode;
```

```
(* The next three procedures implement the typical inserting possibilities. *)
```

```
PROCEDURE InsertAtBeginning (VAR list: List; data: Data);
```

```
VAR
```

```
    new: NodePtr;
```

```
BEGIN
```

```
    new := MYNEW (MkNode (data, list));
```

```
    list := new;
```

```
END InsertAtBeginning;
```

```
PROCEDURE InsertAfter (ptr: NodePtr; data: Data);
```

```
VAR
```

```
    new: NodePtr;
```

```
BEGIN
```

```
    new := MYNEW (MkNode (data, ptr^.next));
```

```
    ptr^.next := new;
```

```
END InsertAfter;
```

```
(* This one is the exported procedure and must select one of the above. *)
```

```
PROCEDURE Insert (VAR list: List; data: Data; position: Nat1);
```

```
BEGIN
```

```
    IF position = 1
```

```
        THEN
```

```

        InsertAtBeginning (list, data);
    ELSE
        InsertAfter (PtrToNode (list, position - 1), data);
    END;
END Insert;

PROCEDURE CanInsert (list: List; position: Nat1): BOOLEAN;
BEGIN
    RETURN (position <= Length (list) + 1) AND StorageAvailable ();
END CanInsert;

PROCEDURE Append (VAR list: List; data: Data);
VAR
    ptr: NodePtr;
BEGIN
    ptr := list;
    IF ptr = NIL
    THEN
        InsertAtBeginning (list, data);
    ELSE
        WHILE ptr^.next <> NIL
        DO
            ptr := ptr^.next;
        END;
        InsertAfter (ptr, data);
    END;
END Append;

PROCEDURE CanAppend (): BOOLEAN;
BEGIN
    RETURN StorageAvailable ();
END CanAppend;

PROCEDURE Update (VAR list: List; data: Data; position: Nat1);
VAR
    ptr: NodePtr;
BEGIN
    ptr := PtrToNode (list, position);
    AssignData (ptr^.data, data);
END Update;

PROCEDURE CanUpdate (list: List; position: Nat1): BOOLEAN;
BEGIN
    RETURN (position <= Length (list)) AND StorageAvailable ();
END CanUpdate;

(* The next three procedures implement the typical deleting possibilities. *)

PROCEDURE DeleteAtBeginning (VAR list: List);
VAR
    temp: NodePtr;
BEGIN
    temp := list;

```

```

    list := list^.next;
    DISPOSE (temp);
END DeleteAtBeginning;

PROCEDURE DeleteAfter (ptr: NodePtr);
VAR
    temp: NodePtr;
BEGIN
    temp := ptr^.next;
    ptr^.next := temp^.next;
    DISPOSE (temp);
END DeleteAfter;

(* This one is the exported procedure and must select one of the above. *)

PROCEDURE Delete (VAR list: List; position: Nat1);
BEGIN
    IF position = 1
    THEN
        DeleteAtBeginning (list);
    ELSE
        DeleteAfter (PtrToNode (list, position - 1));
    END;
END Delete;

PROCEDURE CanDelete (list: List; position: Nat1): BOOLEAN;
BEGIN
    RETURN position <= Length (list);
END CanDelete;

PROCEDURE Traverse (VAR list: List; traversal: Traversal);
VAR
    ptr: NodePtr;
BEGIN
    ptr := list;
    WHILE ptr <> NIL
    DO
        AssignData (ptr^.data, traversal (ptr^.data));
        ptr := ptr^.next;
    END;
END Traverse;

PROCEDURE GetLength (list: List): Nat;
BEGIN
    RETURN Length (list);
END GetLength;

PROCEDURE Empty (list: List): BOOLEAN;
BEGIN
    RETURN IsEmpty (list);
END Empty;

PROCEDURE GetElement (list: List; position: Nat1): Data;

```

```

VAR
    ptr: NodePtr;
BEGIN
    ptr := PtrToNode (list, position);
    RETURN ptr^.data;
END GetElement;

PROCEDURE CanGetElement (list: List; position: Nat1): BOOLEAN;
BEGIN
    RETURN position <= Length (list);
END CanGetElement;

END List.

```

16.7.2 Generic Queues

Here, Matt provides a generic definition module for the type *Queue*. Implementation is fairly simple and can be based directly on the List module in the previous subsection.

```

GENERIC DEFINITION MODULE Queues (Data: TYPE; AssignData: List.AssignProcType);

TYPE
    Queue;
    Nat = CARDINAL;

PROCEDURE InitQueue (VAR q: Queue);
(* Post: Empty (q) = TRUE *)

PROCEDURE CanEnqueue (q: Queue; d: Data): BOOLEAN;
(* Pre: 'memory available' *)
PROCEDURE Enqueue (VAR q: Queue; d: Data);
(* Post: the Length (q) time, Dequeue (q) is called, Dequeue (q) = d *)

PROCEDURE CanDequeue (q: Queue): BOOLEAN;
(* Pre: NOT Empty (q) *)
PROCEDURE Dequeue (VAR q: Queue): Data;
(* Post: Dequeue (q') = GetHead (q') *)

PROCEDURE CanGetHead (q: Queue);
(* Pre: NOT Empty (q) *)
PROCEDURE GetHead (q: Queue): Data;
(* Post: GetHead (q) = the oldest enqueued element *)

PROCEDURE Length (q: Queue): Nat;
(* Post: Length (q) = the number of elements enqueued - number dequeued *)

PROCEDURE Empty (q: Queue): BOOLEAN;
(* Post: Length (q) = 0   Empty (q) = TRUE *)

END Queues.

```

16.7.3 Generic Stacks

Finally, he gives a generic definition module for the type *Stack*. Once again, implementation is fairly simple and is left to the student as an exercise.

```
GENERIC DEFINITION MODULE Stacks (Data: TYPE; AssignData: List.AssignProcType);

TYPE
    Stack;
    Nat = CARDINAL;

PROCEDURE InitStack (VAR s: Stack);
(* Post: Empty (s) = TRUE *)

PROCEDURE CanPush (s: Stack; d: Data): BOOLEAN;
(* Pre: 'memory available' *)
PROCEDURE Push (VAR s: Stack; d: Data);
(* Post: Pop (s) = d *)

PROCEDURE CanPop (s:Stack): BOOLEAN;
(* Pre: NOT Empty (s) *)
PROCEDURE Pop (VAR s:Stack): Data;
(* Post: Pop (s') = GetTop (s') *)

PROCEDURE CanGetTop (s: Stack): BOOLEAN;
(* Pre: NOT Empty (s) *)
PROCEDURE GetTop (s: Stack);
(* Post: GetTop (s) = the list element to be pushed *)

PROCEDURE Length (s: Stack): Nat;
(* Post: Length (s) = the number of elements pushed - number popped *)

PROCEDURE Empty (s: Stack): BOOLEAN;
(* Post: Length (s) = 0  Empty (s) *)

END Stacks.
```

[Contents](#)

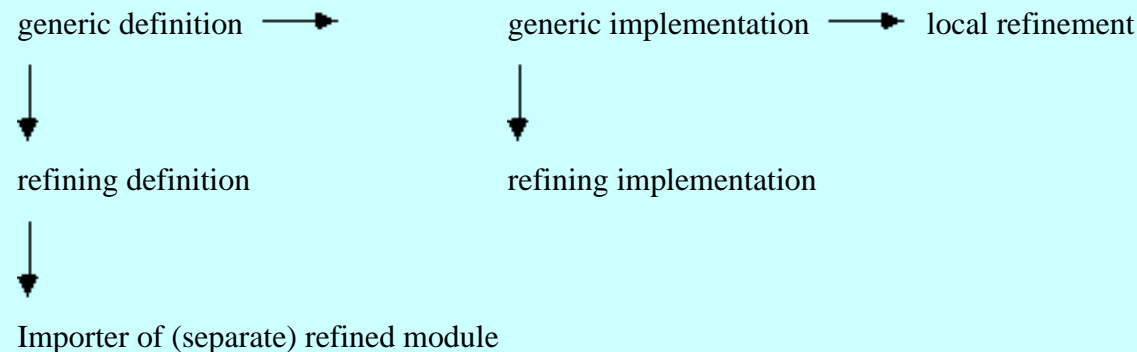
16.8 Dependency and Order in Generic Modula-2

16.8.1 Module Dependencies

The module dependencies of Standard Generic Modula-2 are the same as those in the base language, with the following additions:

- The implementation module of a generic separate module must be translated after the corresponding definition module of that same generic separate module.
- The definition module of a refining separate module must be translated after the definition module of the generic separate module of which it is a refiner.
- The implementation module of a refining separate module must be translated after the corresponding definition module of that same refining separate module.
- The implementation module of a refining separate module must be translated after the implementation module of the generic separate module of which it is a refiner.
- A module containing one or more refining local modules must be translated after all the implementation modules of the generic separate modules for which it contains refining local modules.
- A module importing one or more refining separate modules must be translated after all the definition modules of the refining separate modules that it imports.

Chart of dependencies:



16.8.2 Nested Module Refinement Order

If an implementation module of a generic separate module contains one or more refining local modules, then the order of refinement is the same as the order of initialization would be once the modules were fully refined. This means that the outer module is refined first, then the local modules in the textual order in which they appear, and then any local modules contained in the generic separate modules from which they are being refined, and so on.

This process is not recursive, whether directly or indirectly. If the implementation part of a generic separate module contains refining local module, that refining local module cannot refine from the same generic separate module in which it is contained. Neither may two generic separate modules each contain a refiner to the other. That is generic separate module *A* cannot contain a local module refining from generic separate module *B* which in turn contains a local module refining from *A*.

16.8.3 Module Initialization and Termination Order

The module initialization order in Standard Generic Modula-2 is determined by applying the rules of the base language after the refinements of any generic separate modules have been completed. This means that if a program uses two different refinements of the same generic separate module, these are regarded as different modules in the sense of the base language. Each has its own initialization, and the two initializations are done in an order determined by the import lists of the program.

The module termination order in Standard Generic Modula-2 is determined by applying the rules of the base language after the refinements of any generic separate modules have been completed. This means that if a program uses two different refinements of the same generic separate module, the two modules are terminated in an order determined by their initialization.

16.8.4 Import and Export Lists

The rules for import lists in a module in Standard Generic Modula-2 are identical to those of the base language and are applied after the refinement of any generic separate modules has taken place. In addition, the rules for imports into generic separate modules are the same as the rules for any other separate modules.

NOTE: Because the names of formal parameters are in the scope of a generic separate module, and they are treated as imports, they may not clash with any identifiers named in an import list of that module.

A refined separate module has as its qualified export list the items defined in the generic definition module of which its refining separate module is a refiner. A refined local module, on the other hand, has as its export list the items named in the (possibly qualified) export list of the refining local module. Items not in the generic separate definition module cannot be in this export list.

Thus, if one had:

```
GENERIC DEFINITION MODULE SomeGenericStuff (GenType: TYPE; aConst : CARDINAL);
CONST
    aConst : INTEGER; (* sorry, this is an illegal with the formal parameter name *)
END SomeGenericStuff.
```

then the definition of *aConst* here is a redefinition of an item with the same name in the same scope and so is forbidden. Moreover, if this were completed with an implementation that looked like

```
GENERIC IMPLEMENTATION MODULE SomeGenericStuff (GenType: TYPE; aConst : CARDINAL);
CONST
    bConst : INTEGER; (* can't be exported by a refiner *)
END SomeGenericStuff.
```

then one could not refine the implementation with an export as in the local module contained below:

```
IMPLEMENTATION MODULE DoGen;;

MODULE Local = SomeGenericStuff (CARDINAL, 5);
EXPORT bConst; (* this export is illegal; not in definition *)
END Local;

END DoGen.
```

16.9 Summary and Comparison With Other Notations

The mechanism for producing generic modules that is shown here is of less complexity than similar mechanisms used in Ada C++; it is of similar complexity to the mechanism used in Modula-3. In Ada, Modula-3 and Modula-2 the generic entities coincide with the separate compilation entities. In C++ the generic entities are 'template classes' and 'template functions'. The allowed kinds of generic parameters are more diverse: Ada allows types, constants, subprograms and variables, whereas Modula-2 and C++ only allow types and constants (which includes procedure constants in Modula-2). Modula-3 allows only the names of interfaces to be parameters to generic modules.

Standard Generic Modula-2 is a novel but very slight extension (by a single keyword and some syntax) of ISO Standard Modula-2. For this modest investment one obtains all the functionality of the static generic facilities already present in such languages as Modula-3 and C++. In addition, refinement of generic separate modules as local modules allows the module container to be a mechanism for the composition of one or more existing generic modules into a new module with more or less functionality than the original modules—something not possible, for instance, in Modula-3.

Much of this functionality (and some other things) can also be obtained with object oriented mechanisms in various languages, but static generic modules are simpler, easier to use, and easier to debug than complex object oriented programs. In many cases (indeed in most) such modules are all that are needed to implement abstract data types even for large scale applications. The programmer who also has object oriented facilities available is encouraged to use both facilities, either separately, or in combination. Object oriented Modula-2 extensions are the subject of a separate international standard, and will be discussed in detail in a later chapter of this book.

One should also note that with the mechanisms introduced in Generic Modula-2 for partial, composed, or extended refinement of generic separate modules as local modules, scalability is no longer confined to procedures and objects, for modules become scalable as well. That is, modules can also be easily composed into new modules with different functionality than the ones going into the mix.

This additional functionality does come with a cautionary note, however. It is not a good idea to build complex composed hierarchies, whether of procedures, modules, or objects. When such things are done, greater abstraction may be achieved, but it is too easy to lose track altogether of the details of implementation and thus to make it almost impossible to fix bugs. Moreover, excessive composition can lead to the production of very slow code. Composed hierarchies should therefore be no more than a few levels deep so as to promote code maintainability.

16.10 Chapter Summary

This chapter covered these topics:

- a review of near-generic ideas as they may be employed in the base language
- formal generic templates as defined in Standard Generic Modula-2
 - generic separate modules
 - refining separate modules
 - refining local modules
- the use of generics for parameterizing both techniques and abstract data types
- the use of refinement in local modules to scale software on a modular basis

Reserved Words

GENERIC

Standard Identifier

TYPE (new context)

[Contents](#)

16.11 Assignments

Questions

1. What is meant by the term *generic software*?
2. What new keyword is used in standard Generic Modula-2, and in what contexts is it used?
3. What are the four new kinds of compilation units in Generic Modula-2?
4. What are the three kinds of refining modules in Generic Modula-2?
5. Describe the kinds of formal parameters used in Generic Separate modules.
6. What actual parameters are compatible with these formal parameters?
7. Why are variables not permitted as actual parameters in refining modules when they are allowed for formal value parameters in procedures?
8. "The reserved word *TYPE* is re-used in Generic modula-2 as though it were a pervasive." Explain this statement.
9. If Generic Modula-2 were implemented with a program that simply read a refining module and then created a copy for compilation of the appropriate generic separate module without the parameters but with a few lines renaming the formal parameters as the actual ones in the style of [section 16.1.1](#), what differences would there be between this and an implementation that evaluated the actual parameters and did the substitution as part of the compilation process without creating a copy?
10. The generic module [Lists](#) in section 16.7 has a bare minimum of comments. Study the code carefully and comment it in detail.
11. Give a detailed description of generics in at least two other computing languages and compare both to each other and to Modula-2.
12. Name at least four programs either from the examples in the book or from the assignments you have done that would *not* be appropriate as generic separate modules. Explain why in detail.
13. Discuss ways in which Standard Generic Modula-2 might eliminate the need for variant records. Or, does it?
14. In what ways is refining locally different from refining separately?
15. What additional functionality does the ability to refine locally provide that cannot be had by refining separately only?
16. What can a local refining module have that a separate refining module cannot have?

Problems

17. Consider the module [ArithSeq](#) in section 3.8. Rewrite this module so as to be generic for the data in the sequence and test your code with a client that imports a refinement.
18. Consider the module [SortThree](#) in section 4.3. Rewrite this module so as to be generic for the data to be sorted and test your code with a client that imports a refinement.
19. Consider the module [Points](#) in section 6.9. Rewrite this module so as to be generic for the data in the ordered pair and test your code with a client that imports a refinement.
20. Consider the module [PointToPoint](#) in section 6.9. Rewrite this module so as to be generic for the data

- in the sequence and to have all the facilities of the module *Points* also available for export and then and test your code with a client that imports a refinement.
21. Consider the module [Vectors](#) in section 7.11. Rewrite this module so as to be generic for the data in the components and test your code with a client that imports a refinement.
 22. Produce and test a generic insert sort.
 23. Produce and test a generic shell sort.
 24. Produce and test a generic merge sort.
 25. Produce and test a generic heapsort.
 26. Produce and test a refinement of a sort that can sort an array of your favourite type of records.
 27. Test by using a refinement of the generic module [Lists in section 16.7.1](#).
 28. Complete and test the simple generic separate module [Stacks](#) in section 16.2.1 and test your code with a client that imports a refinement.
 29. Complete and test the simple generic separate module [Matrix](#) in section 16.2.1 and test your code with a client that imports a refinement. You will need to add some simple matrix manipulation procedures as you deem appropriate.
 30. Complete and test the simple generic separate module [Sorts](#) in section 16.2.1 and test your code with a client that imports a refinement.
 31. Complete and test the simple generic separate module [Counter](#) in section 16.2.1 and test your code with a client that imports a couple of refinements.
 32. Provide and test the implementation part of the generic module [Queues](#) in section 16.7.2.
 33. Provide and test the implementation part of the generic module [Stacks](#) in section 16.7.3.
 34. Design, implement, and test a generic binary tree ADT.
 35. From the last question, create, using refinement, a balanced binary tree.
 36. Complete and test the simple generic separate module [ListsSorted](#) and the refinement [IntListsSorted](#) in section 16.6 and test your code. You may make use of the [generic list module](#) in that same section, or another one elsewhere in the chapter.
 37. Complete and test the simple generic separate module *PriorityQueues* and the refinement *StudentPriorityQueues* in section 16.6 and test your code. You may make use of the generic queue module in that same section, or another one elsewhere in the chapter.
 38. Refine and test the generic module [Lists](#) in section 16.7.1.
 39. Design, implement, and test a two way list ADT. Can you base it on the above module [Lists](#)? Why or why not?
 40. Design, implement, and test a circular list ADT. Can you base it on the above module [Lists](#)? What about on the two way list? Why or why not?
 41. Complete, refine and test the generic module *Queues* in section 16.7.2.
 42. Complete, refine and test the generic module *Stacks* in section 16.7.3.
 43. Pretend that your implementation lacks the standard generic Modula-2 extensions. (You may not need to pretend.) Write a program that will take as input a file containing only a refining separate module (implementation or definition) and find and refine the appropriate generic separate module (text file) using renaming as in [section 16.1.1](#).

Challenges

44. Add to the code in the last question the ability to refine as a local module.
 45. Redesign, implement, and test the [statistics modules](#) in section 7.7 so as to take advantage of generics. Show your design to your instructor before producing code. Among several other things, your modules should work for both real types.
 46. Gather several of your generic sorting techniques into a single module. Make a new such module, but with the comparison procedure type having reference semantics rather than the value semantics employed in most such routines. Can the choice of semantics also be made generic? Why or why not?
 47. Design, implement, and test a generic table ADT. Show your design to your instructor before producing code.
 48. Design, implement, and test a generic B-tree ADT.
-

[Contents](#)

Chapter 16

Generic Modula-2

[16.0 Chapter Goals](#)

[16.1 Generics In the Base Language \(Revisited\)](#)

[16.1.1 Semi Generic Methods and Structures](#)

[16.1.2 Limitations of Fully Generic Techniques](#)

[16.1.3 Limitations of Fully Generic Structures](#)

[16.1.4 Summary](#)

[16.2 Generic Separate Library Modules](#)

[16.2.1 Generic Definition Modules](#)

[16.2.2 Generic Implementation Modules](#)

[16.2.3 Formal Module Parameters](#)

[16.3 Refining Separate Library Modules](#)

[16.3.1 Refining Definition Modules](#)

[16.3.2 Refining Implementation Modules](#)

[16.3.3 Multiple Refinements](#)

[16.3.4 Actual Parameters](#)

[16.4 Refining Within a Program Module](#)

[16.5 Refining Within an Implementation Module](#)

[16.6 Making New ADTs from Old With Generics](#)

[16.7 Extended ADT Examples](#)

[16.7.1 Generic Lists](#)

[16.7.2 Generic Queues](#)

[16.7.3 Generic Stacks](#)

[16.8 Dependency and Order in Generic Modula-2](#)

[16.8.1 Module Dependencies](#)

[16.8.2 Nested Module Refinement Order](#)

[16.8.3 Module Initialization and Termination Order](#)

[16.8.4 Import and Export Lists](#)

[16.9 Summary and Comparison With Other Notations](#)

[16.10 Chapter Summary](#)

Contents

17.0 Chapter Goals

The purpose of this chapter is to introduce the remaining pervasive Modula-2 types and to apply some of the theory of the last few chapters to some specific and relatively advanced situations. In the course of this, some additional ISO standard library modules will also be discussed and details on writing them provided.

On completing the chapter, the student should understand and be able to use the following: complex numbers, decimal types, the date and time, simple finite state machine scanners, and high and low level whole number conversion routines.

Data Representation Abstractions

General:

complex numbers, (binary coded) decimals, time and date record structures

Realized in the Modula-2 notation:

COMPLEX, LONGCOMPLEX, BCD (optional) the *SysClock* structures

Data Manipulation Abstractions

General:

I/O for complex numbers, date and time and low level whole number I/O

Realized in the Modula-2 notation:

ComplexIO and LongComplexIO (both non-standard) and the entire suite of Whole number I/O modules

Programming Abstractions

General:

scanning input strings with finite state machines

Realized in the Modula-2 notation:

the scanners in the low level I/O module *WholeConv*

17.1 Standard and Non-Standard Numeric Types

The numeric types encountered thus far--CARDINAL, INTEGER, REAL, and LONGREAL can be counted on to be available in all versions of Modula-2, whether they follow the ISO standard or not. Many non-standard versions will have either SHORT or LONG (or both) forms of INTEGER and CARDINAL and some, to accommodate the idiosyncrasies of the underlying system, will have a variety of REAL types. If the version is ISO standard, any of these additional types will have to be imported from SYSTEM.

There are additional numeric types in common use. These include complex numbers, and (binary coded) decimal numbers. The purpose of this section is to elaborate on the support for and the use of these two types. In the process, there will be some discussion of the formatting of output, and that will be followed up on in connection with the output formatting of items of other number types as well.

[Contents](#)

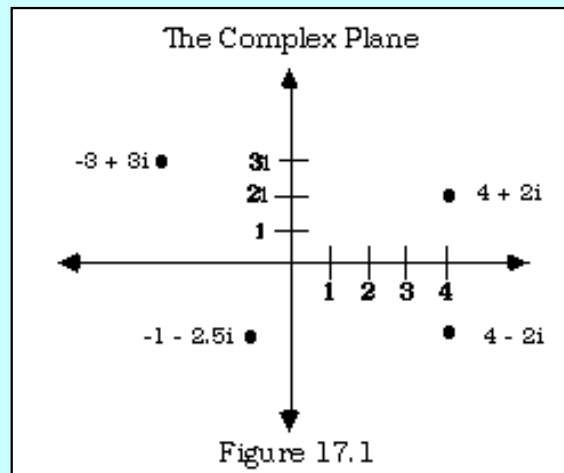
17.2 Complex Numbers

17.2.1 Complex Numbers Defined

There are a number of applications in mathematics and physics for which the real number are insufficient. An extended one of these will be considered later in this chapter. From an aesthetic point of view alone, it is convenient to be able to talk about solutions to equations such as $x^2 = -4$ in much the same way as equations such as $x^2 = 4$. After all, there is not, at first glance, much difference between the two, except that the solution to the former, whatever it is, it is not a real number, because the square of a real number can only be positive. To answer this and other more important needs, we define the complex numbers:

A Complex number has the form $a + bi$ where a and b are Real numbers and $i^2 = -1$. The number " a " is called the real part, and the number " b " is the imaginary part.

In pictorial form, a complex number can be thought of as being a point in the complex plane. This is defined by two axes, much like a two dimensional real plane, except that only the horizontal axis has real numbers. The vertical or *imaginary* axis is marked in units of i (the square root of negative one). Points in the plane can be thought of as ordered pairs (a, b) , but are normally written in the notation above as $a + bi$.



The complex number zero is defined to be the one that represents the origin in the above diagram, that is a complex number is zero if and only if both its real part and its imaginary part are zero ($0 + 0i$).

Addition and subtraction of complex numbers is performed in the same way as it is for vectors (see Chapter 7) where the components are added. Thus, in order to add two complex numbers, one just adds the real parts and we add the imaginary parts. That is,

$$(a + bi) + (c + di) = (a + b) + (c + d)i$$

To multiply two complex numbers, one treats them as any other binomial expressions and multiplies each term of the first by each term of the second:

$$(a + bi) * (c + di) = a * (c + di) + bi * (c + di) = (ac - bd) + (ad + bc)i$$

The reciprocal of a pure imaginary complex number (one with zero real part) is found by multiplying the numerator and denominator by i . Thus:

$$\frac{1}{2i} = \frac{1 \times i}{2i \times i} = \frac{i}{2i^2} = \frac{i}{-2} = -\frac{1}{2}i$$

For other divisions, one multiplies numerator and denominator by the complex conjugate of the denominator. This results in a real denominator and a complex number in the proper form.

The complex conjugate of a complex number $a + bi$ is the number $a - bi$.

$$\frac{(2-i)}{(3+2i)} = \frac{(2-i)(3-2i)}{(3+2i)(3-2i)} = \frac{6-4i-3i+2i^2}{9-6i+6i+4i^2} = \frac{4-7i}{5} = \frac{4}{5} - \frac{7}{5}i$$

In the introductory Calculus course it is shown that certain transcendental functions of real numbers can be expanded (so as to be approximated) as power series in the following way:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \dots$$

If one replaces x with ix in the latter series, it can readily be seen that:

$$e^{ix} = 1 + ix + \frac{(ix)^2}{2!} + \frac{(ix)^3}{3!} + \frac{(ix)^4}{4!} + \frac{(ix)^5}{5!} + \frac{(ix)^6}{6!} + \dots$$

$$e^{ix} = 1 + ix - \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} - \frac{x^6}{6!} - \dots$$

$$e^{ix} = \left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots \right) + i \left(x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots \right)$$

$$e^{ix} = \cos(x) + i\sin(x)$$

This last equation is known as Euler's Formula. From it, one also has that

$$(\cos(x) + i\sin(x))^n = (e^{ix})^n = e^{inx} = \cos(nx) + i\sin(nx)$$

so that it is also possible to write:

$$[r(\cos(x) + i\sin(x))]^n = r^n [\cos(nx) + i\sin(nx)]$$

These formulas provide a means of taking powers and roots of complex numbers readily. First observe that, if in figure 17.1, the distance from the origin to the point representing the complex number is denoted as r and the angle that a ray through the origin and this point makes with the positive real axis is denoted t , then as for vectors, one can write:

$$r = \sqrt{a^2 + b^2} \quad \text{and} \quad t = \arctan\left(\frac{b}{a}\right)$$

Thus, if, for instance $z = (1 - i)^4$ were expressed using these two relations as

$$z = \sqrt{2} \cos\left(\frac{\pi}{4}\right) + i\sqrt{2} \sin\left(\frac{\pi}{4}\right)$$

$$z = \sqrt{2} \left(\cos\left(\frac{\pi}{4}\right) + i\sin\left(\frac{\pi}{4}\right) \right)$$

then one could write

$$z^4 = \left(\sqrt{2} \left(\cos\left(\frac{\pi}{4}\right) + i\sin\left(\frac{\pi}{4}\right) \right) \right)^4$$

$$z^4 = (\sqrt{2})^4 \left(\cos 4\left(\frac{\pi}{4}\right) + i\sin 4\left(\frac{\pi}{4}\right) \right)$$

$$z^4 = 4(\cos(\pi) + i\sin(\pi))$$

$$z^4 = -1 + 0i$$

and this works for fractional exponents as well, allowing one to take various roots, so that, for instance, the sixth root of one can be written as

$$z^{\frac{1}{6}} = (1 + 0i)^{\frac{1}{6}} = 1^{\frac{1}{6}} \left(\cos \frac{1}{6}(0) + i\sin \frac{1}{6}(0) \right) = 1 + 0i$$

and since the angle 0 is coterminal with the angles $2\frac{1}{4}$, $4\frac{1}{4}$, $6\frac{1}{4}$, $8\frac{1}{4}$, and $10\frac{1}{4}$ there are also roots at

$$z^{\frac{1}{6}} = 1^{\frac{1}{6}} \left(\cos \frac{1}{6}(2n\pi) + i\sin \frac{1}{6}(2n\pi) \right)$$

for each of these, yielding

$$\cos\left(\frac{n\pi}{3}\right) + i\sin\left(\frac{n\pi}{3}\right) \quad \text{for } n = 1, 2, 3, 4, \text{ and } 5.$$

At $n = 6$ the results begin to repeat. After all, there can only be six sixth roots of a number. These roots are therefore :

$$(1+0i), (0+i), \text{ and } \left(\pm\frac{1}{2}\pm i\frac{\sqrt{3}}{2}\right)$$

This discussion of power and root finding has been undertaken mainly as illustration. However, the underlying principles will be returned to later in the chapter.

17.2.2 Implementing non-ISO Complex Numbers

There are a variety of approaches to implementing complex numbers in non ISO standard versions of Modula-2. One could do so transparently:

```
DEFINITION MODULE ComplexNumbers;
(* by R. Sutcliffe
   last modified 1996 10 30 *)

TYPE
  Parts = (re, im);
  Complex = ARRAY Parts OF REAL;

PROCEDURE Cmplx (real, imag : REAL) : Complex;
(* constructs a complex from an ordered pair *)
PROCEDURE Re (num : Complex) : REAL;
(* returns the real part of a complex number *)
PROCEDURE Im (num : Complex) : REAL;
(* returns the imaginary part of a complex number *)
PROCEDURE Add (first, second : Complex) : Complex;
PROCEDURE Subtract (first, second : Complex) : Complex;
PROCEDURE Multiply (first, second : Complex) : Complex;
PROCEDURE Divide (first, second : Complex) : Complex;
END ComplexNumbers.
```

Alternately, one could use a record with two fields called *re* and *im*, and otherwise the same arithmetic operations. The implementation module for this data type as defined above is fairly straightforward, and most of it is shown below:

```
IMPLEMENTATION MODULE ComplexNumbers;
(* by R. Sutcliffe
   last modified 1996 10 30 *)

PROCEDURE Cmplx (real, imag : REAL) : Complex;
(* constructs a complex from an ordered pair *)
VAR
  temp : Complex;
BEGIN
  temp [re] := real;
  temp [im] := imag;
  RETURN temp;
END Cmplx;

PROCEDURE Re (num : Complex) : REAL;
(* returns the real part of a complex number *)
BEGIN
  RETURN num[re];
```

```

END Re;

PROCEDURE Im (num : Complex) : REAL;
(* returns the imaginary part of a complex number *)
BEGIN
    RETURN num[im];
END Im;

PROCEDURE Add (first, second : Complex) : Complex;
VAR
    temp : Complex;
BEGIN
    temp [re] := first [re] + second [re];
    temp [im] := first [im] + second [im];
    RETURN temp;
END Add;

PROCEDURE Subtract (first, second : Complex) : Complex;
VAR
    temp : Complex;
BEGIN
    temp [re] := first [re] - second [re];
    temp [im] := first [im] - second [im];
    RETURN temp;
END Subtract;

PROCEDURE Multiply (first, second : Complex) : Complex;
VAR
    temp : Complex;
BEGIN
    temp [re] := first [re] * second [re] - first [im] * second [im];
    temp [im] := first [re] * second [im] + first [im] * second [re];
    RETURN temp;
END Multiply;

PROCEDURE Divide (first, second : Complex) : Complex;
VAR
    temp : Complex;
BEGIN
    (* left as an exercise *)
END Divide;

END ComplexNumbers.

```

There are other things that would still have to be taken care of, such as I/O and various mathematical operations, but these could be done in other library modules. In addition, it might be desirable to have a separate ADT called *LongComplex* whose parts were LONGREAL.

For minimal testing purposes at this juncture, it is worth while to include a small application program that makes use of the data type *Complex*. It incidentally addresses the issue of writing complex numbers.

17.2.3 Testing the non-ISO Complex Implementation

```

MODULE TestComplex;
(* by R. Sutcliffe
   to test transparent implementation of complex numbers
   modified 1996 10 30 *)
FROM ComplexNumbers IMPORT
    Complex, Add, Multiply, Cmplx, Re, Im;
FROM STextIO IMPORT
    WriteChar, WriteLn, WriteString;
FROM SRealIO IMPORT
    WriteReal;
VAR
    number1, number2, number3 : Complex;

PROCEDURE WriteComplex (number : Complex; flen : CARDINAL);
(* Of the space provided in flen, four places are needed to write the + between the
   two reals, and one to write the letter i. The rest is divided equally between the
   two reals. *)
VAR
    realFlen : CARDINAL;
BEGIN
    IF flen < 4
    THEN
        realFlen := 0;
    ELSE
        realFlen := (flen - 4) DIV 2;
    END;
    WriteReal (Re(number), realFlen);
    WriteString (" + ");
    WriteReal (Im(number), realFlen);
    WriteChar ('i');
END WriteComplex;

BEGIN
    (* Initialize complex numbers *)
    number1 := Cmplx (4.0, 16.0);
    number2 := Cmplx (-13.0, 2.0);

    number3 := Add(number1, number2);
    WriteString ("The sum is ");
    WriteComplex (number3, 20);
    WriteLn;
    number3 := Multiply (number1, number2);
    WriteString ("The product is ");
    WriteComplex (number3, 20);
    WriteLn;
END TestComplex.

```

The output from this program, as expected was:

```

The sum is -9.00000 + 18.00000i
The product is -84.0000 + -200.000i

```

17.2.4 Opaque non-ISO Complex Numbers

Notice that the client program "knows" the names of the parts of the type *Complex*. This fact has not actually been used because the procedures *Cmplx*, *Re*, and *Im* were used to construct and find the parts of a complex number, but it could have been. As indicated in earlier chapters, it is a better idea to keep such details hidden, so that the client program does not have access to such information, but rather the data can be operated upon only by the procedures which are also contained in the module along with the data type definition. There have been several instances of such opaque types throughout the text. However, as noted in [section 12.8](#) and illustrated with the module *points* in that chapter, one cannot have a procedure allocate memory for a temporary opaque variable and then return it because every call to such a procedure would use a little more memory unnecessarily, and one cannot deallocate the memory before returning either because then the memory pointed to after the return is no part of a valid allocation. The real problem is that in writing

```
number3 := Add(number1, number2);
```

with opaque variables, access to the memory pointed to by *number3* before the assignment is simply lost. This is a consequence of the fact that opaque types are in fact pointers.

Thus, an implementation of complex numbers in classic Modula-2 requires that one use variable parameters in regular procedures rather than function procedures returning an item of the opaque type. Moreover, procedures are needed to initialize and destroy complex entities. We could have:

DEFINITION MODULE ComplexNumbers0;

(* by R. Sutcliffe

last modified 1996 10 30 *)

TYPE

Complex; (* no details here *)

PROCEDURE Init (VAR complexToInit : Complex);

(* Must be called to create a Complex number. The number created will have both real and imaginary parts zeroed. *)

PROCEDURE Destroy (VAR complexToDestroy : Complex);

(* Call to give back the memory allocated to a Complex number *)

PROCEDURE Cmplx (real, imag : REAL; VAR result : Complex);

(* Takes two reals as the parts and assigns them to the complex. *)

PROCEDURE Re (cNumber : Complex) : REAL;

(* Returns the real part of the given complex number. *)

PROCEDURE Im (cNumber : Complex) : REAL;

(* Returns the imaginary part of the given complex number. *)

PROCEDURE Negate (originalNum: Complex; VAR result : Complex);

(* Returns the opposite of the complex number. *)

PROCEDURE Add (firstNum, secondNum : Complex; VAR result : Complex);

(* Returns the sum of the two complex numbers. *)

PROCEDURE Subtract (firstNum, secondNum : Complex; VAR result : Complex);

(* Returns the difference of the two complex numbers. *)

PROCEDURE Multiply (firstNum, secondNum : Complex; VAR result : Complex);

(* Returns the product of the two complex numbers. *)

PROCEDURE Divide (firstNum, secondNum : Complex; VAR result : Complex);

(* Returns the quotient of the two complex numbers. *)

END ComplexNumbers0.

One must still decide whether to implement the type as an array or a record. A record style implementation is given below.


```

IMPLEMENTATION MODULE ComplexNumbers0;
(* by R. Sutcliffe
   last modified 1996 10 30 *)

FROM Storage IMPORT
    ALLOCATE, DEALLOCATE;

TYPE
    Complex = POINTER TO ComplexData;    (* opaque, so must be a pointer *)
    ComplexData =    (* now here is its structure *)
        RECORD
            re : REAL;
            im : REAL;
        END    (* record *);

PROCEDURE Init (VAR complexToInit : Complex);
BEGIN
    NEW (complexToInit);    (* get memory dynamically *)
    complexToInit^.re := 0.0;    (* and zero it off *)
    complexToInit^.im := 0.0;
END Init;

PROCEDURE Cmplx (re, im : REAL; VAR complexToAssign: Complex);
BEGIN
    complexToAssign^.re := re;
    complexToAssign^.im := im;
END Cmplx;

PROCEDURE Re (cNumber : Complex) : REAL;
BEGIN
    RETURN cNumber^.re;
END Re;

PROCEDURE Im ( cNumber : Complex) : REAL;
BEGIN
    RETURN cNumber^.im;
END Im;

PROCEDURE Negate (originalNum: Complex; VAR result : Complex);
BEGIN
    result^.re := -originalNum^.re;
    result^.im := -originalNum^.im;
END Negate;

PROCEDURE Add (firstNum, secondNum : Complex; VAR result : Complex);
BEGIN
    result^.re := firstNum^.re + secondNum^.re;
    result^.im := firstNum^.im + secondNum^.im;
END Add;

PROCEDURE Subtract (firstNum, secondNum : Complex; VAR result : Complex);
BEGIN
    result^.re := firstNum^.re - secondNum^.re;

```

```

    result^.im := firstNum^.im - secondNum^.im;
END Subtract;

PROCEDURE Multiply (firstNum, secondNum : Complex; VAR result : Complex);
BEGIN
    result^.re := firstNum^.re * secondNum^.re - firstNum^.im * secondNum^.im;
    result^.im := firstNum^.re * secondNum^.im + firstNum^.im * secondNum^.re;
END Multiply;

PROCEDURE Divide (firstNum, secondNum : Complex; VAR result : Complex);
BEGIN
    result^.re :=
        (firstNum^.re*secondNum^.re + firstNum^.im*secondNum^.im)/
        (secondNum^.re*secondNum^.re + secondNum^.im*secondNum^.im);
    result^.im :=
        (firstNum^.im*secondNum^.re + firstNum^.re*secondNum^.im)/
        (secondNum^.re*secondNum^.re + secondNum^.im*secondNum^.im);
END Divide;

PROCEDURE Destroy (VAR complexToDestroy : Complex);
BEGIN
    DISPOSE (complexToDestroy);
END Destroy;

END ComplexNumbers0.

```

As usual, the implementation could be changed without affecting the definition in any way. Now, consider how the client program must be written with the new regular procedure syntax for the calls:

17.2.5 Testing the Opaque non-ISO Complex Implementation

```

MODULE TestComplex0;
(* by R. Sutcliffe
   last modified 1996 10 30 *)

FROM ComplexNumbers0 IMPORT
    Complex, Init, Cmplx, Re, Im, Add, Multiply;
FROM STextIO IMPORT
    WriteLn, WriteString, WriteChar;
FROM SRealIO IMPORT
    WriteReal;
VAR
    number1, number2, number3 : Complex;

PROCEDURE WriteComplex (number : Complex; flen : CARDINAL);
(* Of the space provided in flen, four places are needed to write the + between the
   two reals, and one to write the letter i. The rest is divided equally between the
   two reals. *)
VAR
    realFlen : CARDINAL;
BEGIN
    IF flen < 4

```

```

    THEN
        realFlen := 0;
    ELSE
        realFlen := (flen - 4) DIV 2;
    END;
WriteReal (Re(number), realFlen);
WriteString (" + ");
WriteReal (Im(number), realFlen);
WriteChar ('i');
END WriteComplex;

BEGIN    (* ClientProgram *)
    (* Create the complex numbers *)
    Init (number1);
    Init (number2);
    Init (number3);

    (* Initialize the complex numbers number1 and number2 *)
    Cmplx (4.0, 16.0, number1);
    Cmplx (-13.0, 2.0, number2);

    (* the last part is the same as before. *)
    Add (number1, number2, number3);
    WriteString ("The sum is ");
    WriteComplex (number3, 20);
    WriteLn;
    Multiply (number1, number2, number3);
    WriteString ("The product is ");
    WriteComplex (number3, 20);
    WriteLn;
END TestComplex0.

```

Having to use such syntax for the manipulation of a numeric type is a nuisance. One does not expect any numeric type to have to be initialized or to be added using three parameter regular procedures. However, if all one has is classical Modula-2 the only choice is to implement transparently or to live with the extra apparatus required by an opaque implementation. If it is necessary to go further and define arithmetic and mathematical operations on the type complex, that can be done in a separate module. For details on what such a module should contain, see the next section.

[Contents](#)

17.3 ISO Complex Types and Support

Because of the awkwardness that is inherent in dealing with opaque types, and because complex numbers are a basic numeric type, the ISO committee eventually decided to add support for an abstract type directly into the language.

*The Modula-2 complex types are **COMPLEX**, **LONGCOMPLEX**, and the **C-type**.*

The parts of a **COMPLEX** number can be thought of as corresponding to the type **REAL** and the parts of a **LONGCOMPLEX** number as corresponding to the type **LONGREAL**. *C-type* is the type of complex literals, in the same way that *R-type* is the type of the real literals.

There is a constructor and these are two component extractors for the complex types.

*The Modula-2 complex constructor is **CMPLX** and the extractors are **IM** and **RE**.*

All of **COMPLEX**, **LONGCOMPLEX**, **CMPLX**, **RE**, and **IM** are standard identifiers. **CMPLX**, **RE**, and **IM** work with all complex types, and the operations of addition, subtraction, multiplication and division are written in the same way as for all other numeric types and work on all complex types. (This is much more satisfying than in the last section.) All of the following are correct:

VAR

```
r1, r2 : REAL;  
r3, r4 : LONGREAL;  
z1, z2 : COMPLEX;  
z3, z4 : LONGCOMPLEX;
```

CONST

```
zero = CMPLX (0.0, 0.0);  
i = CMPLX (0.0, 1.0);
```

BEGIN

```
z1 := CMPLX (2.0, 4.5);  
z2 := z1 + i;  
z3 := zero;  
z3 := z3 / CMPLX (1.0, 1.0);  
r3 := 5.7;  
r4 := 4.5;  
z4 := CMPLX (r3, r4);  
r1 := RE (z1);  
r2 := IM (z1);
```

17.3.1 ISO COMPLEX Math Library Support

A number of items are defined in an ISO standard support library for complex mathematics. If using the type **COMPLEX**, the library is called *ComplexMath*. The corresponding library for **LONGCOMPLEX** (which has exactly the same contents except all are **LONG**) is *LongComplexMath*. Here is the definition of *ComplexMath*:

DEFINITION MODULE ComplexMath;

```
(* =====  
      Definition Module from  
      ISO Modula-2  
      ISO/IEC IS10515 by JTC1/SC22/WG13
```

Original COMPLEX specification and
design of ComplexMath
Copyright © 1990-1991 by R. Sutcliffe
Assigned to the BSI for standards work
Last modification date 1994 08 31

=====*)

(* Mathematical functions for the type **COMPLEX** *)

CONST

i = **CMPLX** (0.0, 1.0);
one = **CMPLX** (1.0, 0.0);
zero = **CMPLX** (0.0, 0.0);

PROCEDURE abs (z: **COMPLEX**): **REAL**;

(* Returns the length of z *)

PROCEDURE arg (z: **COMPLEX**): **REAL**;

(* Returns the angle that z subtends to the positive real axis *)

PROCEDURE conj (z: **COMPLEX**): **COMPLEX**;

(* Returns the complex conjugate of z *)

PROCEDURE power (base: **COMPLEX**; exponent: **REAL**): **COMPLEX**;

(* Returns the value of the number base raised to the power exponent *)

PROCEDURE sqrt (z: **COMPLEX**): **COMPLEX**;

(* Returns the principal square root of z *)

PROCEDURE exp (z: **COMPLEX**): **COMPLEX**;

(* Returns the complex exponential of z *)

PROCEDURE ln (z: **COMPLEX**): **COMPLEX**;

(* Returns the principal value of the natural logarithm of z *)

PROCEDURE sin (z: **COMPLEX**): **COMPLEX**;

(* Returns the sine of z *)

PROCEDURE cos (z: **COMPLEX**): **COMPLEX**;

(* Returns the cosine of z *)

PROCEDURE tan (z: **COMPLEX**): **COMPLEX**;

(* Returns the tangent of z *)

PROCEDURE arcsin (z: **COMPLEX**): **COMPLEX**;

(* Returns the arcsine of z *)

PROCEDURE arccos (z: **COMPLEX**): **COMPLEX**;

(* Returns the arccosine of z *)

PROCEDURE arctan (z: **COMPLEX**): **COMPLEX**;

(* Returns the arctangent of z *)

PROCEDURE polarToComplex (abs, arg: **REAL**): **COMPLEX**;

(* Returns the complex number with the specified polar coordinates *)

```

PROCEDURE scalarMult (scalar: REAL; z: COMPLEX): COMPLEX;
  (* Returns the scalar product of scalar with z *)

PROCEDURE IsCMathException (): BOOLEAN;
  (* Returns TRUE if the current coroutine is in the exceptional execution state
  because of the raising of an exception in a routine from this module; otherwise
  returns FALSE. *)

END ComplexMath.

```

Here are a few observations about these contents (and how to implement them) that of course apply to both versions of the library:

The constants i, one, and zero

These constants are provided for convenience and are the implementation defined approximations to the specified values.

The procedure abs

A call to *abs* produces the implementation defined approximation to the Modulus, (or length, or absolute value) of the complex number. As indicated earlier, this is the distance in the complex plane from the origin to the point that represents the complex number. If $z = a + bi$, then $|z| = \sqrt{a^2 + b^2}$. Note that an overflow exception may occur in performing this computation, even when the complex number is itself well defined. So, if one were implementing this function, a better approach to the calculation might be:

modulus (z) =

let t = max(RE(cmplx),IM(cmplx))

let u = min(RE(cmplx),IM(cmplx))

return $\sqrt{1 + \frac{u^2}{t^2}}$ a

end modulus

It can readily be seen with a little algebraic manipulation that the two expressions are equivalent, but the likelihood of an overflow is less in using the second.

The procedure arg

A call to *arg* produces the implementation defined approximation to the angle the complex number makes with the positive real axis in the complex plane:

If RE(cmplx) < 0 it yields

pi-arctan(IM(cmplx)/RE(cmplx)) if IM(cmplx)<0.

If RE(cmplx) = 0 it yields

pi/2 when IM(cmplx) < 0 and

is undefined (an exception occurs) if IM(cmplx) = 0 also.

That is, approximation values over the entire range of $-\pi < \arg \leq \pi$ are produced.

The procedure conj

A call to *conj* with base = a + bi as parameter produces the implementation defined approximation to the complex conjugate of the parameter, that is to a - bi. This is useful in doing divisions by complex numbers.

The procedure power

A call to *power* with base = a + bi as complex parameter and exponent the real parameter produces the implementation defined approximation to the complex number base raised to the power exponent, that is, if $z = a + bi$, then $z^n = (a + bi)^n$

Recall that it is also the case that such numbers can be expressed as $z = r(\cos \theta + i \sin \theta)$, by Euler's formula:

$z^n = (r (\cos \theta + i \sin \theta))^n = (r e^{i\theta})^n = r^n e^{in\theta} = r^n (\cos n\theta + i \sin n\theta)$

In other words, the result of such an exponentiation is a complex number with modulus r^n and argument $n\theta$.

The procedure sqrt

There are, of course, two square roots to a complex number. For instance, both i and -i are square roots of -1. A call to *sqrt* (cmplx) produces the implementation defined approximation to the principal square root of the complex number. That is, the result is the complex number whose argument has minimum absolute value and where *result* * *result* equals *cmplx*.

The procedure exp

A call to exp with $z = a + bi$ as parameter produces the implementation defined approximation to the exponential of $a + bi$, that is, to $e^{a+bi} = e^a e^{bi}$

The procedure ln

A call to ln with $z = a + bi$ as parameter produces the implementation defined approximation to the principal value of the natural logarithm of $a + bi$. This can be found by observing that if $\ln(z) = a + bi$ and the natural logarithm is to be the inverse of exponentiation, then

$$z = e^{\ln z} = e^{a+bi} = e^a e^{bi} = e^a (\cos(b) + i \sin(b))$$

so that $\text{abs}(z) = e^a$ and $\arg(z) = b$. This means that $b = \ln(\text{abs}(z))$ and $d = \arg(z)$. That is, $\ln(z) = \text{CMPLX}(\ln(\text{abs}(z)), \arg(z))$.

In order to handle the trigonometric functions, one assumes the series discussed in [section 17.1](#) apply and then expands both:

$$e^{iz} = \cos(z) + i \sin(z)$$

$$e^{-iz} = \cos(z) - i \sin(z)$$

Solving these as a system of equations for $\cos(z)$ and $\sin(z)$ produces the definitions used in the following:

The procedure sin

A call to sin with $z = a + bi$ as parameter produces the implementation defined approximation to the sine of the parameter, which is defined as:

$$\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}$$

The procedure cos

A call to cos with $z = a + bi$ as parameter produces the implementation defined approximation to the cosine of the parameter, which is defined as:

$$\cos(z) = \frac{e^{iz} + e^{-iz}}{2}$$

It is not difficult to verify that, as is the case for trigonometric functions taking real valued arguments, $\sin^2(z) + \cos^2(z) = 1$.

The tangent function can be defined on complex numbers in the same way as it is on real numbers.

$$\tan(z) = \frac{\sin(z)}{\cos(z)} = \frac{e^{iz} - e^{-iz}}{2i} \times \frac{2}{e^{iz} + e^{-iz}} = \frac{e^{iz} - e^{-iz}}{i(e^{iz} + e^{-iz})}$$

The procedure tan

A call to tan with $z = a + bi$ as parameter produces the implementation defined approximation to the tangent of the parameter, which is defined as:

$$\tan(z) = \frac{\sin(z)}{\cos(z)} \text{ unless } z \text{ lies on the imaginary axis (has } \arg \frac{1}{4}/2 \text{ or } \frac{3}{4}/2 \text{) in which case there shall be an exception raised.}$$

Turning attention to the inverse trigonometric functions, it is easy to require that on the domain for which they are defined, one must have

$$\sin(\text{Arcsin}(z)) = z \text{ and } \cos(\text{Arccos}(z)) = z.$$

Taking these and letting $y = \text{Arc sin}(z)$ and $y = \text{Arc cos}(z)$ for the moment (respectively), one solves for y in $z = \sin(y)$ and in $z = \cos(y)$ to obtain:

$$z = \sin(y)$$

$$z = \frac{e^{iy} - e^{-iy}}{2i}$$

$$2iz = e^{iy} - e^{-iy}$$

$$(2iz = e^{iy} - e^{-iy}) \times e^{iy}$$

$$e^{2iy} - 2ize^{iy} - 1 = 0$$

$$e^{iy} = \frac{2iz \pm \sqrt{-4z^2 + 4}}{2}$$

$$e^{iy} = iz + \sqrt{1 - z^2}$$

$$iy = \ln(iz + \sqrt{1 - z^2})$$

$$y = \operatorname{Arcsin}(z) = \frac{1}{i} \ln(iz + \sqrt{1 - z^2})$$

$$z = \cos(y)$$

$$z = \frac{e^{iy} + e^{-iy}}{2}$$

$$2z = e^{iy} + e^{-iy}$$

$$(2z = e^{iy} + e^{-iy}) \times e^{iy}$$

$$e^{2iy} + 2ze^{iy} - 1 = 0$$

$$e^{iy} = \frac{2z \pm \sqrt{4z^2 + 4}}{2}$$

$$e^{iy} = z + \sqrt{1 + z^2}$$

$$iy = \ln(z + \sqrt{1 + z^2})$$

$$y = \operatorname{Arccos}(z) = \frac{1}{i} \ln(z + \sqrt{1 + z^2})$$

The negative values of the root in the quadratic formula are ignored at the third to last line in favour of using only the principal root in each case. Thus one has:

The procedure arcsin

A call to arcsin with $z = a + bi$ as parameter produces the implementation defined approximation to the arcsine of the parameter, which is defined as the complex number whose argument has minimum absolute value and where $\sin(\text{result})$ produces z . This may be expressed mathematically as:

$$\operatorname{arcsin}(z) = \frac{1}{i} \ln(iz + \sqrt{1 - z^2})$$

The procedure arccos

A call to arccos with $z = a + bi$ as parameter produces the implementation defined approximation to the arccosine of the parameter. That is, the result is the complex number whose argument has minimum absolute value and where $\cos(\text{result})$ produces z . This may be expressed mathematically as:

$$\operatorname{arccos}(z) = \frac{1}{i} \ln(z + \sqrt{z^2 - 1})$$

In a similar manner, one assumes that $\tan(\operatorname{Arctan}(z)) = z$ and works in a similar way to find a form for $\operatorname{Arctan}(z)$ as follows:

$$z = \frac{e^{iy} - e^{-iy}}{i(e^{iy} + e^{-iy})}$$

$$e^{iy} - e^{-iy} = zie^{iy} + zie^{-iy}$$

$$(e^{iy} - e^{-iy} = zie^{iy} + zie^{-iy})e^{iy}$$

$$e^{2iy} - 1 = zie^{2iy} + zi$$

$$e^{2iy}(1 - zi) = 1 + zi$$

$$e^{2iy} = \frac{1 + zi}{1 - zi}$$

$$2iy = \ln\left(\frac{1 + zi}{1 - zi}\right)$$

$$y = \text{Arc tan}(z) = \frac{1}{2i} \ln\left(\frac{1 + zi}{1 - zi}\right)$$

The procedure arctan

A call to arctan with $z = a + bi$ as parameter produces the implementation defined approximation to the arctangent of the parameter. That is, the result is the complex number whose argument has minimum absolute value and where $\tan(\text{result})$ produces z . This may be expressed mathematically as:

$$mz = m(a + bi) = ma + mbi$$

The procedure polarToComplex

A call to polarToComplex with parameters *abs* and *arg* produces the implementation defined approximation to the complex number having modulus equal to *abs* and argument equal to *arg*.

The procedure scalarMult

A call to scalarMult with the parameters *scalar* and *cmplx* produces the implementation defined approximation to the scalar product of the scalar and the complex number, that is $mz = m(a + bi) = ma + mbi$. This result will have modulus mz and the same argument as z .

The actual code is now quite easy to produce. Note how short the procedures have become. This makes them easy to write, easy to understand, and easy to debug.

17.3.2 Implementing ComplexMath

```
IMPLEMENTATION MODULE ComplexMath;
```

```
( * =====
    Initial Coding by Gordon Tisher
    Implementation © 1994
    by R. Sutcliffe
    Trinity Western University
    7600 Glover Rd., Langley, BC Canada V2Y 1Y1
    e-mail: rsutcl@twu.ca
    Last modification date 1996 12 13
    =====* )

(* Mathematical functions for the type COMPLEX *)
(* original 1994 08 31 by GT
   fixed a bug in arccos 1996 12 13 RS *)
```

```

( *****
    Imports
    ***** )

FROM EXCEPTIONS IMPORT
    ExceptionSource, RAISE, AllocateSource, IsCurrentSource, IsExceptionalExecution;

IMPORT RealMath;

( *****
    Constants
    ***** )

CONST
    twoi = CMPLX (0.0, 2.0);

( *****
    Global variables
    ***** )

VAR
    mathExceptionSource : ExceptionSource;

( *****
    Procedures
    ***** )

PROCEDURE abs (z: COMPLEX): REAL;
    (* Returns the length of z *)
VAR
    re, im : REAL;
BEGIN
    re := RE (z);
    im := IM (z);
    RETURN RealMath.sqrt ( (re*re) + (im*im) );
END abs;

PROCEDURE arg (z: COMPLEX): REAL;
    (* Returns the angle that z subtends to the positive real axis *)
VAR
    re, im : REAL;
BEGIN
    re := RE (z);
    im := IM (z);

    IF im > 0.0
        THEN
            RETURN RealMath.arctan ( im/re );
        ELSIF re < 0.0 THEN
            RETURN RealMath.pi + RealMath.arctan ( im/re );
        ELSE
            RETURN RealMath.pi / 2.0;
        END; (* if *)
    ELSIF im < 0.0 THEN

```

```

    IF re < 0.0 THEN
        RETURN -RealMath.pi + RealMath.arctan ( im/re );
    ELSE
        RETURN -RealMath.pi / 2.0;
    END; (* IF *)
ELSE
    IF re < 0.0 THEN
        RETURN RealMath.pi;
    ELSE
        RAISE (mathExceptionSource, 0, "ComplexMath.arg: zero value");
    END; (* IF *)
END (* IF *)
END arg;

PROCEDURE conj (z: COMPLEX): COMPLEX;
    (* Returns the complex conjugate of z *)
BEGIN
    RETURN CMPLX ( RE (z), -IM (z) );
END conj;

PROCEDURE power (base: COMPLEX; exponent: REAL): COMPLEX;
    (* Returns the value of the number base raised to the power exponent *)
VAR
    pow : REAL;
    expTheta, temp : COMPLEX;
    tempRe, tempIm : REAL;
BEGIN
    expTheta := CMPLX (exponent * arg (base), 0.0);
    temp := cos (expTheta) + i * sin (expTheta);
    pow := RealMath.power ( abs (base), exponent );
    tempRe := pow * RE (temp);
    tempIm := pow * IM (temp);
    RETURN CMPLX (tempRe, tempIm);
END power;

PROCEDURE sqrt (z: COMPLEX): COMPLEX;
    (* Returns the principal square root of z *)
VAR
    temp : COMPLEX;
    tempRe, tempIm : REAL;
BEGIN
    tempRe := RealMath.cos ( arg (z) / 2.0 );
    tempIm := RealMath.sin ( arg (z) / 2.0 );
    tempRe := tempRe * RealMath.sqrt ( abs (z) );
    tempIm := tempIm * RealMath.sqrt ( abs (z) );
    RETURN CMPLX ( tempRe, tempIm );
END sqrt;

PROCEDURE exp (z: COMPLEX): COMPLEX;
    (* Returns the complex exponential of z *)
VAR
    temp, outZ : COMPLEX;
    expR,

```

```

    re, im : REAL;
BEGIN
    re := RealMath.cos ( IM (z) );
    im := RealMath.sin ( IM (z) );
    expR := RealMath.exp ( RE (z));
    re := re * expR;
    im := im * expR;
    outZ := CMPLX (re, im);
    RETURN outZ;
END exp;

PROCEDURE ln (z: COMPLEX): COMPLEX;
    (* Returns the principal value of the natural logarithm of z *)
VAR
    tempRe, tempIm : REAL;
BEGIN
    tempRe := RealMath.ln ( abs (z) );
    tempIm := arg (z);
    RETURN CMPLX ( tempRe, tempIm );
END ln;

PROCEDURE sin (z: COMPLEX): COMPLEX;
    (* Returns the sine of z *)
BEGIN
    RETURN ( exp (i * z) - exp (-i * z) ) / CMPLX (0.0, 2.0);
END sin;

PROCEDURE cos (z: COMPLEX): COMPLEX;
    (* Returns the cosine of z *)
BEGIN
    RETURN ( exp (i * z) + exp (-i * z) ) / CMPLX (2.0, 0.0);
END cos;

PROCEDURE tan (z: COMPLEX): COMPLEX;
    (* Returns the tangent of z *)
BEGIN
    IF ( RE (z) = (RealMath.pi / 2.0) + RealMath.pi ) AND ( IM (z) = 0.0 )
        THEN
            RAISE (mathExceptionSource, 0, "ComplexMath.tan: overflow");
        END; (* if *)
    RETURN sin (z) / cos (z);
END tan;

PROCEDURE arcsin (z: COMPLEX): COMPLEX;
    (* Returns the arcsine of z *)
BEGIN
    RETURN ( one / i ) * ln (i * z + sqrt (one - z * z));
END arcsin;

PROCEDURE arccos (z: COMPLEX): COMPLEX;
    (* Returns the arccosine of z *)
BEGIN
    RETURN (one / i) * ln (z + sqrt (z * z - one));
END arccos;

```

```

PROCEDURE arctan (z: COMPLEX): COMPLEX;
  (* Returns the arctangent of z *)
BEGIN
  RETURN (one / (twoi)) * ln ((one + i*z) / (one - i*z));
END arctan;

PROCEDURE polarToComplex (abs, arg: REAL): COMPLEX;
  (* Returns the complex number with the specified polar coordinates *)
VAR
  tempRe, tempIm : REAL;
BEGIN
  tempRe := abs * RealMath.cos (arg);
  tempIm := abs * RealMath.sin (arg);
  RETURN CMPLX (tempRe, tempIm);
END polarToComplex;

PROCEDURE scalarMult (scalar: REAL; z: COMPLEX): COMPLEX;
  (* Returns the scalar product of scalar with z *)
VAR
  tempRe, tempIm : REAL;
BEGIN
  tempRe := RE (z) * scalar;
  tempIm := IM (z) * scalar;
  RETURN CMPLX (tempRe, tempIm);
END scalarMult;

PROCEDURE IsCMathException (): BOOLEAN;
  (* Returns TRUE if the current coroutine is in the exceptional execution state
  because of the raising of an exception in a routine from this module; otherwise
  returns FALSE. *)
BEGIN
  RETURN IsExceptionalExecution () AND IsCurrentSource (mathExceptionSource);
END IsCMathException;

BEGIN (* main *)
  AllocateSource (mathExceptionSource);
END ComplexMath.

```

The nearly identical *LongComplexMath* need not be reproduced here.

17.3.3 Input and Output

The ISO standard does not define a standard library module for I/O. However, it is easy to write one, based on the procedures in RealIO and LongIO and the simple procedure employed in the examples of the last section. As for the standard I/O suite, define libraries for both specified channels and for standard channels.

```

DEFINITION MODULE ComplexIO;

(* =====
   © 1996 by R. Sutcliffe
   Last modification date 1996 10 30
   ===== *)

```

```
(* Input and output of complex numbers in decimal text form over specified channels. The read result is of the type IOConsts.ReadResults. *)
```

```
IMPORT IOChan;
```

```
(* The text form of a complex number is  
    realNumber, [space], ["+" | "-"], [space,] [realnumber, i] |  
    [realNumber, [space],] ["+" | "-"], [space,] realnumber, i  
    where the real numbers in each case are in the  
    format specified for fixed or floating reals.  
*)
```

```
PROCEDURE ReadComplex (cid: IOChan.ChanId; VAR complex: COMPLEX);
```

```
(* Skips leading spaces, and removes any remaining characters from cid that form  
part of a complex number. The value of this number is assigned to complex. The read  
result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or  
endOfInput. *)
```

```
(* following procedure affects all the Write procs below *)
```

```
PROCEDURE SetVerbose (verbose : BOOLEAN);
```

```
(* if true prints both components even if one is zero; else prints only one if the  
other is zero. The default is false. *)
```

```
PROCEDURE WriteFloat (cid: IOChan.ChanId; complex: COMPLEX; sigFigs: CARDINAL; width:  
CARDINAL);
```

```
(* Writes the value of complex to cid in floating-point real text form, with  
sigFigs significant figures, in a field of the given minimum width. The width for the  
real parts is 0 if the supplied width is 3 or less, and it is (width - 4) DIV 2  
otherwise. *)
```

```
PROCEDURE WriteEng (cid: IOChan.ChanId; complex: COMPLEX; sigFigs: CARDINAL; width:  
CARDINAL);
```

```
(* As for WriteFloat, except that the number is scaled with one to three  
digits in the whole number part, and with an exponent that is a multiple of three. *)
```

```
PROCEDURE WriteFixed (cid: IOChan.ChanId; complex: COMPLEX; place: INTEGER; width:  
CARDINAL);
```

```
(* Writes the value of complex to cid in fixed-point text form, with real parts  
rounded to the given place relative to the decimal point, in a field of the given  
minimum width. *)
```

```
PROCEDURE WriteComplex (cid: IOChan.ChanId; complex: COMPLEX; width: CARDINAL);
```

```
(* Writes the value of complex to cid, as WriteFixed if the sign and magnitude can  
be shown in the given width, or otherwise as WriteFloat. The number of places or  
significant digits depends on the given width. *)
```

```
END ComplexIO.
```

The procedure *SetVerbose* changes an internal setting to allow client modules to specify whether complex numbers in which one component is zero should have both components written out or only one. To keep things simple, the default is that *verbose* is *FALSE*, and the option of changing it has been eliminated from the *SComplexIO* module, so that if verbose output is required, the user must import from the *ComplexIO* module. Here, the corresponding module for standard channels is abbreviated.

DEFINITION MODULE SComplexIO;

```
(* =====
      © 1996 by R. Sutcliffe
      Last modification date 1996 10 30
=====*)
```

(* Input and output of complex numbers in decimal text form over the default channels. The read result is of the type IOConsts.ReadResults. *)

IMPORT IOChan;

```
PROCEDURE ReadComplex (VAR complex: COMPLEX);
PROCEDURE WriteFloat (complex: COMPLEX; sigFigs: CARDINAL; width: CARDINAL);
PROCEDURE WriteEng (complex: COMPLEX; sigFigs: CARDINAL; width: CARDINAL);
PROCEDURE WriteFixed (complex: COMPLEX; place: INTEGER; width: CARDINAL);
PROCEDURE WriteComplex (complex: COMPLEX; width: CARDINAL);
END SComplexIO.
```

The only tricky part of the implementation is the procedure *ReadComplex*. This was done in such a way that complex numbers such as $2-i$, $3+5i$, $-7i$, and 56.7 would all be read correctly. That is, reading will work whether there are spaces or not between the parts and whether both parts are present or not. Observe that this means one cannot rely on *ReadReal* to read the sign of the part; that has to be done before handing off control to *ReadReal*. Likewise, the presence of a bare i has to be checked for before looking for a real number.

IMPLEMENTATION MODULE ComplexIO;

```
(* =====
      © 1996 by R. Sutcliffe
      Last modification date 1996 10 30
=====*)
```

(* Input and output of complex numbers in decimal text form over specified channels. The read result is of the type IOConsts.ReadResults. *)

```
IMPORT IOChan, RealIO, IOResult;
FROM TextIO IMPORT
    WriteString, WriteChar, ReadChar;
FROM IOResult IMPORT
    ReadResults;
```

```
VAR
    gverbose : BOOLEAN; (* affects all writing procs *)
```

```
PROCEDURE ReadComplex (cid: IOChan.ChanId; VAR complex: COMPLEX);
    (* Skips leading spaces, and removes any remaining characters from cid that form
    part of a complex number. The value of this number is assigned to complex. The read
    result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or
    endOfInput. *)
```

```
VAR
    re, im : REAL;
    ch: CHAR;
```

```
neg : BOOLEAN;  
res : IOResult.ReadResults;
```

BEGIN

```
IOChan.Look (cid, ch, res);
```

```
IF res = allRight
```

```
  THEN
```

```
    neg := (ch = "-");
```

```
    IF (ch = "+") OR (ch = "-")
```

```
      THEN (* go around leading sign *)
```

```
        IOChan.SkipLook (cid, ch, res);
```

```
      END;
```

```
    IF (ch = "i") OR (ch = "I") (* case of bare i *)
```

```
      THEN
```

```
        re := 1.0
```

```
      ELSE (* not a bare i so get number *)
```

```
        RealIO.ReadReal (cid, re); (* also kills any leading spaces *)
```

```
        IF IOResult.ReadResult (cid) = allRight
```

```
          THEN
```

```
            (* skip any more space there may be before next token *)
```

```
            IOChan.Look (cid, ch, res);
```

```
            WHILE (ch = " ") AND (res = allRight)
```

```
              DO
```

```
                IOChan.SkipLook (cid, ch, res);
```

```
              END; (* while *)
```

```
            END (* if IOResult *)
```

```
          END; (* if ch *)
```

```
    IF neg
```

```
      THEN
```

```
        re := - re;
```

```
      END; (* if neg *)
```

```
    (* now, we could be holding an i here *)
```

```
    IF (ch = "i") OR (ch = "I")
```

```
      THEN (* yes, so pure imaginary *)
```

```
        IOChan.Skip (cid); (* skip that char *)
```

```
        complex := CMPLX (0.0, re);
```

```
        RETURN; (* and done *)
```

```
      ELSIF (ch # "+") AND (ch # "-") THEN
```

```
        (* this means pure real, no imaginary part; so leave that char for next guy
```

```
*)
```

```
        (* set it up as pure real *)
```

```
        complex := CMPLX (re, 0.0);
```

```
        RETURN; (* and done *)
```

```
      END; (* if ch *)
```

```
    (* ok, we get here only if we got the first real ok and a + or a - afterwards
```

```
*)
```

```
    neg := (ch = "-");
```

```
    IOChan.SkipLook (cid, ch, res); (* skip that sign char *)
```

```
    WHILE (ch = " ") AND (res = allRight) (* skip and spaces here *)
```

```
      DO
```

```
        IOChan.SkipLook (cid, ch, res);
```

```
      END; (* while *)
```



```

(* now, they might have a + in the middle and then a sign on the real.
This is a bit dumb, but logical in a way. *)
IOChan.Look (cid, ch, res); (* there had better be something there *)
IF res = allRight
  THEN (* adjust neg if needed *)
    neg := (neg OR (ch = "-")) AND NOT (neg AND (ch = "-"));
    IF (ch = "+") OR (ch = "-")
      THEN (* go around this leading sign too *)
        IOChan.SkipLook (cid, ch, res);
      END;
    IF (ch = "i") OR (ch = "I") (* check case of bare i *)
      THEN
        im := 1.0
      ELSE (* not a bare i so get number *)
        RealIO.ReadReal (cid, im);
        IF IOResult.ReadResult (cid) = allRight (* got second real *)
          THEN
            (* must be followed by an i or bad format *)
            ReadChar (cid, ch);
            IF (ch # "i") AND (ch # "I")
              THEN
                IOChan.SetReadResult (cid, wrongFormat);
                RETURN;
              END; (* if ch *)
            END; (* if IOResult *)
          END; (* if ch *)
        IF neg
          THEN
            im := - im;
          END; (* if neg *)
        (* we only get here if all went OK for two numbers read *)
        complex := CMPLX (re, im);
      ELSE (* nothing there at all *)
        IOChan.SetReadResult (cid, wrongFormat);
      END; (* if res *)
    ELSE (* bad second number *)
      IOChan.SetReadResult (cid, wrongFormat);
    END; (* first if res *)

```

```

END ReadComplex;

```

```

(* following procedure affects all the Write procs below *)

```

```

PROCEDURE SetVerbose (verbose : BOOLEAN);

```

```

  (* if true prints both components even if one is zero; else prints only one if the
  other is zero. The default is false. *)

```

```

BEGIN

```

```

  gverbose := verbose;

```

```

END SetVerbose;

```

```

PROCEDURE WriteFloat (cid: IOChan.ChanId; complex: COMPLEX; sigFigs: CARDINAL; width:
CARDINAL);

```

```

  (* Writes the value of complex to cid in floating-point real text form, with

```

sigFigs significant figures, in a field of the given minimum width. The width for the real parts is 0 if the supplied width is 3 or less, and it is (width - 4) **DIV** 2 otherwise. *)

VAR

realFlen : **CARDINAL**;
re, im : **REAL**;

BEGIN

re := **RE** (complex);
im := **IM** (complex);
IF width < 4
 THEN
 realFlen := 0;
 ELSE
 realFlen := (width - 4) **DIV** 2;
 END;
IF gverbose **OR** (re # 0.0) **OR** (im = 0.0)
 THEN
 RealIO.WriteFloat (cid, re, sigFigs, realFlen);
 END;
IF gverbose **OR** (im # 0.0) (* decide if to print more *)
 THEN (* yes, see if space needed *)
 IF gverbose **OR** (re # 0.0)
 THEN
 WriteChar (cid, " ");
 END;
 IF im < 0.0
 THEN (* always print negative sign *)
 WriteChar (cid, "-");
 ELSIF gverbose **OR** (re # 0.0) **THEN**
 WriteChar (cid, '+'); (* space only *)
 END;
 IF gverbose **OR** (re # 0.0)
 THEN
 WriteChar (cid, " ");
 END;
 RealIO.WriteFloat (cid, **ABS** (im), sigFigs, realFlen);
 WriteChar (cid, 'i');
 END (* first if gverbose *)
END WriteFloat;

PROCEDURE WriteEng (cid: IOChan.ChanId; complex: **COMPLEX**; sigFigs: **CARDINAL**; width: **CARDINAL**);

(* As for WriteFloat, except that the number is scaled with one to three digits in the whole number part, and with an exponent that is a multiple of three. *)

VAR

realFlen : **CARDINAL**;
re, im : **REAL**;

BEGIN

re := **RE** (complex);
im := **IM** (complex);

```

IF width < 4
  THEN
    realFlen := 0;
  ELSE
    realFlen := (width - 4) DIV 2;
  END;
IF gverbose OR (re # 0.0) OR (im = 0.0)
  THEN
    RealIO.WriteEng (cid, re, sigFigs, realFlen);
  END;
IF gverbose OR (im # 0.0) (* decide if to print more *)
  THEN (* yes, see if space needed *)
    IF gverbose OR (re # 0.0)
      THEN
        WriteChar (cid, " ");
      END;
    IF im < 0.0
      THEN (* always print negative sign *)
        WriteChar (cid, "-");
      ELSIF gverbose OR (re # 0.0) THEN
        WriteChar (cid, '+'); (* space only *)
      END;
    IF gverbose OR (re # 0.0)
      THEN
        WriteChar (cid, " ");
      END;
    RealIO.WriteEng (cid, ABS (im), sigFigs, realFlen);
    WriteChar (cid, 'i');
  END (* first if gverbose *)

```

```

END WriteEng;

```

```

PROCEDURE WriteFixed (cid: IOChan.ChanId; complex: COMPLEX; place: INTEGER; width:
CARDINAL);
  (* Writes the value of complex to cid in fixed-point text form, with real parts
rounded to the given place relative to the decimal point, in a field of the given
minimum width. *)

```

```

VAR

```

```

  realFlen : CARDINAL;
  re, im : REAL;

```

```

BEGIN

```

```

  re := RE (complex);
  im := IM (complex);
  IF width < 4
    THEN
      realFlen := 0;
    ELSE
      realFlen := (width - 4) DIV 2;
    END;
  IF gverbose OR (re # 0.0) OR (im = 0.0)
    THEN

```

```

    RealIO.WriteFixed (cid, re, place, realFlen);
END;
IF gverbose OR (im # 0.0) (* decide if to print more *)
THEN (* yes, see if space needed *)
    IF gverbose OR (re # 0.0)
        THEN
            WriteChar (cid, " ");
        END;
    IF im < 0.0
        THEN (* always print negative sign *)
            WriteChar (cid, "-");
        ELSIF gverbose OR (re # 0.0) THEN
            WriteChar (cid, '+'); (* space only *)
        END;
    IF gverbose OR (re # 0.0)
        THEN
            WriteChar (cid, " ");
        END;
    RealIO.WriteFixed (cid, ABS (im), place, realFlen);
    WriteChar (cid, 'i');
END (* first if gverbose *)
END WriteFixed;

PROCEDURE WriteComplex (cid: IOChan.ChanId; complex: COMPLEX; width: CARDINAL);
    (* Writes the value of complex to cid, as WriteFixed if the sign and magnitude can
    be shown in the given width, or otherwise as WriteFloat. The number of places or
    significant digits depends on the given width. *)
    (* Of the space provided in width, four places are needed to write the + between
    the two reals, and one to write the letter i. The rest is divided equally between
    the two reals. *)

VAR
    realFlen : CARDINAL;
    re, im : REAL;

BEGIN
    re := RE (complex);
    im := IM (complex);
    IF width < 4
        THEN
            realFlen := 0;
        ELSE
            realFlen := (width - 4) DIV 2;
        END;
    IF gverbose OR (re # 0.0) OR (im = 0.0)
        THEN
            RealIO.WriteReal (cid, re, realFlen);
        END;
    IF gverbose OR (im # 0.0) (* decide if to print more *)
        THEN (* yes, see if space needed *)
            IF gverbose OR (re # 0.0)
                THEN
                    WriteChar (cid, " ");

```

```

        END;
    IF im < 0.0
    THEN (* always print negative sign *)
        WriteChar (cid, "-");
    ELSIF gverbose OR (re # 0.0) THEN
        WriteChar (cid, '+'); (* space only *)
    END;
    IF gverbose OR (re # 0.0)
    THEN
        WriteChar (cid, " ");
    END;
    RealIO.WriteReal (cid, ABS (im), realFlen);
    WriteChar (cid, 'i');
    END (* first if gverbose *)
END WriteComplex;

BEGIN (* main body *)
    gverbose := FALSE;
END ComplexIO.

```

Once this is done, the implementation of the *SComplexIO* module is quite straightforward.

```

IMPLEMENTATION MODULE SComplexIO;

(* =====
   © 1996 by R. Sutcliffe
   Last modification date 1996 10 30
   =====*)

IMPORT StdChans, ComplexIO;

PROCEDURE ReadComplex (VAR complex: COMPLEX);
BEGIN
    ComplexIO.ReadComplex (StdChans.InChan(), complex);
END ReadComplex;

PROCEDURE WriteFloat (complex: COMPLEX; sigFigs: CARDINAL; width: CARDINAL);
BEGIN
    ComplexIO.WriteFloat (StdChans.OutChan(), complex, sigFigs, width);
END WriteFloat;

PROCEDURE WriteEng (complex: COMPLEX; sigFigs: CARDINAL; width: CARDINAL);
BEGIN
    ComplexIO.WriteEng (StdChans.OutChan(), complex, sigFigs, width);
END WriteEng;

PROCEDURE WriteFixed (complex: COMPLEX; place: INTEGER; width: CARDINAL);
BEGIN
    ComplexIO.WriteFixed (StdChans.OutChan(), complex, place, width);
END WriteFixed;

PROCEDURE WriteComplex (complex: COMPLEX; width: CARDINAL);
BEGIN
    ComplexIO.WriteComplex (StdChans.OutChan(), complex, width);

```

```
END WriteComplex;
```

```
END SComplexIO.
```

From these, it is also rather a simple matter to produce modules called *LongComplexIO* and *SLongComplexIO*. These will not be reproduced here. A small test harness illustrates some of the combinations. For the first group of numbers, *verbose* is turned on; for the second group, only one form of output is used, and *verbose* is off.

```
MODULE TestComplexIO;
```

```
(* by R. Sutcliffe  
   to test ComplexIO modified 1996 11 01 *)
```

```
FROM SComplexIO IMPORT
```

```
    ReadComplex, WriteComplex, WriteFixed, WriteFloat, WriteEng;
```

```
FROM ComplexIO IMPORT
```

```
    SetVerbose;
```

```
FROM ComplexMath IMPORT
```

```
    zero;
```

```
FROM STextIO IMPORT
```

```
    WriteString, WriteLn, SkipLine;
```

```
VAR
```

```
    z : COMPLEX;
```

```
BEGIN
```

```
    SetVerbose (TRUE);
```

```
    REPEAT
```

```
        WriteString ("Type complex; zero to go on  ");
```

```
        ReadComplex (z);
```

```
        SkipLine;
```

```
        WriteString ("complex z is: ");
```

```
        WriteLn;
```

```
        WriteFixed (z, 6, 26);
```

```
        WriteLn;
```

```
        WriteFloat (z, 10, 26);
```

```
        WriteLn;
```

```
        WriteEng (z, 10, 26);
```

```
        WriteLn;
```

```
        WriteComplex (z, 26);
```

```
        WriteLn;
```

```
        WriteLn;
```

```
    UNTIL z = zero;
```

```
    SetVerbose (FALSE);
```

```
    REPEAT
```

```
        WriteString ("Type complex  ");
```

```
        ReadComplex (z);
```

```
        SkipLine;
```

```
        WriteComplex (z, 26);
```

```
        WriteLn;
```

```
    UNTIL z = zero;
```

```
END TestComplexIO.
```

Observe the variety of inputs and outputs used to test the library. As usual,, inputs are highlighted in bold to distinguish them.

```
Type complex; zero to go on 1
complex z is:
    1.000000 +    0.00000i
1.0000000000E+00 + 0.0000000000E+00i
1.0000000000E+00 + 0.0000000000E+00i
1.0000000000 + 0.00000E+00i
```

```
Type complex; zero to go on 3i
complex z is:
    0.000000 +    3.000000i
0.0000000000E+00 + 3.0000000000E+00i
0.0000000000E+00 + 3.0000000000E+00i
0.00000E+00 + 3.0000000000i
```

```
Type complex; zero to go on 2-5i
complex z is:
    2.000000 -    5.000000i
2.0000000000E+00 - 5.0000000000E+00i
2.0000000000E+00 - 5.0000000000E+00i
2.0000000000 - 5.0000000000i
```

```
Type complex; zero to go on 0
complex z is:
    0.000000 +    0.00000i
0.0000000000E+00 + 0.0000000000E+00i
0.0000000000E+00 + 0.0000000000E+00i
0.00000E+00 + 0.00000E+00i
```

```
Type complex 1
1.0000000000
Type complex 2i
2.0000000000i
Type complex 3-i
3.0000000000 - 1.0000000000i
Type complex 7 +5i
7.0000000000 + 5.0000000000i
Type complex -i
-1.0000000000i
Type complex 0
0.00000E+00
```

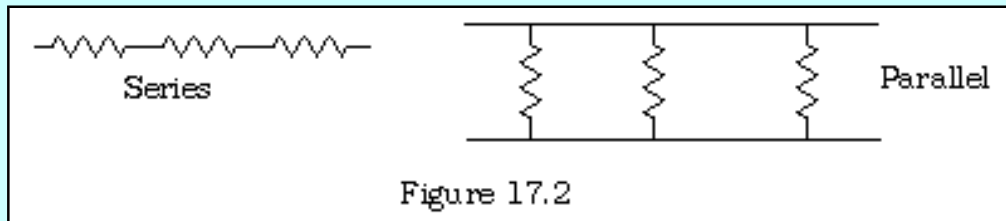
17.4 Electrical Circuits and Complex Numbers

In a direct current circuit, the Voltage (E) current (I) and resistance (R) are related by a simple formula called Ohms Law:

$$E = IR$$

where E is measured in Volts, I in Amperes, and R in ohms. Computations involving the three quantities are straightforward.

It turns out that resistances in series are additive, $R = R_1 + R_2 + R_3 + \dots$



whereas those in parallel add in inverses $1/R = 1/R_1 + 1/R_2 + 1/R_3 + \dots$

When one considers alternating current, the situation becomes more complex. Here, the voltage and current are not constant, but expressed as a sinusoidal wave over time.

$$E(t) = E_0 \cos(\omega t + \theta)$$

and likewise for I

$$I(t) = I_0 \cos(\omega t + \phi).$$

Using Euler's law, ($e^{i\theta} = \cos\theta + i \sin\theta$) these can be expressed as $E(t) = E_0 e^{i(\omega t + \theta)}$ and $I(t) = I_0 e^{i(\omega t + \phi)}$ where i is, as usual, .

The time dependent resistance of the circuit, (called the impedance and denoted Z), is affected by resistance and by two other types of components-capacitors and inductance coils. Denoting the three by Z_R , Z_L , and Z_C for impedance due to resistors, capacitors, and impedance respectively, it can be shown that:

$$Z_R = R$$

$$Z_L = i\omega L \text{ where } L \text{ is the inductance in Henries}$$

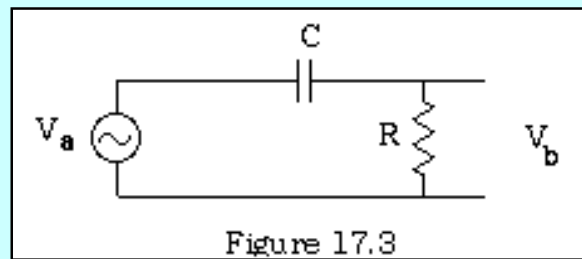
$$Z_C = \frac{1}{i\omega C} = \frac{-1}{\omega C} \text{ where } C \text{ is the capacitance in Farads.}$$

These impedances add in series and parallel in the same way as do resistances.

In general, such quantities have both a real part (still called the resistance) and an imaginary part (called the reactance).

Of course, the magnitude $|Z|$ of an impedance $Z = Z_0 e^{i\theta}$ is $\text{abs}(Z)$ and the phase angle θ is $\arg(Z)$. The effect of an RCL circuit is to filter the current and pass an altered current whose amplitude is reduced by a factor of $|Z|$ compared to the original and whose phase is shifted by a quantity equal to the phase angle of Z.

An interesting property of circuits that have combinations of these three elements is called the transfer function. It is the ratio of the output voltage to the input voltage. Consider the RC circuit in figure 17.3 for instance.



Considering the ratio of the output voltage V_b to the input voltage V_a , and denoting the voltage drop at the capacitor by V_c , one has for this circuit the transfer function \mathbf{H} given by:

$$H(R, C, \omega) = \frac{V_b}{V_c} = \frac{V_b}{V_b - V_a} = \frac{IR}{IR + \frac{1}{\omega C}} = \frac{Ri_{\omega} C}{IR + Ri_{\omega} C}$$

The magnitude of this result is the magnitude of the resulting voltage function, and the phase angle of the result is the shift in phase from the input voltage, which for purposes of making the calculation simple is assumed not to have been shifted at the start. The units of the frequency ω are radians per second. This particular computation has been encapsulated as follows:

```

DEFINITION MODULE TransferFunctions;

(* This module contains one or more procedures to compute the
transfer function or ratio of output voltage to input voltage
as a complex quantity. The magnitude of the result is the amplitude of the
resulting sinusoidal wave, and the phase angle is the phase shift
from the starting phase angle. Each procedure takes parameters
for the resistance, capacitance, and/or inductance of the circuit,
and frequency of the input wave. *)

PROCEDURE RCTransfer (res, cap, freq : REAL) : COMPLEX;
(* computes the transfer function for the RC circuit
-----||-----
|
|
~ input      R      output
|
|
|
-----
*)

END TransferFunctions.

```

```

IMPLEMENTATION MODULE TransferFunctions;

FROM ComplexMath IMPORT
    one, i;

PROCEDURE RCTransfer (res, cap, freq : REAL) : COMPLEX;

VAR
    temp: REAL;

```

```

BEGIN
    temp := freq * res * cap;
    RETURN (CMPLX (temp, 0.0) * i) / (one + CMPLX (temp, 0.0) * i);
END RCTransfer;

END TransferFunctions.

```

The imports *one* and *i* are the complex quantities $1 + 0i$, and $0 + i$, respectively. Here, the built in function **CMPLX** is employed to convert a pair of reals (a, b) to the complex quantity $a + bi$. One may also begin with a complex number and fetch the real and imaginary parts using:

```

re := RE (z);
im := IM (z);

```

where re and im are both of type **REAL** and z is of type **COMPLEX**. **RE** and **IM** are standard identifiers. What follows is a simple module to test this transfer function. It calls for the resistance and capacitance of the RC circuit and a range of frequencies over which to compute the transfer function, then translates this into an amplitude and phase shift for the output wave and prints a little table.

```

MODULE TestTransferFunctions;
(* Module to test transfer functions and
   illustrate the use of complex numbers
   by R. Sutcliffe
   modified 1996 01 12  *)

FROM STextIO IMPORT
    WriteString, WriteLn, SkipLine;

FROM SRealIO IMPORT
    ReadReal, WriteFixed;

FROM ComplexMath IMPORT
    zero, abs, arg;

FROM TransferFunctions IMPORT
    RCTransfer;

VAR
    count : CARDINAL;
    resistance, capacitance,
    frequency, startFreq, endFreq, amplitude, phaseShift :REAL;
    transFactor : COMPLEX;

CONST
    step = 10.0;

BEGIN

```

```

WriteString ("This program determines the magnitude and phase effects ");
WriteLn;
WriteString ("of an RC circuit over an interval of frequencies ");
WriteLn;
WriteString ("entered by the user.");
WriteLn;WriteLn;
WriteString ("Enter the resistance in ohms ");
ReadReal (resistance);
SkipLine;
WriteString ("Enter the capacitance in farads ");
ReadReal (capacitance);
SkipLine;
WriteString ("Enter the starting frequency in radians per second ");
ReadReal (startFreq);
SkipLine;
WriteString ("Enter the ending frequency in radians per second ");
ReadReal (endFreq);
SkipLine;

frequency := startFreq;
WriteString ("Frequency      Magnitude      Phase Shift");
WriteLn;
WHILE frequency <= endFreq
  DO
    transFactor := RCTransfer (resistance, capacitance, frequency);
    amplitude := abs (transFactor);
    IF transFactor = zero (* can't feed to arg or get exception *)
      THEN
        phaseShift := 0.0
      ELSE
        phaseShift := arg (transFactor);
      END;
    WriteFixed (frequency, 2, 10);
    WriteFixed (amplitude, 2, 10);
    WriteFixed (phaseShift, 2, 10);
    WriteLn;
    frequency := frequency + step
  END;
WriteLn;
WriteString ("Press a key to continue");
SkipLine;
END TestTransferFunctions.

```

Like the constants *zero*, *one*, and *i*, the functions *abs* and *arg* are imported from the separate module *ComplexMath*. *abs* returns the magnitude of the complex number, and *arg* returns the angle it makes with the real axis. Runs of this test are printed below:

**** Run log starts here ****

This program determines the magnitude and phase effects of an RC circuit over an interval of frequencies entered by the user.

Enter the resistance in ohms 10000

Enter the capacitance in farads .000002

Enter the starting frequency in radians per second 0

Enter the ending frequency in radians per second 200

Frequency	Magnitude	Phase Shift
-----------	-----------	-------------

0.0	0.0	0.0
10.00	0.20	1.37
20.00	0.37	1.19
30.00	0.51	1.03
40.00	0.62	0.90
50.00	0.71	0.79
60.00	0.77	0.69
70.00	0.81	0.62
80.00	0.85	0.56
90.00	0.87	0.51
100.00	0.89	0.46
110.00	0.91	0.43
120.00	0.92	0.39
130.00	0.93	0.37
140.00	0.94	0.34
150.00	0.95	0.32
160.00	0.95	0.30
170.00	0.96	0.29
180.00	0.96	0.27
190.00	0.97	0.26
200.00	0.97	0.24

Press a key to continue

**** Run log starts here ****

This program determines the magnitude and phase effects of an RC circuit over an interval of frequencies entered by the user.

Enter the resistance in ohms 100

Enter the capacitance in farads .000002

Enter the starting frequency in radians per second 0

Enter the ending frequency in radians per second 200

Frequency	Magnitude	Phase Shift
-----------	-----------	-------------

0.0	0.0	0.0
10.00	0.00	1.57

20.00	0.00	1.57
30.00	0.01	1.56
40.00	0.01	1.56
50.00	0.01	1.56
60.00	0.01	1.56
70.00	0.01	1.56
80.00	0.02	1.55
90.00	0.02	1.55
100.00	0.02	1.55
110.00	0.02	1.55
120.00	0.02	1.55
130.00	0.03	1.54
140.00	0.03	1.54
150.00	0.03	1.54
160.00	0.03	1.54
170.00	0.03	1.54
180.00	0.04	1.53
190.00	0.04	1.53
200.00	0.04	1.53

Press a key to continue

**** Run log starts here ****

This program determines the magnitude and phase effects
of an RC circuit over an interval of frequencies
entered by the user.

Enter the resistance in ohms 30000

Enter the capacitance in farads .000002

Enter the starting frequency in radians per second 0

Enter the ending frequency in radians per second 200

Frequency	Magnitude	Phase Shift
-----------	-----------	-------------

0.0	0.0	0.0
10.00	0.51	1.03
20.00	0.77	0.69
30.00	0.87	0.51
40.00	0.92	0.39
50.00	0.95	0.32
60.00	0.96	0.27
70.00	0.97	0.23
80.00	0.98	0.21
90.00	0.98	0.18
100.00	0.99	0.17
110.00	0.99	0.15
120.00	0.99	0.14
130.00	0.99	0.13
140.00	0.99	0.12

150.00	0.99	0.11
160.00	0.99	0.10
170.00	1.00	0.10
180.00	1.00	0.09
190.00	1.00	0.09
200.00	1.00	0.08

Press a key to continue

The first run ($R = 10000\Omega$ and $C = .000002F$) has low frequency currents scarcely making it through the circuit, whereas the higher frequency one are passed almost without distortion. When the resistance is lowered, the lower frequencies are almost entirely cut off, and the phase shift is large. When R is increased to 30000Ω , the effect is present only for the very lowest frequencies. Such circuits (usually with several more elements) are often called *filters* because of these effects. Numerous other combinations of the resistance, inductance and capacitance are possible in such circuits, and each has its own transfer function.

[Contents](#)

17.5 Very Long Cardinals--The Type Decimal

One of the problems we have when manipulating the type REAL is that numbers stored in this form are subject to being rounded off. In working with dollar amounts, rounding off errors are simply not acceptable, so many computer languages or operating environments provide some means for expressing dollar figures as sequences of digits.

Numbers that are stored as sequences of exact digits are said to be of type Decimal.

If this is done, and we have $472 + 231$, a special procedure is required as in Section 7.6 to add the digits one column at a time starting from the right-hand-side and get the sequence 703.

Subtraction and multiplication present their own challenges, as does printing such quantities out, for one may store them initially as strings of digits, but when it comes to printing them, one would probably want to enter those digits into a format defined in another string and output "\$7.03". (In this case, the format string used is "\$!##" where each # is a digit and the ! indicates the location of the decimal point in the result. See the examples later for details.

A model string used to specify a format for Decimal data I/O is called a format string or a picture or a mask. The purpose of a picture is to indicate where the punctuation marks are in the output string.

Various notations have different provisions for handling such types of data and for performing these operations. In some versions of Pascal, the predecessor of Modula-2, there is a built-in facility to define long integers of any specified number of digits, merely by stating in brackets after the TYPE definition the number of digits for that Integer type. COBOL, Fortran, and some BASICs have the ability to reformat such numbers into strings which can be written out as dollar amounts, social security numbers, or in any other desired fashion.

As may be guessed because of the treatment of the problem of multiplying such quantities in Chapter 7, no such data type or ability is built-in to Modula-2. Modules to implement such types are provided with some versions of Modula-2 and if such a module is available to the reader, there are some straightforward exercises on its use at the end of this chapter. If it is not, the challenge is to write it, using some of the methods of Chapter 7 and later chapters. Only a portion of that work will be shown here; the rest is left as exercises for the reader.

NOTE: Unlike the situation with complex numbers, there is *no* provision in ISO standard Modula-2 for such a data type. The user who needs such a facility is at the mercy of her own ability to write it or the vendor's to provide it.

The module below is one possibility for a definition of such an ADT. It can be modified for Canadian, American, or European style numeric output by changing the constants *decPoint* and *separator* and by supplying a different currency symbol in the picture string.

```
DEFINITION MODULE Decimals;
(* by R. Sutcliffe
   last modified 1996 11 05 *)
```

```
CONST
  MaxDigits = 19;
  decPoint = '.';
  separator = ' ';
```

```
TYPE
  Digit = [0..9];
  DecRange = [0 .. MaxDigits - 1];
  DecState = (allOK, NegOvfl, PosOvfl, Invalid);
  DecHandler = PROCEDURE (DecState);
  CompareResults = (less, equal, greater);
  Decimal =
```

```
  RECORD
    state : DecState;
    isNeg : BOOLEAN;
```

```

    number : ARRAY DecRange OF Digit;
END;

```

CONST

```

    zero = Decimal {allOK, FALSE, {0 BY MaxDigits}};

```

```

PROCEDURE SetHandler (handler: DecHandler);
PROCEDURE Abs (dec : Decimal): Decimal;
PROCEDURE Add (dec1, dec2: Decimal): Decimal;
PROCEDURE Sub (dec1, dec2: Decimal): Decimal;
PROCEDURE Mul (dec1, dec2: Decimal): Decimal;
PROCEDURE Div (dec1, dec2: Decimal): Decimal;
PROCEDURE Remainder (): Decimal;
PROCEDURE Compare (dec1, dec2: Decimal): CompareResults;
PROCEDURE Neg (dec: Decimal): Decimal;
PROCEDURE Status (dec: Decimal): DecState;

END Decimals.

```

For reasons similar to those given in the initial discussions in the last section, the numeric type is here implemented transparently. An opaque implementation would require regular procedures and variable parameters to return results of numeric operations.

The module *Decimals* exports the type *Decimal*, which can be thought of as an 19-digit long integer. Provision is made to store an error state and a sign for each such entity. Of course, *Decimal* should be treated as an opaque type, as though the details were not available in the definition module. *Decimals* also exports an apparatus for error handling that consists of a type *DecState* that defines the error values, a *Status* enquiry procedure to discover the state of any individual item, and a *Handler* type to define the type of an error handler procedure that a client can attach using *SetHandler*. This handler defaults to a procedure that does nothing at all, but a program can define a procedure taking the *DecState* parameter, and set it as desired. The procedure *Remainder* is intended to fetch the stored remainder of the last division performed.

WARNING: The implementation shown below is minimal and incomplete. In particular there are a minimum of comments. Completing it is the subject of some of the exercises at the end of the chapter.

```

IMPLEMENTATION MODULE Decimals;
(* by R. Sutcliffe
   last modified 1996 11 05 *)

VAR
    remainder : Decimal;
    theHandler : DecHandler;

PROCEDURE DefaultHandler (theError : DecState);
(* does nothing *)
END DefaultHandler;

(* exported procs *)

PROCEDURE SetHandler (handler: DecHandler);
BEGIN
    theHandler := handler;
END SetHandler;

PROCEDURE Abs (dec : Decimal): Decimal;
BEGIN
    dec.isNeg := FALSE;

```



```

    RETURN dec;
END Abs;

PROCEDURE Add (dec1, dec2: Decimal): Decimal;
VAR
    count, temp, carry : CARDINAL;
    result : Decimal;
BEGIN
    result := zero;
    carry := 0;
    (* if both pos or both neg, just add the digits up *)
    IF ((dec1.isNeg) AND (dec2.isNeg)) OR NOT ((dec1.isNeg) OR (dec2.isNeg))
    THEN
        FOR count := 0 TO MaxDigits - 1
            DO
                temp := carry + dec1.number[count] + dec2.number[count];
                result.number[count] := temp MOD 10;
                carry := temp DIV 10;
            END;
        (* attach the common sign *)
        result.isNeg := dec1.isNeg;
        IF carry # 0
        THEN
            IF result.isNeg
            THEN
                result.state := PosOvfl
            ELSE
                result.state := NegOvfl
            END;
        END;
    ELSE (* one is neg, the other pos so find difference *)
        IF Compare (Abs (dec1), Abs (dec2)) = greater
        THEN
            FOR count := 0 TO MaxDigits - 1
                DO
                    DEC (dec1.number[count], carry);
                    IF dec1.number[count] >= dec1.number[count]
                    THEN
                        result.number[count] := dec2.number[count] - dec1.number[count];
                        carry := 0;
                    ELSE
                        result.number[count] := 10 + dec2.number[count] -
dec1.number[count];
                        carry := 1;
                    END;
                END;
            (* attach sign of larger in absolute value *)
            result.isNeg := dec2.isNeg;
        END;
    END;
    (* always call error handler before concluding *)
    theHandler (result.state);
    RETURN result;

```

```

END Add;

PROCEDURE Sub (dec1, dec2: Decimal): Decimal;
BEGIN
    RETURN Add (dec1, Neg (dec2));
END Sub;

PROCEDURE Mul (dec1, dec2: Decimal): Decimal;
(* exercise *)
END Mul;

PROCEDURE Div (dec1, dec2: Decimal): Decimal;
(* exercise *)
END Div;

PROCEDURE Remainder (): Decimal;
BEGIN
    RETURN remainder;
END Remainder;

PROCEDURE Compare (dec1, dec2: Decimal): CompareResults;
VAR
    count : INTEGER;
BEGIN
    count := MaxDigits - 1;
    WHILE (count < 0
        THEN
            RETURN equal
        ELSIF dec1.number[count] < dec2.number[count] THEN
            RETURN less
        ELSE
            RETURN greater;
        END;
END Compare;

PROCEDURE Neg (dec: Decimal): Decimal;
BEGIN
    dec.isNeg := NOT dec.isNeg;
    RETURN dec;
END Neg;

PROCEDURE Status (dec: Decimal): DecState;
BEGIN
    RETURN dec.state;
END Status;

BEGIN
    theHandler := DefaultHandler;
END Decimals.

```

Naturally, there have to be procedures for getting data into and out of the internal form. In this case, these are not located in the ADT definition module, but in two other places. First, one can define fairly straightforward input and output for *Decimal* quantities.

DEFINITION MODULE DecimalIO;

(* by R. Sutcliffe
modified 1996 11 04 *)

IMPORT IOChan;

FROM Decimals **IMPORT**
Decimal;

PROCEDURE ReadDecimal (cid : IOChan.ChanId; **VAR** dec : Decimal);

PROCEDURE WriteDecimal (cid : IOChan.ChanId; dec : Decimal; width : **CARDINAL**);

END DecimalIO.

IMPLEMENTATION MODULE DecimalIO;

(* by R. Sutcliffe
modified 1996 11 04 *)

IMPORT IOChan, TextIO, IOResult;

FROM Decimals **IMPORT**
Decimal, MaxDigits, DecRange, zero, DecState, decPoint;

FROM CharClass **IMPORT**
IsNumeric;

FROM WholeIO **IMPORT**
WriteCard;

FROM IOResult **IMPORT**
ReadResults;

FROM STextIO **IMPORT** WriteChar;

IMPORT SWholeIO;

TYPE

DecString = **ARRAY** DecRange **OF** **CHAR**;

(* exported procs *)

PROCEDURE ReadDecimal (cid : IOChan.ChanId; **VAR** dec : Decimal);

VAR

temp : DecString;
count, len : **CARDINAL**;
ch : **CHAR**;
res : IOResult.ReadResults;

BEGIN

count := 0;
IOChan.Look (cid, ch, res);
IF (res = allRight)
 THEN
 dec := zero; (* initialize it *)
 dec.isNeg := (ch = "-")

END;

IF (ch = "-") **OR** (ch = "+")
 THEN

```

        IOChan.SkipLook (cid, ch, res);
    END;
WHILE (count < MaxDigits) AND (res = allRight)
    DO (* skips over all non numerics *)
        IF (IsNumeric (ch))
            THEN
                temp [count] := ch;
                INC (count);
            END;
        IOChan.SkipLook (cid, ch, res);
    END;
IF (res = allRight) OR (res = endOfLine)
    THEN
        len := count - 1;
        WHILE count > 1)
            THEN
                DEC (width);
            END; (* if dec *)
        IF width = 0
            THEN
                WriteChar (" ");
            ELSIF width > 0) AND (countd < MaxDigits)
        DO
            DEC (counts);
            IF IsNumeric (string[counts])
                THEN
                    temp.number[countd] := ORD (string [counts]) - ORD ("0");
                    INC (countd);
                END;
            END;
        IF string [0] = "-"
            THEN
                temp.isNeg := TRUE;
            END;
        RETURN temp;
END StrToDec;

PROCEDURE DecToStr (dec: Decimal; picture: ARRAY OF CHAR; VAR result: ARRAY OF CHAR);
(* Formats the Decimal according to the picture.  Characters with special meaning
are:
#   a leading blank or a digit
9   a leading zero or a digit
!   the decimal character defined in the module Decimals (commonly "." or ",")
=   the sign (+ or -)
,   the separator defined in the module Decimals (commonly "." or "," or " ")
all other characters in the picture string are entered into the result literally *)
VAR
    counts, countd, countr, maxs, maxr, picDigits, pad : CARDINAL;
    ch : CHAR;
    decDone : BOOLEAN;

BEGIN

```

```

decDone := FALSE;
maxs := LENGTH (picture);
maxr := HIGH (result);
picDigits := 0;
FOR counts := 0 TO maxs - 1
    DO
        ch := picture [counts];
        IF (ch = "#") OR (ch = "9")
            THEN
                INC (picDigits)
            END
        END;
counts := 0;
countd := MaxDigits;
countr := 0;
WHILE (countd > countd
    THEN
        pad := picDigits - countd;
    ELSE
        pad := 0;
    END;
(* special case zero *)
IF countd = 0
    THEN
        INC (countd);
    END;
ch := picture [counts];
WHILE (counts < maxs) AND (countd < maxr)
    DO
        IF (ch = "#") OR (ch = "9")
            THEN
                IF pad = 0
                    THEN
                        DEC (countd);
                        result [countr] := CHR (dec.number[countd] + ORD ("0"));
                    ELSE (* fill in spaces or zeros from # and 9 places not used in dec *)
                        IF (ch = "#")
                            THEN
                                result [countr] := " ";
                            ELSIF (ch = "9") THEN
                                result [countr] := "0";
                            END;
                        DEC (pad);
                    END;
                IF countd < picDigits
                    THEN
                        INC (counts);
                    END;
                ELSIF (ch = "!") THEN
                    result [countr] := decPoint;
                    INC (counts);
                ELSIF (ch = ",") THEN
                    result [countr] := separator;

```

```

        INC (counts);
    ELSIF (ch = "=") THEN
        IF dec.isNeg
            THEN
                result [countr] := "-"
            ELSE
                result [countr] := "+"
            END;
        INC (counts);
    ELSE
        result [countr] := ch;
        INC (counts);
    END;
    INC (countr);
    ch := picture [counts];
END;
WHILE (counts < maxs) AND (countr < maxr)
    DO (* copy any stuff left in picture; must be literals *)
        result [countr] := ch;
        INC (counts);
        INC (countr);
        ch := picture [counts];
    END;
IF (countr < maxr)
    THEN
        result [countr] := 0C;
    END;
END DecToStr;

END DecimalStr.

```

As indicated in the examples already discussed, a program can read *Decimal* quantities in the form of strings (perhaps in picture form), assign them to variables of type *Decimal*, manipulate them, and then print them out using pictures (perhaps of a different form than in the way they were entered). Here is an example:

Similarly, one could use this functionality to save and print Social Security (Insurance) or credit card numbers in a form with spaces or dashes at appropriate places.

[Contents](#)

17.6 Binary Coded Decimal Fixed Point Types

An important variation on the idea of storing cardinals in a digit by digit fashion and then using a picture to print out the numbers with a decimal point is to store the digits along with a decimal point position. In other words, each stored item is thought of as a decimal fraction with a specific number of places after the decimal. Again, the storage representation is exact, so no rounding off errors take place. One could modify the definition of the last section as:

```
Decimal =  
  RECORD  
    state : DecState;  
    isNeg : BOOLEAN;  
    number : ARRAY DecRange OF Digit;  
    decPlace : DecRange;  
  END;
```

The code for arithmetic operations will now be considerably more complicated, because the decimal place has to be taken into consideration each time, and each operand can have it in a different place. The complexity of doing arithmetic also slows things down considerably. Yet such a numeric type is attractive for business arithmetic, and some vendors do wish to include such a facility. Indeed, some hardware has built in support in the machine language to which one is compiling for such a type. Ordinarily, as shown in Chapter 8, each machine nibble can code one hexadecimal digit in the range of 0 - F (hex) or 0 - 15 (decimal). Arithmetic is performed by routines that handle carries from nibble to nibble automatically. Thus, F + E produces 1C. The number of nibbles that make up a numeric type (and for which such carrying is performed) determine how many hexadecimal digits can be stored by that type.

Now, if a nibble can store a hexadecimal digit, it can surely store a decimal digit, and do so with room to spare, because there are only ten possibilities rather than sixteen. Many computers allow for a switch to be set to tell the nibbles to do their carry in a decimal fashion rather than a hexadecimal fashion. Thus, the digits A,B,C,D,E, and F are not used when arithmetic is in this mode, and the digit each nibble stores can be thought of as a decimal digit. True, there is a little waste memory space this way, but this technique does answer the need of being able to store each decimal digit individually and therefore to do arithmetic without round off errors.

The name of any data type stored and manipulated in decimal digits at the binary level is binary coded decimal or BCD for short.

Because putting the machine into BCD arithmetic mode (and changing it back again) requires a machine language instruction to set the mode switch, and because not all machines will support BCD at all, the BCD type is not in Modula-2 *per se*. However, some vendors do add such support to their otherwise standard system. When this is done, client modules will normally have to include

```
FROM SYSTEM IMPORT  
  BCD;
```

before making any use of the type. Because this is a numeric type, if the vendor does go to the trouble to provide it, arithmetic operations should work using the normal operators (overloaded) on items of type BCD from this point. However the details will vary from one vendor to another.

17.6.1 BCD Support in p1 Modula-2 (Optional)

NOTE: Information in this section is implementation-specific to one product. The modules described here may exist in other forms, with other names, or not at all in other Modula-2 systems.

The p1 compiler is an example of a package that does include a BCD type that has to be imported from SYSTEM but for which the arithmetic operators are overloaded. As one might expect from an otherwise ISO standard package, there is a module for doing I/O that has a rather standard-looking interface:

```

DEFINITION MODULE BcdIO;
(* Input and output of bcd numbers in decimal text form *)

IMPORT IOChan;
FROM SYSTEM IMPORT BCD;

(* the text form of a signed bcd number is
   ["+" | "-"], decimal digit, {decimal digit}, [".", {decimal digit}] ["$"] *)

PROCEDURE ReadBcd (cid: IOChan.ChanId; VAR bcd: BCD);
  (* Skips leading spaces and removes any remaining characters that form part of a
   signed bcd number. A corresponding value is assigned to the parameter bcd. The read
   result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or
   endOfInput. *)

PROCEDURE WriteFixed (cid: IOChan.ChanId; bcd: BCD; place: INTEGER; width: CARDINAL);
  (* Writes the value of the parameter bcd in fixed-point text form, rounded to the
   given place relative to the decimal point, in a field of the given minimum width. *)

  (* Examples of fixed point output:
   value:      3923009    3.923009    0.0003923009
   places
   -5          3920000            0            0
   -2          3923010            0            0
   -1          3923009            4            0
   0           3923009.            4.           0.
   1           3923009.0           3.9          0.0
   4           3923009.0000        3.9230        0.0004
   *)

END BcdIO.

```

There is also a module *SBcdIO*. Naturally, there are facilities like those in the last section, though a little more sophisticated, for converting to and from formatted strings.

```

DEFINITION MODULE BcdStr;
(* BCD/string conversions *)

IMPORT ConvTypes, BcdConv;
FROM SYSTEM IMPORT BCD;

TYPE
  ConvResults = ConvTypes.ConvResults; (* strAllRight, strOutOfRange, strWrongFormat,
  strEmpty *)
  BcdFormat = BcdConv.BcdFormat; (* formatOk, decMarker, missingChar, illegalValue *)

(* the string form of a signed bcd number is
   ["+" | "-"], decimal digit, {decimal digit}, [".", {decimal digit}] ["$"] *)

PROCEDURE StrToBcd (str: ARRAY OF CHAR; VAR bcd: BCD; VAR res: ConvResults);

```


(* Ignores any leading spaces in str. If the subsequent characters in str are in the format of a signed bcd number, assigns a corresponding value to bcd. Assigns a value indicating the format of str to res. *)

PROCEDURE BcdToFixed (bcd: **BCD**; place: **INTEGER**; **VAR** str: **ARRAY OF CHAR**);

(* Converts the value of bcd to fixed-point string form, rounded to the given place relative to the decimal point, and copies the possibly truncated result to str. *)

PROCEDURE BcdToFree (bcd: **BCD**; format: **ARRAY OF CHAR**;

VAR str: **ARRAY OF CHAR**; **VAR** formatResult: BcdFormat);

(* If format is a well-format format string and bcd can be represented within this format, converts bcd to the given format, and assigns the possibly truncated result to str, and "formatOk" to formatResult. Otherwise assigns a value indicating the error to formatResult. *)

END BcdStr.

The lower level module that this one imports from defines the format or picture string type and also provides facilities for checking on the validity of such a string before making use of it.

DEFINITION MODULE BcdConv;

IMPORT ConvTypes;

FROM SYSTEM **IMPORT** BCD;

TYPE

ConvResults = ConvTypes.ConvResults;

BcdFormat = (

formatOk, (* correct conversion format *)

decMarker, (* none or more than one marker for the position
of the decimal point specified *)

missingChar, (* missing characters after "=" or "\" *)

illegalValue (* a value that is too great to be represented with this

format *)

);

(*

The format string accepted by the conversion routines has the following rules:

Z : digit, leading " "

: digit, leading " "

9 : digit, leading "0"

= : sign, the two following characters represent "+" and "-"

- : same as "= -"

+ : same as "=+ -"

! : marks the position of the decimal point (no required representation)

, : same as "!\,"

. : same as "!\."

\ : escape character, the follow character is printed without
interpretation

other characters: copied to the result string as they are.

Examples:

format: " ###\.\###\.\###\.\##9,999 =HS"

```
23456.78 is converted to : "    .    . 23.456,780 H"
-0.123 is converted to : "    .    .    0,123 S"
```

```
" +999.99999"
```

```
23456.78 formatResult becomes illegalValue
-0.123 is converted to : "-000.12300"
```

```
"DM - ZZZZZZ,ZZ"
```

```
23456.78 is converted to : "DM    23456,78"
-0.123 is converted to : "DM -    ,12"
```

```
"DM - #####9,99"
```

```
23456.78 is converted to : "DM    23456,78"
-0.123 is converted to : "DM -    0,12"
```

```
*)
```

```
PROCEDURE ScanBcd (inputCh: CHAR; VAR chClass: ConvTypes.ScanClass;
    VAR nextState: ConvTypes.ScanState);
```

```
(* Represents the start state of a finite state scanner for bcd numbers - assigns
class of inputCh to chClass and a procedure representing the next state to nextState.
*)
```

```
PROCEDURE FormatBcd (str: ARRAY OF CHAR): ConvResults;
```

```
(* Returns the format of the string value for conversion to BCD *)
```

```
PROCEDURE ValueBcd (str: ARRAY OF CHAR): BCD;
```

```
(* If str is well-formed, returns the value corresponding to the bcd number string
value str, otherwise an exception is raised. *)
```

```
PROCEDURE LengthFixedBcd (bcd: BCD; place: INTEGER): CARDINAL;
```

```
(* Returns the number of characters in the fixed-point string representation of bcd
rounded to the given place relative to the decimal point. *)
```

```
PROCEDURE TestFreeFormat (format: ARRAY OF CHAR): BcdFormat ;
```

```
(* Tests whether format is a format string accepted by conversion routines, value
checks cannot be done. *)
```

```
PROCEDURE LengthFreeBcd (bcd: BCD; format: ARRAY OF CHAR): CARDINAL;
```

```
(* If format is a well-formed format string and bcd can be represented within this
format, returns the number of characters in the given string representation of bcd,
otherwise an exception is raised. *)
```

```
PROCEDURE IsBcdConvException (): BOOLEAN; (*V5.0b6*)
```

```
END BcdConv.
```

Much of this is more or less as expected. Procedures like *ScanBcd* are a little odd, but a fuller explanation of such scanners can be found in a later section of this chapter. As can be seen, the string formatting rules are a little different (and more complicated) than in the simple example of the previous section, but there is no great uniformity in such matters, and the example begun there was deliberately kept as simple as possible so as not to confuse the concepts.

17.7 A Suggested Project--Polynomials

An abstraction commonly used in algebra, calculus, and their applications is the polynomial. This is a sum of terms, each of which is the product of a real number, and a cardinal power of some variable. Thus,

3 , $4x^2$, $15y^7$, and $23.45t^9$ are all terms and also are all one-term polynomials or monomials

$4m - 7$, $3x^3 + 5x$, $7t^2 - 2t$ are all two term polynomials or binomials

$2x^4 - 5x + 7$ and $3d^6 + 4d^5 - 2d^4$ are three term polynomials or trinomials.

In the polynomial term $4.7x^2$, the number 4.7 is called the coefficient, the letter x is called the (independent) variable, and the exponent 2 is called the degree of the term. The highest exponent in a polynomial is called the degree of the polynomial. Two terms that have the same variable and the same exponent are called like.

A polynomial can be multiplied by a scalar simply by multiplying each of its coefficients by the scalar. Thus

$$4(7x - 2) = 28x - 8$$

$$6(x^2 + 3x) = 12x^2 + 18x$$

Two terms can be combined into one by addition only if they are like. Thus:

$$3x^2 + 7x^2 = 10x^2$$

$$2x - 6x + 5x = x$$

$x^2 + 6x - 5$ cannot be further combined.

The sum of two polynomials are added (subtracted) by adding (subtracting) the coefficients of their mutual like terms:

$$(4x - 7) + (7x - 2) = 11x - 9$$

$$(3x^2 + 4x) + (2x^2 - 5) = 5x^2 + 4x - 5$$

$$(5t - 8) - (2t - 7) = 3t - 1$$

Two monomials (terms) are multiplied by multiplying their coefficient factors and their variable factors. For instance,

$$(3x)(5x^2) = 15x^3$$

$$(3.2y^7)(5y^4) = 16y^{11}$$

If one wishes to multiply two longer polynomials, each term of the multiplier must be multiplied by each term of the multiplicand and the result added.

$$(7x - 5)(2x^2 + 4) = (7x)(2x^2) + (7x)(4) + (-5)(2x^2) + (-5)(4) = 14x^3 + 28x - 10x^2 - 20$$

$$(3x - 2)(4x + 5) = 12x^2 + 15x - 8x - 10 = 12x^2 - 7x - 10$$

For practical reasons (such as graphing the polynomial) one often wishes to evaluate a polynomial at a particular value of the variable. In such cases, the polynomial function is usually denoted $P(x)$ and the evaluation using a particular number, say, c , by $P(c)$. This is done by substituting. Thus if

$$P(x) = 3x^2 + 2x - 7$$

$$P(2) = 3(2^2) + 2(2) - 7 = 9$$

$$P(5) = 3(5^2) + 2(5) - 7 = 78$$

With this little review of elementary algebra, it is not difficult to define a Polynomial ADT in Modula-2. One possibility would be:

```
DEFINITION MODULE Polynomials;  
(* specification by R. Sutcliffe  
  1996 11 07 *)
```

```
TYPE  
  Polynomial;
```

```
PROCEDURE Init (VAR p : Polynomial);  
  (* creates a polynomial and sets it to equal to zero *)
```

```

PROCEDURE Destroy (VAR p : Polynomial);
    (* gives back any dynamic memory: the result is an invalid polynomial*)

PROCEDURE UpdateTerm (p : Polynomial; exp : CARDINAL; coef : REAL);
    (* sets the coefficient of the term of degree exp in a valid polynomial to coef *)

PROCEDURE Degree (p: Polynomial) : CARDINAL;
    (* returns the degree of the specified polynomial *)

PROCEDURE NumTerms (p: Polynomial) : CARDINAL;
    (* returns the number of terms of the specified polynomial *)

PROCEDURE Coefficient (p: Polynomial; degree : CARDINAL) : REAL;
    (* returns the coefficient of the term of specified degree in the given polynomial
    *)

(* The valid form of a string representation of a polynomial is
[+|-term]
and the valid string representation of a term is
realnumbercoefficient, "charactervariablename", ["^" cardinalnumberexponent] *)

PROCEDURE PolyToString (source : Polynomial; VAR dest : ARRAY OF CHAR);

PROCEDURE StringToPoly (source : ARRAY OF CHAR; VAR dest : Polynomial);

PROCEDURE Value (p : Polynomial; at : REAL) : REAL;
    (* evaluate the given polynomial at the specified value of the variable *)

PROCEDURE Add (p, q : Polynomial; VAR res : Polynomial);

PROCEDURE Sub (p, q : Polynomial; VAR res : Polynomial);

PROCEDURE Mul (p, q : Polynomial; VAR res : Polynomial);

PROCEDURE ScalarMul (VAR p : Polynomial; scalar : REAL);
    (* 5(4x^2) for example *)

END Polynomials.

```

A few observations are in order. The actual representation could be done in a number of ways. One would be to define a term, then define the *Polynomial* ADT as a linked list of terms. The latter could be done using an earlier and relatively generic linked list module, or the necessary apparatus could be customized within this module. For instance, the types could be:

```

TYPE
    TermPoint = POINTER TO Term;
    Term =
        RECORD
            coefficient : REAL;
            exponent : CARDINAL;
            nextTerm, lastTerm : TermPoint;
        END;
    Polynomial = POINTER TO PolyDataNode;
    PolyDataNode =

```

RECORD`firstTerm : Term;``numTerms, degree : CARDINAL;`**END;**

The string representation for a polynomial uses a fairly common notation that employs the caret symbol ^ to denote that the number following is an exponent. Thus $7x^3 - 3x^2 + 4$ is represented by the string "7x^3 - 3.4x^2 + 4". The spaces are not relevant and can be ignored on input strings, but should be present for legibility on output strings. A full implementation will not be done here, and is left as a challenge to the reader.

[Contents](#)

17.8 The Date and Time

It is often important to time operations or to keep track of elapsed time between user responses. This could be done in order to:

- determine the relative efficiency of sorting procedures
- time user responses in a game or learning environment
- calculate the number of elapsed days for interest calculations
- place a time or date "stamp" in a document, such as a letter

Some computing systems have only an elapsed time clock. These only keep track of how much time has passed since the system was started up for the last time. Better (not all) systems have a battery operated real time clock that keeps track of the current year, month, day, hour, minute, and second. They may also have fractions of a second, information on the difference between local time and Universal Time, and on whether daylight savings time is in effect. The details of what information is available will vary from one system to another.

If there is a clock present, it may be possible for a client program to set the clock as well as get information from it. In a UNIX system, this is unlikely, as users are generally not allowed access to system features. On personal computers, the opposite is the rule--the user can do almost anything.

In order to take as many as possible of these variations in effect, ISO Modula-2 mandates the following low-level module for setting and examining the system clock. Note that some items are implementation defined.

17.8.1 The Module SysClock

DEFINITION MODULE SysClock;

(* =====

Definition Module from

ISO Modula-2

ISO/IEC IS10515 by JTC1/SC22/WG13

Original specification and design of SysClock

Copyright © 1990-1991 by R. Sutcliffe

Assigned to ISO for standards work

Language and Module designs © 1992 - 1995 by

BSI, D.J. Andrews, B.J. Cornelius, R. B. Henry

R. Sutcliffe, D.P. Ward, and M. Woodman

===== *)

(* Facilities for accessing a system clock that records the date and time of day *)

CONST

maxSecondParts = 0; (* this number is implementation defined *)

TYPE

Month = [1 .. 12];

Day = [1 .. 31];

Hour = [0 .. 23];

Min = [0 .. 59];

Sec = [0 .. 59];

Fraction = [0 .. maxSecondParts];

UTCDiff = [-780 .. 720];

DateTime =

RECORD

```
year:      CARDINAL;
month:     Month;
day:       Day;
hour:      Hour;
minute:    Min;
second:    Sec;
fractions: Fraction;      (* parts of a second *)
zone:      UTCDiff;       (* Time zone differential factor which is the number
                           of minutes to add to local time to obtain UTC. *)
summerTimeFlag: BOOLEAN; (* Interpretation of flag depends on local usage. *)
END;
```

```
PROCEDURE CanGetClock (): BOOLEAN;
```

```
(* Returns TRUE if a system clock can be read; FALSE otherwise *)
```

```
PROCEDURE CanSetClock (): BOOLEAN;
```

```
(* Returns TRUE if a system clock can be set; FALSE otherwise *)
```

```
PROCEDURE IsValidDateTime (userData: DateTime): BOOLEAN;
```

```
(* Returns TRUE if the value of userData represents a valid date and time; FALSE
otherwise *)
```

```
PROCEDURE GetClock (VAR userData: DateTime);
```

```
(* If possible, assigns system date and time of day to userData *)
```

```
PROCEDURE SetClock (userData: DateTime);
```

```
(* If possible, sets the system clock to the values of userData *)
```

```
END SysClock.
```

17.8.2 Time and Date I/O

No other facilities are provided in the ISO standard library, as needs vary from one application and system to another. However, if the clock is available, it is not difficult to write procedures to output the last date and/or time read from the clock.

```
MODULE TestClock;
```

```
(* by R. Sutcliffe
to illustrate SysClock *)
```

```
FROM SysClock IMPORT
```

```
DateTime , CanGetClock, GetClock;
```

```
FROM STextIO IMPORT
```

```
WriteString, WriteLn, WriteChar;
```

```
FROM SWholeIO IMPORT
```

```
WriteCard;
```

```
PROCEDURE Pad (number : CARDINAL);
```

```
BEGIN
```

```
IF number < 10
```

```
THEN
```

```
WriteChar("0");
```

```
END;
```

```

END Pad;

PROCEDURE WriteDateTime (dateTime : DateTime);
BEGIN
    WriteString ("Date: ");
    WriteCard (dateTime.year, 1);
    Pad (dateTime.month);
    WriteCard (dateTime.month, 0);
    WriteChar (" ");
    Pad (dateTime.day);
    WriteCard (dateTime.day, 1);

    WriteString ("    Time: ");
    Pad (10* dateTime.hour);
    Pad (dateTime.hour);
    WriteCard (dateTime.hour, 1);
    Pad (dateTime.minute);
    WriteCard (dateTime.minute, 1);
    WriteString (" : ");
    Pad (dateTime.second);
    WriteCard (dateTime.second, 1);

END WriteDateTime;

VAR
    dateTime : DateTime;
    count : CARDINAL;

BEGIN
    IF CanGetClock()
        THEN
            FOR count := 1 TO 10
                DO
                    GetClock (dateTime);
                    WriteDateTime (dateTime);
                    WriteLn;
                END;
            ELSE
                WriteString ("No clock available");
            END;
    END TestClock.

```

When this program was run, the following output was produced.

```

Date: 1996 11 08    Time: 0911 : 05
Date: 1996 11 08    Time: 0911 : 05
Date: 1996 11 08    Time: 0911 : 05
Date: 1996 11 08    Time: 0911 : 05
Date: 1996 11 08    Time: 0911 : 06
Date: 1996 11 08    Time: 0911 : 06
Date: 1996 11 08    Time: 0911 : 06
Date: 1996 11 08    Time: 0911 : 06
Date: 1996 11 08    Time: 0911 : 06

```


Date: 1996 11 08 Time: 0911 : 06

There are a number of possibilities for expanding the primitive I/O for date and time in this module. Consider the following formats for date input and output:

January 7, 1996	month day year	long month name with comma
Jan 7 1996	month day year	short month name without comma
1/7/96	month day year	all numeric with slashes
01/07/1996	month day year	numeric, slashes, leading zeros, century
7th Jan 1996	day month year	day with suffix, short month name with spaces
1996 01 07	year month day	ISO format with century, spaces, leading zeros

These possibilities could be encapsulated in a conversion module in the following way:

TYPE

```
separator = (space, slash, comma);
order = (ymd, dmy, mdy);
monthNames = (numeric, short, long);
Date =
```

RECORD

```
year : CARDINAL;
month : Month;
day : Day;
```

END;

```
Format =
```

RECORD

```
sep : separator;
ord : order;
useNames : monthNames;
useSuffix : BOOLEAN;
```

END;

```
PROCEDURE FormatDate (date: Date, format : Format; VAR result : ARRAY OF CHAR);
```

The task of designing and writing a suite of formatting and I/O modules for date and time has been left for the exercises.

17.8.3 Time and Date Arithmetic

The most common arithmetic that might have to be done on dates and times is to find the elapsed time or date (or both) between two events or to add or subtract a period to a time or a date. For instance, one could define a module:

```
DEFINITION MODULE DateMath;
```

```
FROM SysClock IMPORT
```

```
DateTime, Month, Day;
```

TYPE

```
Date =
```

RECORD

```
year : CARDINAL;
month : Month;
day : Day;
```

END;

PROCEDURE IsValid (da : Date) : **BOOLEAN**;

PROCEDURE AssignDateToDateTime (da : Date) : DateTime;

PROCEDURE AssignDateTimeToDate (dt : DateTime) : Date;

PROCEDURE Inc (**VAR** da : Date; delta : Date);

(* Increase the given date by the year, month and day specified in delta *)

PROCEDURE Dec (**VAR** da : Date; delta : Date);

(* Decrease the given date by the year, month and day specified in delta *)

PROCEDURE DateSpan (from, to : Date) : Date;

(* Increase the given date by the year, month and day specified in delta *)

END DateMath.

A very similar module could be written for doing mathematics on times. Doing so, and implementing both, is left as an exercise for the reader.

[Contents](#)

17.9 A Closer Look at Whole Number I/O

To date, this text has been concerned with ISO standard I/O only on the top level. For instance, the modules *SWholeIO* (and *WholeIO*) have been freely used, but little attention has been paid to the layer below these high level modules. The purpose of this section is to examine more closely what goes on in the process of reading and writing numeric data.

For instance, in order to output a **CARDINAL** value, the internal representation must first be changed into a string of characters. Once this is complete, the string may simply be passed to *TextIO* for output to the screen.

Likewise, for a **CARDINAL** to be read into memory, characters must be read and processed, interpreting the string of characters as a number and then constructing the numeric value in memory.

In ISO Modula-2 these conversions are handled in two levels of modules. The module *WholeConv* (and others like it for real and longreal types) scan input strings to determine if they can legitimately be regarded as **CARDINAL**s, **INTEGER**s, **REAL**s, or **LONGREAL**s (as the case may be.) The module *WholeStr* (and others like it for Real and Long types) handles the actual conversions from numeric to string and vice-versa.

In addition, there are some modules that provide common information to the conversion routines in the form of constants they all use, and in the form of routines designed to classify characters as numeric, alphabetic, control, or white space.

To illustrate this activity, and a technique employed by the standard library that has not as yet been used in this text, the following subsections detail the modules *WholeConv* and *WholeStr*. The corresponding modules for the real types are similar though somewhat more complicated because there are several output forms in use (fixed, floating, and engineering.) These will not be detailed here.

17.9.1 Common Conversion Information Modules

The first of these provides two enumeration types and one procedure type that all the *xxConv* modules use when they scan input to check its validity.

DEFINITION MODULE ConvTypes;

```
(* Common types used in the string conversion modules *)
```

TYPE

```
  ConvResults =      (* Values of this type are used to express the format of a string *)
  ( strAllRight,      (* the string format is correct for the corresponding conversion *)
    strOutOfRange,    (* the string is well-formed but the value cannot be represented *)
    strWrongFormat,   (* the string is in the wrong format for the conversion *)
    strEmpty          (* the given string is empty *) );
```

```
  ScanClass = (* Values of this type are used to classify input to finite state scanners *)
  ( padding,    (* a leading or padding character at this point in the scan - ignore it *)
    valid,      (* a valid character at this point in the scan - accept it *)
    invalid,    (* an invalid character at this point in the scan - reject it *)
    terminator  (* a terminating character at this point in the scan (not part of token) *) );
```

```
  ScanState = (* The type of lexical scanning control procedures *)
```

```
PROCEDURE (CHAR, VAR ScanClass, VAR ScanState);
```

```
END ConvTypes.
```

Because only types are defined here, the implementation part of this module is empty. The values of the type *ConvResults* are for reporting the results of the scan of a string. If, for example the maximum CARDINAL value that can be stored were 10000, and the string "100000" were passed to the scanner checking for cardinal values, it should report the value *strOutOfRange*. If the string "-M123" were passed to a scanner expecting an integer, it would correctly parse the sign at the beginning, but would then expect a digit. Not finding one, it would return the value *strWrongFormat*.

The values of the type *ScanClass* are for the purpose of classifying characters in the string being scanned. For instance, spaces at the beginning are padding and can be ignored. After the padding has all been scanned, subsequent characters are either valid or invalid, depending on what was expected at that point in the scan. A determination that the scan of the string is finished is eventually made, and the character scanned to decide this is called the terminating character. In many cases, this is a space.

However, when scanning a numeric string, it could be a letter.

The procedure type *ScanState* provides a standard type for scanning procedures. Each one will take a character to scan and produce a classification of the character. Moreover, the correct procedure to scan the next character will be returned.

At any given point in the string, the scanner is in a particular *state*. If one is scanning for integer values for instance and presents the string "-234" the scanner will first be in a starting state where it expects padding (skip and stay in the same state) or a sign or digit (read, store and go to a digit reading state). Once a sign or a digit has been read, the scanner is in a digit reading state where it expects either a digit or a terminating character. There is a different procedure for each state, but one procedure variable can be used for the current one, and the procedure can set the next state through the variable parameter before exiting each time. The entire scanning machine (including all its procedures) has only a few states, and this idea is captured in the definition below. For more specific details on the operation of such a machine, study the *WholeConv* module below.

A routine that operates at any given time in one of finitely many states, at least one of which is a terminal state, is called a finite state machine.

Much more can be said about finite state machines, but such remarks are beyond the scope of this course. The reader is encouraged to further study of the theory of such computing machines as Turing machines and finite state machines.

The second module that is used in common by all the conversion modules is *CharClass*. This provides a general classification of characters that can be useful in a variety of contexts, not just the one here. Its definition is:

```
DEFINITION MODULE CharClass;
```

```
  (* Classification of values of the type CHAR *)
```

```
PROCEDURE IsNumeric (ch: CHAR): BOOLEAN;
```

```
  (* Returns TRUE if and only if ch is classified as a numeric character *)
```

```
PROCEDURE IsLetter (ch: CHAR): BOOLEAN;
```

```
  (* Returns TRUE if and only if ch is classified as a letter *)
```

```
PROCEDURE IsUpper (ch: CHAR): BOOLEAN;
```

```
  (* Returns TRUE if and only if ch is classified as an upper case letter *)
```

```
PROCEDURE IsLower (ch: CHAR): BOOLEAN;
```

```
  (* Returns TRUE if and only if ch is classified as a lower case letter *)
```

```
PROCEDURE IsControl (ch: CHAR): BOOLEAN;
```

```
  (* Returns TRUE if and only if ch represents a control function *)
```

```
PROCEDURE IsWhiteSpace (ch: CHAR): BOOLEAN;
```

```
  (* Returns TRUE if and only if ch represents a space character or a format effector *)
```

```
END CharClass.
```

If the ISO/IEEE character set for the Latin alphabet is employed, an implementation could be done as follows:

```
IMPLEMENTATION MODULE CharClass;
```

```
(* =====  
      Definition Module from  
          ISO Modula-2  
Draft Standard CD10515 by JTC1/SC22/WG13  
      Language and Module designs © 1992 by  
BSI, D.J. Andrews, B.J. Cornelius, R. B. Henry  
R. Sutcliffe, D.P. Ward, and M. Woodman  
  
      Implementation © 1993  
      by R. Sutcliffe  
(Portions coded by G. Tischer)  
      Trinity Western University  
7600 Glover Rd., Langley, BC Canada V3A 6H4  
      e-mail: rsutcl@twu.ca  
      Last modification date 1993 10 20  
===== *)
```

```
(* Classification of values of the type CHAR *)
```

```
PROCEDURE IsNumeric (ch: CHAR): BOOLEAN;
```

```
(* Returns TRUE if and only if ch is classified as a numeric character *)
```

```
BEGIN
```

```
    RETURN (ch < 72C);
```

```
END IsNumeric;
```

```
PROCEDURE IsLetter (ch: CHAR): BOOLEAN;
```

```
(* Returns TRUE if and only if ch is classified as a letter *)
```

```
BEGIN
```

```
    RETURN ((ch < 133C)) OR ((ch < 173C));
```

```
END IsLetter;
```

```
PROCEDURE IsUpper (ch: CHAR): BOOLEAN;
```

```
(* Returns TRUE if and only if ch is classified as an upper case letter *)
```

```
BEGIN
```

```
    RETURN (ch < 133C);
```

```
END IsUpper;
```

```
PROCEDURE IsLower (ch: CHAR): BOOLEAN;
```

```
(* Returns TRUE if and only if ch is classified as a lower case letter *)
```

```
BEGIN
```

```
    RETURN (ch < 173C);
```

```
END IsLower;
```

```
PROCEDURE IsControl (ch: CHAR): BOOLEAN;
```

```
(* Returns TRUE if and only if ch represents a control function *)
```

```
BEGIN
```

```
    RETURN (ch < 40C);
```

```
END IsControl;
```

```
PROCEDURE IsWhiteSpace (ch: CHAR): BOOLEAN;
```

```

    (* Returns TRUE if and only if ch represents a space character or a format
effector *)
BEGIN
    RETURN (ch = 11C) OR (ch = 15C) OR (ch = 40C);
END IsWhiteSpace;

END CharClass.

```

17.9.2 Scanning For Whole Number Input

The module *WholeConv* (and corresponding ones for real and longreal types) is employed for this purpose. Its definition is as follows:

```

DEFINITION MODULE WholeConv;

    (* Low-level whole-number/string conversions *)

IMPORT
    ConvTypes;

TYPE
    ConvResults = ConvTypes.ConvResults; (* strAllRight, strOutOfRange,
strWrongFormat, strEmpty *)

PROCEDURE ScanInt (inputCh: CHAR; VAR chClass: ConvTypes.ScanClass;
    VAR nextState: ConvTypes.ScanState);

    (* Represents the start state of a finite state scanner for signed whole
numbers -assigns class of inputCh to chClass and a procedure representing the next
state to nextState. *)

PROCEDURE FormatInt (str: ARRAY OF CHAR): ConvResults;

    (* Returns the format of the string value for conversion to INTEGER. *)

PROCEDURE ValueInt (str: ARRAY OF CHAR): INTEGER;

    (* Returns the value corresponding to the signed whole number string value str if str
is well-formed; otherwise raises the WholeConv exception. *)

PROCEDURE LengthInt (int: INTEGER): CARDINAL;

    (* Returns the number of characters in the string representation of int. *)

PROCEDURE ScanCard (inputCh: CHAR; VAR chClass: ConvTypes.ScanClass;
    VAR nextState: ConvTypes.ScanState);

    (* Represents the start state of a finite state scanner for unsigned whole numbers -
assigns class of inputCh to chClass and a procedure representing the next state to
nextState. *)

PROCEDURE FormatCard (str: ARRAY OF CHAR): ConvResults;

    (* Returns the format of the string value for conversion to CARDINAL. *)

PROCEDURE ValueCard (str: ARRAY OF CHAR): CARDINAL;

    (* Returns the value corresponding to the unsigned whole number string value str if
str is well-formed; otherwise raises the WholeConv exception *)

PROCEDURE LengthCard (card: CARDINAL): CARDINAL;

```

```
(* Returns the number of characters in the string representation of card. *)
```

```
PROCEDURE IsWholeConvException (): BOOLEAN;
```

```
(* Returns TRUE if the current coroutine is in the exceptional execution state  
because of the raising of an exception in a routine from this module; otherwise  
returns FALSE. *)
```

```
END WholeConv.
```

In a manner similar to the way in which device modules import and re-export the channel constants, these modules import and re-export the type *ConvResults*. Notice that there is a starting state (procedure) for scanning for an integer, and another one for starting the scan of a cardinal. Subsequent scanners are hidden away in the implementation module. Observe too that one must know the results of a call to *FormatCard* before attempting to do the actual conversion to a cardinal with *ValueCard* because if the string turns out to be invalid, *ValueCard* will raise an exception. Here is an implementation of this module:

```
IMPLEMENTATION MODULE WholeConv;
```

```
(* =====  
      Definition Module from  
      ISO Modula-2  
Draft Standard CD10515 by JTC1/SC22/WG13  
      Language and Module designs © 1992 by  
BSI, D.J. Andrews, B.J. Cornelius, R. B. Henry  
R. Sutcliffe, D.P. Ward, and M. Woodman  
  
      Implementation © 1994  
      by R. Sutcliffe  
(Portions coded by G. Tischer)  
      Trinity Western University  
7600 Glover Rd., Langley, BC Canada V3A 6H4  
      e-mail: rsutcl@twu.ca  
      Last modification date 1996 12 03  
===== *)
```

```
(* Low-level whole-number/string conversions *)
```

```
(*1994 06 14 First version by R. Sutcliffe  
      1996 11 12 revised by R. Sutcliffe not to use scanClass inputs.  
      ideas considered include those of Norm Black and Keith Hopper *)
```

```
IMPORT
```

```
    ConvTypes;
```

```
FROM ConvTypes IMPORT
```

```
    ScanClass;
```

```
FROM EXCEPTIONS IMPORT
```

```
    ExceptionSource, AllocateSource, RAISE, IsExceptionalExecution, IsCurrentSource;
```

```
FROM CharClass IMPORT
```

```
    IsWhiteSpace, IsNumeric;
```

```
(* two globals to hold last wholes checked by format; use if OK in value *)
```

```
VAR
```

```
    theCard : CARDINAL;
```

```
    theInt : INTEGER;
```

```

CONST
    zeroAsc = ORD ("0");
    maxCardDiv10 = MAX (CARDINAL) / 10;
    lastMaxCardDigitChar = CHR( (MAX (CARDINAL) REM 10) + zeroAsc );
    maxIntDiv10 = MAX (INTEGER) / 10;
    lastMaxIntDigitChar = CHR( (MAX (INTEGER) REM 10) + zeroAsc );
    minIntDiv10 = MIN (INTEGER) / 10;
    lastMinIntDigitChar = CHR( ABS ((MIN (INTEGER) REM 10)) + VAL (INTEGER, zeroAsc)
);
VAR
    WholeConvExSource : ExceptionSource;

(* local procs representing scan states after the initial one. These are named after
the last input classified *)

PROCEDURE Sign (inputCh : CHAR; VAR chClass : ConvTypes.ScanClass; VAR nextState :
ConvTypes.ScanState);
(* after sign must get digit *)
BEGIN
    IF IsNumeric (inputCh)
    THEN
        chClass := ConvTypes.valid;
        nextState := WDigit;
    ELSE
        chClass := ConvTypes.invalid;
        nextState := Sign;
    END;
END Sign;

PROCEDURE WDigit (inputCh : CHAR; VAR chClass : ConvTypes.ScanClass; VAR nextState :
ConvTypes.ScanState);
(* after digit can have another digit; anything else terminates *)
BEGIN
    IF IsNumeric (inputCh)
    THEN
        chClass := ConvTypes.valid;
        nextState := WDigit;
    ELSE
        chClass := ConvTypes.terminator;
        (* no point in changing states *)
    END;
END WDigit;

(* exported procs *)
PROCEDURE ScanInt (inputCh: CHAR; VAR chClass: ConvTypes.ScanClass;
    VAR nextState: ConvTypes.ScanState);
    (* Represents the start state of a finite state scanner for signed whole numbers-
    assigns class of inputCh to chClass and a procedure representing the next state to
    nextState. *)

BEGIN
    IF IsWhiteSpace (inputCh)
    THEN

```



```

    (* say got padding *)
    chClass := ConvTypes.padding;
    (* and next state is same as this one *)
    nextState := ScanInt;
ELSIF IsNumeric (inputCh) THEN (* got digit *)
    chClass := ConvTypes.valid;
    (* switch to digit state *)
    nextState := WDigit;
ELSIF (inputCh = '-') OR (inputCh = '+') THEN
    chClass := ConvTypes.valid;
    (* switch to sign state *)
    nextState := Sign;
ELSE (* anything else is no good *)
    chClass := ConvTypes.invalid;
    nextState := ScanInt;
END;
END ScanInt;

```

```

PROCEDURE FormatInt (str: ARRAY OF CHAR) : ConvResults;
    (* Returns the format of the string value for conversion to INTEGER. *)

```

VAR

```

    count, len : CARDINAL;
    class : ConvTypes.ScanClass;
    Scan : ConvTypes.ScanState;
    pos : BOOLEAN;
    ch : CHAR;

```

BEGIN

```

    pos := TRUE;
    len := LENGTH (str);
    IF len = 0
    THEN
        RETURN ConvTypes.strEmpty
    ELSE
        theInt := 0;
        Scan := ScanInt;
        count := 0;
        LOOP
            ch := str [count];
            Scan (ch, class, Scan);
            CASE class OF
                padding:
                    (* leave it *) |
                valid:
                    IF ch = "-" (* negative sign *)
                    THEN
                        pos := FALSE;
                    ELSIF pos AND (theInt <= maxIntDiv10) THEN
                        theInt := theInt * 10; (* shift number over *)
                        IF (theInt < maxIntDiv10) OR (ch <= lastMaxIntDigitChar)
                        THEN
                            INC (theInt, VAL (INTEGER, (ORD (ch) - zeroAsc))) ; (* add digit
*)

```

```

        ELSE
            RETURN ConvTypes.strOutOfRange
        END
        (* consider the case when the last digit might cause an
overflow. *)
        ELSIF (NOT pos) AND (theInt > minIntDiv10) OR (ch <=
lastMinIntDigitChar))
            THEN
                DEC (theInt, VAL (INTEGER, (ORD (ch) - zeroAsc))); (* add digit
*)
            ELSE
                RETURN ConvTypes.strOutOfRange
            END;
        ELSE
            RETURN ConvTypes.strOutOfRange
        END; |
invalid:
        RETURN ConvTypes.strWrongFormat |
        terminator: (* if get here, all OK so far *)
        RETURN ConvTypes.strAllRight;
    END; (* case *)
    INC (count);
    IF count = len (* end of string and still all ok -- haven't returned *)
    THEN
        RETURN ConvTypes.strAllRight;
    END;
    END; (* loop *)
END; (* if *)

END FormatInt;

PROCEDURE ValueInt (str: ARRAY OF CHAR): INTEGER;
(* Returns the value corresponding to the signed whole number string value str if str
is well-formed; otherwise raises the WholeConv exception. *)

BEGIN
    IF FormatInt (str) # ConvTypes.strAllRight
    THEN
        RAISE (WholeConvExSource, 0, "Can't convert badly formatted string to
integer.");
    ELSE
        RETURN theInt
    END;
END ValueInt;

PROCEDURE LengthInt (int: INTEGER): CARDINAL;
(* Returns the number of characters in the string representation of int. *)

VAR
    count : CARDINAL;
    neg : BOOLEAN;

BEGIN
    IF int < 0

```

```

THEN
    neg := TRUE;
ELSIF int <= maxCardDiv10
    THEN
        theCard := theCard * 10; (* shift *)
        IF (theCard < maxCardDiv10) OR (str [count] <= lastMaxCardDigitChar)
            THEN (* add digit *)
                theCard := theCard + (ORD(str [count]) - zeroAsc);
            ELSE
                RETURN ConvTypes.strOutOfRange
            END;
        ELSE
            RETURN ConvTypes.strOutOfRange
        END;
    |
invalid:
    RETURN ConvTypes.strWrongFormat |
terminator: (* if get here, all OK so far *)
    RETURN ConvTypes.strAllRight;
END; (* case *)
INC (count);
IF count = len (* end of string and still all ok -- haven't returned *)
    THEN
        RETURN ConvTypes.strAllRight;
    END;
END; (* loop *)
END; (* if *)

```

END FormatCard;

PROCEDURE ValueCard (str: **ARRAY OF CHAR**): **CARDINAL**;

(* Returns the value corresponding to the unsigned whole number string value str if str is well-formed; otherwise raises the WholeConv exception *)

BEGIN

```

IF FormatCard (str) # ConvTypes.strAllRight
    THEN
        RAISE (WholeConvExSource, 0, "Can't convert badly formatted string to
cardinal.");
    ELSE
        RETURN theCard; (* stashed globally for us *)
    END;

```

END ValueCard;

PROCEDURE LengthCard (card: **CARDINAL**): **CARDINAL**;

(* Returns the number of characters in the string representation of card. *)

VAR

count : **CARDINAL**;

BEGIN

```

IF card = 0
    THEN
        RETURN 1;
    END;
count := 0;

```

```

WHILE card # 0
  DO
    card := card DIV 10;
    INC (count);
  END;
RETURN count;
END LengthCard;

PROCEDURE IsWholeConvException (): BOOLEAN;
  (* Returns TRUE if the current coroutine is in the exceptional execution state
  because of the raising of an exception in a routine from this module; otherwise
  returns FALSE. *)

BEGIN
  RETURN (IsExceptionalExecution() ) AND (IsCurrentSource (WholeConvExSource) )
END IsWholeConvException ;

BEGIN (* initialization of main module *)

  AllocateSource (WholeConvExSource);

END WholeConv.

```

There are a few interesting things worth noting about this implementation. Since the *Format* and *Value* procedure pairs are closely related, in this implementation a choice has been made to store the value constructed by *Format* in a global variable. When *Value* calls *Format* it can then pick up the result at once. There is a design problem here, however, and that is that an outside client must call *Format* and then *Value* and so do all the work twice. If the outside routine has access to the source code of *WholeConv* some of it can be copied there to improve the speed. Other clients will just have to suffer a performance hit when using both routines.

Another thing that a careful implementor has to worry about is the handling of values of type **INTEGER**. Usually signed numbers are implemented in such a way that the expression **ABS** (**MIN**(**INTEGER**)) causes an overflow because the result is too big. Thus, it is necessary in conversion code to handle the case of **MIN**(**INTEGER**) separately. Note also that when one gets close to the **MAX** and **MIN** values of **INTEGER** and the **MAX** value of **CARDINAL**, it is necessary to look carefully at the last digit so as to ensure that an overflow does not take place. For instance, if the largest cardinal that could be stored were 65535, then *maxCardDiv10* is 6553 and *LastMaxCardDigitChan* is 5. If the scan thus far had 6553 as it added digits on the right and shifted left, the next digit must be looked at carefully to see if an overflow would take place if it were used.

17.9.3 High Level String Conversion Routines

At the next higher level (the one that I/O modules will call to do conversions) the ISO library has such modules as *WholeStr*. Here is the definition:

```

DEFINITION MODULE WholeStr;

  (* Whole-number/string conversions *)

IMPORT
  ConvTypes;

TYPE
  ConvResults = ConvTypes.ConvResults; (* strAllRight, strOutOfRange, strWrongFormat,
  strEmpty *)

  (* the string form of a signed whole number is

```

```

    ["+" | "-"], decimal digit, {decimal digit}
*)

PROCEDURE StrToInt (str: ARRAY OF CHAR; VAR int: INTEGER; VAR res: ConvResults);
    (* Ignores any leading spaces in str. If the subsequent characters in str are in
    the format of a signed whole number, assigns a corresponding value to int. Assigns a
    value indicating the format of str to res. *)

PROCEDURE IntToStr (int: INTEGER; VAR str: ARRAY OF CHAR);
    (* Converts the value of int to string form and copies the possibly truncated
    result to str. *)

(* the string form of an unsigned whole number is decimal digit, {decimal digit} *)

PROCEDURE StrToCard (str: ARRAY OF CHAR; VAR card: CARDINAL; VAR res:
ConvResults);
    (* Ignores any leading spaces in str. If the subsequent characters in str are in
    the format of an unsigned whole number, assigns a corresponding value to card.
    Assigns a value indicating the format of str to res. *)

PROCEDURE CardToStr (card: CARDINAL; VAR str: ARRAY OF CHAR);
    (* Converts the value of card to string form and copies the possibly truncated
    result to str. *)

END WholeStr.

```

Notice that if the string to which a whole number is to be converted is not long enough, then the result is supposed to be simply truncated. In the implementation that follows, this is done by dividing the whole number by 10 until it will fit.

```

IMPLEMENTATION MODULE WholeStr;

(* omit copyright notice from above to save space *)
(* omit long version history *)
(* Whole-number/string conversions *)

FROM WholeConv IMPORT
    ScanCard, ScanInt, LengthInt, LengthCard;
IMPORT
    ConvTypes;
FROM ConvTypes IMPORT
    ScanClass;
    (* debug *)

(* It makes no sense to call FormatInt and ValueInt from WholeConv because then the
work is done twice. So, we dump a copy of FormatInt and FormatCard here *)
(* two globals to hold last wholes checked by format; use if OK in value *)
VAR
    theCard : CARDINAL;
    theInt : INTEGER;
CONST
    zeroAsc = ORD ("0");
    maxCardDiv10 = MAX (CARDINAL) / 10;
    lastMaxCardDigitChar = CHR( (MAX (CARDINAL) REM 10) + zeroAsc);
    maxIntDiv10 = MAX (INTEGER) / 10;

```

```

lastMaxIntDigitChar = CHR( (MAX (INTEGER) REM 10) + zeroAsc);
minIntDiv10  = MIN (INTEGER) / 10;
lastMinIntDigitChar = CHR ((ABS (MIN (INTEGER) + 10)) REM 10 + VAL (INTEGER,
zeroAsc));

(* omit copies of FormatInt and FormatCard from above *)
(* end copies of stuff from WholeConv *)

PROCEDURE StrToInt (str: ARRAY OF CHAR; VAR int: INTEGER; VAR res: ConvResults);
  (* Ignores any leading spaces in str. If the subsequent characters in str are in
the format of a signed whole number, assigns a corresponding value to int. Assigns a
value indicating the format of str to res.  *)
BEGIN
  res := FormatInt (str);
  IF res = ConvTypes.strAllRight
    THEN
      int := theInt; (* pick up the global *)
    END;
END StrToInt;

PROCEDURE IntToStr (int: INTEGER; VAR str: ARRAY OF CHAR);
  (* Converts the value of int to string form and copies the possibly truncated
result to str. *)

(* the string form of an unsigned whole number is
   decimal digit, {decimal digit} *)
VAR
  maxs, maxi, count, digit, stop : CARDINAL;
  neg : BOOLEAN;

BEGIN
  neg := (int < 0);
  maxs := HIGH (str) + 1;
  maxi := LengthInt (int); (* includes any neg *)
  IF maxs > count
    DO
      int := int / 10;
      DEC (maxi);
    END;
  END;
  IF neg
    THEN
      IF int = MIN (INTEGER) (* special case this one, cant do abs else *)
        THEN
          str [count - 1] := lastMinIntDigitChar;
          DEC (count);
          int := int / 10;
        END;
      stop := 1;
      str [0] := "-";
      int := ABS (int);
    ELSE
      stop := 0;
    END;
  END;

```

```

WHILE count > maxi
  THEN
    count := maxi;
    str [count] := "";
  ELSE
    count := maxs;
    (* if string is too short, throw away some digits *)
    WHILE maxi > 0
    DO
      digit := card REM 10;
      str [count - 1] := CHR (digit + zeroAsc);
      card := card / 10;
      DEC (count);
    END;
END CardToStr;

END WholeStr.

```

There are similar modules for the conversion of reals and longreals, but the code is somewhat more complex because of the need to consider the exponent and the mantissa. There is also a lower level module for working with reals called *LoReal*, but that is beyond the scope of this section.

17.9.4 High Level Whole Number I/O

The reader is familiar of course with the use of *WholeIO* and *SWholeIO* from long use throughout the text. With the materials already developed here, it is now possible to produce implementations for both. Rather than reproduce the entire contents of both, only a single procedure is given here to illustrate the calls to the conversion routines above.

```

PROCEDURE WriteInt (cid: IOChan.ChanId; int: INTEGER; width: CARDINAL);
  (* Writes the value of int to cid in text form, in a field of the given
  minimum width. *)

VAR
  temp : String;
  count, len : CARDINAL;

BEGIN
  IntToStr (int, temp);          (* convert the integer to a string *)
  len := LENGTH (temp);
  IF len < width                  (* write spaces *)
  THEN
    WriteString (cid, PadString (width-len));
  ELSIF width = 0 THEN
    WriteChar (cid, GetPadChar());
  END;
  WriteString (cid,temp);        (* write the string *)
END WriteInt;

```

17.10 Chapter Summary

This chapter covered these topics:

- A discussion of the implementation of numeric data types such as complex numbers.
- ISO standard support for complex numbers both within the language and in standard libraries.
- One application of complex numbers in electrical circuits.
- A discussion of complex number input and output, and an implementation of a library for these.
- A discussion of decimal types and how these may be implemented (optional).
- A suggestion for defining and implementing a polynomial type.
- Date and time support in the ISO standard, and some comments on date and time arithmetic.
- An extended discussion of whole number I/O with a look at the ISO libraries and their implementations.

It included discussion of the following Modula-2 built-ins:

Reserved Words

none

Standard Identifier

COMPLEX

LONGCOMPLEX

RE

IM

CMPLX

Standard Libraries

ComplexMath

LongComplexMath

SysClock

ConvTypes

CharClass

WholeConv

WholeStr

(The real and long versions of the latter two were also mentioned.)

17.11 Assignments

Questions

1. What is a complex number?
2. "Complex numbers are not scalar numbers." Explain this statement.
3. Why did the ISO committee decide to include complex numbers as a basic type? What is so difficult about implementing them in a library?
4. In what sense are CMPLX (on the one hand), and RE and IM (on the other) inverses of each other?
5. Why are there two complex types and two associated libraries?
6. What is the C-type and what kinds of entities are of this type?
7. Suppose *zed* is of type COMPLEX. What is wrong with the assignment *zed* := 0?
8. The *norm* of a complex number is the number divided by its absolute value *abs(z)*. Show that *abs(norm(z))* is always equal to *one*.
9. Write a small program to compute the value of $e^{i\pi}$.
10. How may the risk of an overflow be reduced when computing the modulus or abs of a complex number?
11. What criticism was made against the design of *WholeStr* and *WholeConv* and why?
12. What other combinations of resistance and capacitance can be made to construct different types of RC circuits? Develop transfer functions for these.
13. Suppose there is also an inductance in the circuit, say on the bottom part of [figure 17.3](#). What does the transfer function become?
14. How many different combinations of on each of resistance, capacitance, and inductance are there (one, two or three at a time).
15. Suppose two resistors are placed in series. Write a program to compute the complex (AC) voltage drop across each from a 120 Volt AC source at 60Hz (cycles per second).
16. How does a library module allow a client program to handle its errors?
17. What is the difference between a *Decimal* type and a *BCD* type?
18. Add carefully thought out comments to the module *DecimalStr* in [section 17.5](#).

Problems

19. Re-implement *ComplexNumbersO* using an array implementation for the ADT.
20. Write and test your own procedures to compute the modulus and argument of a complex number.
21. Write a program to solve quadratic equations of the form $ax^2 + bx + c = 0$. The user should type in the coefficients *a*, *b*, and *c* and the program should compute the root(s), if any, using the quadratic formula $x = (-b \pm d)/2$ where $d = \sqrt{b^2 - 4ac}$. Report answers that are complex numbers automatically in the form *a* + *bi*.
22. Modify the above program to allow for complex coefficients as well.
23. Write a program that will calculate the principal square root of a complex number and then display both square roots.

24. Write a program that will compute the principal n^{th} root of the number one (unity) and then display all the n^{th} roots.
 25. Some people like to write complex numbers as ordered pairs (a, b) . Moreover, earlier in the text there was a module that implemented *points* as an ADT, and these are also normally written as ordered pairs. Write a module *PairIO* to read and write ordered pairs in the form (a, b) .
 26. Modify the module above to read and write any n -tuple (a, b, c, d, \dots) .
 27. Re-do the Complex I/O in this chapter to employ low level scanners and converters as in Whole I/O. Comment on the usefulness of this.
 28. Complete the implementation of the module *Decimals*.
 29. Implement the module *Polynomials*.
 30. Implement the module *ComplexPolynomials*, which is the same as the module *Polynomials* except that the coefficients and the values of the variables (but not the exponents) may be complex.
 31. Add the procedures *Divide* and *Derivative* to the module *Polynomials*.
 32. Define and implement a module for date and time I/O.
 33. Define and implement a module for date and time arithmetic. You will need to be able to find out how many hours there are between two times, how many days between two dates, add a specified number of hours/minutes/seconds to a time, and so on. (You decide on exactly what needs to be included in the design.)
 34. Write your own version of the ISO module *RealConv*.
 35. Write your own version of the ISO module *RealStr*. If the ISO module *LoReal* is available on your system, you may elect to use it or not as you wish.
 36. Design and write a module to implement the ADT *Array* in a dynamic fashion. A client program should be able to create an array of specified size dynamically, and add or multiply two of these where the sizes permit.
 37. Follow up the last question with a module to do I/O on these dynamic arrays.
 38. Design and implement a module to do polynomial I/O. You will have to decide what the format of a polynomial will be for such purposes.
 39. Redesign the *ComplexIO* suite to input and output complex numbers in the form (r, θ) (that is in polar coordinate form.) You may wish to design the modules so that any pair of this form can be read/written, (lower level; can also be used with vectors) and the complex level just makes simple conversions from these into complex numbers.
 40. Define a number type called a *Quat* by $q = a + bi + cj + dk$ where $i^2 = j^2 = k^2 = ijk = -1$. Create a module defining and implementing this data type along with some basic operations.
 41. Write the basic I/O module for the type *Quat*.
-

Chapter 17

Advanced Applications

[17.0 Chapter Goals](#)

[17.1 Standard and Non-Standard Numeric Types](#)

[17.2 Complex Numbers](#)

[17.2.1 Complex Numbers Defined](#)

[17.2.2 Implementing non-ISO Complex Numbers](#)

[17.2.3 Testing the non-ISO Complex Implementation](#)

[17.2.4 Opaque non-ISO Complex Numbers](#)

[17.2.5 Testing the Opaque non-ISO Complex Implementation](#)

[17.3 ISO Complex Types and Support](#)

[17.3.1 ISO COMPLEX Math Library Support](#)

[17.3.2 Implementing ComplexMath](#)

[17.3.3 Input and Output](#)

[17.4 Electrical Circuits and Complex Numbers](#)

[17.5 Very Long Cardinals--The Type Decimal](#)

[17.6 Binary Coded Decimal Fixed Point Types](#)

[17.6.1 BCD Support in p1 Modula-2 \(Optional\)](#)

[17.7 A Suggested Project--Polynomials](#)

[17.8 The Date and Time](#)

[17.8.1 The Module SysClock](#)

[17.8.2 Time and Date I/O](#)

[17.8.3 Time and Date Arithmetic](#)

[17.9 A Closer Look at Whole Number I/O](#)

[17.9.1 Common Conversion Information Modules](#)

[17.9.2 Scanning For Whole Number Input](#)

[17.9.3 High Level String Conversion Routines](#)

[17.9.4 High Level Whole Number I/O](#)

[17.10 Chapter Summary](#)

[17.11 Assignments](#)

Contents

18.0 Chapter Goals

The purpose of this chapter is to introduce some simple graphics ideas and show how they can be realized on the MacOS and Windows 32 bit systems. On completing the chapter, the student should understand and be able to use the following: a supplied module for graphing simple figures, fractals, and functions. The student should have some acquaintance with aspects of the application program interface (API) for the MacOS and Windows NT.

Data Representation Abstractions

General:

No new standard data types are taken up in chapter 18.

Realized in the Modula-2 notation:

Numerous items of the MacOS and Windows NT API

Data Manipulation Abstractions

General:

Window manipulation at an elementary level; screen graphics, the graphical representation of points, lines, and other figures.

Realized in the Modula-2 notation:

Numerous items of the MacOS and Windows NT API

Programming Abstractions

General:

Event loops.

Realized in the Modula-2 notation:

Calls to event loop handlers in MacOS and Windows NT API.

18.1 Basic Graphics Concepts

Picturing the relationships between two kinds of data is commonly done using points on a Cartesian plane as described in [section 6.9](#). In that section, however, only the algebraic characteristics of points in a plane were realized--no graphical results were achieved. This type of support more difficult to achieve for three reasons:

first because providing support for drawing on the screen of the computer requires at least some platform specific abilities,

second because, although the mathematical model for graphing envisions a coordinate system employing real numbers, the graphics environment on computers uses discrete dots, and these are limited in number. This requires some compromises in the visual presentation,

third because the number of dots (pixels) available in the vertical and horizontal directions cannot be known ahead of time, and

fourth because even in typical mathematical use there is more than one type of coordinate system.

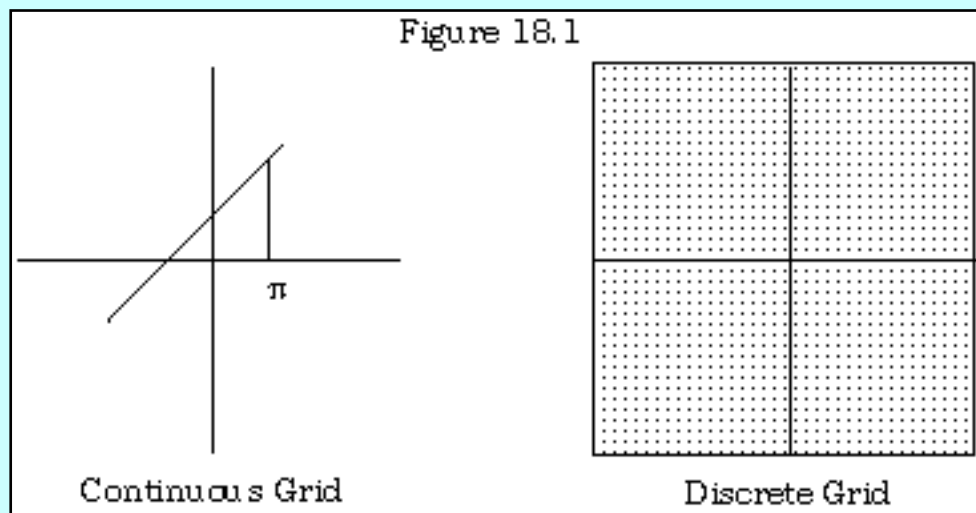
It should be noted at this stage that the ISO committee made *no* attempt to include any form of graphics support in its standard library, so this leaves the student on her own for such important tasks.

The underlying graphics user interface supplied by the computer manufacturer, and to which the compiler vendor will no doubt provide some interface will probably include a number of tools for indicating points on the screen, drawing lines, curves, and even a few predefined shapes. As these are quite system dependent, consideration of them will be postponed for a few sections.

18.1.1 Discrete Grids--Graphing Pixels

The standard rectangular or Cartesian (named after René Descartes) has two perpendicular real number lines (axes) which divide the plane into four quadrants. The place where the number lines cross is the origin (0, 0) and the numbering is positive to the right and up; negative to the left and down. This abstraction includes the ability to graph a line as a set of continuous points, including those such as $(\frac{1}{4}, 1)$ where the decimal representation continues indefinitely without repetition or terminating.

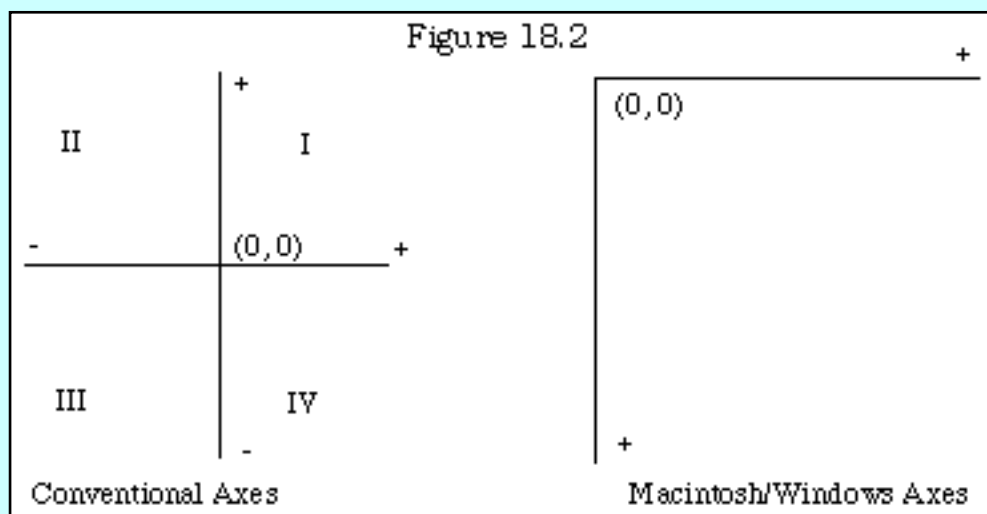
On a typical computer generated graphics raster, there are several hundred pixel points available in both the horizontal and vertical directions, and they are much closer together than suggested in figure 18.1 below. The standard grid could simply be modified to take this into account, but only points with whole number coordinates can be properly depicted. The point $(\frac{1}{4}, 1)$ would have to be depicted as (3,1) unless some form of scaling were used. Indeed, since the individual pixels are rather close together on the screen, it might be better to magnify this and use, say, ten points for a unit, marking (31, 10) on the raster to represent this point.



18.1.2 Where is the origin?

When the Macintosh--the first popular graphics-oriented computer was developed--it had the origin not at the centre of the screen as suggested in the above diagrams, but at the top left corner. The positive horizontal direction was still to the right, but the positive vertical direction goes down, rather than up. There are still four quadrants, as points can have negative coordinates, but three of the quadrants are off the screen. As with many other parts of the interface, this arrangement was subsequently copied into other graphics user systems, including the various versions of Windows. The standard mathematical system and the Mac/Windows screen are illustrated in figure 18.2.

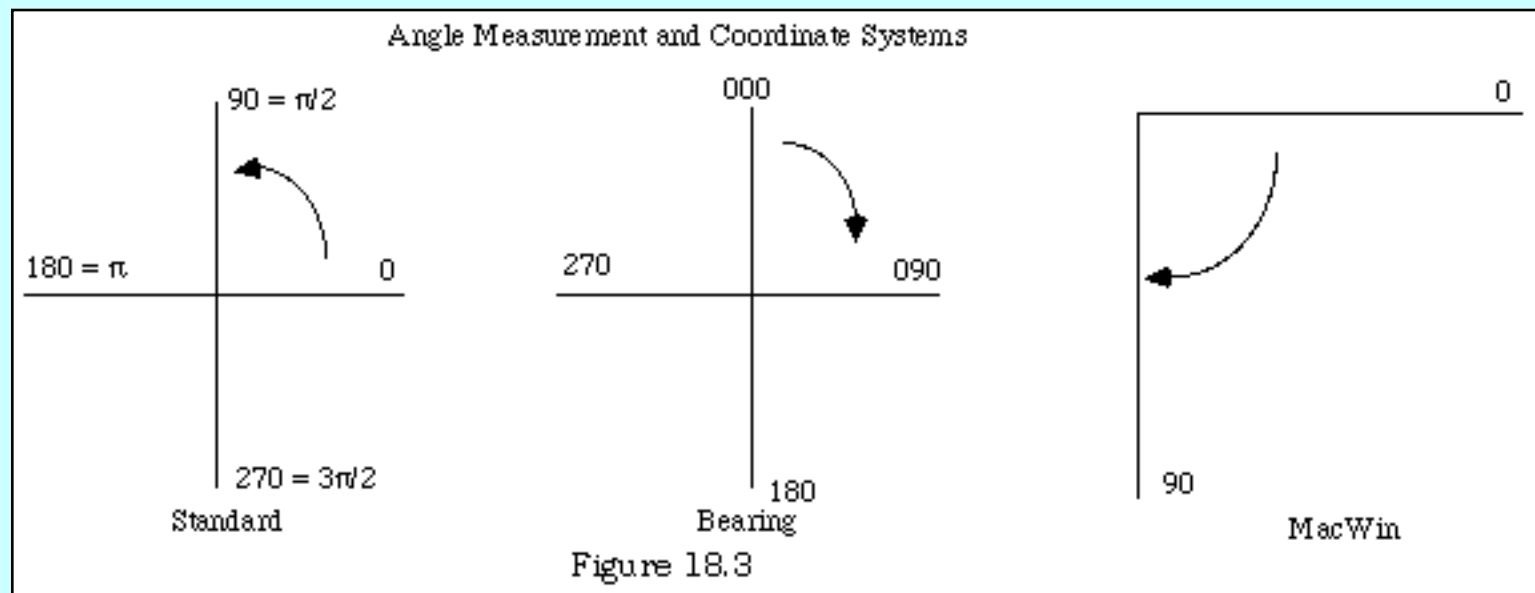
A graphing module used on either of these systems will need to have the option of using the native system directly, or passing conventional mathematical coordinates and then translating to the native screen.



18.1.3 Measuring Angles

Yet another option needed for graphing is created by the fact that when the focus is on polar coordinates--that is, the angle and distance from the origin are given--rather than on rectangular coordinates, there are

two common systems for measuring angles. In the *standard* system used in mathematics and physics, the angle zero is the positive horizontal axis (East; also called the polar axis) and angles are measured counterclockwise (either in degrees or radians.) However, in the *bearing* system used in navigation, the positive vertical axis (North) is taken as the zero angle, and bearings are measured clockwise. For the Macintosh/Windows raster, it makes sense to use the positive horizontal axis for zero and measure counterclockwise, as only one quadrant is on the screen. In addition, because some people prefer to graph in degrees and others in radians, a graphing module perhaps ought to provide the option to do either.



Observe that bearing style angles are by convention recorded in degrees using three digits, including leading zeros where necessary.

[Contents](#)

18.2 A Graphics Environment

For the purposes of carrying the discussion further, the remainder of this chapter will assume that the student has access to the graphics facilities provided by the following definition module. Although the code required to realize this module is quite implementation dependent, it can be done on both Macintosh and Windows systems with a minimum of work, and the necessary detail may all be found in [section 18.6](#) so that the student can easily add the necessary facilities to her own project libraries if they have not been bundled by the compiler vendor.

The basis for drawing on the screen in a graphics user environment is a *pen*. At any given time, the pen is in a specific (coordinatized) position, and can be used to draw a trail of dots as desired. Alternately, it can be moved to another position without drawing.

DEFINITION MODULE GraphPaper;

```
(* Original design copyright 1996 by R. Sutcliffe
   Original implementation 1996 using pl on the Macintosh
   Windows implementation 1998 05 12 by Joel Schwartz
       with use of examples written by Stony Brook
       Last revision by RS 1998 07 11
*)
```

TYPE

```
CoordSystem = (MacWin, bearing, standard); (* default = standard *)
(* The MacWin Coordinate system grows down and has the origin at the top
   left hand corner with angles measured clockwise. In the bearing
   system the home position (0,0) is the middle of the screen
   from which angles are measured clockwise. The standard system also
   starts in mid screen and grows up but measures from east as zero and
   thence counterclockwise. *)

AngleType = (deg, rad, grad); (* Allows for angle type specification *)
LabelType = ARRAY [0..50] OF CHAR; (* Standard format for labels *)
```

PROCEDURE SetCoordSystem (kind : CoordSystem);

```
(* Allows the user to set the system. The default is the standard system so this has
to be called only if a change is desired. This procedure concludes by calling Home.
Any shift in the system origin must be made after calling this procedure. *)
```

PROCEDURE SetAngleType (kind : AngleType);

```
(* Allows the user to set the angle measurement type. The default is degrees so this
has to be called only if a change is desired. This procedure concludes by calling
Home. Any shift in the system origin must be made after calling this procedure. *)
```

PROCEDURE Home;

```
(* moves to 0,0 and then
   In the bearing system:
       - sets angle to 0 (North)
       - sets the rotational direction to clockwise
   In the MacWin & standard system:
       - sets the angle to 0 (East)
       - sets the rotational direction to clockwise (MacWin)
```

or to counterclockwise (standard) *)

PROCEDURE ShiftOrigin (deltaX, deltaY : **INTEGER**);
(* Translate the origin by the amount specified. The direction of the translation on the screen depends, of course, on the coordinate system being used. Drawing is now with respect to the new origin. Does not call home or change any other settings. *)

PROCEDURE GetDimensions (**VAR** x, y: **INTEGER**);
(* obtains the overall width and height of the drawing screen *)

PROCEDURE GetLocation (**VAR** x, y: **INTEGER**);
(* get the drawing pen location in current coordinates *)

(* The following three procedures work in the current coordinate system and on the stored pen position only but do no actual drawing. *)

PROCEDURE MoveBy (distance: **INTEGER**);
(* move in the stored direction by the supplied distance *)

PROCEDURE MoveTo (x, y : **INTEGER**);
(* move the drawing pen to the specified coordinates *)

PROCEDURE Move (dx, dy : **INTEGER**);
(* move the drawing pen to a point (x + dx, y + dy) from the currently stored point *)

(* The following three procedures work in the current coordinate system on the stored pen direction only but do no actual drawing.
The angle is assumed to be in the currently set units. *)

PROCEDURE GetCurrentAngle () : **REAL**;
(* Return the current angle in the current units *)

PROCEDURE Turn (angle : **REAL**);
(* change the stored pen direction by adding its angle to the one supplied *)

PROCEDURE TurnTo (angle : **REAL**);
(* change the stored pen direction by setting its angle to the one supplied *)

(* The following two procedures use the pen to draw a line and change the stored position. *)

PROCEDURE LineBy (distance: **INTEGER**);
(* Draws in the stored pen direction the number of units supplied. *)

PROCEDURE LineTo (x, y : **INTEGER**);
(* Draws a line from the current stored position to the supplied one without using or changing the stored direction. *)

PROCEDURE Line (dx, dy : **INTEGER**);
(* Line to a point (x + dx, y + dy) from the current point without using or changing the stored direction. *)

(* The following two procedures place a dot on the screen, but do not change the pen direction. Measurement is in pixels; not scaled. *)

PROCEDURE Dot;
(* places a dot at the present location *)

PROCEDURE DotAt (x, y : **INTEGER**);
(* does a MoveTo, then a dot *)

```
(* These procedures are for annotating the graph paper with a scale and labels for the axes. *)
```

```
PROCEDURE SetLabels (horix, vert : LabelType);
```

```
    (* Sets the names for the horizontal and vertical axes. *)
```

```
PROCEDURE ShowLabels;
```

```
    (* Show the labels - if no label is set then "x" and "y" are used *)
```

```
PROCEDURE ShowAxes;
```

```
    (* Show the axes for the graph *)
```

(* The following procedures allow for a scaling of the graph paper and the plotting of points according to the scale. If the scale is 10, there is one unit every ten division marks. This will make the plotting of functions more readable. The default is one unit per division mark.

EXAMPLE: setting the scale to 5 using cm's as the unit means 1 cm = 5 division marks.*)

```
PROCEDURE SetScale (dataPerDivision : CARDINAL);
```

```
    (* Set the scale by which the graph is measured *)
```

```
PROCEDURE PlotPoint( x, y : REAL);
```

```
    (* Plot a scaled point on the graph *)
```

```
PROCEDURE PolarPlotPoint (radius, angle : REAL);
```

```
    (* Moves to a given angle and radius and places a scaled dot at that point.
```

```
        The angle is assumed to be in the currently set units. *)
```

```
END GraphPaper.
```

NOTE: Implementations of this module for both the Macintosh and Windows systems will be given later in the chapter.

In this module, all three basic coordinate systems are supplied, with options to graph in either rectangular or the appropriate polar style, and to turn on a grid system and do scaling if desired. It should be noted that when graphing polar style the point (10, 30) is different in all three coordinate systems.

Note that plotting pixels is handled by one set of procedures, and plotting points on a scaled (and possibly labeled) piece of graph paper is handled by another set of procedures. This permits the user to work in either. The latter is more useful when graphing functions.

The user can change the origin to some other place on the screen without changing the coordinate system. Also, the user may decide whether to graph in degrees or radians as both are supported in all three. Other options include the ability to show a set of axes and to set up labels for the axes, for in real situations these are seldom just "x" and "y." For the advanced programmer who needs to use the graphics window for other purposes, a procedure to pass out a reference to the window can normally be found at a lower level, where the graphics window is made available to the implementation of *GraphPaper*. In addition, the user can obtain the screen dimensions (which depend on the monitor and its settings) and determine the location on the screen and angle for the pen.

The defaults for this module are set to not display the axes or labels, and to employ the standard coordinate system.

The underlying implementation stores the current position on the graph paper and also a direction in which any lines drawn with *LineBy* and any moves made with *MoveBy* will go. This direction can be changed by *Turn* which adds the supplied angle to the stored one, or by *TurnTo* which resets the stored angle. The procedures *Line* and *Move* change the position for the next drawing by a specified number of units, so they supply their own direction and do not change the stored one.

18.3 Using The Module GraphPaper

The following module is a demonstration of some simple aspects of the *GraphPaper* library.

```
MODULE TestGraph;

(* by Joel Schwartz 1998 06 12 to test GraphPaper
revised 1998 07 11 by R. Sutcliffe. *)

FROM GraphPaper IMPORT
  AngleType, CoordSystem, SetCoordSystem, SetAngleType, SetLabels, SetScale,
  ShowAxes, ShowLabels, PlotPoint, PolarPlotPoint, TurnTo, LineBy,
  LineTo, MoveTo, Turn, Home;

FROM RealMath IMPORT
  pi, sin;

VAR
  counter : INTEGER;
  x, y : REAL;

PROCEDURE DrawSquare (side : INTEGER);
(* draws a square of the given side length starting at the current angle; concludes
with the same position and angle as it started *)
VAR
  Count : CARDINAL;
BEGIN
  SetAngleType (rad);
  FOR Count := 1 TO 4
    DO
      LineBy (side);
      Turn (pi/2.0);
    END;
  SetAngleType (deg);
END DrawSquare;

BEGIN
  (* Set up the axes *)
  SetCoordSystem (standard); (* use mathematical system *)
  SetLabels ('X', 'Y');
  SetScale (1); (* one dot per division *)
  ShowAxes;
  ShowLabels;

  (* Test the scaled point plotting routines *)
  (* Sine curve *)
  FOR counter := -4000 TO 4000
    DO
```

```

    x := FLOAT (counter)/100.0;
    y := sin (x);
    PlotPoint (x, y);
END;

(* Circle of radius 5 units with a point at every unit degree angle *)
FOR counter := 1 TO 360
    DO
        PolarPlotPoint (5.0, FLOAT (counter));
    END;

(* Angled line *)
FOR counter := 1 TO 300
    DO
        PolarPlotPoint (FLOAT (counter)/10.0 ,10.0);
    END;

(* Test the pixel plotting routines *)
Home;
TurnTo (30.0);
LineBy (200);
LineTo (200, -100);
LineBy (100);

(* Make a square *)
TurnTo (0.0);
DrawSquare (75);

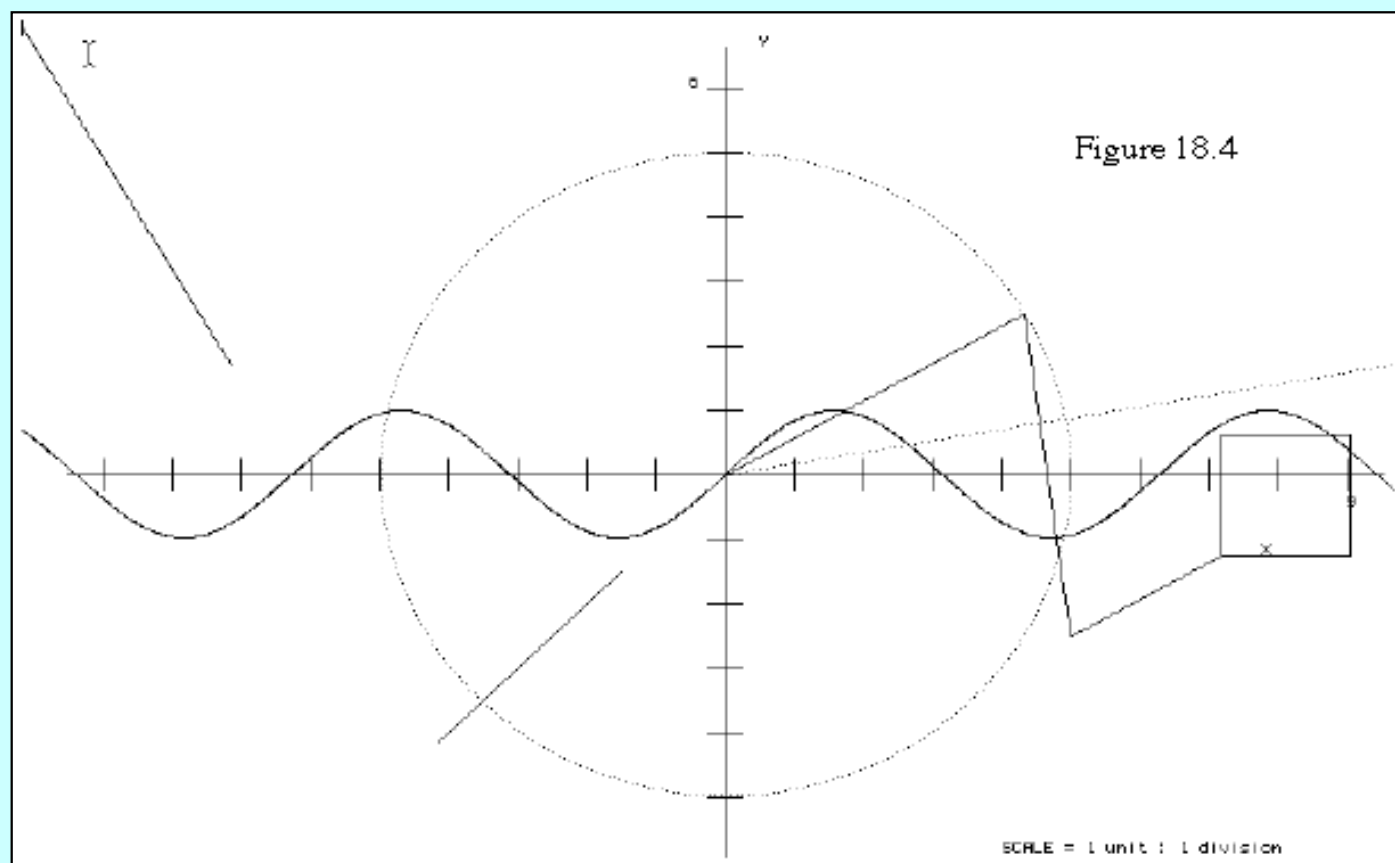
SetCoordSystem (MacWin); (* use Mac system *)
MoveTo (0, 0);
TurnTo (60.0);
LineBy (250); (* should come from top left corner *)

SetCoordSystem (bearing); (* use bearing system *)
MoveTo (-60, -60);
Turn (225.0);
LineBy (150); (* should be in bottom left quadrant *)

END TestGraph.

```

The reader should verify the correctness of each routine in the following (much reduced) screen shot of the output.



[Contents](#)

18.4 Recursive Drawing--Fractals

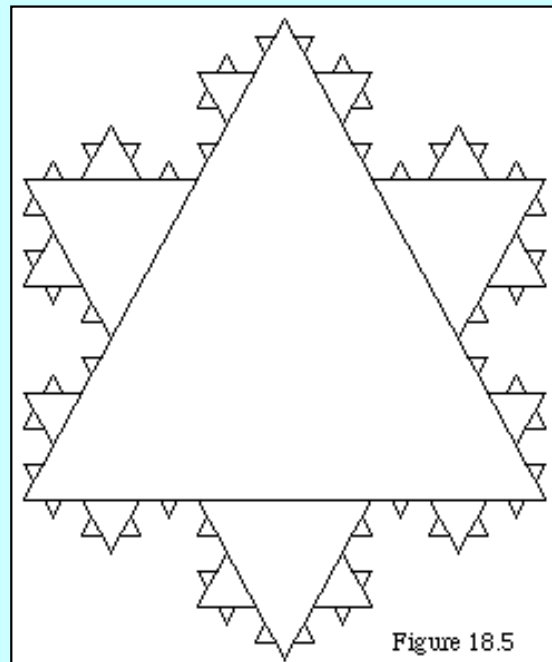
Recursively repetitive figures arise naturally and in a variety of pattern and design work. Such figures are called fractals. The essence of a fractal is that it has the same kind of micro structure as it has a macro structure.

For instance, if the West coast of British Columbia, Canada is viewed from outer space, it has an irregular shape with numerous inlets folded into the larger outline throughout this length. Looked at from a lower altitude airplane, these larger inlets themselves have an irregular and folded structure, and if one were to look closer still down on the ground, these too would be seen to have inlets and folds. That is to say, a fractal is recursive, at least to some number of levels. (In the case of the coastline, the recursion would break down to some extent after one began looking at individual grains of sand under sufficient magnification.)

A number of standard fractal designs can be constructed using recursive drawing techniques, and these are detailed below.

18.4.1 Snowflake Fractals

One of the easiest types of fractals to draw is a snowflake.



The snowflake fractal shown in figure 18.5 is based on triangles and has four levels of recursion or sizes of triangle. Assuming that the drawing starts at the lower left corner and with the direction to the right or East, the drawing of this fractal could be formulated in terms of pseudocode for a three sided figure as follows:

```
draw fractal =  
  select standard coordinates  
  select levelsofrecursion  
  select mainlength  
  select starting point on screen  
  calculate turn angle based on number of sides (60 degrees for triangles)  
  drawfigure
```

```
drawfigure (mainlength, recursionlevel) =  
  save current position and direction
```

```

turnby -60 degrees; (* turn out and away first *)
drawsegment (sidelen, curreclevel); (* go draw a side *)
for count := 2 to numsides - 1
    do (* then turn and draw each successive side except the last *)
        turnby 120 degrees; (* turn back in *)
        drawsegment (sidelen, curreclevel);
    end;

if curreclevel < recursionlevel
    then (* fill in the missing side -- central third *)
        turnto (olddir);
        moveto (oldx, oldy);
        lineto (curx, cury);
    else (* go draw the last side *)
        turnby 120 degrees;
        drawsegment (sidelen, curreclevel);
    end;

drawsegment (dist, curreclevel) =
    disttodraw := dist / 3.0;
    if curreclevel = 0
        then
            lineby (dist);
        else
            drawsegment (disttodraw, curreclevel - 1);
            drawfigure (disttodraw, curreclevel - 1);
            drawsegment (disttodraw, curreclevel - 1);
        end;
end;

```

The basic idea is that drawing a figure (say, a triangle) consists of drawing three successive segments with the appropriate angle between. Drawing each segment consists of drawing a third of that segment, then drawing the original figure at one third size as the middle third (unless one is past the appropriate number of recursion levels) then drawing the last third of the segment. In the code that follows, however, this has been modified to allow for the angle to be calculated for the number of sides. The number of recursion levels and number of sides is initially set to reproduce the drawing above, but both can easily be changed. Here is the code:

MODULE Snowflake;

```

(* by R. Sutcliffe 1996 01 29
   to illustrate Macintosh quickdraw
   revised by Joel Schwartz and R. Sutcliffe to depend on GraphPaper.
   This program draws a recursive snowflake with a specified number of sides and level
   of recursion. Last revision 1998 06 16 *)

```

FROM GraphPaper **IMPORT**

```

    LineBy, LineTo, TurnTo, MoveTo, GetDimensions, SetCoordSystem, CoordSystem,
    GetLocation;

```

CONST

```

    (* set up these first two constants; the rest depend on them. *)
    recursionLevel = 3;
    numSides = 3;

```



```

degreesAvailable = 180 * (numSides - 2);
(* how many degrees around and back *)
turnAngle = degreesAvailable DIV numSides;
(* turn for each corner *)

```

VAR

```

curDir : INTEGER;
curX, curY : INTEGER; (* to globally store position and direction *)
dimX, dimY : INTEGER; (* the dimensions *)

```

```

PROCEDURE DrawSegment (dist : REAL; curRecLevel : INTEGER); FORWARD;
(* omit if compiler is multi pass *)

```

```

PROCEDURE DrawFigure (sideLen : REAL; curRecLevel : INTEGER);

```

VAR

```

oldX, oldY : INTEGER;
oldDir : INTEGER;
count : CARDINAL;

```

BEGIN

```

(* store current directions *)
oldX := curX;
oldY := curY;
oldDir := curDir;
curDir := curDir - turnAngle;
TurnTo (FLOAT (curDir)); (* turn out and away first *)
DrawSegment (sideLen, curRecLevel); (* go draw a side length *)
FOR count := 2 TO numSides - 1
    DO (* then turn and draw each successive one but the last *)
        curDir := curDir - turnAngle + 180;
        TurnTo (FLOAT (curDir));
        DrawSegment (sideLen, curRecLevel);
    END;

```

```

IF curRecLevel < recursionLevel

```

```

    THEN (* fill in the missing side -- central third *)
        curDir := oldDir; (* go back to old position *)
        TurnTo (FLOAT (curDir));
        MoveTo (oldX, oldY);
        LineTo (curX, curY); (* and draw to new one *)
    ELSE (* draw another side like the ones in the loop above *)
        curDir := curDir - turnAngle + 180;
        TurnTo (FLOAT (curDir));
        DrawSegment (sideLen, curRecLevel);
    END;

```

```

END DrawFigure;

```

```

PROCEDURE DrawSegment (dist : REAL; curRecLevel : INTEGER);
(* recursively draw a third of the segment as another one
  then a third size figure on the next segment
  then another segment after that. *)

```

VAR

```

distToDraw : REAL;

```

```

BEGIN
  distToDraw := dist / 3.0;
  IF curRecLevel = 0
    THEN
      LineBy (TRUNC(dist));
      GetLocation (curX, curY); (* reset local storage of position *)
    ELSE
      DrawSegment (distToDraw, curRecLevel - 1);
      DrawFigure (distToDraw, curRecLevel - 1);
      DrawSegment (distToDraw, curRecLevel - 1);
    END;
END DrawSegment;

VAR
  mainLength : INTEGER;

BEGIN (* main *)
  (* calculate a length that will fill the screen nicely *)
  SetCoordSystem (MacWin);
  GetDimensions (dimX, dimY);

  mainLength := 3 * dimY/(2 * numSides); (* arrived at by experimenting *)

  (* starting point is "lower left" of first level recursive figure. *)
  curDir := 0;
  curX := (dimX - mainLength) / 2;
  curY := (dimY + mainLength) / 2;

  (* above formula not bad except for three sides, so adjust *)
  IF numSides = 3
    THEN
      curY := curY - 75;
    END;
  MoveTo (curX, curY);
  DrawFigure (FLOAT (mainLength), recursionLevel);

END Snowflake.

```

18.4.2 A Tree Fractal

The program in this section recursively draws a tree fractal. The idea is that it draws a trunk in the current direction, then two trees at half size at forty-five degrees on either side of the top of the trunk, then a tree at three-quarters size in the same direction as the original trunk. After the first vertical trunk, the remaining trees can be thought of as branches. Recursion is exited if the length of the trunk is less than two units.

```

MODULE DrawTree;
(* Program by R. Sutcliffe
  to illustrate GraphPaper
  revised 1998 06 16
  recursively draws a fractal based tree. *)

FROM GraphPaper IMPORT
  MoveTo, LineBy, Turn, TurnTo, GetLocation;

```

```

PROCEDURE DrawATree (trunkLength : CARDINAL);
VAR
    baseX, baseY : INTEGER;

BEGIN
    IF trunkLength < 2
        THEN
            RETURN
        END (* if *);
    LineBy (trunkLength);
    GetLocation (baseX, baseY);
    Turn (45.0);
    DrawATree (trunkLength DIV 2); (* 1/2 current size right *)
    MoveTo (baseX, baseY);
    Turn (-90.0);
    DrawATree (trunkLength DIV 2); (* 1/2 current size left *)
    MoveTo (baseX, baseY);
    Turn (45.0);
    DrawATree (3 * (trunkLength DIV 4)); (* 3/4 current size up *)
    MoveTo (baseX, baseY);
END DrawATree;

BEGIN
    (* Find a good place to start drawing the tree. *)
    MoveTo (0,-250);
    TurnTo (90.0);
    DrawATree (128);
END DrawTree.

```

Here is a screen shot of the tree that is drawn:

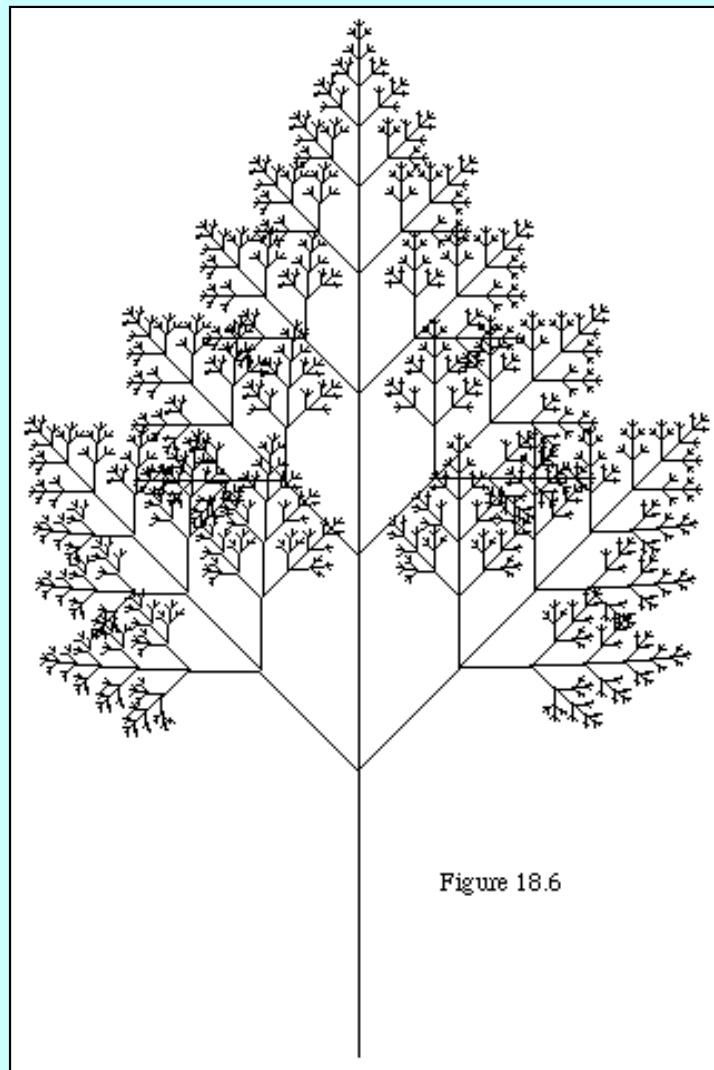


Figure 18.6

18.4.3 Singly-Recursive Snowflake-like Fractals

The tree fractal was drawn in a different way than the snowflake that preceded it. Rather than use a pair of mutually recursive procedures, it employed only a single procedure that drew segment-right fractal-left fractal-central fractal. Snowflakes can be drawn in a similar manner, without the filled in sides, and by using a singly-recursive procedure as in the following:

```
MODULE DrawFractal;
```

```
(* Program to draw a fractal
   by R. Sutcliffe
   revised 1998 06 16 *)
```

```
FROM GraphPaper IMPORT
  LineBy, Turn, TurnTo, MoveTo;
```

```
TYPE
```

```
  CardArray = ARRAY [0..12] OF CARDINAL;
```

```
CONST
```

```
  order = 6; (* fits on most screens *)
```

```
  (* make a constant array with the powers of two for the line lengths. *)
```

```
  Power = CardArray {1,2,4,8,16,32,64,128,256,512,1024,2048,4096};
```

```
PROCEDURE Fractal (level: CARDINAL);
```

```
  (* Recursive drawing of levelth fractal *)
```

```

BEGIN
  Turn (-60.0);
  (* at all but the outer level, we do segment-fractal-segment *)
  LineBy (Power [level]);
  IF level > 1
    THEN
      Fractal (level - 1);
    END;  (* if *)
  LineBy (Power [level]);
  Turn (120.0);
  LineBy (Power [level]);
  IF level > 1
    THEN
      Fractal (level - 1);
    END;  (* if *)
  LineBy (Power [level]);

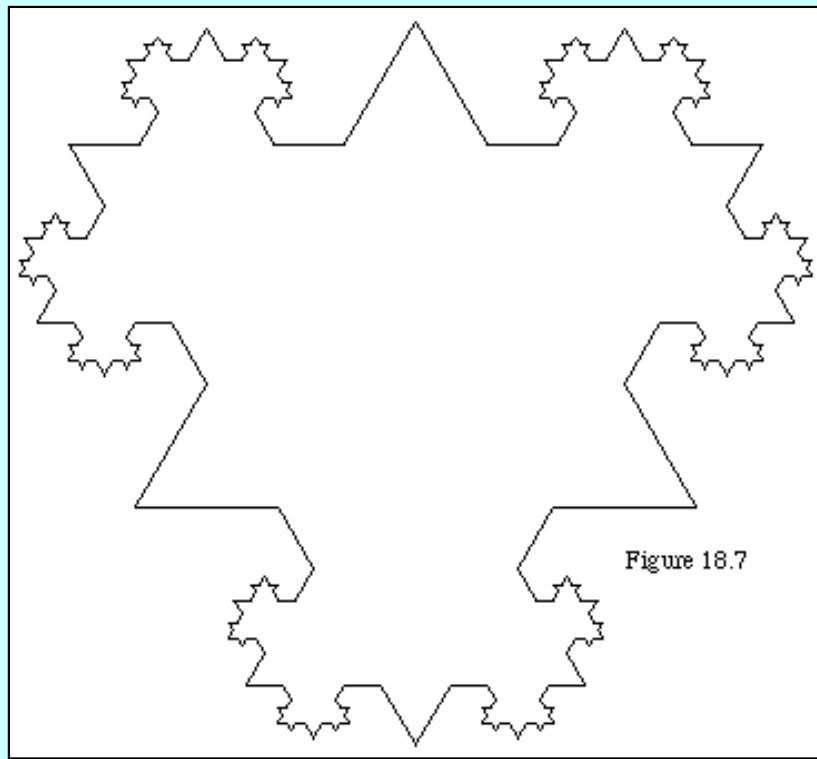
  (* at the outer level we complete the figure so it is closed *)
  IF level = order
    THEN
      Turn (120.0);
      LineBy (Power [level]);
      IF level > 1
        THEN
          Fractal (level - 1);
        END;  (* if *)
      LineBy (Power [level]);
    END;
  Turn (-60.0);
END Fractal;

VAR
  ch : CHAR;

BEGIN  (* DrawFractal *)
  (* orient on the page the way we want *)
  TurnTo (180.0);
  MoveTo (100, -50);
  Fractal (order);
END DrawFractal.

```

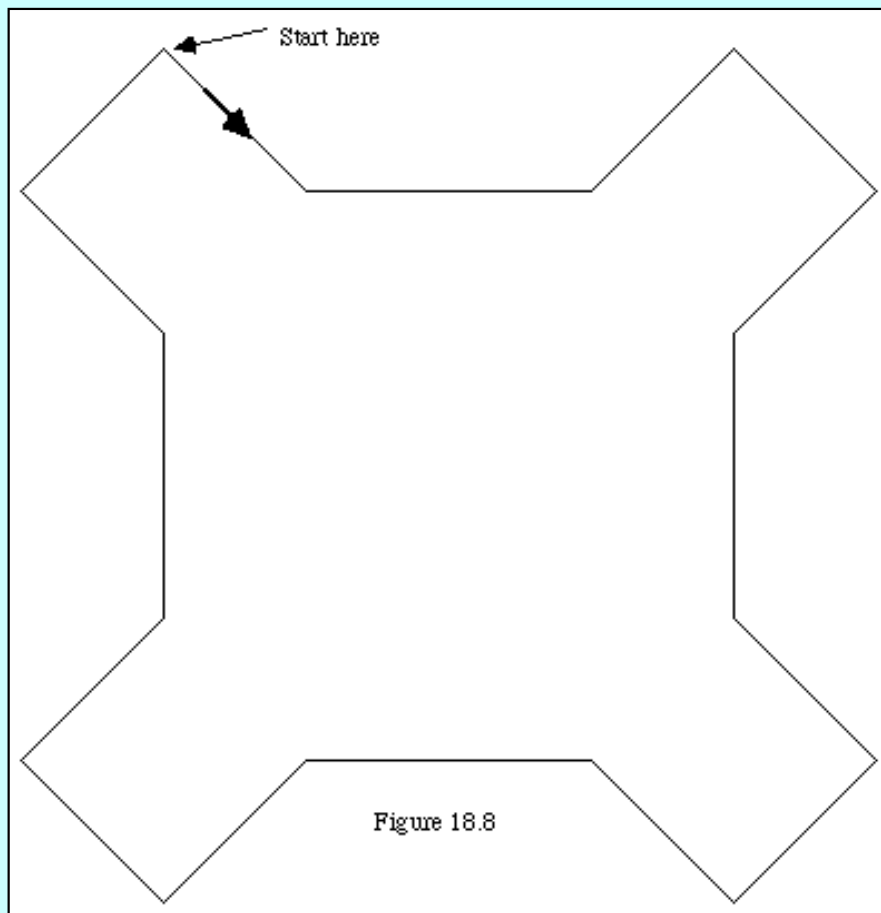
This program produces the following version of the snowflake fractal:



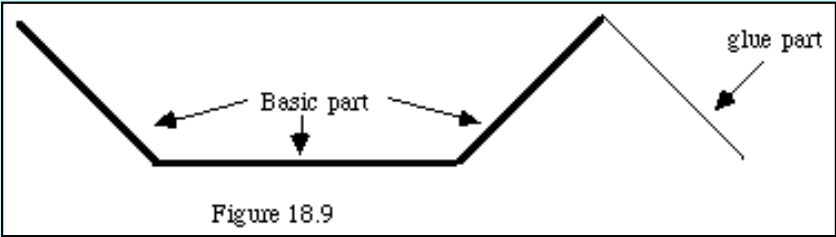
18.4.4 Sierpinski's Curve

The most basic form of the fractal known as Sierpinski's curve consists of a square with squares on its corners in the manner shown in figure 18.8. Assuming standard coordinates with the origin at the centre, and given that the basic length on one of the corners is *dist*, drawing is done in four steps starting from the indicated point. (Recall that *Line* draws to a point with the given displacement from the current point.)

top: Line (*dist*, -*dist*); Line (*2 * dist*, 0); Line (*dist*, *dist*);
 followed by Line (*dist*, -*dist*) to glue to the next piece
right: Line (-*dist*, -*dist*); Line (0, -*2 * dist*); Line (*dist*, -*dist*);
 followed by Line (-*dist*, -*dist*) to glue to the next piece
bottom: Line (-*dist*, *dist*); Line (-*2 * dist*, 0); Line (-*dist*, -*dist*);
 followed y Line (-*dist*, *dist*) to glue to the next piece
left: Line (*dist*, *dist*); Line (0, *2 * dist*); Line (-*dist*, *dist*);
 followed by Line (*dist*, *dist*) to glue to the next piece



That is, each procedure has three steps that draw a copy (in one of four rotations) of what is shown in figure 18.9, followed by a connector to the next part.



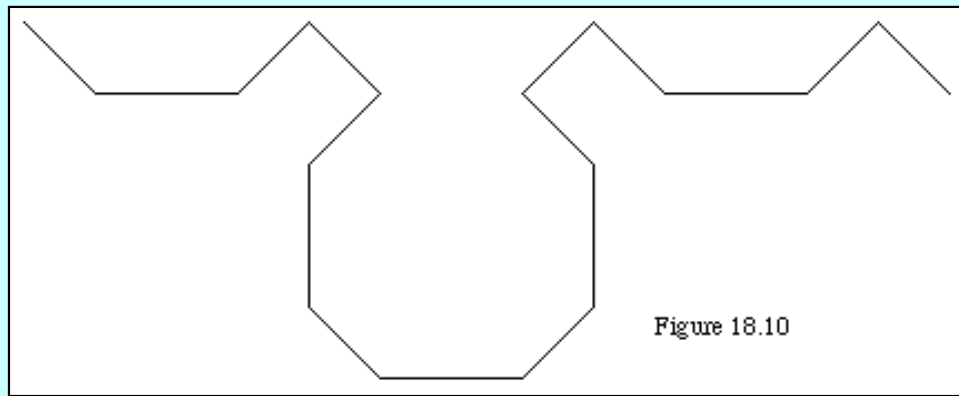
The fun begins when this process is made recursive, in the sense that the prototype procedure *Top* becomes:

```

Top =
  IF level > 0
  THEN
    Top (level - 1);  Line (dist, -dist);
    Right (level - 1); Line (2 * dist, 0);
    Left (level - 1);  Line (dist, dist);
    Top (level - 1)
  END

```

If this is drawn at level two, the top (including the glue) becomes figure 18.10, and this pattern can be replicated around the basic square using the other three procedures in turn.



With a careful calculation of starting points, the level two diagram can be superimposed on top of the level one diagram, and indeed as many levels can be drawn as desired. In the code that is given below, three levels have been drawn by the main loop. Care must be taken not to ask for too many levels, or one could find the program in an infinite loop unable to compute the next smaller side length.

```

MODULE Sierpinski;
(* Heavily adapted from a version in Wirth's PIM-2
   by R. Sutcliffe to illustrate GraphPaper
   last revision: 1998 06 22 *)

FROM GraphPaper IMPORT
    Line, LineTo, MoveTo;

CONST
    size = 512; (* use a power of two that fits on screen. *)
    numOfLevels = 3; (* set to number smaller than the power in the last line; if 512;
use 8 or less here *)

VAR
    dist : INTEGER;

(* Each of the mutually recursive procedures draws one side of the basic square
figure in the orientation given by its name. At the lowest level, this is an angled
part, a straight part, and then an angled part, concluding with a "larger" copy of
itself at a lower level to glue to the next piece. *)

PROCEDURE Top (level: CARDINAL);
BEGIN
    IF level > 0
    THEN
        Top (level - 1);    Line (dist, -dist);
        Right (level - 1); Line (2 * dist, 0);
        Left (level - 1);  Line (dist, dist);
        Top (level - 1)
    END
END Top;

PROCEDURE Right (level: CARDINAL);
BEGIN
    IF level > 0
    THEN
        Right (level - 1);    Line (-dist, -dist);
        Bottom (level - 1); Line (0, -2 * dist);

```



```
Top (level - 1);      Line (dist, -dist);
Right (level - 1)
```

END

END Right;

PROCEDURE Bottom (level: **CARDINAL**);

BEGIN

IF level > 0

THEN

```
Bottom (level - 1); Line (-dist, dist);
Left (level - 1);   Line (-2 * dist, 0);
Right (level - 1);  Line (-dist, -dist);
Bottom (level - 1)
```

END

END Bottom;

PROCEDURE Left (level: **CARDINAL**);

BEGIN

IF level > 0

THEN

```
Left (level - 1);   Line (dist, dist);
Top (level - 1);    Line (0, 2 * dist);
Bottom (level - 1); Line (-dist, dist);
Left (level - 1);
```

END

END Left;

VAR

```
level   : CARDINAL;
startX, startY : INTEGER;
```

BEGIN

```
dist := size DIV 4; (* initial segment distance *)
startX := 0;
startY := dist;
level := 0;
```

(* The following loop draws the level one figure and then overlays it with the level two figure, and so on until the number of specified levels have all been drawn. *)

REPEAT

INC (level); (* begin at one and work up to max set. *)

DEC (startX, dist); (* set up new corner to start *)

dist := dist / 2; (* and basic distance for next level *)

INC (startY, dist);

MoveTo (startX, startY);

(* We end up d units left and d/2 units up from last start for next one. *)

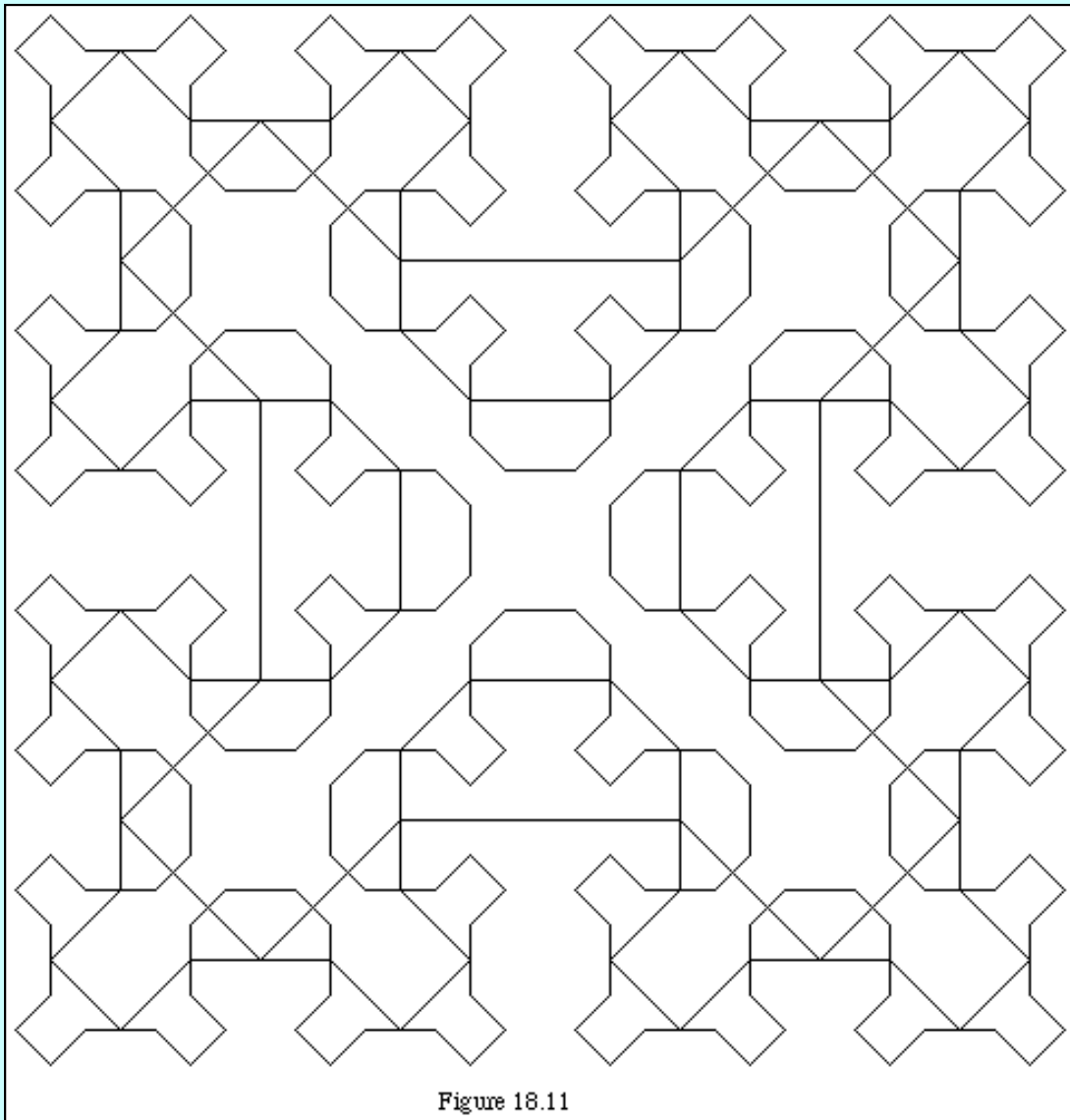
(* In the main program the figure is drawn with the four "sides" in succession, each followed by a line segment to glue to the next piece. *)

```
Top (level);      Line (dist, -dist);
Right (level);    Line (-dist, -dist);
Bottom (level);   Line (-dist, dist);
Left (level);     Line (dist, dist);
```

```
UNTIL level = numOfLevels
```

```
END Sierpinski.
```

When this version of the code was compiled and run, the following figure was produced.



Careful examination will reveal the three levels drawn. More can be called for, but the diagram becomes somewhat cluttered for illustrative purposes.

[Contents](#)

18.5 String Art

A variety of interesting patterns can be achieved by the simple expedient of drawing lines from one side of a shape to the other. Such patterns are commonly known as string art, after the corresponding figures made by attaching coloured threads to nails driven into a board. If the distance between the lines relative to the size of the drawing rectangle is carefully calculated, interference effects can be obtained. The program in this section is presented with little explanation as a simple example of such patterns, and the student is invited to elaborate. The shape can be changed or the step size altered to produce somewhat different effects.

Some things that are different from previous examples are:

- the MacWin coordinate system is used instead of the standard one
- the dimensions of the screen are obtained
- the drawing area is reduced to a square for symmetry

MODULE Moire;

```
(* Draw a Moire pattern in a graphic window
   by R. Sutcliffe
   modified 1998 06 22 *)
```

FROM GraphPaper **IMPORT**

```
CoordSystem, SetCoordSystem, GetDimensions, MoveTo, LineTo;
```

CONST

```
step = 5;  (* determines the granularity of the pattern *)
(* ensure space left is even number of steps *)
(* leave margin; make it square for easy coordinates *)
leftAndTop = 8 * step;
(* do a border within the window all around *)
border = 4 * step;
first = leftAndTop + border; (* begin right in the corner *)
```

VAR

```
maxX, maxY, startX, startY, endX, endY,
bottomAndRight, last : INTEGER;
```

BEGIN

```
SetCoordSystem (MacWin);
GetDimensions (maxX, maxY);
(* set lower right corner to maximum square that will fit on the screen *)

bottomAndRight := maxY;
IF bottomAndRight > maxX
  THEN
```

```

    bottomAndRight := maxX;
END;

(* ensure size is multiple of number of steps *)
bottomAndRight := bottomAndRight - (bottomAndRight MOD step);

(* same border as at topleft *)
last := bottomAndRight - border;
startX := first;
endX := last;

(* draw edge to edge lines in both dimensions *)
FOR startY := first TO last BY step
    DO
        endY := last - startY + first;
        MoveTo (startX, startY);
        LineTo (endX, endY);
    END; (* for *)
startY := first;
endY := last;
FOR startX := first TO last BY step
    DO
        endX := last - startX + first;
        MoveTo (startX, startY);
        LineTo (endX, endY);
    END; (* for *)

END Moire.

```

When the program *Moire* was run with these settings, the pattern produced was as shown in figure 18.12.

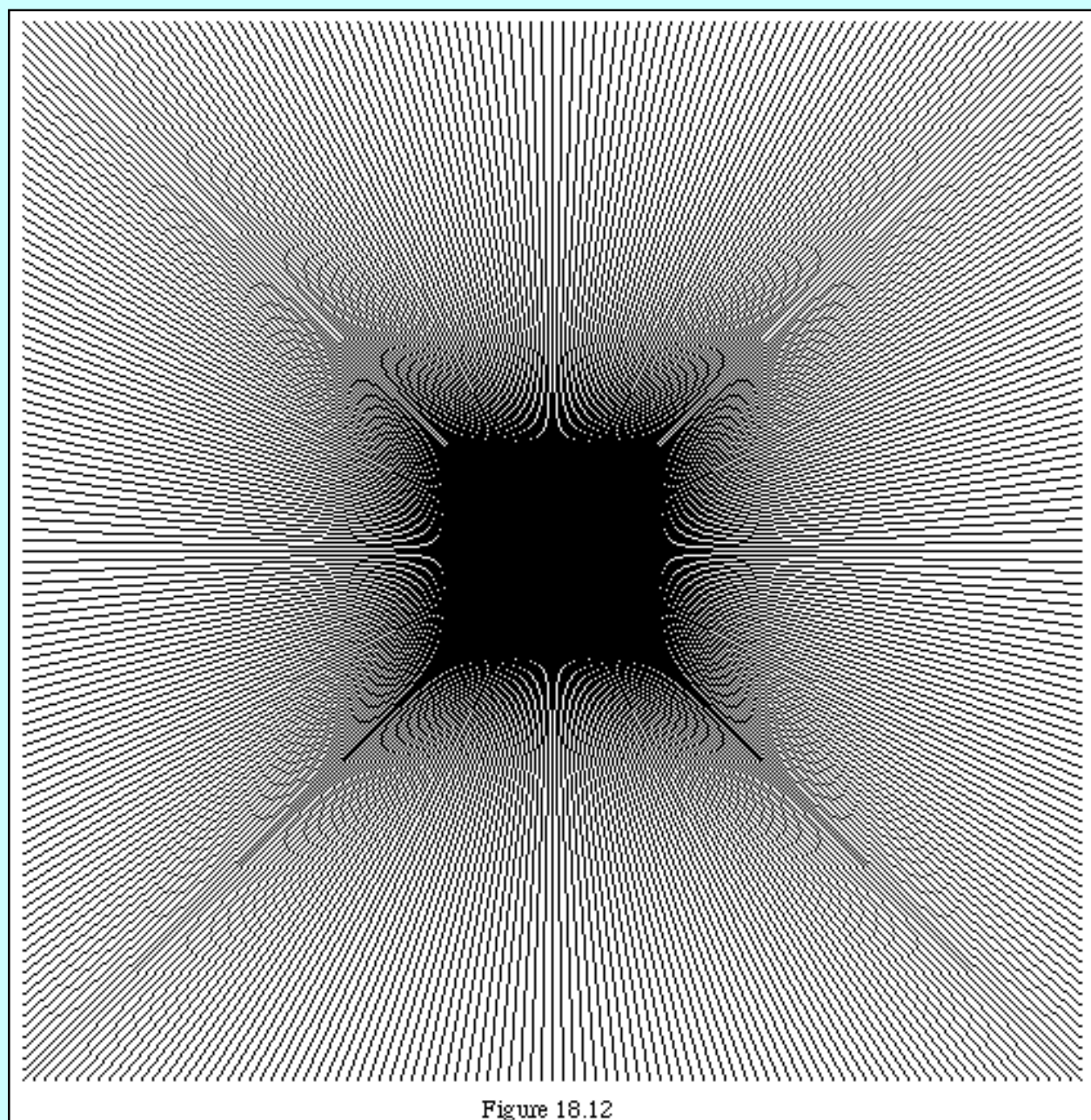


Figure 18.12

[Contents](#)

18.6 An Extended Example--Implementing GraphPaper

Two kinds of functionality have to be combined into the working package in order to implement *GraphPaper*. First, one has to be able to get a window open that can be used for the purpose, and second, one has to have available some drawing routines from a variety of system specific modules. Actually implementing the routines in the definition module then turns out to be relatively routine.

Thus, in the implementations that follow, the functionality is divided between two modules--one that opens and prepares a window for drawing, and the other the actual implementation of *GraphPaper*.

18.6.1 Defining the Module GraphWindow

The only purpose of this module is to isolate the task of preparing the graphics window from the task of implementing the procedures in *GraphPaper*. The MacOS versions are relatively simple, and in the initial implementation one incorporated into *GraphPaper* to make a single module. However, the Windows version was so cumbersome that the details obscured those of *GraphPaper* so the two were separated and the module *GraphWindow* created. Here is the definition:

DEFINITION MODULE GraphWindow;

```
(* Design and Macintosh implementattion by R. Sutcliffe
   Windows implementation by Joel Schwartz
   Last revision: 1998 07 07 *)
```

```
(* This module obtains and passes to applications that need it a simple graphics
   window and its dimensions. Some applications will need only to import this module,
   and possibly get the window dimensions, as graphing takes place in the current
   grafport anyway, and this module will set that port, so GetWindow may not have to be
   imported. *)
```

IMPORT (* MacOS *) Quickdraw; (* Windows: IMPORT WIN32; *)

```
(* for convenience we export the type of the reference; this makes it more compatible
   to the
```

```
Windows version where we import the HDC type and define WindowRef to be an HDC. *)
```

TYPE

```
WindowRef = Quickdraw.WindowRef; (* Windows: WindowRef = WIN32.HDC *)
```

PROCEDURE GetWindow () : WindowRef;

PROCEDURE GetWDimensions (VAR width, height : **INTEGER**);

END GraphWindow.

The Windows version needs the minor change as noted--another reason to separate this functionality from that of *GraphPaper*. One does not want OS-specific items like *WindowRef* turning up in a top level general definition module. If the user of *GraphPaper* needs this reference, perhaps to do some annotating of her own in the window, it is available, but at this lower level. A full listing of the Windows version with the small revisions is given in [section 18.6.5](#) for reference purposes.

18.6.2 Implementing GraphWindow in MacOS

There are a variety of ways to get a window open for graphing in MacOS. In the author's implementation of the ISO library, the I/O system opens a window that comes complete with menu bar and a quit command so that no loop is needed to wait for a key press. The implementation that follows imports *STextIO*, thereby forcing the opening of a window for text (which can also be used to draw) and essentially "steals" this window, setting its graphics port as the current one, and allowing *GraphPaper* to go ahead and draw in it.

```
IMPLEMENTATION MODULE SGraphWindow;
(* Design and Macintosh implementation by R. Sutcliffe
   Ultra simple version that steals a window from elsewhere
   Last revision: 1998 07 07 *)

FROM Quickdraw IMPORT
    SetPort;

FROM MacWindows IMPORT
    FrontWindow;

FROM Types IMPORT
    Rect;

IMPORT STextIO; (* force a window *)

VAR
(* Graphics Variables *)
    graphRect  : Rect;
    gWindow    : WindowRef;
    lwidth, lheight : INTEGER;

(*----- Window Related Procedures ----- *)

PROCEDURE GetWindow () : WindowRef;

BEGIN
    RETURN gWindow;  (* Return the stored window reference *)
END GetWindow;

PROCEDURE GetWDimensions (VAR width, height : INTEGER);

BEGIN
    width := lwidth;
    height := lheight;
END GetWDimensions;

BEGIN (* main *)
    gWindow := FrontWindow (); (* steals the one STextIO puts up *)
    SetPort (gWindow);
    graphRect := gWindow^.visRgn^.rgnBBox;
    lwidth := graphRect.right - graphRect.left;
    lheight := graphRect.bottom - graphRect.top;
END SGraphWindow.
```

The details of the record structure pointed to by a *WindowRef* are not given here. Suffice it to say that once the window has been opened by *STextIO*, it is easy to get its reference using *FrontWindow* and then take information concerning the window size from that data structure as shown.

If the same trick is tried without using the author's implementation of the ISO library, the *Terminal* program that supplies services to the I/O library probably will not have such features as a *Quit* command on the menu bar, and the window will simply vanish as soon as the program has run. In that event, the same implementation as above may be employed, but with the addition of a termination clause. One would include the lines:

```
FROM Keyboard IMPORT
    BusyRead;
FROM Events IMPORT
    Button;

FINALLY
    WriteString ("touch any key to exit");
REPEAT
    BusyRead (ch); (* delay until keypress or mouse button *)
UNTIL (ch # 0C) OR Button ();
```

The addition of the use of *Button* is to ensure that such items as the control, option, and command buttons (which are not keys) will also exit the program if touched.

If the user desires to create graphics windows without using the ones available by stealing from a *Terminal* module employed by *STextIO* then a little more work is necessary, as the module would actually have to open the window itself. Rather than immediately give a variation on *GraphWindow* here that does this, the information necessary is provided in the following module, which opens a graphics window directly and then draws some rectangles and ovals in it. This module does **not** use *GraphPaper*, only built in routines. Extracting the necessary routines from it to produce a stand-alone version of *GraphWindow* for the MacOS is shown later in this section. The only thing this particular module does, besides open the window, is set a pen size to a two-by-two rectangle rather than the usual one pixel each way, and then draw a few figures. Of course, it would be easy to port this module to *GraphPaper* either by importing the routines for framing rectangles and ovals directly, or by writing them in a client of *GraphPaper*.

```
MODULE DrawRectangles;

(* by R. Sutcliffe
   to demonstrate simple graphics on the MacOS using QuickDraw directly
   illustrates simple actions such as opening and preparing a graphics window
   revised 1996 07 14 *)

FROM SYSTEM IMPORT
    ADR, CAST;
FROM Keyboard IMPORT
    BusyRead;
FROM Quickdraw IMPORT
    qd, PenSize, WindowRef, InsetRect, SetPort, SetRect, FrameRect, FrameOval;
FROM MacWindows IMPORT
    WindowRecord, NewWindow, documentProc;
FROM Types IMPORT
    Rect;
FROM Events IMPORT
    Button;

CONST
    inFront = CAST (WindowRef, -1); (* Special constant used when opening window to say
it goes on top. *)
    deltaXY = 20;
    penH = 2;  penV = 2;
```



```

VAR
    curRect, windRect : Rect;
    left, right, top, bot : CARDINAL;
    ch : CHAR;
    wRecord: WindowRecord;
    myWindow: WindowRef;

BEGIN
    windRect := gd.screenBits.bounds;  (* find out screen size *)
    InsetRect (windRect, 50, 50);  (* and make an inset from this *)

    (* now, open a named window using this rect *)
    (* we have to pass the address of a WindowRecord, a rectangle to draw in, a title
    for the window, TRUE to make the window visible, the name of the procedure that draws
    a standard document window, the constant inFront to put it on top, FALSE to indicate
    it has no goAway box and a zero for the refCon.
    *)
    myWindow := NewWindow
        (ADR (wRecord), windRect, 'DrawRectangles', TRUE, documentProc, inFront, FALSE,
    0);

    SetPort (myWindow); (* establish graphics port *)

    (* and the relative coordinates of its size *)
    left := 0;
    right := windRect.right - windRect.left;
    top := 0;
    bot := windRect.bottom - windRect.top;
    PenSize (penH, penV);

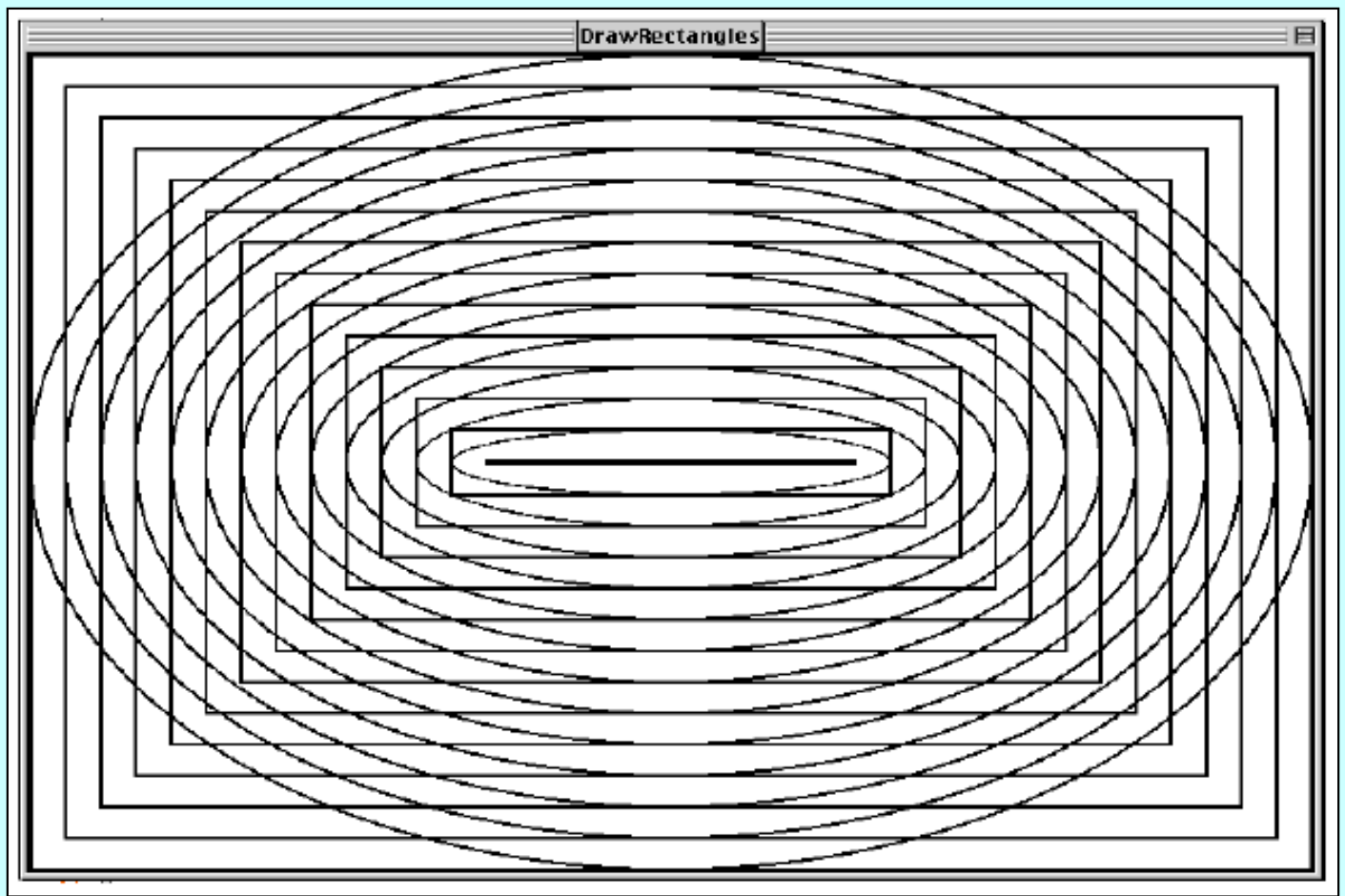
    (* Now, draw a series of contained rectangles and ovals *)
    WHILE left < right
        DO
            SetRect (curRect, left, top, right, bot);
            FrameRect (curRect);
            FrameOval (curRect);
            INC (left, deltaXY);
            INC (top, deltaXY);
            DEC (right, deltaXY);
            DEC (bot, deltaXY);
        END;

    FINALLY
        REPEAT
            BusyRead(ch); (* delay until keypress or mouse button *)
        UNTIL (ch # 0C) OR Button ();

    END DrawRectangles.

```

Here is a reduced screen shot of the output from this simple module.



This module too has a dependency on one of the author's own modules, but the concept has been presented before, and the reader should consult [section 8.4.1](#) for the details of implementing *Keyboard*.

It is not the purpose of this section to give a detailed description of the MacOS toolbox routines, but some careful study of the simple ones employed here should go a long way toward assisting in understanding some of the basics.

Collecting some of these ideas into one module provides a better implementation of *GraphWindow*, this one not dependent on *Keyboard*. Note, however, that this is still not a full-blown MacOS application. It has no menu bar, and it is not possible to switch out of applications based on this window and then back in again, for the graphics drawn on the window will not reappear once they have been erased. Once the client program is finished, the termination clause in this module takes over, and waits for a keypress or a mouse click. In this version, other button clicks (CMD, OPT, CNTRL, SHFT) are left alone so that the normal screen shot process can take place.

This implementation also hints at the very different style of programming needed in a graphics user interface such as MacOS or Windows. When the program is idling, it does so in a loop (called a main event loop) that waits for an event to take place. When one does, the program decides whether or not to handle that event. In this case, the *MainEventLoop* is a very simple one and contains all the code needed to handle the only events of interest. Much more would need to be done in a fully developed application, and the usual method is to have separate procedures for each kind of event and dispatch control to the handling procedures after detecting which event has occurred. The purpose of the *gSleep* variable is to give the system some time to respond to events as well, keeping the system clock and other such things up-to-date. Very few systems will lack colour; those that do will produce a window anyway, but it will not be possible to import routines from *Quickdraw* to change the pen colour for drawing.

IMPLEMENTATION MODULE *GraphWindow*;

```
(* Implements a very simple graphics window on the Mac. There is no menu, and any
keypress or mouse click quits.
```

```
  Screenshots using shift-cmd-4 work, so the results from clients can be captured.
```

```
Design and Macintosh implementation by R. Sutcliffe
```

```
Last revision: 1998 07 22 *)
```

```

FROM SYSTEM  IMPORT
    ADR, CAST;
FROM Types  IMPORT
    OSErr, Rect;
FROM OSUtils IMPORT
    SysEnvirons, SysEnvRec;
FROM Quickdraw IMPORT
    qd, SetPort;
FROM MacWindows IMPORT
    WindowRecord, NewCWindow, NewWindow, documentProc;
FROM Events IMPORT
    EventRecord, GetCaretTime, WaitNextEvent, keyDown, mouseDown, everyEvent;

VAR
    (* Graphics Variables to pass out *)
    gWindow : WindowRef;
    lwidth, lheight : INTEGER;
    (* internal variables *)
    windRect, graphRect : Rect;
    gSleep : INTEGER;
    wRecord : WindowRecord;

CONST
    inFront = CAST (WindowRef, -1); (* special constant to display window *)

    (*-----Exported Procedures ----- *)

PROCEDURE GetWindow () : WindowRef;

BEGIN
    RETURN gWindow; (* Return the stored window reference *)
END GetWindow;

PROCEDURE GetWDimensions (VAR width, height : INTEGER);

BEGIN
    width := lwidth;
    height := lheight;
END GetWDimensions;

    (* -----*)

    (* Initialize everything for the program, make sure we can run. *)
PROCEDURE Initialize;
VAR
    error : OSErr;
    theWorld : SysEnvRec;
    colour : BOOLEAN;

BEGIN
    (* Test the computer to be sure we can do color.  If not we could crash.  Note that
    a client program should do its own test before assuming that the window has colour.
    *)

```

```

error := SysEnvirons (1, theWorld);
colour := theWorld.hasColorQD;

(* The run time system initializes all the needed managers. *)

(* Make a new window for drawing in.  The window is inset from full screen size. *)

windRect := qd.screenBits.bounds; (* get overall dimensions *)
INC (windRect.top, 40); (* drop down from the top *)
IF colour (* make it a colour window if we can *)
    THEN
        gWindow := NewCWindow
            (ADR (wRecord), windRect, 'GraphicsWindow', TRUE, documentProc,
             inFront, FALSE, 0);
    ELSE (* otherwise get one anyway--only on old Macs *)
        gWindow := NewWindow
            (ADR (wRecord), windRect, 'GraphicsWindow', TRUE, documentProc,
             inFront, FALSE, 0);
    END;
SetPort (gWindow);          (* set window to be current graf port *)
graphRect := gWindow^.visRgn^.rgnBBox; (* get local copies of length and width *)
lwidth := graphRect.right - graphRect.left;
lheight := graphRect.bottom - graphRect.top;
(* set idle time used by WaitNextEvent *)
gSleep := GetCaretTime ();

END Initialize;

PROCEDURE MainEventLoop;
VAR
    theEvent : EventRecord;
BEGIN
    LOOP (* wait for something to happen *)
        IF WaitNextEvent (everyEvent, theEvent, gSleep, NIL)
            THEN
                IF (theEvent.what = keyDown) OR (theEvent.what = mouseDown)
                    THEN
                        EXIT (* on any keypress or mouse press *)
                    END
                END (* if WaitNextEvent *)
            END (* loop *)
    END MainEventLoop;

BEGIN
    Initialize;
FINALLY
    MainEventLoop;
END GraphWindow.

```

Once graphics windows are available, one can do some interesting things in them. The following program, often supplied for beginners on the MacOS, draws small coloured balls on the screen at random locations and with random colours. Its only purpose here is to demonstrate *GraphWindow* and to give the students a few more graphics tools to play with. As the output is

dynamic, and the ball drawing stops only when the mouse button is pressed, no output for this module is shown here. The aname Sillyballs, by the way, comes from the original of this program, which was part of tutorial materials for the MacOS. Notice that all drawing is in the context of a rectangle, including that of each individual ball. Notice also that in this case, the interior of the balls has been painted.

```
MODULE SillyBalls;
(* This program draws balls in random colours and at random locations on the screen.
*)

FROM SYSTEM IMPORT
    INT16;
FROM Types IMPORT
    Rect, OSErr, UInt16;
FROM OSUtils IMPORT
    SysEnvirons, SysEnvRec;
FROM DateTimeUtils IMPORT
    GetDateTime;
FROM Sound IMPORT
    SysBeep;
FROM Quickdraw IMPORT
    qd, RGBColor, InsetRect, SetRect, Random, RGBForeColor, PaintOval, MoveTo,
    InvertColor;
FROM QuickdrawText IMPORT
    DrawString, TextSize;
FROM Events IMPORT
    Button;
IMPORT GraphWindow; (* gets window *)

(*-----
#
#   Adapted to Modula-2
#   by R. Sutcliffe
#   Trinity Western University
#   1996 01 29
#   revised 1998 07 22
#   to use GraphWindows
#   last revision 1998 09 09 for Mac 3.1 interfaces
#   from an original program bearing the notice:
#
#   Macintosh Developer Technical Support
#   Simple Color QuickDraw Sample Application
#
#   Copyright © 1988 Apple Computer, Inc.
#   All rights reserved.
#
*)

CONST
    ballWidth  = 25;
    ballHeight = 25;
    TWUSize    = 8;    (* Size of text in each ball. *)

VAR
    height, width : INTEGER;
```

```

(* Initialize everything for the program, make sure we can run. *)
PROCEDURE Initialize;
VAR
    error : OSErr;
    theWorld : SysEnvRec;

BEGIN
    (* Test the computer to be sure we can do color.  If not we would crash, which
    would be bad.  If we can't run, just beep and exit. *)

    error := SysEnvirons (1, theWorld);
    IF NOT theWorld.hasColorQD
    THEN
        SysBeep (50);
        HALT;                (* If no color QD, we must leave. *)
    END;

    (* The run time system initializes all the needed managers. *)

    (* To make the Random sequences truly random, we need to make the seed start at a
    different number.  An easy way to do this is to put the current time and date into
    the seed.  Since it is always incrementing the starting seed will always be
    different.  Don't for each call of Random, or the sequence will no longer be random.
    Only needed once, here in the init. *)

    GetDateTime (qd.randSeed);

    (* Make a new window for drawing in, and it must be a color window.  The window is
    full screen size, made smaller to make it more visible. *)

    TextSize (TWUSize);      (* small font for drawing. *)
    GraphWindow.GetWDimensions (width, height);
    DEC (width, ballWidth); (* don't start any balls too far right *)

END Initialize;

(* NewBall: make another ball in the window at a random location and color. *)
PROCEDURE NewBall;
VAR
    ballColor : RGBColor;
    ballRect : Rect;
    newLeft, newTop : INTEGER;

BEGIN
    (* Make a random new color for the ball. *)
    WITH ballColor
    DO
        red := VAL (UInt16, ABS (Random()));
        green := VAL (UInt16, ABS (Random()));
        blue := VAL (UInt16, ABS (Random()));
    END;

    (* Set that color as the new color to use in drawing. *)
    RGBForeColor (ballColor);

```

```

    (* Make a Random new location for the ball, that is normalized to the window size.
    This makes the Integer from Random into a number that is 0..hieght and 0..width.
    They are normalized so that we don't spend most of our time drawing in places outside
    of the window. *)

```

```

    newTop := Random();
    newLeft := Random();
    newTop := VAL (INT16, ((VAL (INTEGER, newTop) + 32767) * VAL(INTEGER, height)) DIV
65536);
    newLeft := VAL (INT16, ((VAL(INTEGER, newLeft) + 32767) * VAL(INTEGER, width)) DIV
65536);
    SetRect (ballRect, newLeft, newTop, newLeft + ballWidth,
        newTop + ballHeight);

    (* Move pen to the new location, and paint the colored ball. *)
    MoveTo(newLeft, newTop);
    PaintOval (ballRect);

    (* Move the pen to the middle of the new ball position, for the text *)
    MoveTo(ballRect.left + ballWidth DIV 2 - TWUSize,
        ballRect.top + ballHeight DIV 2 + TWUSize DIV 2 -1);

    (* Invert the color and draw the text there. This won't look quite right in 1 bit
    mode, since the foreground and background colors will be the same. Color QuickDraw
    special cases this to not invert the color, to avoid invisible drawing. *)

    InvertColor (ballColor);
    RGBForeColor (ballColor);
    DrawString ('TWU');
END NewBall;

```

```

BEGIN      (* Main body of program SillyBalls *)
    Initialize;
    REPEAT
        NewBall;
    UNTIL Button();
END SillyBalls.

```

18.6.3 Implementing GraphWindow in Windows NT

As a reminder, the (stripped down and relatively uncommented) definition module is given first, with the appropriate small modifications to move from MacOS to Windows NT. This implementation should also work in Windows 95/98.

```

DEFINITION MODULE GraphWindow;
IMPORT WIN32;
TYPE
    WindowRef = WIN32.HDC;

PROCEDURE GetWindow ( ) : WindowRef;

PROCEDURE GetWDimensions (VAR width, height : INTEGER);
END GraphWindow.

```

The implementation is somewhat more work, as the trick of stealing the top available window does not appear to give good results, and more has to be done to get anything to happen at all. Moreover, a lot more information has to be prepared into a data structure before the window is opened. As the purpose here is to supply the necessary code, not to explain every detail, very little other commentary is provided. The reader is invited to compare this code with what was needed in the corresponding MacOS implementation. There are similarities, but several differences as well.

WARNING: This implementation was done for Stonybrook Modula-2 for Win32. As numerous implementation details are certain to vary, the reader cannot expect it to work unmodified on other 32-bit Windows implementations. It is likely that the various imports will come from different places, and it is also likely that the method of handling the translation to C++ classes will also be different. It may be necessary for the reader, as it was for the author, to obtain an example program for the specific implementation, and then modify it to suit, as the available documentation for the Microsoft C++ classes and API is unlikely to shed much light on how to get started.

The reader will note that the style here is a little different in that a window class has to be created and registered before the window itself is created. Then, the main event loop consists of waiting for a *message*, which is then translated and dispatched. Because the system has some built in handlers to which messages can be dispatched, client applications created with this module can be quit in the normal way.

IMPLEMENTATION MODULE GraphWindow;

```
(* Design by R. Sutcliffe
   Implementation by Joel Schwartz for StonyBrook Modula-2
   last modification : 1998 07 14 by RS *)
```

FROM SYSTEM IMPORT

FUNC, ADR, CAST;

FROM WIN32 IMPORT

UINT, WPARAM, LPARAM, LRESULT, BOOL, RECT, HBRUSH, HWND;

FROM WINUSER IMPORT

GetDC, ReleaseDC, SetRect, InvalidateRect, GetClientRect, RegisterClass,
ShowWindow, GetMessage, TranslateMessage, DispatchMessage,
LoadCursor, LoadIcon, IDC_ARROW,
FillRect, DefWindowProc, UpdateWindow, PostQuitMessage,
BeginPaint, EndPaint, CreateWindow, GetSysColor,
LOWORD, HIWORD, COLOR_WINDOW, WS_OVERLAPPEDWINDOW,
CS_VREDRAW, CS_HREDRAW, CS_BYTEALIGNCLIENT,
WM_SIZE, WM_DESTROY, WM_PAINT, WM_SYSCOLORCHANGE, WM_ERASEBKGD,
WNDCLASS, MSG, PAINTSTRUCT;

FROM WINGDI IMPORT

CreateSolidBrush, GetDeviceCaps, VERTRES, HORZRES;

FROM WINX IMPORT

DeleteBrush, NULL_HWND, NIL_RECT, NULL_HBRUSH, NULL_HINSTANCE, NULL_HMENU,
Instance, PrevInstance, CmdShow;

VAR

graphRect : RECT;

```
(* Window Creation / Management Variables *)
```

VRes, HRes : **INTEGER**;

WindowWidth, WindowHeight : **INTEGER**;

szBuffer : **ARRAY** [0..30] **OF** **CHAR**;


```

bFirst : BOOL = TRUE;
hbrBackgnd : HBRUSH;      (* background brush -- system window backbround color *)
Wnd : HWND;
mess : MSG;
DC : WindowRef;

(*----- Public Procedures -----*)

PROCEDURE GetWindow () :WindowRef;
BEGIN
    RETURN DC;  (* Return the window reference *)
END GetWindow;

PROCEDURE GetWDimensions (VAR width, height : INTEGER);
BEGIN
    width := WindowWidth;
    height := WindowHeight;
END GetWDimensions;

(*-----Private Procedures -----*)
<*/PUSH*>
<*/CALLS:WIN32SYSTEM*>

PROCEDURE FrameWndProc (wnd : HWND;
    message : UINT;
    wParam : WPARAM;
    lParam : LPARAM) : LRESULT [EXPORT];

(* Pre: A window event has occurred
 * Post: The window event is handled manually or by the default manager.
 *)

VAR
    ps : PAINTSTRUCT;
    rc : RECT;

BEGIN
    CASE message
    OF
        WM_SIZE:
            SetRect (graphRect, 0, 0, LOWORD (lParam), HIWORD (lParam));
            UpdateWindow (wnd);
        |
        WM_DESTROY:
            (* Delete Tools *)
            FUNC DeleteBrush (hbrBackgnd);
            PostQuitMessage (0);
        |
        WM_PAINT:
            InvalidateRect (wnd, NIL_RECT, TRUE);
            FUNC BeginPaint (wnd, ps);
            FUNC FillRect (DC, graphRect, hbrBackgnd);
            EndPaint (wnd, ps);
        |

```

```

        WM_SYSCOLORCHANGE:
(* Change tools to coincide with system window colors *)
(*Delete Tools*)
        FUNC DeleteBrush (hbrBackgnd);
(* Create Tools *)
        hbrBackgnd := CreateSolidBrush (GetSysColor (COLOR_WINDOW));
|
WM_ERASEBKGD:
        (* Paint over the entire client area *)
        GetClientRect (wnd, rc);
        FUNC FillRect (CAST (WindowRef, wParam), rc, hbrBackgnd);
|
ELSE
        (* Perform the default window processing *)
        RETURN DefWindowProc (wnd, message, wParam, lParam);
END;
RETURN 0;
END FrameWndProc;
<*/POP*>

PROCEDURE FrameInit () : BOOLEAN;

(* Pre: The window is to be displayed for the first time.
   Post: The window class is initialized and registered *)

VAR
    frameClass : WNDCLASS;

BEGIN
    frameClass.lpszClassName := ADR (szBuffer);
    frameClass.hbrBackground := NULL_HBRUSH;
    frameClass.style         := CS_VREDRAW + CS_HREDRAW + CS_BYTEALIGNCLIENT;
    frameClass.hInstance     := Instance;
    frameClass.lpfnWndProc   := FrameWndProc;
    frameClass.hCursor       := LoadCursor (Instance, IDC_ARROW^);
    frameClass.hIcon         := LoadIcon (Instance, 'Graphics');
    frameClass.cbClsExtra    := 0;
    frameClass.cbWndExtra    := 0;
    frameClass.lpszMenuName  := NIL;

    IF RegisterClass (frameClass) = 0
    THEN
        (* Error registering class -- return *)
        RETURN FALSE;
    END;
    RETURN TRUE;
END FrameInit;

(*----- Main Code -----*)

BEGIN
    (* Initialize variables *)
    szBuffer := "The Best of Graphics";

```

```

IF PrevInstance = NULL_HINSTANCE
THEN
    (* First instance -- register window class *)
    IF NOT FrameInit ()
        THEN
            HALT (1);
        END;
    ELSE
        (* Not first instance -- reset bFirst flag *)
        bFirst := FALSE;
    END;

    (* Find the height and width of the screen *)
    DC := GetDC (NULL_HWND);
    VRes := GetDeviceCaps (DC, VERTRES);
    HRes := GetDeviceCaps (DC, HORZRES);
    FUNC ReleaseDC (NULL_HWND, DC);

    (* Create Tools*)
    hbrBackgnd := CreateSolidBrush (GetSysColor (COLOR_WINDOW));

    (* set window height and width *)
    WindowWidth := HRes;
    WindowHeight := VRes - 30;

    Wnd := CreateWindow ( szBuffer,      (* class name                *)
                        szBuffer,      (* The window name          *)
                        WS_OVERLAPPEDWINDOW, (* window style            *)
                        0,              (* Position window at top right *)
                        0,              (* y not used                *)
                        WindowWidth,    (* window Width              *)
                        WindowHeight,   (* window height             *)
                        NULL_HWND,      (* NULL parent handle        *)
                        NULL_HMENU,     (* NULL menu/child handle    *)
                        Instance,       (* program instance          *)
                        NIL,           (* NULL data structure ref. *)
                        );

    DC := GetDC (Wnd); (* Obtain the window reference*)
    FUNC ShowWindow (Wnd, CmdShow);
    (* Pause until user closes the screen *)
FINALLY
    WHILE GetMessage (mess, NULL_HWND, 0, 0)
        DO
            FUNC TranslateMessage (mess);
            FUNC DispatchMessage (mess);
        END;
END GraphWindow.

```

18.6.4 Implementing GraphPaper in MacOS

With the infrastructure now in place, *GraphPaper* is fairly easy to implement. The next listing is the version for the MacOS. A couple of items to note are that many of the drawing routines use the short integer (16 bit) called INT16. As calls into this

module use INTEGER for the most part, users will have to be careful or there will be an overflow. Also, the MacOS uses Pascal strings for most purposes. Thus, the internal string type has to be converted into STR255 type internally.

IMPLEMENTATION MODULE GraphPaper;

```
(* Original design copyright 1996 by R. Sutcliffe
   Original implementation 1996 using pl on the Macintosh
   Windows implementation 1998 05 12 by Joel Schwartz
       with use of examples written by Stony Brook
       added scaling, labelling, showing axes
   Changes ported back to the Mac 1998 05 21 by Joel Schwartz
   Removed all widow-related functionality to a separate module 1998 07 06
       to clean thing up a little more; now imports from GraphWindow
   Last revision: by RS 1998 07 11--added angle measure types
*)
```

FROM GraphWindow **IMPORT**

WindowRef, GetWindow, GetWDimensions;

FROM Strings **IMPORT**

Concat;

FROM WholeStr **IMPORT**

CardToStr, IntToStr;

FROM SYSTEM **IMPORT**

STR255, TOSTR255;

IMPORT Quickdraw; (* use MoveTo and LineTo, and have our own by this name *)

FROM Quickdraw **IMPORT**

PenState, SetPenState, GetPenState, SetPort;

FROM QuickdrawText **IMPORT**

DrawString;

FROM RealMath **IMPORT**

pi, sin, cos;

CONST

convertRad = pi / 180.0;

AspectV = 40; (* vertical raster lines per division mark *)

AspectH = 40; (* Horizontal pixels per division mark *)

VAR

activeSystem : CoordSystem;

angleUnits : AngleType;

width, height : **INTEGER**;

xLabel, yLabel : LabelType;

xScale, yScale : **REAL**;

homeX, homeY, graphX, graphY : **REAL**;

graphArg : **REAL**; (* kept internally in degrees *)

divisionValue : **INTEGER**;

scaleString : LabelType;

scaleSet, labelSet, axesDrawn : **BOOLEAN**;

penPos : PenState;

gWindow : WindowRef;

(* note that internally we use the Mac/Win position angle in which East is 0 and we rotate clockwise but in the bearing system users communicate with bearing angles in

which North is zero and they rotate clockwise. *)

```
PROCEDURE Round (x : REAL) : INTEGER;  
BEGIN  
    RETURN VAL (INTEGER, x + 0.5 );  
END Round;
```

```
PROCEDURE SetCoordSystem (kind : CoordSystem);  
BEGIN  
    activeSystem := kind;  
    CASE activeSystem (* set up appropriate home *)  
        OF  
            bearing, standard:  
                homeX := FLOAT (width) / 2.0;  
                homeY := FLOAT (height) / 2.0; |  
            MacWin:  
                homeX := 0.0;  
                homeY := 0.0  
        END;  
    Home; (* and go there *)  
END SetCoordSystem;
```

```
PROCEDURE SetAngleType (kind : AngleType);  
BEGIN  
    angleUnits := kind;  
END SetAngleType;
```

```
PROCEDURE ToDeg (arg : REAL) : REAL;  
(* used to get angle units into degrees for internal store *)  
BEGIN  
    CASE angleUnits OF  
        deg :  
            RETURN arg |  
        rad :  
            RETURN arg/convertRad |  
        grad :  
            RETURN 0.9 * arg  
    END;  
END ToDeg;
```

```
PROCEDURE FromDeg (arg : REAL) : REAL;  
(* convert from internal store to whatever units have been set. *)  
BEGIN  
    CASE angleUnits OF  
        deg :  
            RETURN arg |  
        rad :  
            RETURN arg * convertRad |  
        grad :  
            RETURN arg / 0.9  
    END;  
END FromDeg;
```

```
PROCEDURE Home; (* moves to 0,0 and sets angle to 0 *)
```

```

BEGIN
  TurnTo (0.0);
  graphX := homeX;
  graphY := homeY;
  Quickdraw.MoveTo (Round (graphX), Round (graphY));
END Home;

PROCEDURE ShiftOrigin (deltaX, deltaY : INTEGER);
BEGIN
  homeX := homeX + FLOAT (deltaX);
  CASE activeSystem
    OF
      bearing, standard:
        homeY := homeY - FLOAT (deltaY); |
      MacWin:
        homeY := homeY + FLOAT (deltaY);
    END;
END ShiftOrigin;

PROCEDURE GetDimensions (VAR x, y: INTEGER);
BEGIN
  GetWDimensions (x, y);
END GetDimensions;

PROCEDURE GetLocation (VAR x,y :INTEGER);
BEGIN
  x := Round (graphX - homeX);
  CASE activeSystem
    OF
      bearing, standard:
        y := Round (homeY - graphY);|
      MacWin:
        y := Round (graphY - homeY)
    END;
END GetLocation;

PROCEDURE Radians (angle : REAL) : REAL;
BEGIN
  RETURN angle * convertRad;
END Radians;

PROCEDURE MoveBy (distance : INTEGER);
BEGIN
  graphX := graphX + (FLOAT (distance) * cos (Radians (graphArg))) ;
  graphY := graphY - (FLOAT (distance) * sin (Radians (graphArg)));
  Quickdraw.MoveTo (Round (graphX), Round (graphY));
END MoveBy;

PROCEDURE MoveTo (x, y : INTEGER);
  (* have to revise coordinates to screen system *)
BEGIN
  graphX := homeX + FLOAT (x);
  CASE activeSystem

```

```

    OF
        bearing, standard:
            graphY := homeY - FLOAT (y);|
        MacWin:
            graphY := homeY + FLOAT (y);
    END;
    Quickdraw.MoveTo (Round (graphX), Round (graphY));
END MoveTo;

PROCEDURE Move (dx, dy: INTEGER);
BEGIN
    graphX := graphX + FLOAT (dx);
    CASE activeSystem
        OF
            bearing, standard:
                graphY := graphY - FLOAT (dy);|
            MacWin:
                graphY := graphY + FLOAT (dy);
        END;
    Quickdraw.MoveTo (Round (graphX), Round (graphY));
END Move;

PROCEDURE ReviseAngle (angle : REAL) : REAL;
(* this procedure is internal, so works strictly in degrees *)
BEGIN
    CASE activeSystem
        OF
            bearing:
                angle := 450.0 - angle;
                graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360);
                RETURN graphArg;|
            MacWin:
                angle := 360.0 - angle;
                graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360);
                RETURN graphArg;|
            standard:
                graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360);
                RETURN graphArg
        END;
END ReviseAngle;

PROCEDURE Turn (angle : REAL);
(* convert if a bearing change *)
BEGIN
    angle := ToDeg (angle);
    CASE activeSystem
        OF
            bearing, MacWin:
                angle := graphArg - angle;
                graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360);|
            standard:
                angle := graphArg + angle;

```

```

        graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360)
    END;
END Turn;

PROCEDURE TurnTo (angle : REAL);
BEGIN
    angle := ReviseAngle (ToDeg (angle));
END TurnTo;

PROCEDURE GetCurrentAngle () : REAL;
BEGIN
    RETURN FromDeg (graphArg);
END GetCurrentAngle;

PROCEDURE LineBy (distance : INTEGER);
BEGIN
    graphX := graphX + (FLOAT (distance) * cos (Radians (graphArg)));
    graphY := graphY - (FLOAT (distance) * sin (Radians (graphArg)));
    Quickdraw.LineTo (Round (graphX), Round (graphY));
END LineBy;

PROCEDURE LineTo (x, y : INTEGER);
BEGIN
    graphX := homeX + FLOAT (x);
    CASE activeSystem
        OF
            bearing, standard:
                graphY := homeY - FLOAT (y);|
            MacWin:
                graphY := homeY + FLOAT (y);
        END;
    Quickdraw.LineTo (Round (graphX), Round (graphY));
END LineTo;

PROCEDURE Line (dx, dy: INTEGER);
BEGIN
    graphX := graphX + FLOAT (dx);
    CASE activeSystem
        OF
            bearing, standard:
                graphY := graphY - FLOAT (dy);|
            MacWin:
                graphY := graphY + FLOAT (dy);
        END;
    Quickdraw.LineTo (Round (graphX), Round (graphY));
END Line;

PROCEDURE Dot;
BEGIN
    Line (0, 0);
END Dot;

PROCEDURE DotAt (x, y: INTEGER);
BEGIN

```



```

    MoveTo (x,y);
    Dot;
END DotAt;

(*----- Graphing related functions -----*)

PROCEDURE DrawXY;
(* Draw the axes of the graph *)
BEGIN
    CASE activeSystem
        OF
            bearing:
                Home;
                MoveTo (0,- (height DIV 2 - 25));
                LineBy (height - 45);
                Home;
                MoveTo (- (width DIV 2 - 30), 0);
                TurnTo (90.0);
                LineBy (width - 60);
            |
            standard:
                Home;
                MoveTo (0,- (height DIV 2 - 25));
                TurnTo (90.0);
                LineBy (height - 45);
                Home;
                MoveTo (- (width DIV 2 - 30), 0);
                LineBy (width - 60);
            |
            MacWin:
                Home;
                MoveTo (0,- (height - 50));
                TurnTo (90.0);
                LineBy (height - 50);
                Home;
                MoveTo (- (width - 30), 0);
                LineBy (width - 30);
        END;
END DrawXY;

PROCEDURE SetLabels (horiz, vert : LabelType);
(* Set the horizontal and vertical axes labels *)
BEGIN
    xLabel := horiz;
    yLabel := vert;
    labelSet := TRUE;
END SetLabels;

PROCEDURE ShowLabels;
(* Show the labels but only if the axes have been drawn *)
BEGIN
    IF axesDrawn
        THEN

```

```

IF ~labelSet
  THEN
    xLabel := "x";
    yLabel := "y";
  END;

CASE activeSystem
OF
  bearing, standard:
    GetPenState (penPos);
    penPos.pnLoc.h := width - 100;
    penPos.pnLoc.v := height DIV 2 + 20;
    SetPenState (penPos);
    DrawString (TOSTR255(xLabel));
    GetPenState (penPos);
    penPos.pnLoc.h := width DIV 2 + 20;
    penPos.pnLoc.v := 20;
    SetPenState (penPos);
    DrawString (TOSTR255(yLabel));
  |
  MacWin:
    GetPenState (penPos);
    penPos.pnLoc.h := width - 100;
    penPos.pnLoc.v := 50;
    SetPenState (penPos);
    DrawString (TOSTR255(xLabel));
    GetPenState (penPos);
    penPos.pnLoc.h := 40;
    penPos.pnLoc.v := height - 70;
    SetPenState (penPos);
    DrawString (TOSTR255(yLabel));
END;
END;
END ShowLabels;

```

```

PROCEDURE ShowAxes;
  (* Show the axes and draw the division marks on the axes *)
BEGIN
  DrawXY;
  IF NOT scaleSet
    THEN
      SetScale (1);
    END;
  DrawDivisionMarks;
  axesDrawn := TRUE;
END ShowAxes;

```

```

PROCEDURE SetScale (dataPerDivision : CARDINAL);

  (* Set the scale if no scale has been set up to this point.
  * NOTE: this does not support mid-graph scale changing *)
VAR
  temp, temp2 : LabelType;

```

```

BEGIN
  IF ~scaleSet
    THEN
      xScale := FLOAT (AspectH * dataPerDivision);
      yScale := FLOAT (AspectV * dataPerDivision);

      (*Set the scale to a string representation *)
      CardToStr (dataPerDivision, temp);
      Concat ('SCALE = 1 unit : ', temp, temp2);
      Concat (temp2, ' division', temp);
      divisionValue := dataPerDivision;
      scaleString := temp;
      scaleSet := TRUE;
    END;
END SetScale;

PROCEDURE DrawDivisionMarks;

VAR
  counter : INTEGER;
  multiple : INTEGER;
  xMax, yMax, tempStore, determineDivision : INTEGER;
  xPos, yPos : INTEGER;
  xString, yString : ARRAY [0..5] OF CHAR;
  widthTest, heightTest, xdivisionPos : INTEGER;

BEGIN
  CASE activeSystem
    OF
      bearing, standard:
        widthTest := (width DIV 2 - 30);
        heightTest := ((height - 70) DIV 2);
        xdivisionPos := -10;
      |
      MacWin:
        widthTest := width - 40;
        heightTest := height - 40;
        xdivisionPos := - 10;
    END;

  (* Draw the division marks 1cm apart while determining where the scale marker will
  go.*)

  CASE activeSystem
    OF
      bearing:  (* for bearing system *)
        counter := AspectH;
        multiple := 2;
        Home;
        WHILE counter < widthTest
          DO
            MoveTo ( - counter, xdivisionPos);
            LineBy (20);

```

```

        counter := (AspectH * multiple);
        INC (multiple);
    END;
counter := AspectH;
multiple := 2;
WHILE counter < widthTest
    DO
        MoveTo (counter, xdivisionPos);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
    END;
xMax := multiple - 2;
xPos := counter - AspectH;

counter := AspectH;
multiple := 2;
WHILE counter < heightTest
    DO
        MoveTo ( -10, -counter);
        TurnTo (90.0);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
    END;
counter := AspectH;
multiple := 2;
WHILE counter < heightTest
    DO
        MoveTo ( -10, counter);
        TurnTo (90.0);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
    END;

yMax := multiple - 2;
yPos := counter - AspectV;
|
standard, MacWin: (* For standard coordinate systems *)
counter := AspectH;
multiple := 2;
WHILE counter < widthTest
    DO
        MoveTo ( - counter, xdivisionPos);
        TurnTo (90.0);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
    END;
counter := AspectH;
multiple := 2;
WHILE counter < widthTest

```

```

    DO
        MoveTo (counter, xdivisionPos);
        TurnTo (90.0);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
    END;

xMax := multiple - 2;
xPos := counter - AspectH;

counter := AspectH;
multiple := 2;
Home;
WHILE counter < heightTest
    DO
        MoveTo ( -10, -counter);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
    END;
counter := AspectH;
multiple := 2;
WHILE counter < heightTest
    DO
        MoveTo ( -10, counter);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
    END;
yMax := multiple - 2;
yPos := counter - AspectV;
END;

tempStore := xMax / divisionValue;
determineDivision := xMax MOD divisionValue;
xPos := xPos - (AspectH * determineDivision);
IntToStr (tempStore, xString);
CASE activeSystem
OF
    bearing, standard:
        GetPenState (penPos);
        penPos.pnLoc.h := width DIV 2 + xPos;
        penPos.pnLoc.v := height DIV 2 + 20;
        SetPenState(penPos);
        DrawString (TOSTR255 (xString));
|
MacWin:
    GetPenState(penPos);
    penPos.pnLoc.h := xPos;
    penPos.pnLoc.v := 20;
    SetPenState(penPos);
    DrawString (TOSTR255 (xString));

```

```

    END;
tempStore := yMax / divisionValue;
determineDivision := yMax MOD divisionValue;
yPos := yPos - (AspectV * determineDivision);
IntToStr (tempStore, yString);
CASE activeSystem
    OF
        bearing, standard:
            GetPenState (penPos);
            penPos.pnLoc.h := width DIV 2 - 20;
            penPos.pnLoc.v := height DIV 2 - yPos;
            SetPenState (penPos);
            DrawString (TOSTR255 (yString));
        |
        MacWin:
            GetPenState(penPos);
            penPos.pnLoc.h := 20;
            penPos.pnLoc.v := yPos;
            SetPenState(penPos);
            DrawString (TOSTR255 (yString));
    END;
GetPenState(penPos);
penPos.pnLoc.h := width - 250;
penPos.pnLoc.v := height - 50;
SetPenState (penPos);
DrawString (TOSTR255 (scaleString));
END DrawDivisionMarks;

PROCEDURE PlotPoint (x, y : REAL);

BEGIN
    IF ~scaleSet
        THEN
            SetScale (1);
        END;
    DotAt (Round (x * xScale), Round (y * yScale));
END PlotPoint;

PROCEDURE PolarPlotPoint (radius, angle : REAL);

VAR
    x,y : REAL;

BEGIN
    IF activeSystem = MacWin
        THEN
            angle := - angle;
        END;
    angle := ReviseAngle (ToDeg (angle));

    x := radius * cos (angle * convertRad);
    y := radius * sin (angle * convertRad);

    IF ~scaleSet

```

```

    THEN      (* Make sure the graph has a scale *)
        SetScale (1);
    END;
    DotAt (Round (x * xScale), Round (y * yScale));
END PolarPlotPoint;

(*----- Main Code -----*)

BEGIN
    (* Initialize variables *)
    (* the import of GraphWindow sets up the window for us.
       but we had better make sure the graph port is current *)
    gWindow := GetWindow ();
    SetPort (gWindow);
    GetWDimensions (width, height);
    scaleSet := FALSE;
    labelSet := FALSE;
    axesDrawn := FALSE;
    graphArg := 0.0;
    SetCoordSystem (standard); (* default *)
    Home;
END GraphPaper.

```

18.6.5 Implementing GraphPaper in Windows NT

Much of the code for implementing the routines is the same in the windows version as in the MacOS version. Some differences to note include:

- imports from WINGDI rather than Quickdraw
- the text drawing routine is in the same module as the line drawing ones
- these routines take a window reference parameter, so this must be obtained from *GraphWindow*
- some routines are called using *SYSTEM.FUNC*

WARNING: This implementation was done for Stonybrook Modula-2 for Win32. As numerous implementation details are certain to vary, the reader cannot expect it to work unmodified on other 32-bit Windows implementations.

```

IMPLEMENTATION MODULE GraphPaper;

```

```

(* Original design copyright 1996 by R. Sutcliffe
   Original implementation 1996 using pl on the Macintosh
   Windows implementation 1998 05 12 by Joel Schwartz
   with use of examples written by Stony Brook
   Changes ported back to the Mac 1998 05 21 by Joel Schwartz
   Removed all widow-related functionality to a separate module 1998 07 06
   to clean thing up a little more; now imports from GraphWindow
   Last revision: by RS 1998 07 11
*)

FROM SYSTEM IMPORT
    FUNC;
IMPORT WINGDI;
FROM WINGDI IMPORT
    MoveToEx, TextOut;

```

```

FROM WINX IMPORT
    NIL_POINT;
FROM GraphWindow IMPORT
    WindowRef, GetWindow, GetWDimensions;
FROM Strings IMPORT
    Length, Concat;
FROM WholeStr IMPORT
    CardToStr, IntToStr;
FROM RealMath IMPORT
    sin, cos, pi;

CONST
    convertRad = pi / 180.0;
    AspectV = 40;    (* vertical raster lines per division mark *)
    AspectH = 40;    (* Horizontal pixels per division mark *)

VAR

    (* Graphics Variables *)
    window : WindowRef;
    activeSystem : CoordSystem;
    angleUnits : AngleType;
    xLabel, yLabel : LabelType;
    xScale, yScale : REAL;
    homeX, homeY, graphX, graphY : REAL;
    graphArg : REAL; (* kept internally in degrees *)
    width, height : INTEGER;
    divisionValue : INTEGER;
    scaleString : LabelType;
    scaleSet, labelSet, axesDrawn : BOOLEAN;

    (* note that internally we use the Mac/Win position angle in which East is 0 and we
    rotate clockwise but in the bearing system users communicate with bearing angles in
    which North is zero and they rotate clockwise. *)

PROCEDURE Round (x : REAL) : INTEGER;
BEGIN
    RETURN VAL (INTEGER, x + 0.5 );
END Round;

PROCEDURE SetCoordSystem (kind : CoordSystem);
BEGIN
    activeSystem := kind;
    CASE activeSystem (* set up appropriate home *)
        OF
            bearing, standard:
                homeX := FLOAT (width) / 2.0;
                homeY := FLOAT (height) / 2.0; |
            MacWin:
                homeX := 0.0;
                homeY := 0.0
        END;
    Home; (* and go there *)
END SetCoordSystem;

```



```

PROCEDURE SetAngleType (kind : AngleType);
BEGIN
    angleUnits := kind;
END SetAngleType;

PROCEDURE ToDeg (arg : REAL) : REAL;
BEGIN
    CASE angleUnits OF
        deg :
            RETURN arg |
        rad :
            RETURN arg/convertRad |
        grad :
            RETURN 0.9 * arg
    END;
END ToDeg;

PROCEDURE FromDeg (arg : REAL) : REAL;
BEGIN
    CASE angleUnits OF
        deg :
            RETURN arg |
        rad :
            RETURN arg * convertRad |
        grad :
            RETURN arg / 0.9
    END;
END FromDeg;

PROCEDURE Home; (* moves to 0,0 and sets angle to 0 *)
BEGIN
    TurnTo (0.0);
    graphX := homeX;
    graphY := homeY;
    FUNC MoveToEx (window, Round (graphX), Round (graphY), NIL_POINT);
END Home;

PROCEDURE ShiftOrigin (deltaX, deltaY : INTEGER);
BEGIN
    homeX := homeX + FLOAT (deltaX);
    CASE activeSystem
        OF
            bearing, standard:
                homeY := homeY - FLOAT (deltaY); |
            MacWin:
                homeY := homeY + FLOAT (deltaY);
    END;
END ShiftOrigin;

PROCEDURE GetDimensions (VAR x, y: INTEGER);
BEGIN
    GetWDimensions (x, y);

```

```

END GetDimensions;

PROCEDURE GetLocation (VAR x,y :INTEGER);
BEGIN
  x := Round (graphX - homeX);
  CASE activeSystem
    OF
      bearing, standard:
        y := Round (homeY - graphY);|
      MacWin:
        y := Round (graphY - homeY)
    END;
END GetLocation;

PROCEDURE Radians (angle : REAL) : REAL;
BEGIN
  RETURN angle * convertRad;
END Radians;

PROCEDURE MoveBy (distance : INTEGER);
BEGIN
  graphX := graphX
    + (FLOAT (distance) * cos (Radians (graphArg))) ;
  graphY := graphY
    - (FLOAT (distance) * sin (Radians (graphArg)));
  FUNC MoveToEx (window, Round (graphX), Round (graphY), NIL_POINT);
END MoveBy;

PROCEDURE MoveTo (x, y : INTEGER);
  (* have to revise coordinates to screen system *)
BEGIN
  graphX := homeX + FLOAT (x);
  CASE activeSystem
    OF
      bearing, standard:
        graphY := homeY - FLOAT (y);|
      MacWin:
        graphY := homeY + FLOAT (y);
    END;
  FUNC MoveToEx (window, Round (graphX), Round (graphY), NIL_POINT);
END MoveTo;

PROCEDURE Move (dx, dy: INTEGER);
BEGIN
  graphX := graphX + FLOAT (dx);
  CASE activeSystem
    OF
      bearing, standard:
        graphY := graphY - FLOAT (dy);|
      MacWin:
        graphY := graphY + FLOAT (dy);
    END;
  FUNC MoveToEx (window, Round (graphX), Round (graphY), NIL_POINT);

```

```

END Move;

PROCEDURE ReviseAngle (angle : REAL) : REAL;
(* this procedure is internal, so works strictly in degrees *)
BEGIN
  CASE activeSystem
    OF
      bearing:
        angle := 450.0 - angle;
        graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360);
        RETURN graphArg;|
      MacWin:
        angle := 360.0 - angle;
        graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360);
        RETURN graphArg;|
      standard:
        graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360);
    RETURN graphArg
  END;
END ReviseAngle;

PROCEDURE Turn (angle : REAL);
(* convert if a bearing change *)
BEGIN
  angle := ToDeg (angle);
  CASE activeSystem
    OF
      bearing, MacWin:
        angle := graphArg - angle;
        graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360);|
      standard:
        angle := graphArg + angle;
        graphArg := angle - FLOAT (VAL (INTEGER, (angle / 360.0)) * 360)
    END;
END Turn;

PROCEDURE TurnTo (angle : REAL);
BEGIN
  angle := ReviseAngle (ToDeg (angle));
END TurnTo;

PROCEDURE GetCurrentAngle () : REAL;
BEGIN
  RETURN FromDeg (graphArg);
END GetCurrentAngle;

PROCEDURE LineBy (distance : INTEGER);
BEGIN
  graphX := graphX + (FLOAT (distance) * cos (Radians (graphArg)));
  graphY := graphY - (FLOAT (distance) * sin (Radians (graphArg)));
  WINGDI.LineTo (window, Round (graphX), Round (graphY));
END LineBy;

```

```

PROCEDURE LineTo (x, y : INTEGER);
BEGIN
    graphX := homeX + FLOAT (x);
    CASE activeSystem
        OF
            bearing, standard:
                graphY := homeY - FLOAT (y);|
            MacWin:
                graphY := homeY + FLOAT (y);
    END;
    WINGDI.LineTo (window, Round (graphX), Round (graphY));
END LineTo;

PROCEDURE Line (dx, dy: INTEGER);
BEGIN
    graphX := graphX + FLOAT (dx);
    CASE activeSystem
        OF
            bearing, standard:
                graphY := graphY - FLOAT (dy);|
            MacWin:
                graphY := graphY + FLOAT (dy);
    END;
    WINGDI.LineTo (window, Round (graphX), Round (graphY));
END Line;

PROCEDURE Dot;
BEGIN
    Line (1, 1);
END Dot;

PROCEDURE DotAt (x, y: INTEGER);
BEGIN
    MoveTo (x,y);
    Dot;
END DotAt;

(*----- Graphing related functions -----*)

PROCEDURE DrawXY;
(* Draw the axes of the graph *)
BEGIN
    CASE activeSystem
        OF
            bearing:
                Home;
                MoveTo (0,- (height DIV 2 - 50));
                LineBy (height - 70);
                Home;
                MoveTo (- (width DIV 2 - 30), 0);
                TurnTo (90.0);
                LineBy (width - 60);
            |

```

```

standard:
  Home;
  MoveTo (0,- (height DIV 2 - 50));
  TurnTo (90.0);
  LineBy (height - 70);
  Home;
  MoveTo (- (width DIV 2 - 30), 0);
  LineBy (width - 60);

```

```

|
MacWin:
  Home;
  MoveTo (0,- (height - 50));
  TurnTo (90.0);
  LineBy (height - 50);
  Home;
  MoveTo (- (width - 30), 0);
  LineBy (width - 30);

```

END;

END DrawXY;

PROCEDURE SetLabels (horiz, vert : LabelType);
 (* Set the horizontal and vertical axes labels *)

BEGIN
 xLabel := horiz;
 yLabel := vert;
 labelSet := **TRUE**;

END SetLabels;

PROCEDURE ShowLabels;
 (* Show the labels but only if the axes have been drawn *)

BEGIN
IF axesDrawn
THEN
IF NOT (labelSet)
THEN
 xLabel := "x";
 yLabel := "y";
END;

CASE activeSystem
OF
 bearing, standard:
 TextOut (window, (width - 100), (height **DIV** 2 + 50), xLabel, Length
(xLabel));
 TextOut (window, (width **DIV** 2 + 20), 20, yLabel, Length (yLabel));
 |
 MacWin:
 TextOut (window, (width - 100), 50, xLabel, Length (xLabel));
 TextOut (window, (40), (height - 70), yLabel, Length (yLabel));
END;
END;

END ShowLabels;

```

PROCEDURE ShowAxes;
  (* Show the axes and draw the division marks on the axes *)
BEGIN
  DrawXY;
  IF NOT scaleSet
    THEN
      SetScale (1);
    END;
  DrawDivisionMarks;
  axesDrawn := TRUE;
END ShowAxes;

```

```

PROCEDURE SetScale (dataPerDivision : CARDINAL);

  (* Set the scale if no scale has been set up to this point.
  * NOTE: this does not support mid-graph scale changing *)
VAR
  temp, temp2 : LabelType;

```

```

BEGIN
  IF ~scaleSet
    THEN
      xScale := FLOAT (AspectH * dataPerDivision);
      yScale := FLOAT (AspectV * dataPerDivision);

      (* Set the scale to a string representation *)
      CardToStr (dataPerDivision, temp);
      Concat ('SCALE = 1 unit : ', temp, temp2);
      Concat (temp2, ' division', temp);
      divisionValue := dataPerDivision;
      scaleString := temp;
      scaleSet := TRUE;
    END;
END SetScale;

```

```

PROCEDURE DrawDivisionMarks;

```

```

VAR
  counter : INTEGER;
  multiple : INTEGER;
  xMax, yMax, tempStore, determineDivision : INTEGER;
  xPos, yPos : INTEGER;
  xString, yString : ARRAY [0..5] OF CHAR;
  widthTest, heightTest, xdivisionPos : INTEGER;

```

```

BEGIN
  CASE activeSystem
    OF
      bearing, standard:
        widthTest := (width DIV 2 - 30);
        heightTest := ((height - 70) DIV 2);
        xdivisionPos := -10;
      |

```

```
MacWin:
  widthTest := width - 40;
  heightTest := height - 40;
  xdivisionPos := - 10;
END;
```

(* Draw the division marks 1cm apart while determining where the scale marker will go.*)

```
CASE activeSystem
OF
  bearing: (* for bearing system *)
    counter := AspectH;
    multiple := 2;
    Home;
    WHILE counter < widthTest
      DO
        MoveTo ( - counter, xdivisionPos);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
      END;
    counter := AspectH;
    multiple := 2;
    WHILE counter < widthTest
      DO
        MoveTo (counter, xdivisionPos);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
      END;
    xMax := multiple - 2;
    xPos := counter - AspectH;

    counter := AspectH;
    multiple := 2;
    WHILE counter < heightTest
      DO
        MoveTo ( -10, -counter);
        TurnTo (90.0);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
      END;
    counter := AspectH;
    multiple := 2;
    WHILE counter < heightTest
      DO
        MoveTo ( -10, counter);
        TurnTo (90.0);
        LineBy (20);
        counter := (AspectH * multiple);
        INC (multiple);
```

```

    END;

    yMax := multiple - 2;
    yPos := counter - AspectV;
|
standard, MacWin: (* For standard coordinate systems *)
    counter := AspectH;
    multiple := 2;
    WHILE counter < widthTest
        DO
            MoveTo ( - counter, xdivisionPos);
            TurnTo (90.0);
            LineBy (20);
            counter := (AspectH * multiple);
            INC (multiple);
        END;
    counter := AspectH;
    multiple := 2;
    WHILE counter < widthTest
        DO
            MoveTo (counter, xdivisionPos);
            TurnTo (90.0);
            LineBy (20);
            counter := (AspectH * multiple);
            INC (multiple);
        END;

    xMax := multiple - 2;
    xPos := counter - AspectH;

    counter := AspectH;
    multiple := 2;
    Home;
    WHILE counter < heightTest
        DO
            MoveTo ( -10, -counter);
            LineBy (20);
            counter := (AspectH * multiple);
            INC (multiple);
        END;
    counter := AspectH;
    multiple := 2;
    WHILE counter < heightTest
        DO
            MoveTo ( -10, counter);
            LineBy (20);
            counter := (AspectH * multiple);
            INC (multiple);
        END;
    yMax := multiple - 2;
    yPos := counter - AspectV;
END;

```



```

tempStore := xMax / divisionValue;
determineDivision := xMax MOD divisionValue;
xPos := xPos - (AspectH * determineDivision);
IntToStr (tempStore, xString);
CASE activeSystem
  OF
    bearing, standard:
      TextOut (window, (width DIV 2 + xPos), (height DIV 2 + 20), xString , Length
(xString));
      |
      MacWin:
        TextOut (window, (xPos), (20), xString , Length (xString));
  END;
tempStore := yMax / divisionValue;
determineDivision := yMax MOD divisionValue;
yPos := yPos - (AspectV * determineDivision);
IntToStr (tempStore, yString);
CASE activeSystem
  OF
    bearing, standard:
      TextOut (window, (width DIV 2 - 20), (height DIV 2 - yPos), yString, Length
(yString));
      |
      MacWin:
        TextOut (window, (20), (yPos), yString, Length (yString));
  END;
IF axesDrawn
  THEN
    TextOut (window, (width - 250), height - 50, scaleString, Length
(scaleString));
  END;
END DrawDivisionMarks;

PROCEDURE PlotPoint ( x, y : REAL);

BEGIN
  IF NOT (scaleSet)
  THEN
    SetScale (1);
  END;
  DotAt (Round (x * xScale), Round (y * yScale));
END PlotPoint;

PROCEDURE PolarPlotPoint (radius, angle : REAL);

VAR
  x,y : REAL;

BEGIN
  IF activeSystem = MacWin
  THEN
    angle := - angle;
  END;

```

```

angle := ReviseAngle (ToDeg (angle));

x := radius * cos (angle * convertRad);
y := radius * sin (angle * convertRad);

IF ~scaleSet
    THEN      (* Make sure the graph has a scale *)
        SetScale (1);
    END;
DotAt (Round (x * xScale), Round (y * yScale));
END PolarPlotPoint;

(*----- Main Code -----*)

BEGIN
    (* Initialize variables *)
    (* the import of GraphWindow sets up the window for us. *)
    window := GetWindow (); (* obtain the variable associated with the graph window *)
    GetWDimensions (width, height);
    scaleSet := FALSE;
    labelSet := FALSE;
    axesDrawn := FALSE;
    graphArg := 0.0;
    SetCoordSystem (standard); (* default *)
    Home;
    (* Pause until user closes the screen *)

END GraphPaper.

```

Again, the reader is invited to peruse the details of the API, but they are not going to be explained in detail here. However, the implementations of *GraphWindow* and *GraphPaper* between them, ought to provide a means of getting started on other programs in Windows NT.

[Contents](#)

18.7 Chapter Summary

This chapter covered these topics:

- a review of basic graphics concepts
- a discussion of coordinate systems
- the definition and use of the high level module *GraphPaper*
- an elementary introduction to fractals and string art
- an extended discussion of the implementation of *GraphPaper* in both MacOS and Windows NT

It included discussion of the following Modula-2 items:

Reserved Words

none

Standard Identifiers

none

Imports:

Standard:

- none

Non-Standard:

- From GraphPaper:
CoordSystem, AngleType, LabelType, SetCoordSystem, SetAngleType, Home, ShiftOrigin, GetDimensions, GetLocation, MoveBy, MoveTo, Move, GetCurrentAngle, Turn, TurnTo, LineBy, LineTo, Line, Dot, DotAt, SetLabels, ShowLabels, ShowAxes, SetScale, PlotPoint, PolarPlotPoint
- From GraphWindows:
GetWindow, GetWDimensions

Macintosh Specific:

- From SYSTEM (p1):
INT16
- From Events:
Button, keyDown, mouseDown

- From Quickdraw:
qd, PenSize, WindowRef, InsetRect, SetPort, SetRect, FrameRect, FrameOval, PaintOval, RGBColorRandom, RGBForeColor, InvertColor, MoveTo, LineTo
- From QuickdrawText:
DrawString, TextSize
- From MacWindows:
FrontWindow, WindowRecord, NewWindow, documentProc, NewCWindow
- From Types:
Rect, OSErr, UInt16;
- From OSUtils:
SysEnviroms, SysEnvRec;
- From DateTimeUtils:
GetDateTime;
- From Sound:
SysBeep;

Windows-32 Specific (Stonybrook):

- From SYSTEM:
FUNC
- From WIN32:
UINT, WPARAM, LPARAM, LRESULT, BOOL, RECT, HBRUSH, HWND
- From WINUSER:
GetDC, ReleaseDC, SetRect, InvalidateRect, GetClientRect, RegisterClass, ShowWindow, GetMessage, TranslateMessage, DispatchMessage, LoadCursor, LoadIcon, IDC_ARROW, FillRect, DefWindowProc, UpdateWindow, PostQuitMessage, BeginPaint, EndPaint, CreateWindow, GetSysColor, LOWORD, HIWORD, COLOR_WINDOW, WS_OVERLAPPEDWINDOW, CS_VREDRAW, CS_HREDRAW, CS_BYTEALIGNCLIENT, WM_SIZE, WM_DESTROY, WM_PAINT, WM_SYSCOLORCHANGE, WM_ERASEBKGD, WNDCLASS, MSG, PAINTSTRUCT
- From WINGDI:
MoveToEx, TextOut, CreateSolidBrush, GetDeviceCaps, VERTRES, HORZRES

- From WINX:

Instance, PrevInstance, CmdShow, DeleteBrush, NULL_HWND, NIL_RECT,
NULL_HBRUSH, NULL_HINSTANCE, NULL_HMENU, NIL_POINT

[Contents](#)

18.8 Assignments

Questions

1. What is meant by the term "screen resolution"?
2. Who was René Descartes, and what were some of major contributions to mathematics?
3. What procedure of the API is employed in (a) MacOS and (b) Windows NT to determine the current size of a window?
4. What is a pixel?
5. Why do you suppose that no graphics module was standardized by the ISO committee?
6. Give a detailed explanation of the three kinds of coordinate systems used in this chapter.
7. Give a detailed explanation of the three kinds of angle measurement units.
8. What is a fractal?
9. Look up a typical *turtlegraphics* module and discuss how it differs from *GraphPaper*. Write a definition module for *Turtlegraphics*.
10. The moire pattern in [figure 18.12](#) is just made with straight lines. Yet, the result does not have lines right across the figure in most instances. Why not?
11. Look up a MacOS programming reference and thoroughly comment everything in the MacOS versions of *GraphWindow*, explaining what every import is and what it does. Be sure to detail the Window record type.
12. Look up a Windows NT programming reference and thoroughly comment everything in the NT version of *GraphWindow*, explaining what every import is and what it does.
13. Detail the differences between the style of event-based programming in MacOS and in Windows.

Problems

14. Produce a module *ComplexPlane* that graphs points in a complex plane. It will take arguments that are complex numbers, not ordered pairs (x, y). As this is drawing complex numbers as points, the contents are likely to have little relationship with those of *GraphPaper*. Be sure to get your design approved by your instructor before implementation.
15. Implement a *turtlegraphics* module as you described in question 9.
16. Write and test a module to graph functions. It should be able to read from the keyboard and parse such expressions as $y = 3x^5 + 4x^{-2} + \sin(x) + 1/x + \ln(x)$, where the "^" symbol is taken to mean "to the power of." That is, it should correctly evaluate polynomial expressions, reciprocal, negative exponents, and the names of the functions in *RealMath*, and then graph them over some user-specified range of values.
17. Look up (in a calculus book) how to draw such figures as the rose, limaçon, and lemniscate in polar coordinates. Construct and test a module to read formulas in polar form in terms of r and θ . (similar to the previous module in Cartesian form) , parse such formulas, and graph them.
18. Parametric equations are given in the form $x = f(t)$; $y = g(t)$, where t is another variable, called a parameter, and s equation forms are called parametric equations. Construct and test a module to read from the keyboard functions in parametric form and graph them, as in the last two questions.

19. Write a module that graphically illustrates the results of adding two dimensional vectors.
20. The module *TestGraph* in [section 18.3](#) did not completely test all aspects of *GraphPaper*. Extend it so that it does test everything.
21. Look up at least two more fractals and write code to construct them.
22. Look up Hilbert's space filling curve and write a module to produce it.
23. Sierpinski's curve in [section 18.4.3](#) was based on a rectangle. Write a module to make the analogous figure bases on five sides.
24. Write a module to draw a table and four chairs around it.
25. Produce a piece of string art of your own design. Perhaps you can make the bounding figure something other than a rectangle.
26. Add to *GraphPaper* the ability to clear the screen from one graph and start over.
27. Write a module to draw pie charts and/or bar graphs. The data could be read from a file.
28. Port *SillyBalls* to WindowsNT. Detail all the changes you have to make.
29. Implement the module *Keyboard* for Windows NT.
30. If you have a Windows 32-bit compiler other than StonyBrook, port the *GraphWindow* and *GraphPaper* modules to it.

Projects

31. Add to *GraphPaper* the ability to save and load graphics files in some standard format (such as PICT or TIFF.)
 32. Add to the MacOS version of *GraphWindow* the ability to switch to another application, switch back and get the window redrawn.
 33. Write a module to graphically illustrate the results of a tree traversal as it is taking place.
 34. Write a module to graphically illustrate the contents of a heap in tree form as it is being sorted.
-

[Contents](#)

Chapter 18

Introduction To Graphics

[18.0 Chapter Goals](#)

[18.1 Basic Graphics Concepts](#)

[18.1.1 Discrete Grids--Graphing Pixels](#)

[18.1.2 Where is the origin?](#)

[18.1.3 Measuring Angles](#)

[18.2 A Graphics Environment](#)

[18.3 Using The Module GraphPaper](#)

[18.4 Recursive Drawing--Fractals](#)

[18.4.1 Snowflake Fractals](#)

[18.4.2 A Tree Fractal](#)

[18.4.3 Singly-Recursive Snowflake-like Fractals](#)

[18.4.4 Sierpinski's Curve](#)

[18.5 String Art](#)

[18.6 An Extended Example--Implementing GraphPaper](#)

[18.6.1 Defining the Module GraphWindow](#)

[18.6.2 Implementing GraphWindow in MacOS](#)

[18.6.3 Implementing GraphWindow in Windows NT](#)

[18.6.4 Implementing GraphPaper in MacOS](#)

[18.6.5 Implementing GraphPaper in Windows NT](#)

[18.7 Chapter Summary](#)

[18.8 Assignments](#)

19.0 Chapter Goals

The purpose of this chapter is to extend the concept of abstract data types to include object classes. On completing the chapter, the student should understand and be able to use the concepts of Object Oriented Modula-2 to design and construct both traced and untraced object classes and instantiate and manipulate their objects:

Data Representation Abstractions

General:

Dynamic objects and their semantics.

Realized in the Modula-2 notation:

Traced and untraced objects and their safeguarded and unsafeguarded module containers.

Data Manipulation Abstractions

General:

Design and creation of ADTs in the form of object classes.

Realized in the Modula-2 notation:

Creation and manipulation of objects, their attributes and methods.

Programming Abstractions

General:

Controlling component access and visibility, inheritance.

Realized in the Modula-2 notation:

The use of READONLY and REVEAL to control use and visibility, INHERIT, and the GUARD statement for dynamic class membership.

19.1 Introduction to Object Oriented Thinking

At this stage of the discussion, the reader is must be very much aware of the value, for both design and debugging purposes, of breaking large software projects down into constituent parts. Procedures are one such constituent, or piece of reusable software, and modules are another. Procedures are activity or flow-centred constituents; they deal with actions that either change data or alter the user perception or experience of the data. Modules are usually collections of such procedural activities, but may sometimes be centred around a particular data type and its appropriate manipulations. In the latter case, the term *Abstract Data Type* is appropriate to describe the entire collection (with its associated procedures) encapsulated in the module, and thinking in terms of such ADTs is a somewhat more data-centred approach to software design and implementation.

However, such ADTs are still rather activity-centric, for they offer collections of procedures, each of which can be passed a data entity of the appropriate type. A more data-centric approach than even the module ADT involves subordinating the manipulative activities and attributes to each individual data item, rather than to the type as a whole. Notationally, this is usually reflected by writing:

for a procedural or activity-centric approach:

```
Manipulate (a);  
c := SomeAttributeOf (a);  
Combine (a, b);
```

but, for a data-centric approach:

```
a.Manipulate;  
c := a.SomeAttribute;  
a.Combine (b);
```

To make things more concrete, consider an abstract data entity *BankAccount* with associated procedures *Create*, *Destroy*, *Balance*; *Credit* and *Debit*. In the conventional and action-oriented approach used so far in this text, one would write a module called *Accounts* containing, say, a type *Account* and the two procedures. A client program would declare the entity *BankAccount* to be of type *Account* and then issue invocations such as:

```
Create (BankAccount);  
Credit (BankAccount, 23.45);  
Debit (BankAccount, mortgage);
```

or, possibly, if the ADT were imported qualified,

```
Accounts.Debit (BankAccount, orthodonticsPayment);  
Current := Accounts.Balance (BankAccount);
```

For each of the procedure invocations, an account name has to be provided in the client code in order for the action to be valid.

In the object oriented approach, *BankAccount* would be the name of an object that would have its own access to the specified procedures and information (in this case, the balance) and the code of the procedures would be invoked by sending a message to the object to act as an agent in performing the required task, thus:

```
BankAccount.Credit (23.45);  
BankAccount.Debit (mortgage);  
Current := BankAccount.Balance;
```

The action of *Credit* does not stand alone waiting, as it were, to be given an account on which to act. Rather, it is owned by the account, which "knows" how to credit itself. That is, both attributes or qualities on the one hand, and activities or manipulative procedures on the other are tied to the data item itself, rather than being independent entities in their own right.

An astute reader ought to see some similarities between objects and modules and be able to take some things for granted here.

- an object defines a scope within which its attributes and procedures are visible
- an object can hide some information items or procedures and prevent clients from using them

However, some other things are implicit in the data-centricity of this approach:

- responsibility for carrying the message lies with the agent, not the procedure
- two similar agents might interpret messages with the same name in different ways
- there ought to be a way to re-use some of the code declared for one object in another object

[Contents](#)

19.2 Object Oriented Terminology

19.2.1 Basic Definitions

Here is an initial summary of common definitions, based on the discussion so far:

A data-centric entity with its own information and manipulative activities is called an object. Each information item of an object is called an attribute, and each activity it can engage in is called a method. A use of "object.attribute" or of "object.method" is called a message to the object, and in the context of carrying such a message, an object may be termed an agent.

If each individual object and its attributes and methods had to be individually declared, these ideas would not give programs much expressive power. However, like variables, constants, and procedures, objects with the same structure are said to have a common type.

The type of an object is called its class. The attributes and methods of a class taken collectively are referred to as the members (a common term in other notations) or components (the Modula-2 term) of the class.

NOTE: In notations other than Modula-2, the term *component* is usually used of a more or less self-contained routine or collection of routines, such as a module, rather than of an entity constituting part of a class.

An entity that is of a class type is said to be an instance or a member of that class.

Thus, objects are not created alone, but as entities belonging to a class (usually declared in a fashion somewhat as in the pseudocode:

```
Class
  Account =
    attributes
      balance : real
    methods
      procedure credit (amount : real)
      procedure debit (amount : real)
    begin initialization
      balance <= 0.0
    end Account
```

In practice, the actual syntax of a class declaration may loosely resemble that of a module or a record, but in most object notations, there are several differences from either syntax.

The generation of a specific object (that is, an instance) from some class is called instantiation.

When an object of this class is generated, the data items can be initialized by some kind of internal code, in the manner shown. One might have something like:

```
var
  bankAccount : Account;
```

and then, later,

```
Create (bankAccount);
```

where *Create* is a built in function to allow for the generation and initialization of an object before it is used.

As suggested at the end of [section 1](#), object notations almost always allow for information hiding. As with political systems, which either allow everything not prohibited, or prohibit everything not allowed, there are two common approaches to object scope data hiding, depending on the notation rules:

- all attributes and methods of the object are visible to clients unless marked *private* OR (the approach taken in Modula-2, by the way)
- all attributes and methods of the object are hidden unless they are marked *reveal*

Similarly, object notations usually allow attributes that are variables to be protected in some way, so that clients can read the value but not change it. This eliminates the necessity (recommended for modules) of keeping the value in the implementation, and only putting a procedure to fetch it into the definition. In most object notations, a visible attribute variable can be changed by clients unless it is marked by a reserved word such as *protected* or *readonly*. Thus, one might have, on the one hand, something like:

```
Class
  Account =
    protected balance
    private overdrawn
    attributes
      balance : real
      overdrawn : boolean
    methods
      procedure credit (amount : real)
      procedure debit (amount : real)
    begin initialization
      balance <= 0.0
      overdrawn <= false
    end Account
```

or, on the other hand, (the Modula-2 hand) something like:

```
Class
  Account =
```

```

reveal credit, debit, Readonly balance
attributes
  balance : real
  overdrawn : boolean
methods
  procedure credit (amount : real)
  procedure debit (amount : real)
begin initialization
  balance <= 0.0
  overdrawn <= false
end Account

```

In either of these renditions, the method components *credit* and *debit* are available for use by objects in the surrounding scope. Of the attributes, *balance* can be read, but not written to, and *overdrawn* is kept private. It should be noted, in accord with the preceding discussion, that object notations can achieve these effects in a variety of ways. The discussion is deliberately being kept general at this point, and can apply to any such notation, but the definition below is used in the OOM-2 standard.

An entity whose value is not permitted to be changed is said to be immutable.

19.2.2 Reference versus Value Objects

The distinction between a value and a reference to a value been made earlier in the text in connection with *first* value and variable parameters; *second* the name of a data store and its address (or a pointer).. In brief;

If an identifier denotes the name of the data item itself, it is the name of a value. If it denotes a means of access to a value then it is a reference.

In many object notations, an object name denotes a reference rather than the actual value of the data. Thus, if *obj1* and *obj2* are both objects, then an assignment like

```
obj1 := obj2
```

using value semantics would take the data value in *obj2* and copy it to the location of *obj2*. However, if reference semantics underlie the object notation (the usual), it is the reference that is copied, not the value. Since an object, as with a record or an array, could be rather large, this is probably a good choice for language designers. It has the consequence, however that such objects, like the entities accessed by pointers, can only be created dynamically, not statically.

However, as in the case of pointers, such semantics does raise the issues of dangling references and garbage. It is all very well to do things this way, but what if a series of operations leaves some object with no reference at all (garbage)? After all, if objects are declared and initialized dynamically, that sounds a lot like there is memory allocation and deallocation going on, at least behind the scenes, even if not out in the open as with pointers. Indeed, some object notations require that their object entities be explicitly destroyed by the program so that the memory can be reclaimed, and programs written with such semantics face the very real possibility that if the programmer is not careful with references, the available memory will become exhausted.

However, nearly all implementations of object notation have some form of garbage collection to scour the program memory space for objects that have no current valid references, and collect their memory. That is, an object with no references can be automatically destroyed by the garbage collection routines. It may be possible to invoke these with an explicit procedure call, but often they simply lurk in the background and the program is unaware of them.

Objects that are tracked by a garbage collector and cannot be destroyed manually by the program code are said to be traced. Objects that are subject to manual destruction by explicit code and so are not tracked by a garbage collector are said to be untraced.

In addition, it is worthwhile to distinguish between program units containing one or the other of these two kinds of object:

A program unit (such as a Modula-2 module) that contains only traced objects is called safeguarded. One that contains even one untraced object is called unsafeguarded.

Of course, in some object notations, this distinction will not have to be made, because there will only be one kind of object. Indeed, if language designers (or ISO committees) are dealing with a language that has no pointers, it is easy to declare that all objects are to be traced (essentially what happens in Java). Not so in Modula-2, which has both partly because the base language already had pointers and so had the possibility of dynamic allocation and deallocation already.

There are echoes here of comments on loop control variables and whether they can be threatened or not. In some languages, they can be reassigned inside the FOR loop they control; in Modula-2, they cannot be threatened in any way. It should be clear that if objects are to be traced, they have to be protected from attempts to manually dispose of them. For instance, taking the address of a traced value (or casting a traced object reference into a pointer type) constitutes an obvious threat, and is one of the things that would have to be forbidden.

[Contents](#)

19.3 Getting Started with Object Oriented Modula-2

19.3.1 A Little History

Classical Modula-2 as originally defined by Wirth did not have any object notation whatsoever. Later, Wirth himself devised some simple object extensions for Pascal, and this Object Oriented Pascal became the language in which the entire Macintosh operating system was written. Because of the surface similarities between Pascal and Modula-2, some vendors adopted these extensions for their versions of Modula-2 as well.

When the ISO committee produced the initial base standard (ISO/IEC 10514-1) for the Modula-2 notation, it also lacked any object notation. However, by the time this standard was published WG13 (modula-2) was already developing the supplementary standards for Generic Modula-2 (ISO/IEC 10514-2) and Object Oriented Modula-2 (ISO/IEC 10514-3), and these were subsequently approved as supplementary and independent standards. That is, vendors could implement only the base standard if they wished, or they could include in their package either or both of the additional parts. Most are expected to include both.

In the light of the introductory discussions, it is worthwhile noting some of the decisions the committee took with respect to Object Oriented Modula-2 (often abbreviated to OOM-2).

- objects are always references, never values
- both traced and untraced objects are available, hence both safeguarded and unsafeguarded modules exist
- safety of traced modules is enforced; threatening is not allowed
- garbage collection is available for traced classes
- all attributes and methods of the object are hidden unless they are marked *reveal*
- syntax differs from both that of records and that of modules
- initialization of objects is via an object body
- an object body is a block body so it can have an exception clause
- the term *component* was used for an item declared within a class declaration
- the term *member* was selected for the relationship between an object and its class

There are several other issues on which the committee had to make decisions, but these will be pointed out when the related concepts are introduced later in the chapter.

19.3.2 Some Simple OOM-2 Programs

Example 1:

The purpose of this example is to illustrate some of the simple syntax for declaring and using OOM-2 classes and objects. The module establishes a class of objects called *rectangle* with some attribute and method components, and generates two messages to those components. Its only output is to print the area before (0) and after (12) the call to *SetDims*.

The keyword **CLASS** is used to declare a class, in like manner to the use of **TYPE** in the base language. Observe that the keyword **REVEAL** is used to allow visibility outside the class, and that the class maintains the hidden variables *length* and *width* internally. In Modula-2 classes are untraced unless marked with **TRACED**, and modules are safeguarded unless marked with **UNSAFEGUARDED**. Note the initialization code, which is automatically applied whenever an object of this class is instantiated with the OOM-2 pervasive **CREATE**. It is important to remember that the declaration of entities of the class *rectangle* in a **VAR** clause only sets aside enough static memory for the reference variable; no dynamic memory is allocated until the object is instantiated with the **CREATE** statement.

```
MODULE TestRectangleClass1;
```



```

(* to demonstrate simple traced class syntax
   by R. Sutcliffe 1998 09 24 *)

IMPORT SWholeIO;

TRACED CLASS rectangle;
  REVEAL SetDims, Area, sides;
  (* declare components of class *)
  (* first the attribute components *)
  CONST
    sides = 4;
  VAR
    length, width : INTEGER;

  (* and then the method components *)
  PROCEDURE SetDims (len, wid : INTEGER);

  BEGIN
    length := len;
    width := wid;
  END SetDims;

  PROCEDURE Area () : INTEGER;
  BEGIN
    RETURN length * width;
  END Area;

BEGIN (* initialization *)
  SetDims (0,0);
END rectangle;

VAR
  theRect : rectangle;

BEGIN (* main *)
  CREATE (theRect);
  (* print out initial area *)
  SWholeIO.WriteInt (theRect.Area(), 10);
  theRect.SetDims (4, 3);
  (* print out area after new dimensions set *)
  SWholeIO.WriteInt (theRect.Area(), 10);

END TestRectangleClass1.

```

Observe the prettyprinting style. Here, the components of the class (but not its body) have been indented for easy reading, and the first line has been treated somewhat like a procedure heading by putting it all on one line, rather than as a type heading and starting a new line. This is because TYPE introduces a whole section that could include several declarations, whereas CLASS heads only one such declaration at a time.

Example 2:

Here is an implementation in OOM-2 of the declaration of the *Account* class discussed in earlier sections. No code has been written to declare and instantiate individual accounts at this time.

```

MODULE TestAccount1;

```

```
(* to demonstrate simple traced class syntax
   by R. Sutcliffe 1998 09 24 *)
```

```
TRACED CLASS Account;
  REVEAL Credit, Debit, READONLY balance;
VAR
  balance : REAL;
  overdrawn : BOOLEAN;

  PROCEDURE Credit (amount : REAL);
BEGIN
  balance := balance + amount;
  CheckOverdraw;
END Credit;

  PROCEDURE Debit (amount : REAL);
BEGIN
  balance := balance - amount;
  CheckOverdraw;
END Debit;

  PROCEDURE CheckOverdraw;
BEGIN
  IF balance < 0.0
  THEN
    overdrawn := TRUE
  ELSE
    overdrawn := FALSE;
  END;
END CheckOverdraw;

BEGIN (* initialization *)
  balance := 0.0;
  overdrawn := FALSE;
END Account;

END TestAccount1.
```

Any number of accounts can be declared in a VAR clause and then instantiated by a CREATE statement. Each one has its own copies of the attribute components, so that when any of these is used in a method component or is referred to by a client, the data obtained is that belonging to the object that invoked the member. The reader should observe that this is a major difference between classes and modules, for any data global to an ADT declared in a module is global to all entities of the type that are declared, not associated with each one individually.

19.3.3 Summary of Basic OOM-2 Traced Class Semantics

Both examples in the previous section were of traced classes. The reserved word TRACED must precede any such class to indicate to the garbage collector that it is to trace objects of this class from the time they are instantiated and, if all references to the object cease to exist, it may destroy the object by reclaiming its memory.

A traced class declaration may contain constant, type, variable, and method (procedure) declarations. It also defines a scope of visibility for these items, and unless one or more of them is revealed outside that scope, they are not visible in the surrounding module. Thus, REVEAL serves a purpose in the context of a class within a module similar to that served by

EXPORT in the context of a module within a module.

On the other hand, Modula-2 class declarations clearly have some similarity to records. The components of a class are in a sense similar to the fields of a record.

Constants and types declared in a class declaration can be referred to by using the appropriate identifier qualified by the class name in the surrounding scope, provided they have been revealed there. Of course, constants can also be referred to qualified by any object name instantiated from that class. However, the base rules of the language prohibit use of a type name qualified by a variable name, so such references are not useful. Moreover, OOM-2 states that the reference to a constant qualified by an object name is not a constant expression, so it cannot be used in such situations.

However, variables declared in a class differ from variables declared in a module in that there is no memory associated with them until an instance of the class has been created. The memory is associated with each individual instance or member, not with the class as a whole, and it is reserved by a call to CREATE. They cannot be referred to qualified by the class name, only qualified by an object instance name of that class. This memory also cannot be manually returned by a corresponding destructor; it is completely under the control of the garbage collector thereafter.

Moreover, although methods are declared with the reserved word PROCEDURE, and they do look and behave very much like procedures, they are useful only with members of the class in which they are declared. Unlike procedures declared elsewhere in a module, they do not have a type; therefore, there are no method variables or method constants like there are procedure variables and constants, and method names also cannot be referred to qualified by the class name, only by an object name.

Not only do Modula-2 object references get their memory when CREATE is called, the data fields of the object can be initialized automatically at that time by placing statements in the body of the class declaration.

Should the programmer decide to re-use a traced object reference, say, in a loop, by calling CREATE on an object reference a second time, a new memory allocation is done, just as when doing this with NEW on a pointer. Thus,

```
CREATE (thingy);
  (* some code here *)
CREATE (thingy); (* again *)
```

results in a different reference for *thingy*. As with pointers, unless the previous value has been stored elsewhere, garbage may now have been generated, for the first memory set aside for *thingy* no longer has this reference (unless it has been assigned.) For traced objects, this is of no consequence. The garbage collector merely takes note of the fact that this reference no longer accesses the memory in question, and when it next activates, if it finds the memory has no references at all, the memory is collected.

To illustrate a few of the things that *cannot* be done, consider the following (erroneous) module.

```
MODULE TestObjectErrors;

CLASS Bad; (* because it is not marked as traced *)
END Bad;

TRACED CLASS One;
  MODULE NotAllowed; (* only const, var, type and procedure *)
  END NotAllowed;

  PROCEDURE Hit;
    MODULE Allowed; (* but a dynamic module in method is ok *)
    END Allowed;
  END Hit;

END One;

TRACED CLASS Two;
  CONST
```

```

    secret = 3;
END Two;

CONST
    badConstant1 = 2 * secret; (* didn't reveal secret *)

TRACED CLASS Three;
    REVEAL secret, thingy;
    CONST
        secret = 3;
    VAR
        thingy: CARDINAL;
    PROCEDURE Empty ();
    END Empty;
END Three;

CONST
    badConstant2 = 2 * secret; (* even if we had, it can't be referred to like this *)
    goodConstant = 2 * Three.secret; (* but rather this way *)
VAR
    three : Three;
CONST
    badConstant3 = 2 * three.secret; (* reference to such a constant not allowed in
constant expression *)

TRACED CLASS Four;
    REVEAL inClassType;
    TYPE
        inClassType = CARDINAL;
END Four;

VAR
    mine : Four.inClassType; (* ok to do this with a type *)
VAR
    four : Four;
VAR
    another : four.inClassType; (* but base language rules don't allow this *)

BEGIN
    mine := 5;
    mine := three.secret; (* this reference OK *)
    Three.thingy := 5; (* can't refer to a variable by class name*)
    Three.Empty (); (* or to a method, whether revealed or not *)
    three.thingy := 5; (* can refer to a variable by object name *)
    three.Empty (); (* or to a method *)
END TestObjectErrors.

```

When this piece of "code" was fed to the compiler, it responded as follows:

```

#      3  CLASS Bad; (* because it is not marked as traced *)
#####      ^ 188: untraced classes not allowed in safeguarded modules
File "TestObjectErrors.MOD"; Line 3
#      7      MODULE NotAllowed; (* only const, var, type and procedure *)

```

```

#####          ^ 173: module declaration not allowed in class declaration
File "TestObjectErrors.MOD"; Line 7
#   23      badConstant1 = 2 * secret; (* didn't reveal secret *)
#####          ^ 73: identifier not declared
File "TestObjectErrors.MOD"; Line 23
#   36      badConstant2 = 2 * secret; (* even if we had, it can't be referred to like
this *)
#####          ^ 73: identifier not declared
File "TestObjectErrors.MOD"; Line 36
#   41      badConstant3 = 2 * three.secret; (* reference to such a constant not
allowed in constant expression *)
#####          ^ 74: wrong class of identifier
File "TestObjectErrors.MOD"; Line 41
#   54      another : four.inClassType; (* but base language rules don't allow this *)
#####          ^ 74: wrong class of identifier
File "TestObjectErrors.MOD"; Line 54
#   59      Three.thingy := 5; (* can't refer to a variable by class name*)
#####          ^ 170: variable attributes cannot be accessed via class type
File "TestObjectErrors.MOD"; Line 59
#   60      Three.Empty (); (* or to a method, whether revealed or not *)
#####          ^ 171: entity not revealed in defining class
File "TestObjectErrors.MOD"; Line 60
#   62      three.Empty (); (* or to a method *)
#####          ^ 171: entity not revealed in defining class
File "TestObjectErrors.MOD"; Line 62
Modula2 - Execution terminated!
### MPW Shell - Execution of makeout terminated.

```

The reasons for most of the error messages should be obvious in the light of the discussion. One that may escape the reader with no previous experience of objects is the second to last one. Just as the use of items from an unrefined generic module on their own is meaningless, so also is the use of variables and methods qualified by only their class name outside the class. However, it is acceptable to refer to constants and types this way. Only if an object of class *Three* has been instantiated with a declaration and *CREATE* does it make sense to refer to *Entity.secret*. No reference to *Secret* on its own is valid outside the class declaration.

Experimentation (or the writing of bad code) ought to reveal some other things that are forbidden--specifically things that might threaten a traced object reference.

One cannot:

- declare a pointer to an object of a traced class reference
- declare a pointer to a structure that contains an object of a traced class reference
- make a reference to an object of a traced class in a variant field of a record
- manually deallocate or destroy an object of a traced class

One cannot in a safeguarded module:

- declare or import an object of an untraced class type
- take the ADR of a reference to an object of a traced class
- CAST a traced object
- pass a reference to an object of a traced class (or a component thereof) to an ARRAY OF LOC parameter

If the programmer does subvert the safety of traced objects using something from the module *SYSTEM* to obtain a reference

to a traced object in an unsafeguarded module, the reference so obtained is not tracked by the garbage collector, and the validity of the program becomes questionable (a supposedly traced object has an untraced reference). Such attempts should not therefore be made.

[Contents](#)

19.4 Untraced Objects

Untraced OOM-2 objects (in the sense described in [section 19.2](#)) are declared and used in almost exactly the same way as are traced objects. They are allowed to have the same kinds of components, and the same things are forbidden. The differences are as follows:

- the reserved word TRACED is omitted
- the reserved word UNSAFEGUARDED must be placed in front of the word MODULE to mark the compilation unit as containing untraced objects
- the programmer is responsible for destroying the object when it is no longer needed
- the object body can have a FINALLY clause (a finalization body) that is executed when any of its objects are destroyed (this is of course forbidden for traced objects).

Example: The class in this demonstration allows actions to be timed. When a member of the class *Timer* is instantiated, the current time is stored. At any time desired, the elapsed time on that timer can be displayed. The code here is only a demonstration, and is not very complete, but it is adequate for a minute/second timer. In this example, the class has a private procedure to calculate a *DateTime* difference, and a revealed one to invoke this and display the result. The FINALLY clause is there just to demonstrate that it is invoked when the object is destroyed. In practice, this clause would only be necessary if some other memory deallocation had to be done as part of the destruction of the object.

```
UNSAFEGUARDED MODULE TimeClassDemo;

FROM SysClock IMPORT
    DateTime, GetClock;
FROM SWholeIO IMPORT
    WriteCard;
FROM STextIO IMPORT
    WriteLn, WriteString, ReadChar, WriteChar;
FROM Storage IMPORT
    ALLOCATE, DEALLOCATE;

CLASS Timer;
    REVEAL WriteElapsed;
    VAR
        start, elapsed : DateTime;

    PROCEDURE CalcElapsed;
    VAR
        current : DateTime;
        borrow : BOOLEAN;
        difference : INTEGER;
    BEGIN
```

```

GetClock (current);
(* note the necessity to use integer arithmetic on the cardinal fields *)
difference := INT (current.second) - INT (start.second);
borrow := (difference < 0);
elapsed.second := difference MOD 60; (* put back as a cardinal *)
difference := INT (current.minute) - INT (start.minute);
IF borrow
    THEN
        DEC (difference)
    END;
borrow := (difference < 0);
elapsed.minute := difference MOD 60;
(* repeat for hours, days, and years to finish off. *)
(* only enough code here for a minute/second timer *)
END CalcElapsed;

PROCEDURE WriteElapsed;
BEGIN
    CalcElapsed;
    WITH elapsed
    DO
        (* add more code for years, months, days, hours if desired *)
        WriteCard (minute, 1);
        WriteChar (":");
        WriteCard (second, 1);
    END;
END WriteElapsed;

BEGIN (* initialize *)
    GetClock (start);
FINALLY
    WriteLn;
    WriteString ("This timer has been destroyed.");
    WriteLn;
    (* well, we could put something useful here, anyway. *)
END Timer;

VAR
    time : Timer;
    ch : CHAR;

BEGIN (* main *)
    CREATE (time);
    WriteString ("Timer started; press return to see elapsed time ");
    ReadChar (ch);
    time.WriteElapsed;
    DESTROY (time);
END TimeClassDemo.

```


There may be a surprise or two in this code, so it ought to be read over carefully. *First*, observe that when untraced objects are created and destroyed, the memory management is turned over to `ALLOCATE` and `DEALLOCATE` respectively, which must therefore be made visible, as is the case with using pointers with `NEW` and `DISPOSE`. These imports did not have to be made for working with traced classes, because the garbage collector is given authority over the relevant memory. Second, note that the module has to be marked `UNSAFEGUARDED` because it contains an untraced class. This would be the case even if it contained several traced classes as well; the presence of even one untraced class means that the module is unsafeguarded. This is also true if any imports are made from an unsafeguarded module, so safeguarded modules cannot do that either.

When the code was run, and the *return* key pressed almost immediately, the result was as shown below, verifying the correct action of the destructor:

```
Timer started; press return to see elapsed time
0:2
This timer has been destroyed.
```

The reader should note that, although there is little difference between the outward effects of using traced or untraced objects, the efficacy of secure memory management by the garbage collector is lost when untraced objects are employed. It would be advisable therefore, to use only traced objects unless there are compelling reasons to do otherwise. Almost all the remaining examples in this chapter will be of traced objects in safeguarded modules. To drive this point home, consider what happens when `CREATE` is called more than once on an untraced object reference.

```
CREATE (thingy);
(* some code here, but no DESTROY *)
CREATE (thingy); (* again *)
```

Unlike the situation with traced objects, the garbage that is generated by leaving memory that was allocated to the reference *thingy* in the first call to `CREATE` without a valid reference is not collected, and indeed it cannot be, for the garbage collector has never been told to trace objects of such a class. Such garbage will simply pile up until the program runs out of memory and (probably) crashes.

It should also be apparent that it is an error to attempt to use `DESTROY` on an object of a traced class.

[Contents](#)

19.5 Assignment and Comparison of Objects

NOTE: Except where otherwise indicated, the comments in this section apply to all OOM-2 objects, whether members of traced classes or of untraced classes.

As might be expected, two object references that are members of the same class are assignment compatible. (It turns out that there is somewhat more to object assignment compatibility than this, however; see [section 19.7](#).) Thus, if one has:

```
VAR
    obj1, obj2 : ClassyClass;
```

it is perfectly legitimate in the course of a program to write

```
obj1 := obj2;
```

as long as the programmer understands that this is a copying of references, not of the values accessed by those references. Thus, there is little point in such an assignment being made before the second of the two objects referred to has even been created, as the call to CREATE will change the reference value. Likewise, two objects can be compared in expressions such as those found in

```
IF obj1 = obj 2
WHILE obj1 # obj2
```

and again, it is the references that are being compared, not the values accessed by those references. Of course, if two references are to the same memory, the values stored there are indeed the same, but if two different but compatible objects do happen to have the same values in their attribute components, this kind of code will never discover that fact.

Earlier, when dealing with pointers, there was a useful value (**NIL**) that could be used for initialization. A similar value is used for initialization of object references.

The pervasive value EMPTY is assignment compatible to all object references.

It will, of course, cause an exception to be raised if one attempts to use a component of an object that has the value EMPTY, just as it would to attempt to dereference the pointer value NIL. (The object in question is the empty object; it currently has no components.)

In yet another parallel with pointers, should the attempt to call CREATE on an object fail to allocate memory for the object, its reference will have the value EMPTY after the failure, but no exception will be raised. Bullet-proof code ought to have a check for success after a CREATE:

```
CREATE (myObject);
```

```
IF myObject # EMPTY  
  THEN (* carry on *)
```

For traced objects only, there is an automatic initialization to the value **EMPTY** on declaration. This includes traced objects declared in static modules, in procedures, and even in dynamic modules inside procedures. This will help prevent references to objects not yet created. Setting the value of a traced object reference to **EMPTY** is also a way of informing the garbage collector that the memory it previously held may be collected (as least, insofar as this reference is concerned.)

On the other hand, if a reference to an object of a traced class that currently has the value **EMPTY** is assigned some other value, the garbage collector is informed of that fact as well.

In addition, any references to objects of traced classes declared in procedures (including parameters) are reported to the garbage collector as empty (that is, defunct) when the procedure is exited (whether normally or exceptionally). This is also true of any traced object references that may have been used to construct components of an untraced object that is subsequently destroyed.

For references to objects of untraced classes, the standard does not say what the value of the reference is after a call to **DESTROY**. It has to be assumed, therefore, that the value is undefined at that point, not that it is necessarily **EMPTY**.

Although pointers and object references behave in similar ways, they are different kinds of entities. This should be evident if for no other reason than that pointers are dereferenced using the dereferencing operator **^**, whereas the components of an object are accessed using a qualified identifier. These differences mean, among other things, that there is no assignment compatibility between object references and pointers.

[Contents](#)

19.6. Encapsulation of Classes in Separate Modules

The base Modula-2 language provides for the creation of new ADTs using separate library modules with definition and implementation parts. Those ADTs may be object classes, and if so, the class has to be defined in the definition module, and then declared (with code) in the implementation, in the manner of any other type. The following rules need to be observed regarding compatibility between the two parts of the separate library module that contains such a class:

- all methods defined in the definition module must be declared in the implementation module
- items revealed in the definition are not revealed again in the implementation
- other attributes and methods may be revealed in the implementation if desired
- attributes defined in the definition are not repeated in the implementation
- attributes and methods not defined in the definition but declared in the implementation are hidden
- syntax must match for objects and methods
- objects defined as traced must be declared as traced
- the safeguarding (or not) of the implementation module must be the same as the definition module

There is nothing startling in these rules. They are much the same as one would expect from the rules for declarations of procedures that have been defined in a definition module. In general, if one does the expected thing in this respect, all will be well.

One that may need comment is that if components are declared in the implementation (but were not defined in the definition part) then, even if they are revealed in the implementation module, they are only visible there, not in any client software--which can only look at the definition.

The following example illustrates these rules with a class that stores and can display a date. Obviously, the example is quite simple-minded, in that one would want, at the very least, to be able to do the display in a variety of formats, and only one is provided here.

```
DEFINITION MODULE Dates;  
(* Simple class definition  
by R. Sutcliffe 1998 09 21 *)
```

```
TRACED CLASS Date;  
  REVEAL SetDate, WriteDate;  
  VAR  
    year, month, day : CARDINAL;  
  
  PROCEDURE SetDate (yr,mo,da : CARDINAL);  
  PROCEDURE WriteDate;  
END Date;
```

```
END Dates.
```

```
IMPLEMENTATION MODULE Dates;  
(* Simple class implementation  
by R. Sutcliffe 1998 09 21 *)
```

```
FROM SWholeIO IMPORT  
    WriteCard;
```

```
TRACED CLASS Date;  
    (* reveals and attributes not repeated here *)
```

```
PROCEDURE SetDate (yr,mo,da : CARDINAL);
```

```
BEGIN
```

```
    year := yr;  
    month := mo;  
    day := da;
```

```
END SetDate;
```

```
PROCEDURE WriteDate;
```

```
BEGIN
```

```
    WriteCard (year, 0);  
    WriteCard (month, 0);  
    WriteCard (day, 0);
```

```
END WriteDate;
```

```
END Date;
```

```
END Dates.
```

This was tested with the simple application below. Observe again the necessity to *CREATE* the instance of the class before using it. Failure to do so will result in an empty class exception being raised.

```
MODULE TestDates;  
(* to test class Dates  
R. Sutcliffe 1998 09 21 *)
```

```
FROM Dates IMPORT  
    Date;
```

```
VAR
```

```
    today : Date;
```

```
BEGIN
```

```
CREATE (today);  
today.SetDate (1998, 12, 25);  
today.WriteDate;  
END TestDates.
```

The result of running this was of course:

```
1998 12 25
```

Observe that if the implementation needs to reveal additional attributes or methods it can do so, but this is of no consequence to the class interface in the definition and is entirely a local matter. Moreover, if additional class components are declared in the implementation, they may be revealed there as well.

[Contents](#)

19.7 Inheritance

19.7.1 Why Inherit

All that has been said thus far is merely syntactical sugar for things that could have been done other ways in the base syntax, without involving OOM-2 at all. Indeed, for many data types, static modules implement ADTs quite nicely, and for most of the rest, Generic templates pick up the remaining slack. However, the real power of using object methods lies in the ability to create objects that are based on pre-existing ones, with perhaps a few changes. This allows the code already written for use in one object class to be reused in another object class, or, perhaps more accurately, it allows the functionality desired for some new object to be handed off to an existing object. With either way of looking at it, the wheel does not have to be re-invented quite so often.

The ability to take an existing object and make it the basis of a new one is called inheritance.

A class that inherits from another one is called a derived class or a child class or a subclass.

A class that has one or more children is called a parent class or an ancestor class or a superclass.

Object oriented software also differs from generic approaches in that much of what is done with objects is dynamic, whereas generic templates are simply means of generating new and specialized static modules. There are various strategies for inheritance among the common object oriented languages. In some, a class can inherit from several other classes, incorporating all their features into itself (multiple inheritance). In others, a class is permitted to inherit only from one other class--though it in turn could inherit from another one--(single inheritance.)

19.7.2 Inheritance in OOM-2

The basic model for OOM-2 is that of single inheritance. One class can say that it is inheriting another class, and this acts in a sense as an import does into a module scope. All the components of a parent class are available inside the scope of the subclass, not just the ones that have been revealed for clients. This allows the subclass declaration to add additional attributes or methods of its own while having full access to those of the original class. It can reveal new components that it defines, but it cannot reveal. An exception is that hidden components (found only in the implementation of a separate library module, not in its definition) are not accessible to subclasses. Apart from this, each class controls the visibility of its own components.

However, a chain of inheritances may be built in OOM-2, wherein a class inherits from a second class which in turn inherits from an third class, and so on, with each new class making changes to the previous component structure.

WARNING: In OOM-2 a traced class can inherit only from a traced class, and an untraced class can inherit only from an untraced class.

The reader should be able to see the logic of the above restriction. It would make no sense to derive a class the objects of which the garbage collector is supposed to trace from a class whose objects it cannot trace, and vice-versa.

Starting with the class *Date* defined in the previous section, we can create a new subclass with additional properties in the following manner, but this time in a program rather than in a library:

```
MODULE DemoDates1;
(* simple demonstration of inheritance in OOM-2
   by R. Sutcliffe; revised: 1998 09 22 *)
IMPORT Dates;
FROM SWholeIO IMPORT
  WriteCard;
```

```

FROM STextIO IMPORT
    WriteString, WriteChar;

TRACED CLASS Dates1;
    INHERIT Dates.Date;
    REVEAL WriteUSDate;

PROCEDURE WriteUSDate;
VAR
    monthStr : ARRAY [0..8] OF CHAR;
BEGIN
    IF month = 1
    THEN
        monthStr := "January"
    ELSIF month = 2 THEN
        monthStr := "February"
    ELSIF month = 3 THEN
        monthStr := "March"
    ELSIF month = 4 THEN
        monthStr := "April"
    ELSIF month = 5 THEN
        monthStr := "May"
    ELSIF month = 6 THEN
        monthStr := "June"
    ELSIF month = 7 THEN
        monthStr := "July"
    ELSIF month = 8 THEN
        monthStr := "August"
    ELSIF month = 9 THEN
        monthStr := "September"
    ELSIF month = 10 THEN
        monthStr := "October"
    ELSIF month = 11 THEN
        monthStr := "November"
    ELSIF month = 12 THEN
        monthStr := "December"
    ELSE
        monthStr := "xxxxxxx"
    END;
    WriteString (monthStr);
    WriteCard (day, 0);
    WriteChar (",");
    WriteCard (year, 0);
END WriteUSDate;
END Dates1;

VAR
    aDate : Dates1;
BEGIN
    CREATE (aDate);

```



```
    aDate.SetDate (1998, 11, 5);
    aDate.WriteUSDate;
END DemoDates1.
```

This program has the output:

November 5, 1998

If the classes have class bodies (constructors), the order that these are run is from root to leaf, that is from the eldest parent through its child, and so on down to the last subclass.

If one is dealing with a chain of untraced classes, and these each have a destructor or **FINALLY** clause (recall that this is impossible with traced classes) then these are executed in reverse order, that is from child class to parent to grandparent and so on up to the top superclass. Care must be taken with the use of **FINALLY** clauses in this respect; this order might well be different from the order of finalization of the modules in which these classes are defined, and if the module finalizations could interfere with the object finalizations. Such complications are a good reason to use traced classes; then the programmer does not have to worry about object finalization.

19.7.3 Assignment Compatibility between Classes and Subclasses

In OOM-2, a subclass *BClass* of a superclass *AClass* is also a subtype of *AClass*. This means that if one has, for instance:

```
VAR
    superObject : AClass;
    subObject   : BClass;
```

then *subObject* is assignment compatible with *superObject*, but not vice versa, so that

```
subObject := superObject;
```

is incorrect, but

```
superObject := subObject;
```

is correct.

What happens here (in the latter case) is that the variable *superObject*, which has been statically declared to be of (class) type *AClass*, has been dynamically been assigned an object of (class) type *BClass*. That is to say, the dynamic type of an object referenced by an identifier may not be the same as its static declaration, it may dynamically become that of a subtype.

It may at time be necessary to check on the dynamic type status of a class variable, and this can be done with the function **ISMEMBER**, which returns a boolean according to the following rule:

```
ISMEMBER (param1, param2);
```

allows either parameter to be a class name or an object reference. **ISMEMBER** returns **TRUE** if objects of the class type of the first parameter are assignment compatible (subclass or same class) to objects of the type of the second parameter and **FALSE** otherwise. That is, **ISMEMBER** can be invoked as

```
ISMEMBER (objectRef, Classname2);
```

and returns TRUE if *objectRef* is a member of (a subclass of) *Classname2*. If the first parameter is a class and the second an object reference, ISMEMBER returns TRUE if objects of the class are assignment compatible with the given object. Moreover,

```
ISMEMBER (ClassName1, Classname2);
```

returns true if objects of *ClassName1* are assignment compatible (subclass or same class) to objects of the *ClassName2* Class, and false otherwise (that is, if the class of the first parameter is equal to or is a subclass of the second parameter). As another fine point on static vs dynamic semantics, the note at the end of [section 19.7.3](#) is to be taken in the sense that the constructor chain is evaluated in static order, but the destructor chain (where present) is evaluated dynamically.

19.7.4 Overriding Methods in Subclasses

A class can do more than just decide to inherit the components of a superclass. It can also replace some of the method components (though not the attribute ones) with different ones, provided they have the same interface. That is, a method of the superclass can be overridden in a subclass by a new method provided it has the same definition (name and parameters) as the original. The new method applies to objects of the subclass, but not to those of the superclass. It has the same syntax as the original method but different semantics. All this is accomplished merely by placing the reserved word **OVERRIDE** in front of the new method.

The subclass can have attribute components added to the original list, but none of the attribute components can be overridden, because the only point to doing that would be to alter their type, and then the interface would have changed, and the new class would not properly be a child class at all.

Here is a simple example to illustrate this syntax and semantics. The first class shown implements adding accumulators. These are commonly used in programs to keep running sums and are initialized to zero. The class exports *Clear* to reset the accumulator, *Accumulate* to add to the register variable (which is kept private) and *Display* to print the result. We could just abandon one accumulator after using it a while and create a new one, not bothering with *Clear*, but this seems unnecessarily messy.

The reader should observe that it is considered to be good taste to have an object know how to display itself (if this is needed) rather than allowing outside information to be leaked so that it can be displayed external to the object. That is, one keeps data hidden and provides procedures to manipulate it where possible.

The second class has the same interface, but accumulates a running product instead. It can use the same *Display*, and reveals the same components, but overrides both *Clear* and *Accumulate* for different semantics, because a multiplication register has to be initialized to zero, and of course, has to multiply each new item rather than add it.

```
MODULE OverrideDemo;
(* demonstration of inheritance and overriding methods
   R. Sutcliffe 1998 09 21 *)

FROM SLongIO IMPORT
    WriteReal;
FROM STextIO IMPORT
    WriteLn;

(* first class implements an adding accumulator *)
TRACED CLASS Accumulator;
    REVEAL Clear, Accumulate, Display;
```

```
VAR
    register : LONGREAL;
```

```
PROCEDURE Clear;
BEGIN
    register := 0.0;
END Clear;
```

```
PROCEDURE Accumulate (new : LONGREAL);
BEGIN
    register := register + new;
END Accumulate;
```

```
PROCEDURE Display ;
BEGIN
    WriteReal (register, 15);
END Display;
```

```
BEGIN (* class body *)
    Clear;
END Accumulator;
```

```
VAR
    acc : Accumulator;
```

```
(* second class does the same thing but with a multiplying version *)
```

```
TRACED CLASS AccumulatorM;
    INHERIT Accumulator; (* same interface *)
    OVERRIDE PROCEDURE Clear;
    BEGIN
        register := 1.0; (* if cleared to zero, multiplications wouldn't do much. *)
    END Clear;
```

```
    OVERRIDE PROCEDURE Accumulate (new : LONGREAL);
    BEGIN
        register := register * new;
    END Accumulate;
```

```
BEGIN
    Clear; (* uses this class' clear *)
END AccumulatorM;
```

```
VAR
    accM : AccumulatorM;
```

```
BEGIN
    CREATE (acc); (* set up an adder *)
    acc.Accumulate (9.8); (* and exercise it *)
    acc.Accumulate (6.7);
    acc.Accumulate (11.35);
    acc.Display;
```

```

WriteLn;

CREATE (accM); (* now do a multiplier *)
accM.Accumulate (9.8);
accM.Accumulate (6.7);
accM.Accumulate (11.35);
accM.Display; (* uses the original; not overridden *)
WriteLn;

END OverrideDemo.

```

The output from this module is:

```

27.8500000000000
745.241000000000

```

Observe that even within the subclass, a reference to *Clear* is to the overridden method, and not to the one in the superclass. In this case, there were no additions needed to the list of things to be made public, so the subclass did not need a *REVEAL* clause at all.

Outside the class, client software references are to the dynamic class of the object. Thus, if one were to write

```
accM := acc;
```

and thereby change the dynamic class of *acc*, it would become an adder rather than a multiplier, because the methods would be chosen according to the dynamic class.

It is important to realize that overriding a method does not create a new component or do anything with scope. Rather, it replaces the implementation of the method with a new one and retains the interface as it was. Thus, the overriding method must have the same procedure type as the original one. If it is a regular procedure, the parameters must be the same in number, position, and type; if a function procedure, the return type must also be identical. This also means that an inherited identifier is in the scope of the subclass; it cannot simply be re-used for something else. Overriding is changing the implementation; it is not replacing the name with a new one that happens to be spelled the same way. In the event that the class inherited from had a definition and declaration in a separate library module, and one plans to override methods of a superclass, this must be signalled in the definition with the word *OVERRIDE* and then implemented accordingly. Moreover, where such a class definition has been made, inheritance must be specified in the definition only and not in the declaration. The *INHERIT* clause was allowed in the previous example within a declaration only because there was no definition already in existence.

So, for instance, if one wanted to define and declare a new separate *Date* class, this time overriding the method for date display rather than adding a new one as previously, the modules would be written as follows:

```

DEFINITION MODULE Dates2;
(* Simple class override definition
by R. Sutcliffe 1998 09 21 *)
FROM Dates IMPORT
    Date;

TRACED CLASS Date2;
    INHERIT Date;
    OVERRIDE PROCEDURE WriteDate;
END Date2;

```

```
END Dates2.
```

Observe the import of the superclass and its inheritance here, followed by the flagging of an override for the one method. Everything else has been left as it was. In the implementation below, no inheritance clause is needed (the one in the definition suffices) and all that remains is to implement the overridden method.

```
IMPLEMENTATION MODULE Dates2;  
(* Simple class override implementation  
by R. Sutcliffe 1998 09 21 *)
```

```
FROM SWholeIO IMPORT  
  WriteCard;
```

```
TRACED CLASS Date2;  
  OVERRIDE PROCEDURE WriteDate;  
  BEGIN  
    WriteCard (day, 0);  
    WriteCard (month, 0);  
    WriteCard (year, 0);  
  END WriteDate;  
END Date2;  
  
END Dates2.
```

Of course, once having flagged a method for overriding in the definition, it must actually be done in the implementation or the compiler will generate an error.

19.7.5 Class and Object References

There are several situations worth noting where code has to use object or class references in somewhat unusual ways.

Accessing an Overridden Method

Sometimes in the declaration of a class it may be necessary to refer to an overridden method from a superclass. Because it has been overridden, the original name is not available for use in the subclass. If this is the case, it can be referred to by the syntax

```
superclassName.originalMethod
```

even if the superclass referenced is several levels up the inheritance chain. By this means, all the methods up the chain are available for use in a subclass, if necessary. Such methods are not directly available outside the subclass, however, unless it defines and reveals a new method that simply calls the old one under a new name.

```
TRACED CLASS sub;  
  INHERIT oldClass;  
  REVEAL newMethod;  
  OVERRIDE PROCEDURE oldMethod;  
  . . . .
```

```

PROCEDURE newMethod;
BEGIN
    oldClass.oldMethod;
END;
END sub;

```

However, in such a case, it seems to make more sense to keep the old method under its original name and define a new one with a different name. This type of reference is more likely to be used if the old method is incorporated into and made a part of the new one, or used in the initialization body.

Using a Class Name as a Qualifier

As indicated in [section 19.3.3](#), one cannot refer to a method of a class by using its class name as qualifying identifier. After all, without the name of an object to qualify them, such a method has nothing to act upon. One also may not refer to a variable declared in a class in such a manner. However, constants and types declared in a class can be referred to outside the class in this manner, providing they have been revealed. Indeed, this is the only way to refer outside the class to a type declared in a class.

```

TRACED CLASS Classy;
    REVEAL myType,    (* can refer to outside as Classy.myType *)
            myVar,    (* qualify outside only by an object name of this class *)
            myConst, (* OK to use outside as Classy.myConst, not in constant
expression *)
            myMethod; (* qualify outside only by an object name of this class *)
    TYPE
        myType : ARRAY [1..2] OF CARDINAL;
    VAR
        myVar : myType;
    CONST
        pi = 3.14159;
    PROCEDURE myMethod;
        some body
    END myMethod;
END Classy;

```

SELF--The Hidden Parameter

On occasion, it may be necessary for an object to refer to itself somewhere in its declaration. This could come about, for instance, if one of its methods took as a parameter an item whose type is that of the object. In such cases, there is always a reference available to the object using the identifier SELF, and this can in turn of course qualify any component if desired.

```

TRACED CLASS Naval;
    VAR
        ok : BOOLEAN;
    PROCEDURE Gaze (at : Naval);
        some code
    END Gaze;

    PROCEDURE check (VAR allRight : BOOLEAN);

```

```

        some code
    END Gaze;

BEGIN (* initialization body or another method body *)
    Gaze (SELF);
    Check (SELF.ok)
END Naval;

```

Inside the various methods, and the initialization body, SELF is available as a kind of hidden parameter, so that every method "knows" what object invoked it and has access to its other components.

It should be obvious that SELF cannot be re-assigned; after all, to do so within itself would create a logical tangle. It is called an immutable entity, just as are any class variables marked as READONLY.

Yet another time that SELF may be useful is if it become necessary to assign the object reference to some variable defined globally to the class scope.

```

TRACED CLASS SomeClass;
    PROCEDURE AMethod;
    BEGIN
        IF condition
            THEN
                globalVar := SELF;
            END;
        END AMethod;
    END SomeClass;

```

Circular Definitions and Declarations

Suppose that two class definitions or declarations make references to each other, in the same manner that two procedures may invoke each other. Perhaps each declares an object variable of the other class type. In this case, neither may come first if there is a strict "declare it before use" rule, so OOM-2 allows the use of FORWARD in such a context, in a manner similar to the way it is used in mutually recursive procedures.

```

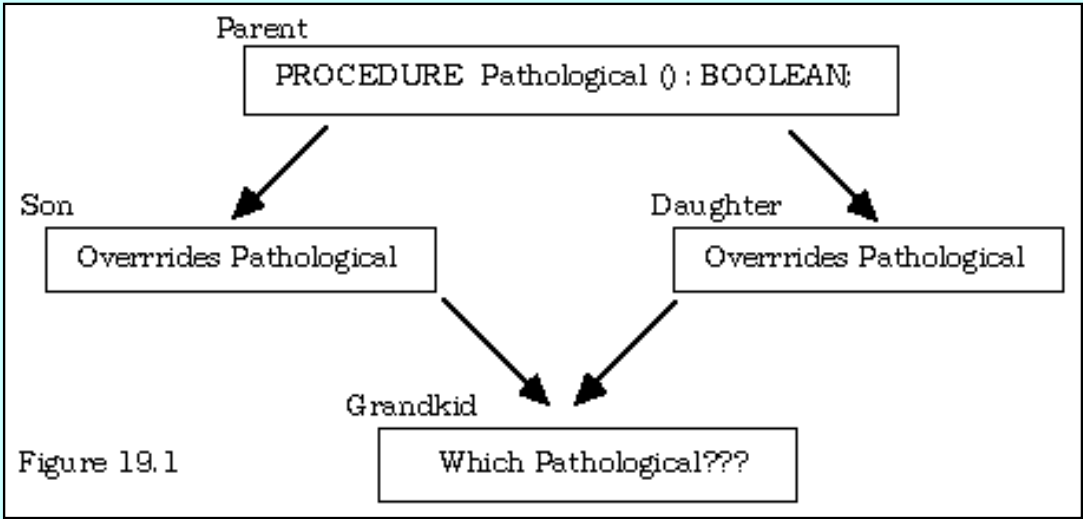
CLASS EventHandler; FORWARD
CLASS MyWindow;
    VAR
        event : EventHandler;
    ...
END MyWindow;
CLASS EventHandler;
    VAR
        theWindow : MyWindow;
    ...
END EventHandler;

```

19.7.6 Why Single Inheritance?

Some OO languages use multiple inheritance. That is, a class is permitted to inherit from more than superclass. The reader who immediately thinks of a score of projects that such a facility would enable ought to consider the following example. Suppose one defines four classes *Parent*, *Daughter*, *Son*, and *Grandkid*. Class *Parent* defines a component

Pathological. *Daughter* and *Son* both override *Pathological* and define their own version. *Grandkid* inherits from both *Daughter* and *Son*, and does not override *Pathological*. In this admittedly incestuous tangle, which version of *Pathological* is a component of *Grandkid*?



Because this problem cannot be resolved without introducing some special magical rule (such as: all versions of a contested method have to be qualified by a class name), many languages, including OOM-2 decline to get involved in such problems by allowing only single inheritance in the first place.

19.8 Abstract Classes

Sometimes, one wishes to derive a whole family of classes from a single superclass. The interface for them all is common, but one or more components is known to be different ahead of time for each subclass. Perhaps only one method is different, but in any case there isn't a specific class that can be thought of as ancestor to all the variations, despite the commonality.

In such a situation, the common ancestor class might have the method in question with only a procedure heading and no implementation, with the intention that this (vacuous) method will have to be overridden by a subclass to produce anything useful.

A class that contains unimplemented methods intended to be overridden by implemented methods in a subclass is called an abstract class.

A method that is intended to be overridden and so has no implementation is called a deferred method or an abstract method.

In OOM-2 both the abstract class itself and any deferred method that causes the class to be abstract must be tagged by preceding them with the keyword **ABSTRACT**. The example that follows has some simple classes that do little more than illustrate these rules.

Here, the plan is to declare a number of geometric classes, each of which will have a method to obtain its dimension(s) and methods to compute the area and perimeter. Initially, an abstract class is defined that reveals three abstract methods for these tasks.

```
DEFINITION MODULE GeoFigures;  
(* demo of abstract class syntax using plane figures  
  R. Sutcliffe 1998 09 25 *)
```

```
TRACED ABSTRACT CLASS GeoFigure;  
  REVEAL InitSize, Area, Perim;  
  ABSTRACT PROCEDURE InitSize;  
  ABSTRACT PROCEDURE Area () : REAL;  
  ABSTRACT PROCEDURE Perim () : REAL;  
END GeoFigure;
```

```
END GeoFigures.
```

NOTE: The keyword **ABSTRACT**, when used, comes immediately before **CLASS** (and after **TRACED**, where the latter is used).

In the event that a method is marked as abstract in a definition, it must not be implemented. Indeed, if, as in this case, the declaration will not be implementing any methods at all, the class declaration still has to be present, but can be empty.

```
IMPLEMENTATION MODULE GeoFigures;
```

```
TRACED ABSTRACT CLASS GeoFigure;  
END GeoFigure;
```

```
END GeoFigures.
```

For simplicity, several classes have been derived from this abstract one in a single module below.

Observe that each gathers the data necessary for its own version of the object and does its own version of the calculation, while maintaining a consistent interface for the whole family of classes. Moreover, the class *Square* is a subclass of the class *Rectangle*, which in turn is a specialization of the abstract class *GeoFigures* above. *Square* employs only one of the two dimensions defined in *Rectangle*.

One might be tempted to make the setting of the class attribute variables part of the initialization block of the class, but this is not a good idea in this case, because *Square* would then end up running both blocks, doing the one for *Rectangle* first, then its own. Thus, this code has been made a method called from the outside.

```
MODULE DemoGeos;
```

```
(* demonstration of subclassing with abstract superclasses  
  by R. Sutcliffe 1998 09 25 *)
```

```
IMPORT GeoFigures;
```

```
FROM STextIO IMPORT
```

```
  WriteString, WriteLn, SkipLine;
```

```
FROM SRealIO IMPORT
```

```
  ReadReal, WriteReal;
```

```
FROM RealMath IMPORT
```

```
  pi;
```

```
(* first subclass of GeoFigures *)
```

```
TRACED CLASS Rectangle;
```

```
  INHERIT GeoFigures.GeoFigure;
```

```
  VAR
```

```
    length, width : REAL;
```

```
  OVERRIDE PROCEDURE Area () : REAL;
```

```
  BEGIN
```

```
    RETURN length * width;
```

```
  END Area;
```

```
  OVERRIDE PROCEDURE Perim () : REAL;
```

```
  BEGIN
```

```
    RETURN 2.0 * (length + width);
```

```
  END Perim;
```

```

OVERRIDE PROCEDURE InitSize;
BEGIN
    WriteString ("enter length ==> ");
    ReadReal (length);
    SkipLine;
    WriteString ("enter width ==> ");
    ReadReal (width);
    SkipLine;
END InitSize;
END Rectangle;

(* now subclass this in turn, making more overrides *)
TRACED CLASS Square;
INHERIT Rectangle;

OVERRIDE PROCEDURE Area () : REAL;
BEGIN
    RETURN length * length;
END Area;

OVERRIDE PROCEDURE Perim () : REAL;
BEGIN
    RETURN 4.0 * length;
END Perim;

OVERRIDE PROCEDURE InitSize;
BEGIN
    WriteString ("enter side length ==> ");
    ReadReal (length);
    SkipLine;
END InitSize;
END Square;

(* and, a completely different subclass of the original *)
TRACED CLASS Circle;
INHERIT GeoFigures.GeoFigure;
VAR
    radius : REAL;

OVERRIDE PROCEDURE Area () : REAL;
BEGIN
    RETURN pi * radius * radius;
END Area;

```

```

OVERRIDE PROCEDURE Perim ( ) : REAL;
BEGIN
    RETURN 2.0 * pi * radius;
END Perim;

OVERRIDE PROCEDURE InitSize;
BEGIN
    WriteString ("enter radius ==> ");
    ReadReal (radius);
    SkipLine;
END InitSize;
END Circle;

VAR (* for main block to test this stuff *)
    rect : Rectangle;
    square : Square;
    circ : Circle;

BEGIN (* main module *)
    CREATE (rect);
    rect.InitSize;
    WriteString ("The area is ");
    WriteReal (rect.Area(), 15); WriteLn;
    WriteString ("The perimeter is ");
    WriteReal (rect.Perim(), 15); WriteLn;WriteLn;

    CREATE (square);
    square.InitSize;
    WriteString ("The area is ");
    WriteReal (square.Area(), 15); WriteLn;
    WriteString ("The perimeter is ");
    WriteReal (square.Perim(), 15); WriteLn;WriteLn;

    CREATE (circ);
    circ.InitSize;
    WriteString ("The area is ");
    WriteReal (circ.Area(), 15); WriteLn;
    WriteString ("The perimeter is ");
    WriteReal (circ.Perim(), 15); WriteLn;WriteLn;

END DemoGeos.

```

One run of this module is shown below, with the inputs in bold.

```
enter length ==> 2
enter width ==> 3
The area is 6.00000000000000
The perimeter is 10.0000000000000
```

```
enter side length ==> 5
The area is 25.0000000000000
The perimeter is 20.0000000000000
```

```
enter radius ==> 1
The area is 3.1415925025940
The perimeter is 6.2831850051880
```

This is very simple, and very minimal, both in form and in content. Many more figures and their associated methods could be derived, and there are more interesting methods such as displaying the figure on the screen that could be written. Also, the verbal clues in the output need work; the student who is interested may clean this code up considerably.

[Contents](#)

19.9 Guard Statement

Sometimes, the code one may wish to execute may depend on the (dynamic) type of the object reference. This situation is similar to ones that result in the use of a CASE statement, where various courses of action are selected based on the value of some variable at the time. However, the CASE statement will not serve in this instance, because it is selection on the dynamic type of the object, not on its value that one is interested in.

A statement that selects actions based on some dynamic state is said to be a guard statement, and the actions that are protected by the statement are said to be guarded.

Thus, OOM-2 introduces a new kind of selection, or guarded statement with a syntax similar to that of CASE, but specialized to the dynamic type of an object reference. The simplest syntax is shown in the skeleton that follows:

```
GUARD objectDenoter AS
  : className1 DO
    statementSequence1 |
  : className2 DO
    statementSequence2 |
    ....
ELSE
    statementSequence(n+1)
END;
```

Although the guarded statement sequence uses DO, there is no END, but rather, the next selection in the list is delimited from the last by the vertical bar. Thus, a prettyprinting style has been chosen that lines up the items in the list of selections and the ELSE and END. Here, the names of the classes to which the object's dynamic class is being matched are preceded by a colon.

The first (in order of appearance) class name to which the object denoter is assignment compatible is selected and its statement sequence is executed. As with a CASE statement, execution of the GUARD statement then terminated and control goes to the next statement after the END.

WARNING: An empty object reference selects the ELSE clause. If there is no match and no ELSE clause, an exception is generated.

The following trivial example only detects the class of the variable and prints a message. It is provided only to illustrate the syntax and semantics. *ClassA* is the superclass of the others, with classes B and D as subclasses of *ClassA*. *ClassC* is a subclass of *ClassB*.

```
UNSAFEGUARDED MODULE guardTest;
(* to illustrate simple guard statement
  by R. Sutcliffe 1998 09 28
```

This example done in a StonyBrook beta version
in which TRACED objects had not yet been implemented. *)

```
FROM STextIO IMPORT
  WriteString, ReadChar;
FROM Storage IMPORT
  ALLOCATE;
CLASS ClassA;
  END ClassA;
CLASS ClassB;
INHERIT ClassA;
  END ClassB;
CLASS ClassC;
INHERIT ClassB;
  END ClassC;
CLASS ClassD;
INHERIT ClassA;
  END ClassD;

VAR
  ch : CHAR;
  a : ClassA;
  b : ClassB;
  c : ClassC;
  d : ClassD;

BEGIN
  CREATE (a);
  CREATE (b);
  CREATE (c);
  CREATE (d);
  a := d;
  GUARD a AS
    : ClassA DO
      WriteString ("Class a found ");|
    : ClassB DO
      WriteString ("Class b found ");|
    : ClassC DO
      WriteString ("Class c found ");|
    : ClassD DO
      WriteString ("Class d found ");
  ELSE
    WriteString ("none found ");
  END; (* guard *)
  ReadChar (ch);
```

```
END guardTest.
```

Look at the above code carefully in the light of the semantics given. It will always output the message for *ClassA*. Since the object *a* is of that class, and it comes first in the list, it will always be selected. What if the order of the list is reversed to be D-C-B-A? *ClassD* would then be selected, because the object *a* is of the dynamic type *ClassD* and it comes first in the list. If *ClassD* were not present as a selection, *ClassA* would eventually be chosen. If we changed the reversed code and wrote:

```
b := c;
GUARD b AS
  : ClassD DO
    WriteString ("Class d found ");|
  : ClassC DO
    WriteString ("Class c found ");|
  : ClassB DO
    WriteString ("Class b found ");|
  : ClassA DO
    WriteString ("Class a found ");
ELSE
  WriteString ("none found ");
END; (* guard *)
```

we would select and get the message for *ClassC*, the first one to which *b* is assignment compatible (because *ClassC* is the dynamic type of *b* at that point.) Care must therefore be taken when writing a **GUARD** statement that the list is in an appropriate order for what the programmer wants the code to do. There is more to the **GUARD** statement than this, however. The components of the selector can be accessed by **GUARDing** it **AS** a specific entity of the given class type. To do this, an object denoter is placed before the colon for that item in the list, and this effectively opens up a scope for that name, which reference is assigned the original selector. Via the new reference denoter, the components of the original object are now accessible. Here is a simple illustration, this time with only two classes. The object reference *a* is passed to a procedure parameter of the base class type (which can also accept any reference to a subclassed object). Depending on the dynamic type of the actual parameter, the procedure selects and prints information correctly. Observe that the denoters *itema* and *itemb* do not need to be declared; they are generated (and are only visible) locally to the statement sequence. These denoters become local aliases within the portion of the **GUARD** statement where they are in scope, and this prevents inappropriate use of attribute or method components of a class by ensuring that they are only referred to in a dynamic context where they make sense.

WARNING: The denoters indicated at the beginning of each list item (where used) are not only local to their own statement sequence, they are immutable there (they cannot be reassigned.)

```
MODULE GuardTest2;
(* to illustrate guard statement with new denoter access
   by R. Sutcliffe 1998 09 28 *)
```



```

FROM STextIO IMPORT
    WriteString, WriteLn;
FROM SWholeIO IMPORT
    WriteCard, WriteInt;
TRACED CLASS ClassA;
    REVEAL card;
VAR
    card : CARDINAL;
END ClassA;
TRACED CLASS ClassB;
INHERIT ClassA;
    REVEAL int;
VAR
    int : INTEGER;
END ClassB;

PROCEDURE Check (item : ClassA);
BEGIN
    GUARD item AS
        itemb : ClassB DO
            WriteInt (itemb.int, 10);
            WriteLn; |
        itema : ClassA DO
            WriteCard (itema.card, 10);
            WriteLn; |
        ELSE
            WriteString ("none found ");
        END; (* guard *)
END Check;

VAR
    a : ClassA;
    b : ClassB;
BEGIN
    CREATE (a);
    CREATE (b);
    a.card := 12;
    b.int := -15;
    Check (a); (* should print value of a.card *)
    Check (b); (* should print value of b.int *)
    a := b;    (* change dynamic type of a *)
    Check (a); (* should print value of b.int *)
END GuardTest2.

```

The output from this program was, as expected:

```
12  
-15  
-15
```

[Contents](#)

19.10 Additions to the Libraries

Unlike the Generic extensions, those for OOM-2 require some library support. That for exceptions parallels the one in the base language; coroutines requires one additional procedure, and garbage collection also has a library.

19.10.1 Exceptions

OOM-2 extensions to the base language define four new exceptions. Detection of the first three is mandatory, but detection of the last one is optional. They are:

1. `emptyException`

raised whenever an attempt is made to access an object via an empty reference

2. `abstractException`

raised whenever an attempt is made to call an abstract method

3. `guardException`

raised if there is no match on the list of selections (possibly due to an empty reference) and no ELSE

4. `immutableException`

an implementation may choose to raise this if there is an attempt to change an immutable entity.

In addition, there is a new system module, which behaves as if it had the following definition:

```
DEFINITION MODULE M2OOEXCEPTION;
```

```
(* Provides facilities for identifying exceptions of the extended language *)
```

```
TYPE
```

```
  M2OOExceptions =  
    (emptyException, abstractException, immutableException, guardException);
```

```
PROCEDURE M2OOException (): M2OOExceptions;
```

```
(* If the current coroutine is in the exceptional execution state because  
   of the raising of an exception of the language extensions, returns the  
   corresponding enumeration value, and otherwise raises an exception. *)
```

```
PROCEDURE IsM2OOException (): BOOLEAN;
```

```
(* If the current coroutine is in the exceptional execution state because  
   of the raising of an exception of the language extensions, returns  
   TRUE, and otherwise returns FALSE. *)
```

```
END M2OOEXCEPTION.
```

The entities of this system module behave in exactly the same way as do those of the module `M2EXCEPTION` described in [section 10.2.1](#) except that they act on the OOM-2 exception enumeration instead of that of the base language.

19.10.2 Coroutines

The module `COROUTINES` (see the chapter on that subject) in the base standard has one procedure added to provide support for OOM-2. It is

```
PROCEDURE DISPOSECOROUTINE (VAR cr: COROUTINE);
```

```
(* Declare that the coroutine identified by cr has reached the end of its lifetime. *)
```

The procedure `DISPOSECOROUTINE` is used to inform the garbage collector that the coroutine of the actual parameter has reached the end of its lifetime. The garbage collector takes this to mean that all the traced variables in that coroutine have become defunct and may be collected.

19.10.3 The Module Garbage Collection

In some implementations it may be possible to turn garbage collection off and on, or to force it to take place at specified points in a program when it is convenient, rather than leave it up to automatic routines. Thus, the procedures of the following module *may* do something, but only if the implementation allows this. Their semantics are described in the comments and need no further explanation.

```
DEFINITION MODULE GARBAGECOLLECTION;

(* Provides facilities for controlling the garbage collector. *)

PROCEDURE IsCollectionEnabled (): BOOLEAN;
  (* If garbage collection is enabled then returns TRUE and otherwise returns FALSE. *)

PROCEDURE CollectionEnable (on: BOOLEAN);
  (* If on is TRUE then enable garbage collection; otherwise if on is FALSE and
     garbage collection can be disabled then disable garbage collection. *)

PROCEDURE ForceCollection;
  (* If garbage collection can be forced then force it else do nothing. *)

END GARBAGECOLLECTION.
```

19.11 Extended Example--Points and Vectors

The module Vectors in [section 7.11](#) and the module Points in section 6.9 share some basic concepts in common. These two data types could be implemented as classes with the latter as the superclass as shown below.

```
DEFINITION MODULE PointClassA;

TRACED CLASS Points;
  REVEAL assignR, firstCoord, secondCoord, abs, arg, assignP, reflectX, reflectY,
    reflect0, reflect45, scale, rotate, translate;
  TYPE
    Point = ARRAY [1 .. 2] OF REAL;
  VAR
    point : Point;

  PROCEDURE assignR (x, y : REAL);
  PROCEDURE firstCoord () : REAL;
  PROCEDURE secondCoord () : REAL;
  PROCEDURE abs () : REAL;
  PROCEDURE arg () : REAL;
  PROCEDURE assignP (abs, arg : REAL);
  PROCEDURE reflectX;
  PROCEDURE reflectY;
  PROCEDURE reflect0;
  PROCEDURE reflect45;
  PROCEDURE scale (scaleFactor : REAL);
  PROCEDURE rotate (rotAngle : REAL);
  PROCEDURE translate (deltaX, deltaY : REAL);
END Points;

END PointClassA.
```

A few names have been changed here to make them more generic for subclasses. The reader should compare this with the original and note the simplicity of the definition when the parameter of type Point can be left out because the procedure acts only on the variable point that is its attribute component. When moving from here to subclass Vectors, it seems appropriate to have a new type name, and to remove all functional duplicates. This results in an even simpler interface:

```
DEFINITION MODULE VectorClassA;
FROM PointClassA IMPORT
  Points;
TRACED CLASS Vectors;
  INHERIT Points;
  REVEAL neg, add, sub, dotProduct;
  TYPE
    Vector = Point;
```

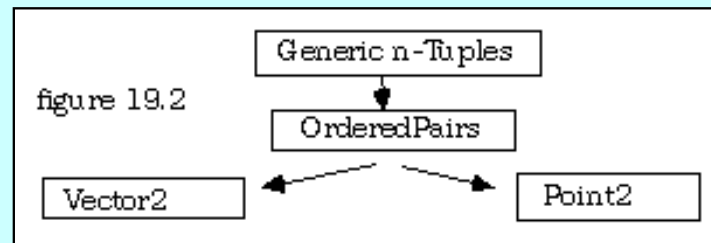
```

PROCEDURE neg;
PROCEDURE add (v : Vector);
PROCEDURE sub (v : Vector);
PROCEDURE dotProduct (v : Vector) : REAL;
END Vectors;

END VectorClassA.

```

However, this design leaves something to be desired, because reflections and rotations, while relevant to points are not relevant to vectors. Moreover, while one might *scale* a point, it is thought of as a multiplication with respect to vectors. It is better design to come up with a base or superclass that has only the bare essentials that are common, and then derive both *Points* and *Vectors* from this. Such a more basic concept is the idea of an ordered pair, or more generally, of an ordered n-tuple. This base class can have in it only the facilities for working with such tuples, and let both *Points* and *Vectors* define their own components. Figure 19.2 illustrates the relationship among the classes:



Indeed, in the design of this suite of classes, the base class or first superclass is a generic module.

```

GENERIC DEFINITION MODULE TupleClass (length : CARDINAL);
(* base class for specific n-tuples
   R. Sutcliffe 1998 10 02 *)

TRACED CLASS Tuples;
REVEAL assignCoord, fetchCoord, abs;
CONST
    len = length; (* for inheritors *)
VAR
    coords : ARRAY [1 .. len] OF REAL;

PROCEDURE assignCoord (coordNum : CARDINAL; value : REAL);
PROCEDURE fetchCoord (coordNum : CARDINAL) : REAL;
PROCEDURE abs () : REAL;
END Tuples;

END TupleClass.

```

To work with two dimensional vectors and points in a plane, this has to be refined by:

```

DEFINITION MODULE OrderedPair = TupleClass (2);
(* produce a 2-dimensional base class of tuples for
   corresponding vectors and points
   R. Sutcliffe 1998 09 28 *)
END OrderedPair.

```

This in turn can be subclassed on the one hand to work with points in the plane.

```

DEFINITION MODULE Point2Class;
(* subclasses OrderedPairs for the geometric notions of points
   R. Sutcliffe 1998 10 02 *)

FROM OrderedPair IMPORT
    Tuples;
TRACED CLASS Points;
    INHERIT Tuples;
    REVEAL reflectInAxis, reflect0, reflect45, scale, rotate, translate;

    PROCEDURE arg () : REAL;
    PROCEDURE reflectInAxis (n : CARDINAL);
    PROCEDURE reflect0;
    PROCEDURE reflect45;
    PROCEDURE scale (scaleFactor : REAL);
    PROCEDURE rotate (rotAngle : REAL);
    PROCEDURE translate (deltaX, deltaY : REAL);
    END Points;

END Point2Class.

```

On the other hand, it can be subclassed to work with two-dimensional vectors.

```

DEFINITION MODULE Vector2Class;
(* subclasses OrderedPairs for vectors
   R. Sutcliffe 1998 10 01 *)

FROM OrderedPair IMPORT
    Tuples;
TRACED CLASS Vector;
    INHERIT Tuples;
    REVEAL abscissa, ordinate, neg, add, sub, dotProduct, arg;

    PROCEDURE abscissa () : REAL;
    PROCEDURE ordinate () : REAL;
    PROCEDURE neg;
    PROCEDURE add (v : Vector);
    PROCEDURE sub (v : Vector);
    PROCEDURE dotProduct (v : Vector) : REAL;
    PROCEDURE arg () : REAL;

    END Vector;

END Vector2Class.

```

Observe that a method such as *Add* needs only to be passed the vector to add to SELF. In effect, such methods are messages to the object to add another vector to themselves. This is shown in some of the procedures of the partial implementation below.

```

IMPLEMENTATION MODULE Vector2Class;
(* subclasses OrderedPairs for vectors
   R. Sutcliffe 1998 10 01 *)

TRACED CLASS Vector;

PROCEDURE abscissa () : REAL;
BEGIN
    RETURN coords [1];
END abscissa;

PROCEDURE ordinate () : REAL;
BEGIN
    RETURN coords [2];
END ordinate;

PROCEDURE neg;
VAR
    count : CARDINAL;
BEGIN
    FOR count := 1 TO len
        DO
            coords [count] := - coords [count];
        END;
END neg;

PROCEDURE add (v : Vector);
VAR
    count : CARDINAL;
BEGIN
    FOR count := 1 TO len
        DO
            coords [count] := coords [count] + v.coords [count];
        END;
END add;

PROCEDURE sub (v : Vector);
    (* similar to add *)
END sub;

PROCEDURE dotProduct (v : Vector) : REAL;
VAR
    count : CARDINAL;
    sum : REAL;
BEGIN
    sum := 0.0;
    FOR count := 1 TO len
        DO
            sum := sum + coords [count] * v.coords [count];
        END;
    RETURN sum;

```



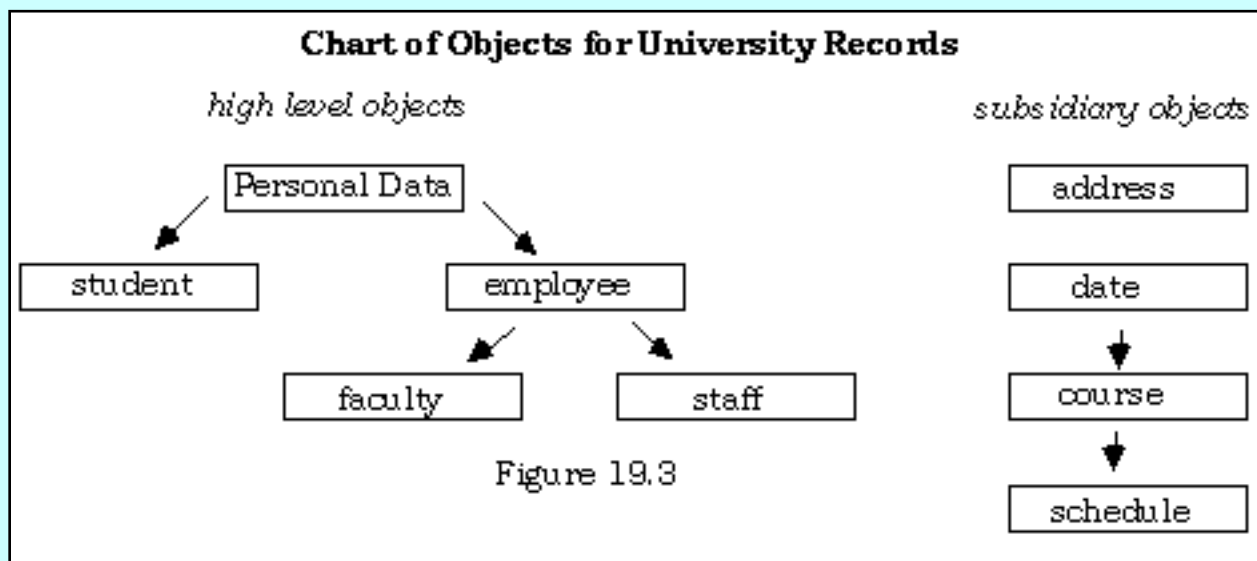
```
END dotProduct;  
  
PROCEDURE arg ( ) : REAL;  
BEGIN  
  (* code to work out angle here. Specific to two-vectors. *)  
END arg;  
  
END Vector;  
  
END Vector2Class.
```

It should be evident to the reader that if complex numbers were not built in, they could be subclassed from ordered pairs as well, with new method components to handle such operations as multiplication and division.

[Contents](#)

19.12 Example Outline--Personnel Records

One of the things computers spend most of their time doing is keeping track of various types of business records. These come in a wide variety of different forms, but often have many elements in common. For instance, in a university environment, every member of the community comes attached to such information as their name, birthdate, address, and sex. Students major or minor in a discipline, take courses, and generate a variety of other data exclusive to them. Others are employees, with certain data common to all of them, but falling into the two broad categories of staff and faculty, each with data not shared by the others. A sketch of a partial design for the data structures is shown in figure 19.3.



The base class for personnel data does not inherit from such classes as address, but it does make use of them in its own structures. When doing so, a class has to create the instances it needs of any other classes when it initializes. If the class were unsafeguarded, it should dispose of the subsidiary objects in its destructor. What follows is an outline of some of the code. It is far from complete, (no methods have been included), and is intended to show only some of the relationships among the classes and types of data needed. In addition, the class *CommonInfo* illustrates the correct positioning of an exception handler for a class body.

```
MODULE URecords;
```

```
(* a rough sketch of some University record keeping classes
   to demonstrate class design
   by R. Sutcliffe 1998 10 05 *)
```

```
FROM Dates IMPORT Date;
```

```
TRACED CLASS Address;
```

```
REVEAL SetAddress, WriteAddress;
```

```
VAR
```

```
    street1, street2, town, country : ARRAY [0..20] OF CHAR;
```

```
    code : ARRAY [0..10] OF CHAR;
```

```
PROCEDURE SetAddress;
```

```
(* code to do so *)
```

```
END SetAddress;
```

```
PROCEDURE WriteAddress;
```

```
(* code to do so *)
```

```
END WriteAddress;
```

```
END Address;
```

```
(* the base information class common to everybody *)
```

```
TRACED CLASS CommonInfo;
```

```
    REVEAL birthDate, adr, sin, sex;
```

```
VAR
```

```
    birthDate : Date;
```

```
    adr : Address;
```

```
    sin : CARDINAL;
```

```
    sex : BOOLEAN;
```

```
(* methods *)
```

```
BEGIN
```

```
    CREATE (birthDate);
```

```
    CREATE (adr);
```

```
(* if untraced, these have to be destroyed in finally clause *)
```

```
EXCEPT
```

```
(* exception handling goes here if necessary *)
```

```
END CommonInfo;
```

```
(* the base class for all employees *)
```

```
TRACED CLASS Employee;
```

```
    INHERIT CommonInfo;
```

```
    REVEAL department; (* and whatever *)
```

```
VAR
```

```
    department : ARRAY [0..20] OF CHAR;
```

```
    startDate : Date;
```

```
    salary : REAL;
```

```
    bankName : ARRAY [0..15] OF CHAR;
```

```
    fullTime : BOOLEAN;
```

```
(* methods *)
```

```
BEGIN
```

```
    CREATE (startDate);
```

```

    (* if untraced, has to be destroyed in finally clause *)
END Employee;

(* specialized employee classes *)
TRACED CLASS FacultyMember;
    INHERIT Employee;
    REVEAL rank; (* and whatever *)
    VAR
        rank, faculty : ARRAY [0..20] OF CHAR;
        (* methods *)
    END FacultyMember;

TRACED CLASS StaffMember;
    INHERIT Employee;
    REVEAL jobTitle; (* and whatever *)
    VAR
        jobTitle : ARRAY [0..20] OF CHAR;
        supervisor : StaffMember;
        (* methods *)
    END StaffMember;

(* data types for courses and schedules *)
TYPE
    DayAbbrev = (M, T, W, R, F, S); (* no sunday classes *)
    DaySet = SET OF DayAbbrev;

TRACED CLASS Course;
    VAR
        dept : ARRAY [0..3] OF CHAR;
        courseNum : [1..999];
        credits : [0..3]; (* needs a fix if half credits allowed *)
        startDate, endDate : Date;
        daysOffered : DaySet;
        (* need times too *)

    BEGIN
        CREATE (startDate);
        CREATE (endDate);

    END Course;

TRACED CLASS Schedule;
    VAR
        indivSchedule : ARRAY [0..10] OF Course;

```

```
(* methods *)  
END Schedule;
```

```
TRACED CLASS Student;
```

```
  INHERIT CommonInfo;
```

```
  REVEAL major; (* and more *)
```

```
  VAR
```

```
    major : ARRAY [0..20] OF CHAR;
```

```
    year : (freshman, sophomore, junior, senior);
```

```
    sched : Schedule;
```

```
    (* info and methods *)
```

```
END Student;
```

```
END URecords.
```

[Contents](#)

19.13 On the Use of Programming Paradigms

With the inclusion of object oriented techniques in the repertoire, the programmer has the ability to produce code in a number of styles. It would be a good idea to ask when a given style of programming is appropriate, and when a different one should be used. Here are a few suggestions:

Monolithic style:

All the code is in a single compilation unit, and there are few or no procedures.

Advantages: There is less writing to do because there are few or no imports; the style is simple; and the program may link and run rather quickly without the overhead of libraries and procedure calls. All the code is available to the programmer for debugging.

Disadvantages: This method does not scale well even to medium sized programs, much less to large ones. Debugging time goes up dramatically with the length of the single program. All code has to be produced by the writer for each program.

When to use: Use for one-of-a-kind simple throw-away programs to do a calculation or simple data manipulation, where reusability is not a concern, and the program is less than a hundred lines.

Modular style:

Code is grouped into procedures and modules according to its type relationship and/or functionality.

Advantages: Modular design scales well even to very large programs. When used to hide data intelligently, static modules are a good tool to factor code into manageable packets that can be designed and debugged quickly and efficiently. This method can be used for the vast majority of programming needs. Existing libraries can be imported, and new ones written that may, if well designed, be reusable.

Disadvantages: Static modules in themselves do not well serve the need for dynamic data. Pointers used for dynamic allocation are notorious for being error prone, even when encapsulated into libraries. Some code has to be rewritten many times for just minor changes in data.

When to use: Use for programs of any size or complexity, from hundreds to millions of lines of code.

Generic style:

Some code is parameterized, particularly as to a data type.

Advantages: This method extends the idea of static modules to solve a particular kind of problem. Code for common structures such as lists or routines such as sorts can be written once, and then easily specialized for a particular type of data.

Disadvantages: This is still a static method. It does not address the problems associated with the safety of dynamic data.

When to use: Use in those cases where a data type such as a list has all its code common regardless of the type of data going into the structure, or when a routine such as a sort has common code regardless of what data is being sorted. Generic templates can be refined for the particular types with a minimum of effort.

Object Oriented style:

The core of the program's focus is on classes of data; actions are secondary to data.

Advantages: This method allows for a different view of problems that are data-centric. Like modular and

generic techniques, it promotes the reusability of code by handing off responsibility for some actions to other data classes. If traced classes are used, many problems with pointers vanish, as memory does not need to be manually managed.

Disadvantages: Object oriented programming has a greater overhead in planning, and its code runs slower than conventional procedure calls. If care is not taken in the management of objects, the tangle of data relationships can produce a kind of spaghetti approach to problem solving that rivals that found in languages employing GOTO for transfer of control

When to use: Use in those cases where there are related classes of data that lend themselves naturally to a class hierarchy, and/or as a means to avoid the problems of dynamic data associated with pointers. Object oriented programming combines well with generic approaches, and the two are often used together.

[Contents](#)

19.14 Chapter Summary

This chapter covered these topics:

- Object oriented concepts and vocabulary
- Object oriented notation
- Standard object oriented Modula-2 extensions.
- Classes, attributes, methods, data access, visibility and inheritance
- Subclasses, overriding methods in OOM-2
- Traced and untraced object classes
- Safeguarded and unsafeguarded module containers
- Circular definitions and self-reference
- Assignment, comparison, reference, and membership test of OOM-2 objects
- Instantiation and destruction
- Classes, attributes, methods, data access, visibility and inheritance
- Some design examples

It included discussion of the following Modula-2 items:

Reserved Words	Standard Identifiers
-----------------------	-----------------------------

FORWARD (new)	none
---------------	------

FINALLY (new)

and of the following Object Oriented Modula-2 items:

Reserved Words	Standard Identifiers
-----------------------	-----------------------------

AS	CREATE
ABSTRACT	DESTROY
CLASS	EMPTY
GUARD	ISMEMBER
INHERIT	SELF
OVERRIDE	
READONLY	
REVEAL	
TRACED	
UNSAFEGUARDED	

Imports:

Standard Library Imports

From M200EXCEPTION:

 M200Exceptions, M200Exception, IsM200Exception

From COROUTINES

 DISPOSECOROUTINE

[Contents](#)

19.15 Assignments

Questions

1. What is the difference between a class and a module for implementing ADTs?
2. What kinds of components may an object have?
3. What do you call the generation of a specific object from a declared class?
4. Describe two approaches to component visibility for clients of a class and say which one was taken in Modula-2.
5. What does the concept of an immutable entity apply to in OOM-2?
6. What parts of a Modula-2 class are visible to a subclass?
7. What is the difference between objects with value semantics and those with reference semantics? Which approach does Modula-2 take?
8. What is the difference between a reference and a pointer?
9. What is the difference between an object definition and an object declaration?
10. What is the difference between a traced class and an untraced class?
11. What may a traced class not have in it?
12. What is the difference between a safeguarded module and an unsafeguarded one?
13. What may a safeguarded module not have in it?
14. What is another name for an object component?
15. Why is a method sometimes called a message?
16. In what way are Modula-2 classes like modules, and in what ways are they like records?
17. How is a Modula-2 object instantiated? How is it initialized?
18. May a Modula-2 object be manually destroyed? If so, under what circumstances?
19. What is meant by inheritance?
20. What is the purpose of inheritance?
21. What restrictions does OOM-2 place on what can be inherited and from where?
22. What kinds of components does OOM-2 allow to be overridden, and how is this done?
23. How does one access an overridden component?
24. What is the difference between static type and dynamic type?
25. If a chain of subclasses is declared, and an object is instantiated in the last child on the chain, is it assignment compatible to a variable of the eldest parent class? What about the other way around?
26. What is the GUARD statement for? In what way is it like a CASE statement and in what way is it like a WITH statement? How is it an assertion?
27. What is unusual about the semantics of ISMEMBER?
28. What is the difference between the way OOM-2 treats constants and type components, on the one hand, and attribute variables and methods on the other?
29. What imports are necessary when using untraced OOM-2 classes?
30. How may two classes refer to each others' declarations (or definitions)?

31. How does an object get access to itself?
32. Why does OOM-2 use single inheritance rather than multiple inheritance?
33. Propose a means (some syntax) that would, if included in OOM-2, avoid the problems of multiple inheritance.
34. What is an abstract class?
35. Some implementations do not implement traced objects, only untraced ones. What else will be missing from such implementations?
36. Determine whether your implementation allows you to turn garbage collection off.
37. In the example of [section 19.11](#), there was no generic module for Vectors. Consider the chapter on generics carefully, and explain why it is not a simple matter to interpose a generic module for vectors between the generic module for Tuples and the module for vectors of a specific length. What would have to be done to accomplish this?
38. What change would have to be made to the base language in order to better accommodate classes in generic definition modules inheriting from classes in other generic definition modules?
39. Some languages are designed without any pointers, just reference variables. Thus, they have no untraced memory allocation at all (not just none for objects) and all memory allocated dynamically is traced by the garbage collector. What are the pros and cons of such an approach?
40. In some languages, class declaration is done by saying what class the new one is a subclass of. Show that this implies the existence of a standard identifier that names a particular built-in class. What would you call such a class?
41. Give an example to illustrate that allowing user defined operator overloading for OOM-2 would make life easier for the users of classes.

Problems:

42. Create a series of derived classes to illustrate the constructor and destructor chain order.
43. Devise a means to test whether there is a greater overhead (time) associated with sending a message to an object (invoking a procedure that is a method) or calling a procedure containing the same code directly.
44. Consider the abstract class demonstration in [section 19.8](#). Add proper commenting, better verbiage in the output, and the classes Triangle, Parallelogram, Rhombus, Trapezoid, and regular n-gon to the family, each in an appropriate place in the class hierarchy.
45. Complete and test the suite of objects in section 19.11.
46. Use the class points to implement a class *line* (using two points.)
47. Add a generic vector module to the example of [section 19.11](#). See question 37.
48. Complete and test the suite of objects in [section 19.12](#).
49. Implement a database for your Church, business, or hobby using an object oriented approach. Be sure to do a careful design of your chart of classes and their contents before writing any code.
50. Implement and test the data type *fraction* as a class. Include standard operations, reduction to lowest terms, and retrieval of numerator and denominator, as well as a display procedure.
51. Derive a complex number class from the suite of classes in [section 19.11](#).
52. Design, implement and test a class for a deck of standard playing cards, including a procedure to deal

hands of Bridge.

53. Use the same class *CardDeck* from the previous question to implement a game of solitaire.

54. Solve the eight queens problem using objects.

55. Solve the knight's tour problem using objects.

56. Implement a linked list as a class.

57. Derive a queue class from the above list class.

58. Derive a priority queue class from the above queue class.

59. Produce a concordance program that examines a text file and produces a sorted list of all the words in the file, their frequency, and a means to display the word in the context of its original occurrences.

60. Implement a binary tree as a class.

61. Implement a quicksort as a class.

62. Use *GraphPaper* or other graphics capabilities on your computer to implement a bouncing ball on the screen as an instance of a class *ball*.

[Contents](#)

Chapter 19

Object Oriented Modula-2

[19.0 Chapter Goals](#)

[19.1 Introduction to Object Oriented Thinking](#)

[19.2 Object Oriented Terminology](#)

[19.2.1 Basic Definitions](#)

[19.2.2 Reference versus Value Objects](#)

[19.3 Getting Started with Object Oriented Modula-2](#)

[19.3.1 A Little History](#)

[19.3.2 Some Simple OOM-2 Programs](#)

[19.3.3 Summary of Basic OOM-2 Traced Class Semantics](#)

[19.4 Untraced Objects](#)

[19.5 Assignment and Comparison of Objects](#)

[19.6 Encapsulation of Classes in Separate Modules](#)

[19.7 Inheritance](#)

[19.7.1 Why Inherit](#)

[19.7.2 Inheritance in OOM-2](#)

[19.7.3 Assignment Compatibility between Classes and Subclasses](#)

[19.7.4 Overriding Methods in Subclasses](#)

[19.7.5 Class and Object References](#)

[19.7.6 Why Single Inheritance?](#)

[19.8 Abstract Classes](#)

[19.9 Guard Statement](#)

[19.10 Additions to the Libraries](#)

[19.10.1 Exceptions](#)

[19.10.2 Coroutines](#)

[19.10.3 The Module Garbage Collection](#)

[19.11 Extended Example--Points and Vectors](#)

[19.12 Example Outline--Personnel Records](#)

[19.13 On the Use of Programming Paradigms](#)

[19.14 Chapter Summary](#)

[19.15 Assignments](#)

[Programming Note--Comparisons With Other Object Notations](#)

[Contents](#)

Programming Note--Comparisons With Other Object Notations

	OO Modula-2	C++	Java
Inheritance : Arity	Single	multiple	single, but multiple interfaces
: syntax	INHERIT	public: (heading)	extends
Assignment semantics	reference	pointer (can be overloaded)	reference
Predefined Base Class	no	no	Object
Traced objects tag	TRACED (class)	n/a	all are traced
Untraced objects tag	UNSAFEGUARDED (module)	all are untraced	n/a
Class Definition/declaration	CLASS	class	class
Abstract classes	ABSTRACT	n/a	abstract
methods	ABSTRACT	virtual; =0 (after heading)	abstract
Instantiation	declare & CREATE	declare or new, automatic	new
Automatic Constructor	class body	method of class name	method of class name
Automatic initialization	class body	constructor initializer clause	use constructor
Destruction : manual	DESTROY (untraced)	~ in front of class name	n/a
: extra steps	FINALLY (untraced)	n/a	Finalize (called on collection)
Overload constructors, operators	n/a	yes	yes
Make data attributes immutable	READONLY	const	final (not strictly immutable)
Override attribute components	n/a	n/a	yes, statically
Parent permission to override	n/a	virtual	n/a
keyword	OVERRIDE	virtual (optional)	redeclare in child class
forbidding of	n/a	n/a	final
Visibility to clients	REVEAL	public	public (each member)

to implementation	put in implementation module	private	private (each member)
to subclasses only	place in declaration	protected	protected
to friend classes	n/a	friend	n/a
Object Selection/assertion	GUARD..AS	n/a	n/a
Class membership	ISMEMBER	n/a	instanceof
Reference to : empty object	EMPTY	null	null
: self	SELF	this	this
Combining OO and Generics	define in generic def mod	use container classes	n/a

[Contents](#)

Appendix 1 The Lexis of Modula-2

A1.1 Reserved Words (Keywords)

AND	ARRAY	BEGIN	BY
CASE	CONST	DEFINITION	DIV
DO	ELSE	ELSIF	END
EXIT	EXCEPT	EXPORT	FINALLY
FOR	FORWARD	FROM	IF
IMPLEMENTATION	IMPORT	IN	LOOP
MOD	MODULE	NOT	OF
OR	PACKEDSET	POINTER	PROCEDURE
QUALIFIED	RECORD	REM	RETRY
REPEAT	RETURN	SET	THEN
TO	TYPE	UNTIL	VAR
WHILE	WITH		

The keywords EXCEPT, FINALLY, FORWARD, PACKEDSET, REM, and RETRY were added for ISO Modula-2. The keywords AND, DIV, IN, MOD, NOT, OR, and REM are called operator keywords; the rest are punctuation keywords.

A1.2 Standard (Pervasive) Identifiers

ABS	BITSET	BOOLEAN	CARDINAL
CAP	CHR	CHAR	COMPLEX
CMPLX	DEC	DISPOSE	EXCL
FALSE	FLOAT	HALT	HIGH
IM	INC	INCL	INT
INTERRUPTIBLE	INTEGER	LENGTH	LFLOAT
LONGCOMPLEX	LONGREAL	MAX	MIN
NEW	NIL	ODD	ORD
PROC	PROTECTION	RE	REAL
SIZE	TRUE	TRUNC	UNINTERRUPTIBLE
VAL			

The pervasive identifiers COMPLEX, CMPLX, IM, INT, INTERRUPTIBLE, LENGTH, LFLOAT, LONGCOMPLEX, LONGREAL, PROTECTION, RE and UNINTERRUPTIBLE were added for ISO Modula-2.

A1.3 Standard Symbols

:	colon	seperates variable from type or case label from statements
,	comma	seperates items in lists
..	ellipsis	range indicator
=	equals	constant or type declarations
.	period	decimal point, qualified identifier selector, end of program
;	semicolon	statement separator
()	parentheses	expressions, parameter lists or enumerations

[]	brackets	index, range, module priority, or absolute address brackets
(! !)	brackets - alternate form	
{ }	braces	set delimiters
(: :)	braces - alternate form	
	bar	separates each variant in case statement
< &	logical disjunction operator	
NOT, ~	logical negation operator	
=	equals operator	
# <<	less than operator	
>	greater than operator	
<=	less than or equal and subset operator	
>=	greater than or equal and superset operator	
IN	set membership operator	
^,@	dereferencing operator (caret or hat)	

[Contents](#)

[Show outer \(navigation\) frames](#)

Warning:

Appendix 2 has over 150 inline images in it. It will take a *long* time to load.

[Continue](#)

[Contents](#)

Appendix 4--Classical Library Modules

The main modules described by Niklaus Wirth and suggested for adoption in Modula-2 are described in this Appendix. Although the detailed semantics varies widely in some cases, they can be assumed to be a part of all older (pre-ISO) versions, and are included alongside the ISO modules by some vendors as well.

A4.1 High Level Input and Output

A4.1.1 InOut

```
DEFINITION MODULE InOut;
CONST
  EOL = 15C;    (* hardware dependant--could also be 36C *)

VAR
  Done: BOOLEAN;
  termCH: CHAR;

PROCEDURE OpenInput (defext: ARRAY OF CHAR);
PROCEDURE OpenOutput (defext: ARRAY OF CHAR);
PROCEDURE CloseInput;
PROCEDURE CloseOutput;
PROCEDURE Read (VAR ch: CHAR);
PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
PROCEDURE ReadInt (VAR x: INTEGER);
PROCEDURE ReadCard (VAR x: CARDINAL);
PROCEDURE Write (ch: CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteString (s: ARRAY OF CHAR);
PROCEDURE WriteInt (x: INTEGER; n: CARDINAL);
PROCEDURE WriteCard (x, n: CARDINAL);
PROCEDURE WriteOct (x, n: CARDINAL);
PROCEDURE WriteHex (x, n: CARDINAL);
END InOut.
```

- NOTES: 1. In some cases *InOut* may incorporate *RealInOut*.
2. Many implementations place *termCH* in *Terminal*.
3. Some include the procedure *ClearScreen*.
4. The syntax and semantics of *OpenInput* and *OpenOutput* vary widely from one classical implementation to another.

A4.1.2 RealInOut

```
DEFINITION MODULE RealInOut;
VAR
  Done: BOOLEAN;

PROCEDURE ReadReal (VAR x: REAL);
PROCEDURE WriteReal (x: REAL; n: CARDINAL);
PROCEDURE WriteRealOct (x: REAL);
```

```
END RealInOut.
```

NOTE: The syntax and semantics of *WriteReal* widely from one classical implementation to another.

A4.1.3 Terminal

```
DEFINITION MODULE Terminal;  
PROCEDURE Read (VAR ch: CHAR);  
PROCEDURE ReadLn (VAR s: ARRAY OF CHAR); (* stops at end-of-line *)  
PROCEDURE BusyRead (VAR ch: CHAR);  
PROCEDURE ReadAgain;  
PROCEDURE Write (ch: CHAR);  
PROCEDURE WriteString (s: ARRAY OF CHAR);  
PROCEDURE WriteLn;  
END Terminal.
```

NOTE: Where supplied, the contents of this module vary. The following module may be included in *Terminal* or be separate.

```
DEFINITION MODULE Screen;  
PROCEDURE HomeCursor;  
PROCEDURE ClearScreen;  
PROCEDURE EraseLine;  
PROCEDURE GotoXY (x, y: CARDINAL);  
END Screen.
```

A4.2 Mathematical Functions

```
DEFINITION MODULE MathLib0;  
PROCEDURE sqrt (x: REAL): REAL;  
PROCEDURE exp (x: REAL): REAL;  
PROCEDURE ln (x: REAL): REAL;  
PROCEDURE sin (x: REAL): REAL;  
PROCEDURE cos (x: REAL): REAL;  
PROCEDURE arctan (x: REAL): REAL;  
PROCEDURE real (x: INTEGER): REAL;  
PROCEDURE entier (x: REAL): INTEGER;  
END MathLib0.
```

Several versions add:

```
DEFINITION MODULE MathLibLong; (* may be called "LongMath" *)  
PROCEDURE sqrt (x: LONGREAL): LONGREAL;  
PROCEDURE exp (x: LONGREAL): LONGREAL;  
PROCEDURE ln (x: LONGREAL): LONGREAL;  
PROCEDURE sin (x: LONGREAL): LONGREAL;  
PROCEDURE cos (x: LONGREAL): LONGREAL;  
PROCEDURE arctan (x: LONGREAL): LONGREAL;  
PROCEDURE real (x: LONGINTEGER): LONGREAL;  
PROCEDURE entier (x: LONGREAL): LONGINTEGER;  
END MathLibLong.
```

NOTE: The contents of mathematical libraries vary widely in classical versions. Some may include:

```
CONST
    pi = 3.1415926536;
    e  = 2.7182818284;
```

A4.3 SYSTEM and Other Low Level and System Access Modules

The module SYSTEM is properly termed a *psuedo-Module* as it does not exist in the library, but is entirely contained inside the compiler. Nonetheless, it behaves in most classical versions as though it had the definition given.

```
DEFINITION pseudo-MODULE SYSTEM;
TYPE
    ADDRESS; WORD;
PROCEDURE ADR (anyObject: ARRAY OF WORD): ADDRESS;
PROCEDURE SIZE (anyObject: ARRAY OF WORD): CARDINAL;
    (* older versions--newer ones have it as a built-in *) PROCEDURE TSIZE (anyType
[,optional variant list] ): CARDINAL;
PROCEDURE NEWPROCESS (nameOfProcess : PROC;
    workspace   : ADDRESS;
    sizeOfSpace  : CARDINAL;
    VAR newProc  : ADDRESS);
PROCEDURE TRANSFER (VAR oldProcess : ADDRESS; VAR newProcess : ADDRESS);
PROCEDURE IOTRANSFER (VAR oldProcess : ADDRESS; VAR newProcess : ADDRESS);
(* Not all implement the latter *)
END SYSTEM.
```

NOTE: The contents of this library vary widely in classical versions. Many other entities may be included, such as:

```
TYPE
    BYTE;
    SHORTWORD;
    LONGWORD;
    QUADWORD;
    OCTWORD;
    HEXWORD;
```

A4.4 Storage

```
DEFINITION MODULE Storage;
FROM SYSTEM IMPORT
    ADDRESS;
PROCEDURE ALLOCATE (VAR p: ADDRESS; size: CARDINAL);
PROCEDURE DEALLOCATE (VAR p: ADDRESS; size: CARDINAL);
PROCEDURE Available (size: CARDINAL): BOOLEAN;
END Storage.
```

A4.5 String Handling

```
DEFINITION MODULE Strings;
```

TYPE

```
STRING = ARRAY [0..80] OF CHAR;
```

```
PROCEDURE Assign (VAR source, dest: ARRAY OF CHAR);
```

```
PROCEDURE Insert (substr: ARRAY OF CHAR; VAR str: ARRAY OF CHAR; index: CARDINAL);
```

```
PROCEDURE Delete (VAR str: ARRAY OF CHAR; index: CARDINAL; len: CARDINAL);
```

```
PROCEDURE Pos (substr, str: ARRAY OF CHAR): CARDINAL;
```

```
PROCEDURE Copy (str: ARRAY OF CHAR; index: CARDINAL; len: CARDINAL;
```

```
    VAR result: ARRAY OF CHAR);
```

```
PROCEDURE Concat (s1, s2: ARRAY OF CHAR; VAR result: ARRAY OF CHAR);
```

```
PROCEDURE Length (VAR str: ARRAY OF CHAR): CARDINAL;
```

```
PROCEDURE CompareStr (s1, s2: ARRAY OF CHAR): INTEGER;
```

(* returns -1 if s1 < s2 0 if s1 = s2 1 if s1 "standard" set of modules, so that readers could all use the same file interface. Those suggestions were taken to heart by many vendors, and gradually a model similar to the one in this section became common in the classical versions, though still with numerous variations in style, syntax, and semantics.

The general characteristics of this model are as follows:

1. *File* (or *FILE*) may be either a transparent or an opaque type. This is the type of the logical (program) variables which are associated with physical files through the procedure *Open*.

2. Only minimal text writing facilities are provided in the module *Files* (or *FileSystem*, or *Filer*) , as it is designed more for random access files.

3. Text I/O is generally handled by a separate module, but using the type *File* directly.

4. The type *FilePos* is usually not an opaque type.

5. Facilities are sometimes provided for looking up file names and for reading and parsing them from the terminal.

6. The main file handling module includes some or all of:

A4.6.1 The File System Module

```
DEFINITION MODULE FileSystem;
```

```
FROM SYSTEM IMPORT
```

```
    WORD, ADDRESS, BYTE;
```

TYPE

```
    File;
```

```
    Response = (done, notdone, notsupported, callerror, unknownmedium, unknownfile, paramerror, toomanyfiles, eom, deviceoff, softparityerror, softprotected, softerror, hardparityerror, hardprotected, timeout, harderror);
```

```
(* File Management *)
```

```

PROCEDURE FileState (f: File) : Response;
PROCEDURE Lookup (VAR f: File; name: ARRAY OF CHAR; new: BOOLEAN);
    (* new = permission to create if not found *)
PROCEDURE Create (VAR f: File; mediumName: ARRAY OF CHAR);
    (* file created is temporary and nameless*)
PROCEDURE Rename (VAR f: File; filename: ARRAY OF CHAR);
    (* needed to make a temporary file permanent. The reverse happens if the name
given is empty. *)
PROCEDURE Close (VAR f: File);
    (* Only those with nonempty names will remain in the directory after being closed.
    *)
PROCEDURE Delete (filename: ARRAY OF CHAR);
PROCEDURE SetRead (VAR f: File);
PROCEDURE SetWrite (VAR f: File);
PROCEDURE SetModify (VAR f: File);
PROCEDURE SetOpen (VAR f: File); (* where present, cancels any of the last three *)

    (* File Information *)
PROCEDURE Eof (f: File): BOOLEAN;

    (* Sequential File Access -- Textual Material *)
PROCEDURE ReadChar (f: File; VAR ch: CHAR);
PROCEDURE WriteChar (f: File; ch: CHAR);

    (* Binary File Access *)
PROCEDURE ReadWord (f: File; w: WORD);
PROCEDURE WriteWord (f: File; w: WORD);
PROCEDURE Again (VAR f: File);
    (* sets position back to beginning or previously read WORD or CHAR *)
PROCEDURE ReadByte (f: File; w: BYTE);
PROCEDURE WriteByte (f: File; w: BYTE);
PROCEDURE ReadNBytes (VAR f: File; buf: ADDRESS; numBytesRequested: CARDINAL; VAR
numRead: CARDINAL);
PROCEDURE WriteNBytes (VAR f: File; buf: ADDRESS; numBytesToWrite: CARDINAL; VAR
numWritten: CARDINAL);

    (* Random Access Files *)
PROCEDURE GetPos (f: File; VAR highpos, lowpos: CARDINAL);
PROCEDURE SetPos (f: File; highpos, lowpos: CARDINAL);
PROCEDURE Length (f: File; highpos, lowpos: CARDINAL);
    (* last three use a single LONGCARD for position if this type is available *)
PROCEDURE Reset (VAR f: File);
    (* sets state to open and position to beginning of file *)

END FileSystem.

```

An alternate is the module *Filer*. See [A6.2](#).

A4.6.2 The Classical Module TextIO

In classical versions, the module in this section is commonly provided in conjunction with the above and used to provide all textual I/O using the type *File*

```
DEFINITION MODULE TextIO;

FROM Files IMPORT
    File;

CONST
    EOL = 15C;

VAR
    Done : BOOLEAN;
    termCH : CHAR;

PROCEDURE WriteString (f : File; s : ARRAY OF CHAR);
PROCEDURE WriteLn (f : File);
PROCEDURE WriteInt (f : File; i : INTEGER; flen : CARDINAL);
PROCEDURE WriteCard (f : File; c : CARDINAL; flen : CARDINAL);
PROCEDURE WriteReal (f : File; r : REAL; flen : CARDINAL; digits: INTEGER);
PROCEDURE ReadString (f : File; VAR s : ARRAY OF CHAR);
PROCEDURE ReadInt (f : File; VAR i : INTEGER);
PROCEDURE ReadCard (f : File; VAR c : CARDINAL);
PROCEDURE ReadReal (f : File; VAR r : REAL);
END TextIO.
```

WARNING: The least reliance can be placed on comments made here about the contents of file handling modules. In classical versions, even the convergence on the common model found in this section never meant that there was much uniformity. This was one of the principal reasons for the ISO standards effort.

A4.7 Character Information--ASCII

```
DEFINITION MODULE ASCII;

CONST
    nul = 00C; soh = 01C; stx = 02C; etx = 03C; eot = 04C; enq = 05C;
    ack = 06C; bel = 07C; bs  = 10C; ht  = 11C; lf  = 12C; vt  = 13C;
    ff  = 14C; cr  = 15C; so  = 16C; si  = 17C; dle = 20C; dc1 = 21C;
    dc2 = 22C; dc3 = 23C; dc4 = 24C; nak = 25C; syn = 26C; etb = 27C;
    can = 30C; em  = 31C; sub = 32C; esc = 33C; fs  = 34C; gs  = 35C;
    rs  = 36C; us  = 37C; del = 177C;

END ASCII.
```

Some versions add:

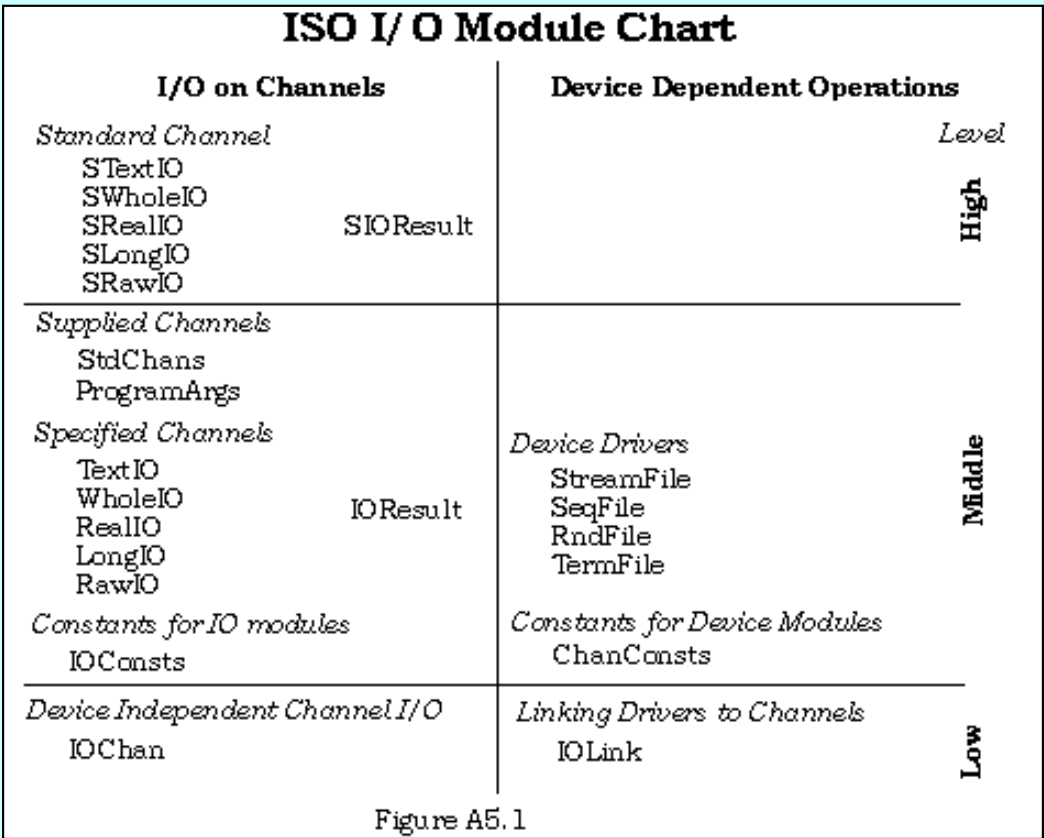
```
CONST
    EOL = 36C;    space = 40C;
```

[Contents](#)

Appendix 5--ISO I/O Library

A5.1 An Overview of the ISO I/O Library

The modules in this section are taken from ISO/IEC IS 10514, the international standard for Modula-2. See the copyright notice in the acknowledgements section. The chart below shows the modules in the collection and the level at which users employ them.



A5.2 I/O On Standard Channels

A5.2.1 STextIO

DEFINITION MODULE STextIO;

(* Input and output of character and string types over default channels. The read result is of the type IOConsts.ReadResults. *)

(* The following procedures do not read past line marks *)

PROCEDURE ReadChar (VAR ch: CHAR);

(* If possible, removes a character from the default input stream, and assigns the corresponding value to ch. The read result is set to allRight, endOfLine or endOfInput. *)

PROCEDURE ReadRestLine (VAR s: ARRAY OF CHAR);

(* Removes any remaining characters from the default input stream before the next line mark, copying to s as many as can be accommodated as a string value. The read result is set to the value allRight, outOfRange, endOfLine, or endOfInput. *)

PROCEDURE ReadString (VAR s: **ARRAY OF CHAR**);

(* Removes only those characters from the default input stream before the next line mark that can be accommodated in s as a string value, and copies them to s. The read result is set to the value allRight, endOfLine, or endOfInput. *)

PROCEDURE ReadToken (VAR s: **ARRAY OF CHAR**);

(* Skips leading spaces, and then removes characters from the default input stream before the next space or line mark, copying to s as many as can be accommodated as a string value. The read result is set to the value allRight, outOfRange, endOfLine, or endOfInput. *)

(* The following procedure reads past the next line mark *)

PROCEDURE SkipLine;

(* Removes successive items from the default input stream up to and including the next line mark or until the end of input is reached. The read result is set to the value allRight, or endOfInput. *)

(* Output procedures *)

PROCEDURE WriteChar (ch: **CHAR**);

(* Writes the value of ch to the default output stream. *)

PROCEDURE WriteLn;

(* Writes a line mark to the default output stream. *)

PROCEDURE WriteString (s: **ARRAY OF CHAR**);

(* Writes the string value of s to the default output stream. *)

END STextIO.

A5.2.2 SWholeIO

DEFINITION MODULE SWholeIO;

(* Input and output of whole numbers in decimal text form over default channels. The read result is of the type IOConsts.ReadResults. *)

(* The text form of a signed whole number is ["+" | "-"], decimal digit, {decimal digit}

The text form of an unsigned whole number is decimal digit, {decimal digit} *)

PROCEDURE ReadInt (VAR int : **INTEGER**);

(* Skips leading spaces, and removes any remaining characters from the default input channel that form part of a signed whole number. The value of this number is assigned to int. The read result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or endOfInput. *)

PROCEDURE WriteInt (int: **INTEGER**; width: **CARDINAL**);

(* Writes the value of int to the default output channel in text form, in a field of the given minimum width. *)

```

PROCEDURE ReadCard (VAR card: CARDINAL);
  (* Skips leading spaces, and removes any remaining characters from the default
  input channel that form part of an unsigned whole number. The value of this
  number is assigned to card. The read result is set to the value allRight,
  outOfRange, wrongFormat, endOfLine, or endOfInput. *)

PROCEDURE WriteCard (card: CARDINAL; width: CARDINAL);
  (* Writes the value of card to the default output channel in text form, in a
  field of the given minimum width. *)

END SWholeIO.

```

A5.2.3 SRealIO

```

DEFINITION MODULE SRealIO;

  (* Input and output of real numbers in decimal text form over default
  channels. The read result is of the type IOConsts.ReadResults.
  *)

  (* The text form of a signed fixed-point real number is
  ["+" | "-"], decimal digit, {decimal digit},
  [".", {decimal digit}]

  The text form of a signed floating-point real number is
  signed fixed-point real number,
  "E", ["+" | "-"], decimal digit, {decimal digit}
  *)

PROCEDURE ReadReal (VAR real: REAL);
  (* Skips leading spaces, and removes any remaining characters from the default
  input channel that form part of a signed fixed or floating point number. The
  value of this number is assigned to real. The read result is set to the value
  allRight, outOfRange, wrongFormat, endOfLine, or endOfInput. *)

PROCEDURE WriteFloat (real: REAL; sigFigs: CARDINAL; width: CARDINAL);
  (* Writes the value of real to the default output channel in floating-point
  text form, with sigFigs significant figures, in a field of the given minimum
  width. *)

PROCEDURE WriteEng (real: REAL; sigFigs: CARDINAL; width: CARDINAL);
  (* As for WriteFloat, except that the number is scaled with one to three
  digits in the whole number part, and with an exponent that is a multiple of
  three. *)

PROCEDURE WriteFixed (real : REAL; place: INTEGER; width : CARDINAL);
  (* Writes the value of real to the default output channel in fixed-point
  text form, rounded to the given place relative to the decimal point, in a
  field of the given minimum width. *)

PROCEDURE WriteReal (real: REAL; width: CARDINAL);
  (* Writes the value of real to the default output channel, as WriteFixed if
  the sign and magnitude can be shown in the given width, or otherwise as
  WriteFloat. The number of places or significant digits depends on the given
  width. *)

END SRealIO.

```

A5.2.4 SLongIO

DEFINITION MODULE SLongIO;

```
(* Input and output of real numbers in decimal text form over default
   channels.  The read result is of the type IOConsts.ReadResults.
   *)
```

```
(* The text form of a signed fixed-point real number is
   ["+" | "-"], decimal digit, {decimal digit},
   [".", {decimal digit}]
```

```
   The text form of a signed floating-point real number is
   signed fixed-point real number,
   "E", ["+" | "-"], decimal digit, {decimal digit}
   *)
```

PROCEDURE ReadReal (**VAR** real: **LONGREAL**);

```
(* Skips leading spaces, and removes any remaining characters from the default
   input channel that form part of a signed fixed or floating point number. The value of
   this number is assigned to real.  The read result is set to the value allRight,
   outOfRange, wrongFormat, endOfLine, or endOfInput.
   *)
```

PROCEDURE WriteFloat (real: **LONGREAL**; sigFigs: **CARDINAL**; width: **CARDINAL**);

```
(* Writes the value of real to the default output channel in floating-point text
   form, with sigFigs significant figures, in a field of the given minimum width. *)
```

PROCEDURE WriteEng (real: **LONGREAL**; sigFigs: **CARDINAL**; width: **CARDINAL**);

```
(* As for WriteFloat, except that the number is scaled with one to three digits in
   the whole number part, and with an exponent that is a multiple of three. *)
```

PROCEDURE WriteFixed (real: **LONGREAL**; place : **INTEGER**; width: **CARDINAL**);

```
(* Writes the value of real to the default output channel in fixed-point text form,
   rounded to the given place relative to the decimal point, in a field of the given
   minimum width. *)
```

PROCEDURE WriteReal (real: **LONGREAL**; width: **CARDINAL**);

```
(* Writes the value of real to the default output channel, as WriteFixed if the
   sign and magnitude can be shown in the given width, or otherwise as WriteFloat.  The
   number of places or significant digits depends on the given width. *)
```

END SLongIO.

A5.2.5 SRawIO

DEFINITION MODULE SRawIO;

```
(* Reading and writing data over default channels using raw operations, that is,
   with no conversion or interpretation. The read result is of the type
   IOConsts.ReadResults. *)
```

IMPORT SYSTEM;

```

PROCEDURE Read (VAR to: ARRAY OF SYSTEM.LOC);
  (* Reads storage units from the default input channel, and assigns them to
  successive components of to. The read result is set to the value allRight,
  wrongFormat, or endOfInput. *)

PROCEDURE Write (from: ARRAY OF SYSTEM.LOC);
  (* Writes storage units to cid from successive components of from. *)

END SRawIO.

```

A5.2.6 SIOResult

```

DEFINITION MODULE SIOResult;

(* Read results for the default input channel *)

IMPORT IOConsts;

TYPE
  ReadResults = IOConsts.ReadResults;

PROCEDURE ReadResult () : ReadResults;
(* Returns the result for the last read operation on the default input channel *)

END SIOResult.

```

A5.3 Supplied Channels

A5.3.1 StdChans

```

DEFINITION MODULE StdChans;

  (* Access to standard and default channels *)

IMPORT IOChan;

TYPE
  ChanId = IOChan.ChanId;
  (* Values of this type are used to identify channels *)

  (* The following functions return the standard channel values.
  These channels cannot be closed. *)

PROCEDURE StdInChan () : ChanId;
  (* Returns the identity of the implementation-defined standard source for program
  input. *)

PROCEDURE StdOutChan () : ChanId;
  (* Returns the identity of the implementation-defined standard source for program
  output. *)

PROCEDURE StdErrChan () : ChanId;
  (* Returns the identity of the implementation-defined standard destination for

```

```

program error messages. *)

PROCEDURE NullChan (): ChanId;
  (* Returns the identity of a channel open to the null device. *)

  (* The following functions return the default channel values *)

PROCEDURE InChan (): ChanId;
  (* Returns the identity of the current default input channel. *)

PROCEDURE OutChan (): ChanId;
  (* Returns the identity of the current default output channel. *)

PROCEDURE ErrChan (): ChanId;
  (* Returns the identity of the current default error message channel. *)

  (* The following procedures allow for redirection of the default channels *)

PROCEDURE SetInChan (cid: ChanId);
  (* Sets the current default input channel to that identified by cid. *)

PROCEDURE SetOutChan (cid: ChanId);
  (* Sets the current default output channel to that identified by cid. *)

PROCEDURE SetErrChan (cid: ChanId);
  (* Sets the current default error channel to that identified by cid. *)

END StdChans.

```

A5.3.2 ProgramArgs

```

DEFINITION MODULE ProgramArgs;

  (* Access to program arguments *)

IMPORT IOChan;

TYPE
  ChanId = IOChan.ChanId;

PROCEDURE ArgChan (): ChanId;
  (* Returns a value that identifies a channel for reading program arguments *)

PROCEDURE IsArgPresent (): BOOLEAN;
  (* Tests if there is a current argument to read from. If not, read <=
  IOChan.CurrentFlags() will be FALSE, and attempting to read from the argument channel
  will raise the exception notAvailable. *)

PROCEDURE NextArg ();
  (* If there is another argument, causes subsequent input from the argument device
  to come from the start of the next argument. Otherwise there is no argument to read
  from, and a call of IsArgPresent will return FALSE. *)

END ProgramArgs.

```


A5.4 Specified Channels

A5.4.1 TextIO

DEFINITION MODULE TextIO;

```
(* Input and output of character and string types over specified channels.
   The read result is of the type IOConsts.ReadResults.
   *)
```

IMPORT IOChan;

```
(* The following procedures do not read past line marks *)
```

PROCEDURE ReadChar (cid: IOChan.ChanId; **VAR** ch: **CHAR**);

```
(* If possible, removes a character from the input stream cid and assigns the
   corresponding value to ch. The read result is set to the value allRight, endOfLine,
   or endOfInput.
   *)
```

PROCEDURE ReadRestLine (cid: IOChan.ChanId; **VAR** s: **ARRAY OF CHAR**);

```
(* Removes any remaining characters from the input stream cid before the next line
   mark, copying to s as many as can be accommodated as a string value. The read result
   is set to the value allRight, outOfRange, endOfLine, or endOfInput. *)
```

PROCEDURE ReadString (cid: IOChan.ChanId; **VAR** s: **ARRAY OF CHAR**);

```
(* Removes only those characters from the input stream cid before the next line
   mark that can be accommodated in s as a string value, and copies them to s. The read
   result is set to the value allRight, endOfLine, or endOfInput. *)
```

PROCEDURE ReadToken (cid: IOChan.ChanId; **VAR** s: **ARRAY OF CHAR**);

```
(* Skips leading spaces, and then removes characters from the input stream cid
   before the next space or line mark, copying to s as many as can be accommodated as a
   string value. The read result is set to the value allRight, outOfRange, endOfLine, or
   endOfInput. *)
```

```
(* The following procedure reads past the next line mark *)
```

PROCEDURE SkipLine (cid: IOChan.ChanId);

```
(* Removes successive items from the input stream cid up to and including the next
   line mark, or until the end of input is reached. The read result is set to the
   value allRight, or endOfInput. *)
```

```
(* Output procedures *)
```

PROCEDURE WriteChar (cid: IOChan.ChanId; ch: **CHAR**);

```
(* Writes the value of ch to the output stream cid. *)
```

PROCEDURE WriteLn (cid: IOChan.ChanId);

```
(* Writes a line mark to the output stream cid. *)
```

PROCEDURE WriteString (cid: IOChan.ChanId; s: **ARRAY OF CHAR**);

```
(* Writes the string value in s to the output stream cid. *)
```

END TextIO.

A5.4.2 WholeIO

DEFINITION MODULE WholeIO;

```
(* Input and output of whole numbers in decimal text form over specified
channels. The read result is of the type IOConsts.ReadResults.
*)
```

IMPORT IOChan;

```
(* The text form of a signed whole number is
   ["+" | "-"], decimal digit, {decimal digit}
```

```
   The text form of an unsigned whole number is
   decimal digit, {decimal digit}
```

```
*)
```

PROCEDURE ReadInt (cid: IOChan.ChanId; **VAR** int: **INTEGER**);

```
(* Skips leading spaces, and removes any remaining characters from cid that form
part of a signed whole number. The value of this number is assigned to int. The
read result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or
endOfInput. *)
```

PROCEDURE WriteInt (cid: IOChan.ChanId; int: **INTEGER**; width: **CARDINAL**);

```
(* Writes the value of int to cid in text form, in a field of the given
minimum width. *)
```

PROCEDURE ReadCard (cid: IOChan.ChanId; **VAR** card: **CARDINAL**);

```
(* Skips leading spaces, and removes any remaining characters from cid that form
part of an unsigned whole number. The value of this number is assigned to card. The
read result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or
endOfInput. *)
```

PROCEDURE WriteCard (cid: IOChan.ChanId; card: **CARDINAL**; width: **CARDINAL**);

```
(* Writes the value of card to cid in text form, in a field of the given
minimum width. *)
```

END WholeIO.

A5.4.3 RealIO

DEFINITION MODULE RealIO;

```
(* Input and output of real numbers in decimal text form over specified
channels. The read result is of the type IOConsts.ReadResults. *)
```

IMPORT IOChan;

```
(* The text form of a signed fixed-point real number is
   ["+" | "-"], decimal digit, {decimal digit},
   [".", {decimal digit}]
```

```
   The text form of a signed floating-point real number is
```

```

signed fixed-point real number,
"E", ["+" | "-"], decimal digit, {decimal digit}
*)

```

```

PROCEDURE ReadReal (cid: IOChan.ChanId; VAR real: REAL);

```

```

(* Skips leading spaces, and removes any remaining characters from cid that form
part of a signed fixed or floating point number. The value of this number is
assigned to real. The read result is set to the value allRight, outOfRange,
wrongFormat, endOfLine, or endOfInput. *)

```

```

PROCEDURE WriteFloat (cid: IOChan.ChanId; real: REAL; sigFigs: CARDINAL; width:
CARDINAL);

```

```

(* Writes the value of real to cid in floating-point text form, with sigFigs
significant figures, in a field of the given minimum width. *)

```

```

PROCEDURE WriteEng (cid: IOChan.ChanId; real: REAL; sigFigs: CARDINAL; width:
CARDINAL);

```

```

(* As for WriteFloat, except that the number is scaled with one to three
digits in the whole number part, and with an exponent that is a multiple of three. *)

```

```

PROCEDURE WriteFixed (cid: IOChan.ChanId; real: REAL; place: INTEGER; width:
CARDINAL);

```

```

(* Writes the value of real to cid in fixed-point text form, rounded to the
given place relative to the decimal point, in a field of the given minimum width. *)

```

```

PROCEDURE WriteReal (cid: IOChan.ChanId; real: REAL; width: CARDINAL);

```

```

(* Writes the value of real to cid, as WriteFixed if the sign and magnitude can be
shown in the given width, or otherwise as WriteFloat. The number of places or
significant digits depends on the given width. *)

```

```

END RealIO.

```

A5.4.4 LongIO

```

DEFINITION MODULE LongIO;

```

```

(* Input and output of real numbers in decimal text form over specified
channels. The read result is of the type IOConsts.ReadResults. *)

```

```

IMPORT IOChan;

```

```

(* The text form of a signed fixed-point real number is
["+" | "-"], decimal digit, {decimal digit},
[".", {decimal digit}]

```

```

The text form of a signed floating-point real number is
signed fixed-point real number,
"E", ["+" | "-"], decimal digit, {decimal digit}

```

```

*)

```

```

PROCEDURE ReadReal (cid: IOChan.ChanId; VAR real: LONGREAL);

```

```

(* Skips leading spaces, and removes any remaining characters from cid that form
part of a signed fixed or floating point number. The value of this number is
assigned to real. The read result is set to the value allRight, outOfRange,

```

```
wrongFormat, endOfLine, or endOfInput.
```

```
*)
```

```
PROCEDURE WriteFloat (cid: IOChan.ChanId; real: LONGREAL; sigFigs: CARDINAL; width: CARDINAL);
```

```
  (* Writes the value of real to cid in floating-point text form, with sigFigs significant figures, in a field of the given minimum width. *)
```

```
PROCEDURE WriteEng (cid: IOChan.ChanId; real: LONGREAL; sigFigs: CARDINAL; width: CARDINAL);
```

```
  (* As for WriteFloat, except that the number is scaled with one to three digits in the whole number part, and with an exponent that is a multiple of three. *)
```

```
PROCEDURE WriteFixed (cid: IOChan.ChanId; real: LONGREAL; place: INTEGER; width: CARDINAL);
```

```
  (* Writes the value of real to cid in fixed-point text form, rounded to the given place relative to the decimal point, in a field of the given minimum width. *)
```

```
PROCEDURE WriteReal (cid: IOChan.ChanId; real: LONGREAL; width: CARDINAL);
```

```
  (* Writes the value of real to cid, as WriteFixed if the sign and magnitude can be shown in the given width, or otherwise as WriteFloat. The number of places or significant digits depends on the given width. *)
```

```
END LongIO.
```

A5.4.5 RawIO

```
DEFINITION MODULE RawIO;
```

```
  (* Reading and writing data over specified channels using raw operations, that is, with no conversion or interpretation. The read result is of the type IOConsts.ReadResults. *)
```

```
IMPORT IOChan, SYSTEM;
```

```
PROCEDURE Read (cid: IOChan.ChanId; VAR to: ARRAY OF SYSTEM.LOC);
```

```
  (* Reads storage units from cid, and assigns them to successive components of to. The read result is set to the value allRight, wrongFormat, or endOfInput. *)
```

```
PROCEDURE Write (cid: IOChan.ChanId; from: ARRAY OF SYSTEM.LOC);
```

```
  (* Writes storage units to cid from successive components of from. *)
```

```
END RawIO.
```

A5.4.6 IOResult

```
DEFINITION MODULE IOResult;
```

```
  (* Read results for specified channels *)
```

```
IMPORT IOConsts, IOChan;
```

```
TYPE
```

```
  ReadResults = IOConsts.ReadResults;
```

```

PROCEDURE ReadResult (cid: IOChan.ChanId): ReadResults;
  (* Returns the result for the last read operation on the channel cid. *)

END IOResult.

```

A5.5 Channel Constants--IOConsts

```

DEFINITION MODULE IOConsts;

(* Types and constants for input/output modules *)

TYPE
  ReadResults =
    (* This type is used to classify the result of an input operation *)
    (
      notKnown,      (* no data read result is set *)
      allRight,      (* data is as expected or as required *)
      outOfRange,    (* data cannot be represented *)
      wrongFormat,   (* data not in expected format *)
      endOfLine,     (* end of line seen before expected data *)
      endOfInput     (* end of input seen before expected data *)
    );

END IOConsts.

```

A5.6 Device Independent Channel I/O--IOChan

```

DEFINITION MODULE IOChan;

(* Types and procedures forming the interface to channels for
device-independent data transfer modules *)

IMPORT IOConsts, ChanConsts, SYSTEM;

TYPE
  ChanId; (* Values of this type are used to identify channels *)

  (* There is one pre-defined value identifying an invalid channel on which no data
transfer operations are available. It may be used to initialize variables of type
ChanId. *)

PROCEDURE InvalidChan (): ChanId;
  (* Returns the value identifying the invalid channel. *)

  (* For each of the following operations, if the device supports the operation on
the channel, the behaviour of the procedure conforms with the description below. The
full behaviour is defined for each device module. If the device does not support the
operation on the channel, the behaviour of the procedure is to raise the exception
notAvailable. *)

  (* Text operations - these perform any required translation between the internal
and external representation of text. *)

PROCEDURE Look (cid: ChanId; VAR ch: CHAR; VAR res: IOConsts.ReadResults);

```

```

    (* If there is a character as the next item in the input stream cid, assigns its
value to ch without removing it from the stream; otherwise the value of ch is not
defined.  res (and the stored read result) are set to the value allRight, endOfLine,
or endOfInput. *)

PROCEDURE Skip (cid: ChanId);
    (* If the input stream cid has ended, the exception skipAtEnd is raised; otherwise
the next character or line mark in cid is removed, and the stored read result is set
to the value allRight. *)

PROCEDURE SkipLook (cid: ChanId; VAR ch: CHAR; VAR res: IOConsts.ReadResults);
    (* If the input stream cid has ended, the exception skipAtEnd is raised; otherwise
the next character or line mark in cid is removed.  If there is a character as the
next item in cid stream, assigns its value to ch without removing it from the stream.
    Otherwise, the value of ch is not defined.  res (and the stored read result) are
set to the value allRight, endOfLine, or endOfInput. *)

PROCEDURE WriteLn (cid: ChanId);
    (* Writes a line mark over the channel cid. *)

PROCEDURE TextRead (cid: ChanId; to: SYSTEM.ADDRESS; maxChars: CARDINAL; VAR
charsRead: CARDINAL);
    (* Reads at most maxChars characters from the current line in cid, and assigns
corresponding values to successive components of an ARRAY OF CHAR variable for which
the address of the first component is to. The number of characters read is assigned
to charsRead. The stored read result is set to allRight, endOfLine, or endOfInput. *)

PROCEDURE TextWrite (cid: ChanId; from: SYSTEM.ADDRESS; charsToWrite:
CARDINAL);
    (* Writes a number of characters given by the value of charsToWrite, from
successive components of an ARRAY OF CHAR variable for which the address of the first
component is from, to the channel cid. *)

    (* Direct raw operations - these do not effect translation between the internal
and external representation of data  *)

PROCEDURE RawRead (cid: ChanId; to: SYSTEM.ADDRESS; maxLocs: CARDINAL; VAR locsRead:
CARDINAL);
    (* Reads at most maxLocs items from cid, and assigns corresponding values to
successive components of an ARRAY OF LOC variable for which the address of the first
component is to. The number of characters read is assigned to charsRead. The stored
read result is set to the value allRight, or endOfInput. *)

PROCEDURE RawWrite (cid: ChanId; from: SYSTEM.ADDRESS; locsToWrite: CARDINAL);
    (* Writes a number of items given by the value of charsToWrite, from successive
components of an ARRAY OF LOC variable for which the address of the first component
is from, to the channel cid. *)

    (* Common operations *)

PROCEDURE GetName (cid: ChanId; VAR s: ARRAY OF CHAR);
    (* Copies to s a name associated with the channel cid, possibly truncated
(depending on the capacity of s).  *)

PROCEDURE Reset (cid: ChanId);

```

```
(* Resets the channel cid to a state defined by the device module. *)
```

```
PROCEDURE Flush (cid: ChanId);
```

```
(* Flushes any data buffered by the device module out to the channel cid. *)
```

```
(* Access to read results *)
```

```
PROCEDURE SetReadResult (cid: ChanId; res: IOConsts.ReadResults);
```

```
(* Sets the read result value for the channel cid to the value res. *)
```

```
PROCEDURE ReadResult (cid: ChanId): IOConsts.ReadResults;
```

```
(* Returns the stored read result value for the channel cid. (This is initially the value notKnown). *)
```

```
(* Users can discover which flags actually apply to a channel *)
```

```
PROCEDURE CurrentFlags (cid: ChanId): ChanConsts.FlagSet;
```

```
(* Returns the set of flags that currently apply to the channel cid. *)
```

```
(* The following exceptions are defined for this module and its clients *)
```

```
TYPE
```

```
ChanExceptions =
```

```
(wrongDevice,      (* device specific operation on wrong device *)
```

```
notAvailable,      (* operation attempted that is not available on that channel *)
```

```
skipAtEnd,         (* attempt to skip data from a stream that has ended *)
```

```
softDeviceError,   (* device specific recoverable error *)
```

```
hardDeviceError,   (* device specific non-recoverable error *)
```

```
textParseError,    (* input data does not correspond to a character or line mark - optional detection *)
```

```
notAChannel        (* given value does not identify a channel - optional detection *)
```

```
);
```

```
PROCEDURE IsChanException (): BOOLEAN;
```

```
(* Returns TRUE if the current coroutine is in the exceptional execution state because of the raising of an exception from ChanExceptions;
```

```
otherwise returns FALSE. *)
```

```
PROCEDURE ChanException (): ChanExceptions;
```

```
(* If the current coroutine is in the exceptional execution state because of the raising of an exception from ChanExceptions, returns the corresponding enumeration value, and otherwise raises an exception. *)
```

```
(* When a device procedure detects a device error, it raises the exception softDeviceError or hardDeviceError. If these exceptions are handled, the following facilities may be used to discover an implementation-defined error number for the channel. *)
```

```
TYPE
```

```
DeviceErrNum = INTEGER;
```

```
PROCEDURE DeviceError (cid: ChanId): DeviceErrNum;
```

```
(* If a device error exception has been raised for the channel cid, returns the
```



```
error number stored by the device module. *)
```

```
END IOChan.
```

A5.7 Device Drivers

A5.7.1 StreamFile

```
DEFINITION MODULE StreamFile;
```

```
  (* Independent sequential data streams *)
```

```
IMPORT IOChan, ChanConsts;
```

```
TYPE
```

```
  ChanId = IOChan.ChanId;
```

```
  FlagSet = ChanConsts.FlagSet;
```

```
  OpenResults = ChanConsts.OpenResults;
```

```
  (* Accepted singleton values of FlagSet *)
```

```
CONST
```

```
  read = FlagSet{ChanConsts.readFlag};    (* input operations are requested/available *)
```

```
  write = FlagSet{ChanConsts.writeFlag};  (* output operations are requested/available *)
```

```
  old = FlagSet{ChanConsts.oldFlag};      (* a file may/must/did exist before the channel is opened *)
```

```
  text = FlagSet{ChanConsts.textFlag};    (* text operations are requested/available *)
```

```
  raw = FlagSet{ChanConsts.rawFlag};      (* raw operations are requested/available *)
```

```
PROCEDURE Open (VAR cid: ChanId; name: ARRAY OF CHAR; flags: FlagSet; VAR res: OpenResults);
```

```
  (* Attempts to obtain and open a channel connected to a sequential stream of the given name. The read flag implies old; without the raw flag, text is implied. If successful, assigns to cid the identity of the opened channel, and assigns the value opened to res. If a channel cannot be opened as required, the value of res indicates the reason, and cid identifies the invalid channel.  *)
```

```
PROCEDURE IsStreamFile (cid: ChanId): BOOLEAN;
```

```
  (* Tests if the channel identified by cid is open to a sequential stream. *)
```

```
PROCEDURE Close (VAR cid: ChanId);
```

```
  (* If the channel identified by cid is not open to a sequential stream, the exception wrongDevice is raised; otherwise closes the channel, and assigns the value identifying the invalid channel to cid.  *)
```

```
END StreamFile.
```

A5.7.2 SeqFile

```
DEFINITION MODULE SeqFile;
```


(* Rewindable sequential files *)

IMPORT IOChan, ChanConsts;

TYPE

ChanId = IOChan.ChanId;

FlagSet = ChanConsts.FlagSet;

OpenResults = ChanConsts.OpenResults;

(* Accepted singleton values of FlagSet *)

CONST

read = FlagSet{ChanConsts.readFlag}; (* input operations are
requested/available *)

write = FlagSet{ChanConsts.writeFlag}; (* output operations are
requested/available *)

old = FlagSet{ChanConsts.oldFlag}; (* a file may/must/did exist before the
channel is opened *)

text = FlagSet{ChanConsts.textFlag}; (* text operations are
requested/available *)

raw = FlagSet{ChanConsts.rawFlag}; (* raw operations are
requested/available *)

PROCEDURE OpenWrite (**VAR** cid: ChanId; name: **ARRAY OF CHAR**; flags: FlagSet; **VAR** res:
OpenResults);

(* Attempts to obtain and open a channel connected to a stored rewindable file of
the given name.

The write flag is implied; without the raw flag, text is implied.

If successful, assigns to cid the identity of the opened channel, assigns
the value opened to res, and selects output mode, with the write position at the
start of the file (i.e. the file is of zero length).

If a channel cannot be opened as required, the value of res indicates the
reason, and cid identifies the invalid channel.

*)

PROCEDURE OpenAppend (**VAR** cid: ChanId; name: **ARRAY OF CHAR**; flags: FlagSet; **VAR** res:
OpenResults);

(* Attempts to obtain and open a channel connected to a stored rewindable file of
the given name.

The write and old flags are implied; without the raw flag, text is
implied. If successful, assigns to cid the identity of the opened channel, assigns
the value opened to res, and selects output mode, with the write position
corresponding to the length of the file.

If a channel cannot be opened as required, the value of res indicates the
reason, and cid identifies the invalid channel.

*)

PROCEDURE OpenRead (**VAR** cid: ChanId; name: **ARRAY OF CHAR**; flags: FlagSet; **VAR** res:
OpenResults);

(* Attempts to obtain and open a channel connected to a stored rewindable file of
the given name.

The read and old flags are implied; without the raw flag, text is implied.

If successful, assigns to cid the identity of the opened channel, assigns

the value opened to res, and selects input mode, with the read position corresponding to the start of the file.

If a channel cannot be opened as required, the value of res indicates the reason, and cid identifies the invalid channel.

*)

PROCEDURE IsSeqFile (cid: ChanId): **BOOLEAN**;

(* Tests if the channel identified by cid is open to a rewindable sequential file. *)

PROCEDURE Reread (cid: ChanId);

(* If the channel identified by cid is not open to a rewindable sequential file, the exception wrongDevice is raised; otherwise attempts to set the read position to the start of the file, and to select input mode.

If the operation cannot be performed (perhaps because of insufficient permissions) neither input mode nor output mode is selected.

*)

PROCEDURE Rewrite (cid: ChanId);

(* If the channel identified by cid is not open to a rewindable sequential file, the exception wrongDevice is raised; otherwise, attempts to truncate the file to zero length, and to select output mode.

If the operation cannot be performed (perhaps because of insufficient permissions) neither input mode nor output mode is selected.

*)

PROCEDURE Close (VAR cid: ChanId);

(* If the channel identified by cid is not open to a rewindable sequential file, the exception wrongDevice is raised; otherwise closes the channel, and assigns the value identifying the invalid channel to cid.

*)

END SeqFile.

A5.7.3 RndFile

DEFINITION MODULE RndFile;

(* Random access files *)

IMPORT IOChan, ChanConsts, SYSTEM;

TYPE

ChanId = IOChan.ChanId;

FlagSet = ChanConsts.FlagSet;

OpenResults = ChanConsts.OpenResults;

(* Accepted singleton values of FlagSet *)

CONST

read = FlagSet{ChanConsts.readFlag}; (* input operations are requested/available *)

write = FlagSet{ChanConsts.writeFlag}; (* output operations are requested/available *)

```

    old = FlagSet{ChanConsts.oldFlag};      (* a file may/must/did exist before the
channel is opened *)
    text = FlagSet{ChanConsts.textFlag};    (* text operations are
requested/available *)
    raw = FlagSet{ChanConsts.rawFlag};      (* raw operations are
requested/available *)

PROCEDURE OpenOld (VAR cid: ChanId; name: ARRAY OF CHAR; flags: FlagSet; VAR res:
OpenResults);
    (* Attempts to obtain and open a channel connected to a stored random access file
of the given name.
    The old flag is implied; without the write flag, read is implied; without the
text flag, raw is implied.
    If successful, assigns to cid the identity of the opened channel, assigns
the value opened to res, and sets the read/write position to the start of the file.
    If a channel cannot be opened as required, the value of res indicates the
reason, and cid identifies the invalid channel.  *)

PROCEDURE OpenClean (VAR cid: ChanId; name: ARRAY OF CHAR; flags: FlagSet; VAR res:
OpenResults);
    (* Attempts to obtain and open a channel connected to a stored random access file
of the given name.
    The write flag is implied; without the text flag, raw is implied.
    If successful, assigns to cid the identity of the opened channel, assigns
the value opened to res, and truncates the file to zero length.
    If a channel cannot be opened as required, the value of res indicates the
reason, and cid identifies the invalid channel.  *)

PROCEDURE IsRndFile (cid: ChanId): BOOLEAN;
    (* Tests if the channel identified by cid is open to a random access file. *)

PROCEDURE IsRndFileException (): BOOLEAN;
    (* Returns TRUE if the current coroutine is in the exceptional execution state
because of the raising of a RndFile exception; otherwise returns FALSE.  *)

CONST
    FilePosSize = 4;  (***** version defined*****)

TYPE
    FilePos = ARRAY [1 .. FilePosSize] OF SYSTEM.LOC;

PROCEDURE StartPos (cid: ChanId): FilePos;
    (* If the channel identified by cid is not open to a random access file, the
exception wrongDevice is raised; otherwise returns the position of the start of the
file.  *)

PROCEDURE CurrentPos (cid: ChanId): FilePos;
    (* If the channel identified by cid is not open to a random access file, the
exception wrongDevice is raised; otherwise returns the position of the current
read/write position.  *)

PROCEDURE EndPos (cid: ChanId): FilePos;
    (* If the channel identified by cid is not open to a random access file, the
exception wrongDevice is raised; otherwise returns the first position after which
there have been no writes.  *)

```

```

PROCEDURE NewPos (cid: ChanId; chunks: INTEGER; chunkSize: CARDINAL; from: FilePos):
FilePos;
    (* If the channel identified by cid is not open to a random access file, the
exception wrongDevice is raised; otherwise returns the position (chunks * chunkSize)
relative to the position given by from, or raises the exception posRange if the
required position cannot be represented as a value of type FilePos.  *)

PROCEDURE SetPos (cid: ChanId; pos: FilePos);
    (* If the channel identified by cid is not open to a random access file, the
exception wrongDevice is raised; otherwise sets the read/write position to the value
given by pos.  *)

PROCEDURE Close (VAR cid: ChanId);
    (* If the channel identified by cid is not open to a random access file, the
exception wrongDevice is raised; otherwise closes the channel, and assigns the value
identifying the invalid channel to cid.  *)

END RndFile.

```

A5.7.4 TermFile

```

DEFINITION MODULE TermFile;

```

```

    (* Access to the terminal device *)

```

```

    (* Channels opened by this module are connected to a single terminal device; typed
characters are distributed between channels according to the sequence of read
requests. *)

```

```

IMPORT IOChan, ChanConsts;

```

```

TYPE

```

```

    ChanId = IOChan.ChanId;

```

```

    FlagSet = ChanConsts.FlagSet;

```

```

    OpenResults = ChanConsts.OpenResults;

```

```

    (* Accepted singleton values of FlagSet *)

```

```

CONST

```

```

    read = FlagSet{ChanConsts.readFlag};    (* input operations are
requested/available *)

```

```

    write = FlagSet{ChanConsts.writeFlag}; (* output operations are
requested/available *)

```

```

    text = FlagSet{ChanConsts.textFlag};    (* text operations are
requested/available *)

```

```

    raw = FlagSet{ChanConsts.rawFlag};      (* raw operations are
requested/available *)

```

```

    echo = FlagSet{ChanConsts.echoFlag};    (* echoing by interactive device on reading
of characters from input stream requested/applies *)

```

```

PROCEDURE Open (VAR cid: ChanId; flags: FlagSet; VAR res: OpenResults);

```

```

    (* Attempts to obtain and open a channel connected to the terminal.
    Without the raw flag, text is implied.

```

```

    *)

```

Without the echo flag, line mode is requested, otherwise single character mode is requested.

If successful, assigns to cid the identity of the opened channel, and assigns the value opened to res.

If a channel cannot be opened as required, the value of res indicates the reason, and cid identifies the invalid channel. *)

PROCEDURE IsTermFile (cid: ChanId): **BOOLEAN**;

(* Tests if the channel identified by cid is open to the terminal. *)

PROCEDURE Close (VAR cid: ChanId);

(* If the channel identified by cid is not open to the terminal, the exception wrongDevice is raised; otherwise closes the channel and assigns the value identifying the invalid channel to cid. *)

END TermFile.

A5.8 Device Module Constants--ChanConsts

DEFINITION MODULE ChanConsts;

(* Common types and values for channel open requests and results *)

TYPE

ChanFlags = (* Request flags possibly given when a channel is opened *)
(readFlag, (* input operations are requested/available *)
writeFlag, (* output operations are requested/available *)
oldFlag, (* a file may/must/did exist before the channel is opened *)
textFlag, (* text operations are requested/available *)
rawFlag, (* raw operations are requested/available *)
interactiveFlag, (* interactive use is requested/applies *)
echoFlag (* echoing by interactive device on removal of characters from
input stream requested/applies *)
);

FlagSet = **SET OF** ChanFlags;

(* Singleton values of FlagSet, to allow for example, read + write *)

CONST

read = FlagSet{readFlag}; (* input operations are requested/available *)
write = FlagSet{writeFlag}; (* output operations are requested/available *)
old = FlagSet{oldFlag}; (* a file may/must/did exist before the channel is
opened *)
text = FlagSet{textFlag}; (* text operations are requested/available *)
raw = FlagSet{rawFlag}; (* raw operations are requested/available *)
interactive = FlagSet{interactiveFlag}; (* interactive use is
requested/applies *)
echo = FlagSet{echoFlag}; (* echoing by interactive device on removal of
characters from input stream requested/applies *)

TYPE

OpenResults = (* Possible results of open requests *)
(opened, (* the open succeeded as requested *))

```

        wrongNameFormat,    (* given name is in the wrong format for the
implementation *)
        wrongFlags,         (* given flags include a value that does not apply to the
device *)
        tooManyOpen,        (* this device cannot support any more open channels *)
        outOfChans,         (* no more channels can be allocated *)
        wrongPermissions,   (* file or directory permissions do not allow request *)
        noRoomOnDevice,     (* storage limits on the device prevent the open *)
        noSuchFile,         (* a needed file does not exist *)
        fileExists,         (* a file of the given name already exists when a new one is
required *)
        wrongFileType,      (* the file is of the wrong type to support the required
operations *)
        noTextOperations,   (* text operations have been requested, but are not supported
*)
        noRawOperations,    (* raw operations have been requested, but are not supported
*)
        noMixedOperations,  (* text and raw operations have been requested, but they are
not supported in combination *)
        alreadyOpen,        (* the source/destination is already open for operations not
supported in combination with the requested operations *)
        otherProblem        (* open failed for some other reason *)
    );

```

END ChanConsts.

A5.9 Linking Drivers To Channels--IOLink

DEFINITION MODULE IOLink;

(* Types and procedures for the standard implementation of channels *)

IMPORT IOChan, IOConsts, ChanConsts, SYSTEM;

TYPE

DeviceId;

(* Values of this type are used to identify new device modules, and are normally obtained by them during their initialization.

*)

PROCEDURE AllocateDeviceId (**VAR** did: DeviceId);

(* Allocates a unique value of type DeviceId, and assigns this value to did. *)

PROCEDURE MakeChan (did: DeviceId; **VAR** cid: IOChan.ChanId);

(* Attempts to make a new channel for the device module identified by did. If no more channels can be made, the identity of the invalid channel is assigned to cid. Otherwise, the identity of a new channel is assigned to cid. *)

PROCEDURE UnMakeChan (did: DeviceId; **VAR** cid: IOChan.ChanId);

(* If the device module identified by did is not the module that made the channel identified by cid, the exception wrongDevice is raised; otherwise the channel is deallocated, and the value identifying the invalid channel is assigned to cid. *)

TYPE

```
DeviceTablePtr = POINTER TO DeviceTable;
(* Values of this type are used to refer to device tables *)
```

TYPE

```
LookProc = PROCEDURE (DeviceTablePtr, VAR CHAR, VAR IOConsts.ReadResults);
SkipProc = PROCEDURE (DeviceTablePtr);
SkipLookProc = PROCEDURE (DeviceTablePtr, VAR CHAR, VAR IOConsts.ReadResults);
WriteLnProc = PROCEDURE (DeviceTablePtr);
TextReadProc = PROCEDURE (DeviceTablePtr, SYSTEM.ADDRESS, CARDINAL, VAR CARDINAL);
TextWriteProc = PROCEDURE (DeviceTablePtr, SYSTEM.ADDRESS, CARDINAL);
RawReadProc = PROCEDURE (DeviceTablePtr, SYSTEM.ADDRESS, CARDINAL, VAR CARDINAL);
RawWriteProc = PROCEDURE (DeviceTablePtr, SYSTEM.ADDRESS, CARDINAL);
GetNameProc = PROCEDURE (DeviceTablePtr, VAR ARRAY OF CHAR);
ResetProc = PROCEDURE (DeviceTablePtr);
FlushProc = PROCEDURE (DeviceTablePtr);
FreeProc = PROCEDURE (DeviceTablePtr);
(* Carry out the operations involved in closing the corresponding channel,
including flushing buffers, but do not unmake the channel. *)
```

TYPE

```
DeviceData = SYSTEM.ADDRESS;
```

```
DeviceTable =
```

```
RECORD (* Initialized by MakeChan to: *)
  cd: DeviceData; (* the value NIL *)
  did: DeviceId; (* the value given in the call of MakeChan *)
  cid: IOChan.ChanId; (* the identity of the channel *)
  result: IOConsts.ReadResults; (* the value notKnown *)
  errNum: IOChan.DeviceErrNum; (* undefined *)
  flags: ChanConsts.FlagSet; (* ChanConsts.FlagSet{} *)
  doLook: LookProc; (* raise exception notAvailable *)
  doSkip: SkipProc; (* raise exception notAvailable *)
  doSkipLook: SkipLookProc; (* raise exception notAvailable *)
  doLnWrite: WriteLnProc; (* raise exception notAvailable *)
  doTextRead: TextReadProc; (* raise exception notAvailable *)
  doTextWrite: TextWriteProc; (* raise exception notAvailable *)
  doRawRead: RawReadProc; (* raise exception notAvailable *)
  doRawWrite: RawWriteProc; (* raise exception notAvailable *)
  doGetName: GetNameProc; (* return the empty string *)
  doReset: ResetProc; (* do nothing *)
  doFlush: FlushProc; (* do nothing *)
  doFree: FreeProc; (* do nothing *)
```

```
END;
```

```
(* The pointer to the device table for a channel is obtained using the following
procedure: *)
```

```
PROCEDURE DeviceTablePtrValue (cid: IOChan.ChanId; did: DeviceId): DeviceTablePtr;
(* If the device module identified by did is not the module that made the channel
identified by cid, the exception wrongDevice is raised; otherwise returns a pointer
to the device table for the channel. *)
```

```
PROCEDURE IsDevice (cid: IOChan.ChanId; did: DeviceId): BOOLEAN;
(* Tests if the device module identified by did is the module that made the channel
```


identified by cid. *)

TYPE

DevExceptionRange = [IOChan. notAvailable .. IOChan. textParseError];

PROCEDURE RAISEdevException (cid: IOChan.ChanId; did: DeviceId;

 x: DevExceptionRange; s: **ARRAY OF CHAR**);

 (* If the device module identified by did is not the module that made the channel
 identified by cid, the exception wrongDevice is raised; otherwise the given exception
 is raised, and the string value in s is included in the exception message. *)

PROCEDURE IsIOException (): **BOOLEAN**;

 (* If the current coroutine is in the exceptional execution state
 because of the raising of an exception from ChanExceptions;
 otherwise returns FALSE. *)

PROCEDURE IOException (): IOChan.ChanExceptions;

 (* Returns TRUE if the current coroutine is in the exceptional execution state
 because of the raising of an exception from ChanExceptions, returns the corresponding
 enumeration value, and otherwise raises an exception. *)

END IOLink.

[Contents](#)

Appendix 6--ISO Support Modules For This Text

All the modules in this section are system specific modules. They were written by the author in support of a personal trial implementation of the ISO library on the Macintosh computer. Since some references are made to these modules in the text, their definitions are reproduced here from highest level to lowest level.

A6.1 RedirStdIO

```
DEFINITION MODULE RedirStdIO;

(* =====
   Definition and Implementation © 1993-1997
       by R. Sutcliffe
       Trinity Western University
       7600 Glover Rd., Langley, BC Canada V3A 6H4
       e-mail: rsutcl@twu.ca
       Last modification date 1997 07 02
   ===== *)

IMPORT ChanConsts;

TYPE
    OpenResults = ChanConsts.OpenResults;

PROCEDURE OpenResult () : OpenResults;
(* returns the result of the last attempt to open a file for redirection *)

PROCEDURE OpenOutput;
(* engages the user in a dialog to obtain a file for redirection of standard Output
and attempts to open the file so obtained *)

PROCEDURE OpenOutputFile (VAR fileName: ARRAY OF CHAR);
(* opens the file specified by fileName for redirection of output.  If the name
supplied is the empty string or the file could not be opened, control passes to
OpenOutput and the filename eventually used is returned in the parameter. *)

PROCEDURE CloseOutput;
(* returns the standard output channel to the default value *)

PROCEDURE OpenInput;
(* engages the user in a dialog to obtain a file for redirection of standard Input
and attempts to open the file so obtained *)

PROCEDURE OpenInputFile (VAR fileName: ARRAY OF CHAR);
(* Opens the file specified by fileName for redirection of input.  If the name
supplied is the empty string or is not found, control passes to OpenInput and the
filename eventually used is returned in the parameter. *)

PROCEDURE CloseInput;
```

```
(* returns the standard input channel to the default value *)
```

```
END RedirStdIO.
```

A6.2 Filer

As can be seen from the credits, this module has a long history. With minor modifications, it has served the author for a decade of Modula-2 work, and it proved a simple matter to implement the ISO library on top of it.

```
DEFINITION MODULE Filer;
```

```
(* =====
```

```
Definition and Implementation © 1985-1995
```

```
by R. Sutcliffe
```

```
Trinity Western University
```

```
7600 Glover Rd., Langley, BC Canada V3A 6H4
```

```
e-mail: rsutc@twu.ca
```

```
Last modification date 1995 01 10
```

```
===== *)
```

```
(*
```

```
Copyright 1985 by R. Sutcliffe as "Files"
```

```
Version 1 for Apple ][ UCSD p-system 1985 by R. Sutcliffe
```

```
Version 2 for MS-DOS PCollier compiler 1987 by R. Sutcliffe
```

```
Version 3 for Sempersoft Modula-2 compiler using MPW
```

```
September 1988
```

```
research and first design by Greg Manning
```

```
Implementation by R. Sutcliffe
```

```
Version 4 for Sun Modula-2 (UNIX) 1989 by R. Sutcliffe
```

```
Version 5 Redesign version3 for the MetCom compiler 1990 09 12
```

```
by R. Sutcliffe
```

```
MPW and PSE versions implemented
```

```
Version 6 For Metrowerks PSE 4.03 compiler
```

```
Renamed "Channels" for a mid-level ISO-like implementation 1993 08 01 by R.  
Sutcliffe
```

```
initial research by Gord Tischer
```

```
redesign and implementation as part of ISO-like I/O suite by R. Sutcliffe
```

```
name changed to "Filer" to avoid toolbox conflict
```

```
changed order of parameters in Read/Write Bytes to typical ISO order
```

```
length parameter in Read/Write Bytes changed to VAR and CARDINAL
```

```
Version 6.1 last modification date 1993 10 21 read/open => var param
```

```
Version 6.2 last modification date 1994 01 28
```

```
added the concept of mode to support ISO
```

```
changed most file parameters from VAR to value
```

```
Version 6.2.1 last modification date 1994 05 18
```

```
to p1 MPW; requires LONGINT ==> INTEGER
```

```
Version 6.2.2 last modification date 1994 10 22
```

```
add OpenMode to record mode of original opening for additional ISO utility
```

```
added flush capability
```

```
fixed close finally
```

```
changed open semantics so that if it fails, the memory is deallocated
```

```
fixed a bug in reading by making the chrBuf and chrRead file by file 1995 01 06
```

Version 6.2.3 Added Lookup modified 1995 01 10 *)

(* This module provides basic file capabilities in a consistent interface to other modules. *)

FROM SYSTEM IMPORT

BYTE, WORD, ADDRESS;

FROM Types IMPORT

Str255;

FROM MacFileTypes IMPORT

NameType;

TYPE

File;

(* Each file procedure will return a result of type "FileErr".
The meaning of each possible result is explained below. *)

FileErr = (FileOK, (* everything worked *)
EOF, (* an read was made past the end of a file *)
DiskError, (* media is corrupted *)
FileExists, (* a file with that name already exists *)
FSErr, (* error in external file system *)
FileAbsent, (* no file with that name exists *)
TooManyOpen, (* tried to open too many files*)
BadName, (* an illegal file name was used: perhaps too long, or contains a
colon *)
DiskFull, (* there is no more room on the disk *)
FileLocked, (* fileges can't be made to locked files *)
FileBusy, (* that file is already open *)
NotOpen, (* attempt to access file not open *)
OtherError); (* one of the rarer errors has occurred *)

Mode = (none, input, output, both, invalid);
OpenMode = (notSet, read, write, readwrite, append, other);

VAR

FileDone : FileErr;
useDialogs : **BOOLEAN**; (* set to **FALSE**, but if reset by user, create and open bring
up standard dialog boxes *)

PROCEDURE Lookup (filename : **ARRAY OF CHAR**) : **BOOLEAN**;

(* If the file with the given name is present returns **TRUE** and sets FileDone to
FileOK
if not, returns **FALSE** and sets FileDone to FileAbsent or another error *)

PROCEDURE Create (**VAR** filename : **ARRAY OF CHAR**; creator, type : NameType);

(* Creates a new, empty file in the current directory.
The new file will be named as specified in 'name'.
The new file will have a signature as specified in 'creator'.
The new file will have a type as specified in 'type'.
Latter two only in Mac version
NB. A newly created file must still be opened to be used.

The standard dialog is used if useDialogs is set to **TRUE** or if the filename passed is an empty string. In that case, the actual filename opened by the user is passed back in filename. *)

PROCEDURE Open (VAR file : File; VAR filename : **ARRAY OF CHAR**);

(* Creates the variable "file"

Opens an already existent file in the current directory.

The file whose name is specified in 'filename' will be opened.

'file' is set to refer to the opened file.

Any information previously stored in 'file' will be lost.

It is important to close all opened files before ending a program.

The standard dialog is used if the filename passed is an empty string.

The mode is set to "both" as a default as this is a low level proc

The OpenMode is set to notSet

If the open fails, the memory for the file variable is deallocated *)

(* The following procedures operate on files specified by name only. *)

PROCEDURE Delete (filename : **ARRAY OF CHAR**);

(* Deletes the file specified in 'filename'. Use with caution! *)

PROCEDURE Rename (oldname, newname : **ARRAY OF CHAR**);

(* Renames the file specified by 'oldname' as specified by 'newname'. *)

(*=====

The following procedures require that 'file' refer to a previously opened file.

Calling one of them when 'file' is invalid may produce unpredictable results.

Hopefully an error will result if 'file' was invalid, but it is possible that 'file' contains a valid value by accident. *)

PROCEDURE GetName (file: File; VAR filename : **ARRAY OF CHAR**);

(* Returns the name of the file that was used when it was opened *)

PROCEDURE Eof (file : File) : **BOOLEAN**;

(* Returns TRUE if the file position is at or past logical EOF

The return result is defined only if the error state is FileOK *)

PROCEDURE GetMode (file : File) : Mode;

(* Returns the currently mode on this file *)

PROCEDURE SetMode (file : File; mode : Mode);

(* Sets the current mode on this file *)

PROCEDURE GetOpenMode (file : File) : OpenMode;

(* Returns the open mode on this file *)

PROCEDURE SetOpenMode (file : File; mode : OpenMode);

(* If the OpenMode is notSet, Sets the open mode on this file;

If the OpenMode is anything else, does nothing, but no error condition is set should only be done just after opening *)

PROCEDURE FileState (file : File) : FileErr;

(* Returns the currently set error on this file *)

PROCEDURE Flush (file : File);

(* The volume on which the file is located will be flushed.

There is no guarantee that any data for a file has been written to the disk until the associated volume has been flushed, either by this procedure or by closing the file. *)

PROCEDURE Close (file : File);

(* The file specified in 'file' will be closed.

After calling this procedure, if the close was successful, the memory is deallocated and variable is set to NIL; if it is not successful, the mode becomes invalid, and the variable is retained. The file cannot be reopened with the same file record, but another attempt to close can be made. *)

PROCEDURE ReadByte (file : File; **VAR** input : BYTE);

(* Reads the next byte from the file 'file' and stores it in 'input'. *)

PROCEDURE Look (file : File; **VAR** ch : **CHAR**);

(* Reads but does not remove from the input stream the next character from the file 'file' and stores it in 'ch'. *)

PROCEDURE Skip (file : File);

(* Removes from the input stream the next byte from the file 'file' *)

PROCEDURE SkipLook (file : File; **VAR** ch : **CHAR**);

(* Does a skip followed by a look *)

PROCEDURE ReadChar (file : File; **VAR** ch : **CHAR**);

(* Reads the next character from the file 'file' and stores it in 'ch'. *)

PROCEDURE ReadWord (file : File; **VAR** input : WORD);

(* Reads the next word from the file 'file' and stores it in 'input'. *)

PROCEDURE ReadBytes (file : File; buffer : ADDRESS; **VAR** length : **CARDINAL**);

(* Reads 'length' bytes from the file 'file' and stores them at 'buffer'.

Actual number of bytes read returned in length

If the buffer is too small, data will be overwritten.

This is a low level procedure -- dont use unless you know what you are doing *)

PROCEDURE ReadRec(file : File; **VAR** rec : **ARRAY OF** BYTE);

(* This is a safer high level procedure for reading records *)

PROCEDURE WriteByte(file : File; output : BYTE);

(* Writes the byte in 'output' to the file 'file'. *)

PROCEDURE WriteChar(file : File; ch : **CHAR**);

(* Writes the char in 'ch' to the file 'file'. *)

PROCEDURE WriteWord(file : File; output : WORD);

(* Writes the word in 'output' to the file 'file'. *)

PROCEDURE WriteBytes(file : File; buffer : ADDRESS; **VAR** length : **CARDINAL**);

(* Writes the 'length' bytes starting at 'buffer' to the file 'file'.

Actual number of bytes read returned in length

If the buffer is too small, undefined bytes will be written.

So don't use this unless you know what you are doing *)

```

PROCEDURE WriteRec(file : File; rec : ARRAY OF BYTE);
    (* This is a safe high level procedure for writing records *)

(* Following for random access files *)

PROCEDURE GetPos (file : File; VAR pos : INTEGER);

PROCEDURE GetEOF (file : File; VAR pos : INTEGER);

PROCEDURE SetPos (file : File; pos : INTEGER);
    (* sets from beginning of file *)

PROCEDURE SetEOF (file : File; pos : INTEGER);

END Filer.

```

A6.3 Keyboard

As suggested in the appendix on classical I/O modules, a keyboard reading module is a common item in many implementations. The author's follows.

```

DEFINITION MODULE Keyboard;

(* =====
   Definition and Implementation © 1993
       by R. Sutcliffe
   ===== *)

PROCEDURE BusyRead (VAR ch: CHAR);
    (* return character if one is there, otherwise return 0C *)

PROCEDURE Read (VAR ch: CHAR);
    (* return character *)

END Keyboard.

```

A6.4 CharBuffer

In order to implement the ISO library *Look* functionality, the author decided to employ a buffer to store text coming from the keyboard until it was "read" by the I/O routines. Because this seemed like a general problem, the required functionality was included in a library. The maximum buffer size is arbitrary.

```

DEFINITION MODULE CharBuffer;

(* =====
   Definition and Implementation © 1993
       by R. Sutcliffe
   ===== *)

(* provides a first in first out 1024 character buffer facility *)

TYPE

```

```

    Buffer;

PROCEDURE Init (VAR b : Buffer);
(* create a new empty buffer *)

PROCEDURE Destroy (VAR b: Buffer);
(* give all memory back to the system *)

PROCEDURE Flush (b : Buffer);
(* empty a buffer *)

PROCEDURE Full (b: Buffer) : BOOLEAN;
(* returns TRUE if the buffer cannot take any more characters
FALSE if it can *)

PROCEDURE Empty (b: Buffer) : BOOLEAN;
(* returns TRUE if the buffer cannot give back any more characters
FALSE if it can *)

PROCEDURE Enter (b : Buffer; ch: CHAR);
(* Enters the character.  If it was full, the first in is lost.
   If you don't like that way of doing it, write your own. *)

PROCEDURE Look (b : Buffer; VAR ch: CHAR);
(* get the first in without removing it *)

PROCEDURE Skip (b : Buffer);
(* remove the first in *)

PROCEDURE Erase (b : Buffer);
(* remove the last in *)

PROCEDURE Fetch (b : Buffer; VAR ch: CHAR);
(* get the first in and removes it *)

PROCEDURE Size (b : Buffer) : CARDINAL;
(* returns number of characters in the buffer *)

END CharBuffer.

```

A6.5 STerminal

A student terminal package was implemented so as to allow programs to have pull down menus and require an explicit quit command to exit. This makes it easier to copy or print material from the terminal window.

```

DEFINITION MODULE STerminal;
FROM SYSTEM IMPORT ADDRESS;
(* exported procedures *)
PROCEDURE WriteChar (ch : CHAR);
PROCEDURE WriteCharRef (adr : ADDRESS);
PROCEDURE WriteString (str : ARRAY OF CHAR);
PROCEDURE WriteStringRef (adr : ADDRESS; howMany : CARDINAL);
PROCEDURE WriteLn;
PROCEDURE BusyRead (VAR ch : CHAR);

```

```

PROCEDURE ReadChar (VAR ch : CHAR);
PROCEDURE ReadString (VAR str : ARRAY OF CHAR);
END STerminal.

```

A6.6 ASCII

```

DEFINITION MODULE ASCII;

```

```

(* R. Sutcliffe. Last Modification 1997 10 15 *)

```

```

CONST
  nul = 00C; soh = 01C; stx = 02C; etx = 03C; eot = 04C; enq = 05C;
  ack = 06C; bel = 07C; bs  = 10C; ht  = 11C; lf  = 12C; vt  = 13C;
  ff  = 14C; cr  = 15C; so  = 16C; si  = 17C; dle = 20C; dc1 = 21C;
  dc2 = 22C; dc3 = 23C; dc4 = 24C; nak = 25C; syn = 26C; etb = 27C;
  can = 30C; em  = 31C; sub = 32C; esc = 33C; fs  = 34C; gs  = 35C;
  rs  = 36C; us  = 37C; space = 40C; del = 177C;

```

```

END ASCII.

```

A6.7 SComplexIO

```

DEFINITION MODULE SComplexIO;

```

```

(* =====
   © 1996 by R. Sutcliffe
   Last modification date 1996 10 30
   =====*)

```

```

(* Input and output of complex numbers in decimal text form over the default
channels. The read result is of the type IOConsts.ReadResults. *)

```

```

IMPORT IOChan;

```

```

(* The text form of a complex number is
   realNumber, space ["+" | "-"], [space,] [realnumber, i] |
   [realNumber, space] ["+" | "-"], [space,] realnumber, i
   where the real numbers in each case are in the
   format specified for fixed or floating reals.
   *)

```

```

PROCEDURE ReadComplex (VAR complex: COMPLEX);

```

```

(* Skips leading spaces, and removes any remaining characters from cid that form
part of a complex number. The value of this number is assigned to complex. The read
result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or
endOfInput. *)

```

```

PROCEDURE WriteFloat (complex: COMPLEX; sigFigs: CARDINAL; width: CARDINAL);

```

```

(* Writes the value of complex to cid in floating-point real text form, with
sigFigs significant figures, in a field of the given minimum width. The width for the
real parts is 0 if the supplied width is 3 or less, and it is (width - 4) DIV 2

```



```
otherwise.  *)
```

```
PROCEDURE WriteEng (complex: COMPLEX; sigFigs: CARDINAL; width: CARDINAL);
  (* As for WriteFloat, except that the number is scaled with one to three
  digits in the whole number part, and with an exponent that is a multiple of three. *)

PROCEDURE WriteFixed (complex: COMPLEX; place: INTEGER; width: CARDINAL);
  (* Writes the value of complex to cid in fixed-point text form, with real parts
  rounded to the given place relative to the decimal point, in a field of the given
  minimum width. *)

PROCEDURE WriteComplex (complex: COMPLEX; width: CARDINAL);
  (* Writes the value of complex to cid, as WriteFixed if the sign and magnitude can
  be shown in the given width, or otherwise as WriteFloat. The number of places or
  significant digits depends on the given width. *)

END SComplexIO.
```

A6.8 SLongComplexIO

```
DEFINITION MODULE SLongComplexIO;

(* =====
   © 1996 by R. Sutcliffe
   Last modification date 1996 11 01
   =====*)

  (* Input and output of longcomplex numbers in decimal text form over the default
  channels. The read result is of the type IOConsts.ReadResults. *)

IMPORT IOChan;

  (* The text form of a LongComplex number is
     realNumber, [space], ["+" | "-"], [space,] [realnumber, i] |
     [realNumber, [space],] ["+" | "-"], [space,] realnumber, i
     where the real numbers in each case are in the
     format specified for fixed or floating longreals.
  *)

PROCEDURE ReadComplex (VAR complex: LONGCOMPLEX);
  (* Skips leading spaces, and removes any remaining characters from cid that form
  part of a complex number. The value of this number is assigned to complex. The read
  result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or
  endOfInput. *)

PROCEDURE WriteFloat (complex: LONGCOMPLEX; sigFigs: CARDINAL; width: CARDINAL);
  (* Writes the value of complex to cid in floating-point real text form, with
  sigFigs significant figures, in a field of the given minimum width. The width for the
  real parts is 0 if the supplied width is 3 or less, and it is (width - 4) DIV 2
  otherwise. *)

PROCEDURE WriteEng (complex: LONGCOMPLEX; sigFigs: CARDINAL; width: CARDINAL);
  (* As for WriteFloat, except that the number is scaled with one to three
```

digits in the whole number part, and with an exponent that is a multiple of three. *)

PROCEDURE WriteFixed (complex: **LONGCOMPLEX**; place: **INTEGER**; width: **CARDINAL**);

(* Writes the value of complex to cid in fixed-point text form, with real parts rounded to the given place relative to the decimal point, in a field of the given minimum width. *)

PROCEDURE WriteComplex (complex: **LONGCOMPLEX**; width: **CARDINAL**);

(* Writes the value of complex to cid, as WriteFixed if the sign and magnitude can be shown in the given width, or otherwise as WriteFloat. The number of places or significant digits depends on the given width. *)

END SLongComplexIO.

A6.9 ComplexIO

DEFINITION MODULE ComplexIO;

(* =====

© 1996 by R. Sutcliffe

Last modification date 1996 10 30

=====*)

(* Input and output of complex numbers in decimal text form over specified channels. The read result is of the type IOConsts.ReadResults. *)

IMPORT IOChan;

(* The text form of a complex number is

realNumber, [space], ["+" | "-"], [space,] [realnumber, i] |

[realNumber, [space],] ["+" | "-"], [space,] realnumber, i

where the real numbers in each case are in the
format specified for fixed or floating reals.

*)

PROCEDURE ReadComplex (cid: IOChan.ChanId; **VAR** complex: **COMPLEX**);

(* Skips leading spaces, and removes any remaining characters from cid that form part of a complex number. The value of this number is assigned to complex. The read result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or endOfInput. *)

(* following procedure affects all the Write procs below *)

PROCEDURE SetVerbose (verbose : **BOOLEAN**);

(* if true prints both components even if one is zero; else prints only one if the other is zero. The default is false. *)

PROCEDURE WriteFloat (cid: IOChan.ChanId; complex: **COMPLEX**; sigFigs: **CARDINAL**; width: **CARDINAL**);

(* Writes the value of complex to cid in floating-point real text form, with sigFigs significant figures, in a field of the given minimum width. The width for the real parts is 0 if the supplied width is 3 or less, and it is (width - 4) **DIV** 2 otherwise. *)

```

PROCEDURE WriteEng (cid: IOChan.ChanId; complex: COMPLEX; sigFigs: CARDINAL; width:
CARDINAL);
    (* As for WriteFloat, except that the number is scaled with one to three
    digits in the whole number part, and with an exponent that is a multiple of three. *)

PROCEDURE WriteFixed (cid: IOChan.ChanId; complex: COMPLEX; place: INTEGER; width:
CARDINAL);
    (* Writes the value of complex to cid in fixed-point text form, with real parts
    rounded to the given place relative to the decimal point, in a field of the given
    minimum width. *)

PROCEDURE WriteComplex (cid: IOChan.ChanId; complex: COMPLEX; width: CARDINAL);
    (* Writes the value of complex to cid, as WriteFixed if the sign and magnitude can
    be shown in the given width, or otherwise as WriteFloat. The number of places or
    significant digits depends on the given width. *)

END ComplexIO.

```

A6.10 LongComplexIO

```

DEFINITION MODULE LongComplexIO;

(* =====
   © 1996 by R. Sutcliffe
   Last modification date 1996 11 01
   =====*)

    (* Input and output of LongComplex numbers in decimal text form over specified
    channels. The read result is of the type IOConsts.ReadResults. *)

IMPORT IOChan;

    (* The text form of a LongComplex number is
       realNumber, [space], ["+" | "-"], [space,] [realnumber, i] |
       [realNumber, [space,] ["+" | "-"], [space,] realnumber, i
       where the real numbers in each case are in the
       format specified for fixed or floating longreals.
    *)

PROCEDURE ReadComplex (cid: IOChan.ChanId; VAR complex: LONGCOMPLEX);
    (* Skips leading spaces, and removes any remaining characters from cid that form
    part of a complex number. The value of this number is assigned to complex. The read
    result is set to the value allRight, outOfRange, wrongFormat, endOfLine, or
    endOfInput. *)

    (* following procedure affects all the Write procs below *)
PROCEDURE SetVerbose (verbose : BOOLEAN);
    (* if true prints both components even if one is zero; else prints only one if the
    other is zero. The default is false. *)

PROCEDURE WriteFloat (cid: IOChan.ChanId; complex: LONGCOMPLEX; sigFigs: CARDINAL;
width: CARDINAL);
    (* Writes the value of complex to cid in floating-point real text form, with
    sigFigs significant figures, in a field of the given minimum width. The width for the

```

real parts is 0 if the supplied width is 3 or less, and it is (width - 4) DIV 2 otherwise. *)

PROCEDURE WriteEng (cid: IOChan.ChanId; complex: **LONGCOMPLEX**; sigFigs: **CARDINAL**; width: **CARDINAL**);

(* As for WriteFloat, except that the number is scaled with one to three digits in the whole number part, and with an exponent that is a multiple of three. *)

PROCEDURE WriteFixed (cid: IOChan.ChanId; complex: **LONGCOMPLEX**; place: **INTEGER**; width: **CARDINAL**);

(* Writes the value of complex to cid in fixed-point text form, with real parts rounded to the given place relative to the decimal point, in a field of the given minimum width. *)

PROCEDURE WriteComplex (cid: IOChan.ChanId; complex: **LONGCOMPLEX**; width: **CARDINAL**);

(* Writes the value of complex to cid, as WriteFixed if the sign and magnitude can be shown in the given width, or otherwise as WriteFloat. The number of places or significant digits depends on the given width. *)

END LongComplexIO.

A6.11 GraphPaper

DEFINITION MODULE GraphPaper;

(* Original design copyright 1996 by R. Sutcliffe
Original implementation 1996 using pl on the Macintosh
Windows implementation 1998 05 12 by Joel Schwartz
with use of examples written by Stony Brook
Last revision by RS 1998 07 11
*)

TYPE

CoordSystem = (MacWin, bearing, standard); (* default = standard *)
(* The MacWin Coordinate system grows down and has the origin at the top left hand corner with angles measured clockwise. In the bearing system the home position (0,0) is the middle of the screen from which angles are measured clockwise. The standard system also starts in mid screen and grows up but measures from east as zero and thence counterclockwise. *)

AngleType = (deg, rad, grad); (* Allows for angle type specification *)

LabelType = **ARRAY** [0..50] **OF CHAR**; (* Standard format for labels *)

PROCEDURE SetCoordSystem (kind : CoordSystem);

(* Allows the user to set the system. The default is the standard system so this has to be called only if a change is desired. This procedure concludes by calling Home. Any shift in the system origin must be made after calling this procedure. *)

PROCEDURE SetAngleType (kind : AngleType);

(* Allows the user to set the angle measurement type. The default is degrees so this has to be called only if a change is desired. This procedure concludes by calling Home. Any shift in the system origin must be made after calling this procedure. *)

```

PROCEDURE Home;
(* moves to 0,0 and then
   In the bearing system:
   - sets angle to 0 (North)
   - sets the rotational direction to clockwise
   In the MacWin & standard system:
   - sets the angle to 0 (East)
   - sets the rotational direction to clockwise (MacWin)
   or to counterclockwise (standard) *)

PROCEDURE ShiftOrigin (deltaX, deltaY : INTEGER);
(* Translate the origin by the amount specified. The direction of the translation on
the screen depends, of course, on the coordinate system being used. Drawing is now
with respect to the new origin. Does not call home or change any other settings. *)

PROCEDURE GetDimensions (VAR x, y: INTEGER);
(* obtains the overall width and height of the drawing screen *)
PROCEDURE GetLocation (VAR x, y: INTEGER);
(* get the drawing pen location in current coordinates *)

(* The following three procedures work in the current coordinate
system and on the stored pen position only but do no actual drawing. *)

PROCEDURE MoveBy (distance: INTEGER);
(* move in the stored direction by the supplied distance *)
PROCEDURE MoveTo (x, y : INTEGER);
(* move the drawing pen to the specified coordinates *)
PROCEDURE Move (dx, dy : INTEGER);
(* move the drawing pen to a point (x + dx, y + dy) from the currently stored point
*)

(* The following three procedures work in the current coordinate system on the stored
pen direction only but do no actual drawing.
The angle is assumed to be in the currently set units. *)

PROCEDURE GetCurrentAngle () : REAL;
(* Return the current angle in the current units *)
PROCEDURE Turn (angle : REAL);
(* change the stored pen direction by adding its angle to the one supplied *)
PROCEDURE TurnTo (angle : REAL);
(* change the stored pen direction by setting its angle to the one supplied *)

(* The following two procedures use the pen to draw a line and change the stored
position. *)

PROCEDURE LineBy (distance: INTEGER);
(* Draws in the stored pen direction the number of units supplied. *)
PROCEDURE LineTo (x, y : INTEGER);
(* Draws a line from the current stored position to the supplied one without using or
changing the stored direction. *)
PROCEDURE Line (dx, dy : INTEGER);
(* Line to a point (x + dx, y + dy) from the current point without using or changing
the stored direction. *)

```

```
(* The following two procedures place a dot on the screen, but do not change the pen
direction. Measurement is in pixels; not scaled. *)
```

```
PROCEDURE Dot;
```

```
    (* places a dot at the present location *)
```

```
PROCEDURE DotAt (x, y : INTEGER);
```

```
    (* does a MoveTo, then a dot *)
```

```
(* These procedures are for annotating the graph paper with a scale and labels for
the axes. *)
```

```
PROCEDURE SetLabels (horix, vert : LabelType);
```

```
    (* Sets the names for the horizontal and vertical axes. *)
```

```
PROCEDURE ShowLabels;
```

```
    (* Show the labels - if no label is set then "x" and "y" are used *)
```

```
PROCEDURE ShowAxes;
```

```
    (* Show the axes for the graph *)
```

```
(* The following procedures allow for a scaling of the graph paper and the plotting
of points according to the scale. If the scale is 10, there is one unit every ten
division marks. This will make the plotting of functions more readable. The default
is one unit per division mark.
```

```
    EXAMPLE: setting the scale to 5 using cm's as the unit means 1 cm = 5 division
marks. *)
```

```
PROCEDURE SetScale (dataPerDivision : CARDINAL);
```

```
    (* Set the scale by which the graph is measured *)
```

```
PROCEDURE PlotPoint( x, y : REAL);
```

```
    (* Plot a scaled point on the graph *)
```

```
PROCEDURE PolarPlotPoint (radius, angle : REAL);
```

```
    (* Moves to a given angle and radius and places a scaled dot at that point.
```

```
        The angle is assumed to be in the currently set units. *)
```

```
END GraphPaper.
```

A6.12 GraphWindow

```
DEFINITION MODULE GraphWindow;
```

```
(* Design and Macintosh implementattion by R. Sutcliffe
```

```
    Windows implementation by Joel Schwartz
```

```
    Last revision: 1998 07 07 *)
```

```
(* This module obtains and passes to applications that need it a simple graphics
window and its dimensions. Some applications will need only to import this module,
and possibly get the window dimensions, as graphing takes place in the current
grafport anyway, and this module will set that port, so GetWindow may not have to be
imported. *)
```

```
IMPORT (* MacOS *) Quickdraw; (* Windows: IMPORT WIN32; *)
```

```
(* for convenience we export the type of the reference; this makes it more compatible
to the
```

```
Windows version where we import the HDC type and define WindowRef to be an HDC. *)

TYPE
    WindowRef = Quickdraw.WindowRef;    (* Windows: WindowRef = WIN32.HDC *)

PROCEDURE GetWindow () : WindowRef;
PROCEDURE GetWDimensions (VAR width, height : INTEGER);

END GraphWindow.
```

[Contents](#)

Appendix 7--ISO Required System Modules

All the modules in this section are pseudo-modules. that is, they behave *as if* they had the definitions given, but are not actually separate library modules because their implementation is generally sufficiently low-level that they need to be part of the compiler itself. Only the contents of SYSTEM may vary to suit the needs of an implementation; all the others are to be supplied *as is* by ISO conforming implementations.

A7.1 SYSTEM

DEFINITION MODULE SYSTEM;

(* Gives access to system programming facilities that are probably non portable. *)

(* The constants and types define underlying properties of storage *)

CONST

BITSPERLOC = <implementation-defined constant<implementation-defined constant<implementation-defined constant<anytype<a packedset type<type of first parameter<a packedset type<type of first parameter<targettype<anytype<targettype<targettype<type<type<implementation-defined>;

PROCEDURE NEWCOROUTINE (procBody: **PROC**; workspace: SYSTEM.ADDRESS; size: **CARDINAL**; **VAR** cr: COROUTINE[; initProtection: **PROTECTION**]);

(* Creates a new coroutine whose body is given by procBody, and returns the identity of the coroutine in cr. workspace is a pointer to the work space allocated to the coroutine; size specifies the size of this workspace in terms of SYSTEM.LOC. initProtection is an optional parameter that specifies the initial protection level of the coroutine. *)

PROCEDURE TRANSFER (**VAR** from: COROUTINE; to: COROUTINE);

(* Returns the identity of the calling coroutine in from, and transfers control to the coroutine specified by to. *)

PROCEDURE IOTRANSFER (**VAR** from: COROUTINE; to: COROUTINE);

(* Returns the identity of the calling coroutine in from and transfers control to the coroutine specified by to. On occurrence of an interrupt, associated with the caller, control is transferred back to the caller, and the identity of the interrupted coroutine is returned in from. The calling coroutine must be associated with a source of interrupts. *)

PROCEDURE ATTACH (source: INTERRUPTSOURCE);

(* Associates the specified source of interrupts with the calling coroutine. *)

PROCEDURE DETACH (source: INTERRUPTSOURCE);

(* Dissociates the specified source of interrupts from the calling coroutine. *)

PROCEDURE IsATTACHED (source: INTERRUPTSOURCE): **BOOLEAN**;

(* Returns TRUE if and only if the specified source of interrupts is

currently associated with a coroutine; otherwise returns FALSE. *)

PROCEDURE HANDLER (source: INTERRUPTSOURCE): COROUTINE;

(* Returns the coroutine, if any, that is associated with the source of interrupts. The result is undefined if IsATTACHED(source) = FALSE. *)

PROCEDURE CURRENT (): COROUTINE;

(* Returns the identity of the calling coroutine. *)

PROCEDURE LISTEN (p: **PROTECTION**);

(* Momentarily changes the protection of the calling coroutine to p. *)

PROCEDURE PROT (): **PROTECTION**;

(* Returns the protection of the calling coroutine. *)

END COROUTINES.

A7.3 EXCEPTIONS

DEFINITION MODULE EXCEPTIONS;

(* Provides facilities for raising user exceptions and for making enquiries concerning the current execution state. *)

TYPE

ExceptionSource; (* values of this type are used within library modules to identify the source of raised exceptions *)

ExceptionNumber = **CARDINAL**;

PROCEDURE AllocateSource (VAR newSource: ExceptionSource);

(* Allocates a unique value of type ExceptionSource *)

PROCEDURE RAISE (source: ExceptionSource; number: ExceptionNumber; message: **ARRAY OF CHAR**);

(* Associates the given values of source, number and message with the current context and raises an exception. *)

PROCEDURE CurrentNumber (source: ExceptionSource): ExceptionNumber;

(* If the current coroutine is in the exceptional execution state because of the raising of an exception from source, returns the corresponding number, and otherwise raises an exception. *)

PROCEDURE GetMessage (VAR text: **ARRAY OF CHAR**);

(* If the current coroutine is in the exceptional execution state, returns the possibly truncated string associated with the current context.

Otherwise, in normal execution state, returns the empty string. *)

PROCEDURE IsCurrentSource (source: ExceptionSource): **BOOLEAN**;

(* If the current coroutine is in the exceptional execution state because of the raising of an exception from source, returns TRUE, and otherwise returns FALSE. *)

PROCEDURE IsExceptionalExecution (): **BOOLEAN**;

(* If the current coroutine is in the exceptional execution state because of the raising of an exception, returns TRUE, and otherwise returns FALSE. *)

END EXCEPTIONS.

A7.4 TERMINATION

DEFINITION MODULE TERMINATION;

(* Provides facilities for enquiries concerning the occurrence of termination events. *)

PROCEDURE IsTerminating (): **BOOLEAN** ;

(* Returns true if any coroutine has started program termination and false otherwise. *)

PROCEDURE HasHalted (): **BOOLEAN** ;

(* Returns true if a call to HALT has been made and false otherwise. *)

END TERMINATION.

A7.5 M2EXCEPTION

DEFINITION MODULE M2EXCEPTION;

(* Provides facilities for identifying language exceptions *)

TYPE

M2Exceptions =
 (indexException, rangeException, caseSelectException,
 invalidLocation, functionException, wholeValueException,
 wholeDivException, realValueException, realDivException,
 complexValueException, complexDivException, protException,
 sysException, coException, exException *)

PROCEDURE M2Exception () : M2Exceptions;

(* If the current coroutine is in the exceptional execution state because of a language exception, returns the corresponding enumeration value, and otherwise raises an exception *)

PROCEDURE IsM2Exception () : **BOOLEAN**;

(* If the current coroutine is in the exceptional execution state because of a language exception, returns TRUE, and otherwise returns FALSE *)

END M2EXCEPTION.

Appendix 8--ISO Utility and Information Modules

A8.1 Characters and Strings

DEFINITION MODULE CharClass;

(* Classification of values of the type **CHAR** *)

PROCEDURE IsNumeric (ch: **CHAR**): **BOOLEAN**;

(* Returns **TRUE** if and only if ch is classified as a numeric character *)

PROCEDURE IsLetter (ch: **CHAR**): **BOOLEAN**;

(* Returns **TRUE** if and only if ch is classified as a letter *)

PROCEDURE IsUpper (ch: **CHAR**): **BOOLEAN**;

(* Returns **TRUE** if and only if ch is classified as an upper case letter *)

PROCEDURE IsLower (ch: **CHAR**): **BOOLEAN**;

(* Returns **TRUE** if and only if ch is classified as a lower case letter *)

PROCEDURE IsControl (ch: **CHAR**): **BOOLEAN**;

(* Returns **TRUE** if and only if ch represents a control function *)

PROCEDURE IsWhiteSpace (ch: **CHAR**): **BOOLEAN**;

(* Returns **TRUE** if and only if ch represents a space character or a format effector *)

END CharClass.

DEFINITION MODULE Strings;

(* Facilities for manipulating strings *)

TYPE

String1 = **ARRAY** [0..0] **OF** **CHAR**;

(* String1 is provided for constructing a value of a single-character string type from a single character value in order to pass **CHAR** values to **ARRAY OF CHAR** parameters. *)

PROCEDURE Length (stringVal: **ARRAY OF CHAR**): **CARDINAL**;

(* Returns the length of stringVal (the same value as would be returned by the pervasive function **LENGTH**). *)

(* The following seven procedures construct a string value, and attempt to assign it to a variable parameter. They all have the property that if the length of the constructed string value exceeds the capacity of the variable parameter, a truncated value is assigned, while if the length of the constructed string value is less than the capacity of the variable parameter, a string terminator is appended before assignment is performed. *)

```

PROCEDURE Assign (source: ARRAY OF CHAR; VAR destination: ARRAY OF CHAR);
    (* Copies source to destination *)

PROCEDURE Extract (source: ARRAY OF CHAR; startIndex, numberToExtract: CARDINAL; VAR
destination: ARRAY OF CHAR);
    (* Copies at most numberToExtract characters from source to destination, starting
at position startIndex in source. *)

PROCEDURE Delete (VAR stringVar: ARRAY OF CHAR; startIndex, numberToDelete:
CARDINAL);
    (* Deletes at most numberToDelete characters from stringVar, starting at position
startIndex. *)

PROCEDURE Insert (source: ARRAY OF CHAR; startIndex: CARDINAL;
VAR destination: ARRAY OF CHAR);
    (* Inserts source into destination at position startIndex *)

PROCEDURE Replace (source: ARRAY OF CHAR; startIndex: CARDINAL;
VAR destination: ARRAY OF CHAR);
    (* Copies source into destination, starting at position startIndex. Copying stops
when all of source has been copied, or when the last character of the string value in
destination has been replaced. *)

PROCEDURE Append (source: ARRAY OF CHAR; VAR destination: ARRAY OF CHAR);
    (* Appends source to destination. *)

PROCEDURE Concat (source1, source2: ARRAY OF CHAR; VAR destination: ARRAY OF CHAR);
    (* Concatenates source2 onto source1 and copies the result into destination. *)

(* The following predicates provide for pre-testing of the operation-completion
conditions for the procedures above. *)

PROCEDURE CanAssignAll (sourceLength: CARDINAL; VAR destination: ARRAY OF CHAR):
BOOLEAN;
    (* Returns TRUE if a number of characters, indicated by sourceLength, will fit into
destination; otherwise returns FALSE. *)

PROCEDURE CanExtractAll (sourceLength, startIndex, numberToExtract: CARDINAL; VAR
destination: ARRAY OF CHAR): BOOLEAN;
    (* Returns TRUE if there are numberToExtract characters starting at startIndex and
within the sourceLength of some string, and if the capacity of destination is
sufficient to hold numberToExtract characters; otherwise returns FALSE. *)

PROCEDURE CanDeleteAll (stringLength, startIndex, numberToDelete: CARDINAL): BOOLEAN;
    (* Returns TRUE if there are numberToDelete characters starting at startIndex and
within the stringLength of some string; otherwise returns FALSE. *)

PROCEDURE CanInsertAll (sourceLength, startIndex: CARDINAL;
VAR destination: ARRAY OF CHAR): BOOLEAN;
    (* Returns TRUE if there is room for the insertion of sourceLength characters from
some string into destination starting at startIndex; otherwise returns FALSE. *)

PROCEDURE CanReplaceAll (sourceLength, startIndex: CARDINAL;
VAR destination: ARRAY OF CHAR): BOOLEAN;

```

```

    (* Returns TRUE if there is room for the replacement of sourceLength characters in
destination starting at startIndex; otherwise returns FALSE.
*)

PROCEDURE CanAppendAll (sourceLength: CARDINAL; VAR destination: ARRAY OF CHAR):
BOOLEAN;
    (* Returns TRUE if there is sufficient room in destination to append a string of
length sourceLength to the string in destination; otherwise returns FALSE. *)

PROCEDURE CanConcatAll (source1Length, source2Length: CARDINAL;
VAR destination: ARRAY OF CHAR): BOOLEAN;
    (* Returns TRUE if there is sufficient room in destination for a two strings of
lengths source1Length and source2Length; otherwise returns FALSE. *)

(* The following type and procedures provide for the comparison of string values, and
for the location of substrings within strings. *)

TYPE
    CompareResults = (less, equal, greater);

PROCEDURE Compare (stringVal1, stringVal2: ARRAY OF CHAR): CompareResults;
    (* Returns less, equal, or greater, according as stringVal1 is lexically less than,
equal to, or greater than stringVal2. *)

PROCEDURE Equal (stringVal1, stringVal2: ARRAY OF CHAR): BOOLEAN;
    (* Returns Strings.Compare(stringVal1, stringVal2) = Strings.equal *)

PROCEDURE FindNext (pattern, stringToSearch: ARRAY OF CHAR; startIndex: CARDINAL; VAR
patternFound: BOOLEAN; VAR posOfPattern: CARDINAL);
    (* Looks forward for next occurrence of pattern in stringToSearch, starting the
search at position startIndex. If startIndex < LENGTH(stringToSearch) and pattern is
found, patternFound is returned as TRUE, and posOfPattern contains the start position
in stringToSearch of pattern. Otherwise patternFound is returned as FALSE, and
posOfPattern is unchanged. *)

PROCEDURE FindPrev (pattern, stringToSearch: ARRAY OF CHAR; startIndex: CARDINAL; VAR
patternFound: BOOLEAN; VAR posOfPattern: CARDINAL);
    (* Looks backward for the previous occurrence of pattern in stringToSearch and
returns the position of the first character of the pattern if found. The search for
the pattern begins at startIndex. If pattern is found, patternFound is returned as
TRUE, and posOfPattern contains the start position in stringToSearch of pattern in
the range [0..startIndex]. Otherwise patternFound is returned as FALSE, and
posOfPattern is unchanged. *)

PROCEDURE FindDiff (stringVal1, stringVal2: ARRAY OF CHAR;
VAR differenceFound: BOOLEAN; VAR posOfDifference: CARDINAL);
    (* Compares the string values in stringVal1 and stringVal2 for differences. If they
are equal, differenceFound is returned as FALSE, and TRUE otherwise. If
differenceFound is TRUE, posOfDifference is set to the position of the first
difference; otherwise posOfDifference is unchanged. *)

PROCEDURE Capitalize (VAR stringVar: ARRAY OF CHAR);
    (* Applies the function CAP to each character of the string value in stringVar. *)

END Strings.

```

A8.2 High Level String Conversion Modules

The modules in this section can be employed to convert strings into the corresponding numeric values. They may (but are not necessarily) be called by routines in WholeIO, RealIO and LongIO to return their values, after (presumed) calls to TextIO.ReadToken.

A8.2.1 WholeStr

DEFINITION MODULE WholeStr;

 (* Whole-number/string conversions *)

IMPORT

 ConvTypes;

TYPE

 ConvResults = ConvTypes.ConvResults; (* strAllRight, strOutOfRange, strWrongFormat, strEmpty *)

 (* the string form of a signed whole number is
 ["+" | "-"], decimal digit, {decimal digit}
 *)

PROCEDURE StrToInt (str: **ARRAY OF CHAR**; **VAR** int: **INTEGER**; **VAR** res: ConvResults);
 (* Ignores any leading spaces in str. If the subsequent characters in str are in
 the format of a signed whole number, assigns a corresponding value to int. Assigns a
 value indicating the format of str to res. *)

PROCEDURE IntToStr (int: **INTEGER**; **VAR** str: **ARRAY OF CHAR**);
 (* Converts the value of int to string form and copies the possibly truncated
 result to str. *)

 (* the string form of an unsigned whole number is
 decimal digit, {decimal digit} *)

PROCEDURE StrToCard (str: **ARRAY OF CHAR**; **VAR** card: **CARDINAL**; **VAR** res:
ConvResults);
 (* Ignores any leading spaces in str. If the subsequent characters in str are in
 the format of an unsigned whole number, assigns a corresponding value to card.
 Assigns a value indicating the format of str to res. *)

PROCEDURE CardToStr (card: **CARDINAL**; **VAR** str: **ARRAY OF CHAR**);
 (* Converts the value of card to string form and copies the possibly truncated
 result to str. *)

END WholeStr.

A8.2.2 RealStr

DEFINITION MODULE RealStr;

 (* **REAL**/string conversions *)

```

IMPORT
    ConvTypes;

TYPE
    ConvResults = ConvTypes.ConvResults; (* strAllRight, strOutOfRange, strWrongFormat,
    strEmpty *)

(* the string form of a signed fixed-point real number is
    ["+" | "-"], decimal digit, {decimal digit}, [".", {decimal digit}] *)

(* the string form of a signed floating-point real number is
    signed fixed-point real number, "E", ["+" | "-"], decimal digit, {decimal digit}
    *)

PROCEDURE StrToReal (str: ARRAY OF CHAR; VAR real: REAL; VAR res: ConvResults);
    (* Ignores any leading spaces in str. If the subsequent characters in str are in
    the format of a signed real number, assigns a corresponding value to real. Assigns a
    value indicating the format of str to res.  *)

PROCEDURE RealToFloat (real: REAL; sigFigs: CARDINAL; VAR str: ARRAY OF CHAR);
    (* Converts the value of real to floating-point string form, with sigFigs
    significant figures, and copies the possibly truncated result to str.  *)

PROCEDURE RealToEng (real: REAL; sigFigs: CARDINAL; VAR str: ARRAY OF CHAR);
    (* Converts the value of real to floating-point string form, with sigFigs
    significant figures, and copies the possibly truncated result to str. The number is
    scaled with one to three digits in the whole number part and with an exponent that is
    a multiple of three.  *)

PROCEDURE RealToFixed (real: REAL; place: INTEGER; VAR str: ARRAY OF CHAR);
    (* Converts the value of real to fixed-point string form, rounded to the given
    place relative to the decimal point, and copies the possibly truncated result to str.
    *)

PROCEDURE RealToStr (real: REAL; VAR str: ARRAY OF CHAR);
    (* Converts the value of real as RealToFixed if the sign and magnitude can be shown
    within the capacity of str, or otherwise as RealToFloat, and copies the possibly
    truncated result to str. The number of places or significant digits are
    implementation-defined. *)

END RealStr.

```

A8.2.3 LongStr

```

DEFINITION MODULE LongStr;

    (* LONGREAL/string conversions *)

IMPORT
    ConvTypes;

TYPE
    ConvResults = ConvTypes.ConvResults; (* strAllRight, strOutOfRange,
    strWrongFormat, strEmpty *)

```



```

(* the string form of a signed fixed-point real number is
   ["+" | "-"], decimal digit, {decimal digit}, [".", {decimal digit}]
*)

(* the string form of a signed floating-point real number is
   signed fixed-point real number, "E", ["+" | "-"], decimal digit, {decimal digit}
*)

PROCEDURE StrToReal (str: ARRAY OF CHAR; VAR real: LONGREAL; VAR res:
ConvResults);
  (* Ignores any leading spaces in str. If the subsequent characters in str are in
the format of a signed real number, assigns a corresponding value to real. Assigns a
value indicating the format of str to res.  *)

PROCEDURE RealToFloat (real: LONGREAL; sigFigs: CARDINAL; VAR str: ARRAY OF
CHAR);
  (* Converts the value of real to floating-point string form, with sigFigs
significant figures, and copies the possibly truncated result to str.  *)

PROCEDURE RealToEng (real: LONGREAL; sigFigs: CARDINAL; VAR str: ARRAY OF CHAR);
  (* Converts the value of real to floating-point string form, with sigFigs
significant figures, and copies the possibly truncated result to str. The number is
scaled with one to three digits in the whole number part and with an exponent that is
a multiple of three. *)

PROCEDURE RealToFixed (real: LONGREAL; place: INTEGER; VAR str: ARRAY OF CHAR);
  (* Converts the value of real to fixed-point string form, rounded to the given
place relative to the decimal point, and copies the possibly truncated result to str.
  *)

PROCEDURE RealToStr (real: LONGREAL; VAR str: ARRAY OF CHAR);
  (* Converts the value of real as RealToFixed if the sign and magnitude can be shown
within the capacity of str, or otherwise as RealToFloat, and copies the possibly
truncated result to str. The number of places or significant digits depend on the
capacity of str. *)

END LongStr.

```

A8.3 Low Level String Conversion Modules

There is no requirement in the standard that the high level conversion modules above depend on the low level ones presented below. However, they are available for those who wish to write their own string conversion routines.

A8.3.1 ConvTypes

```

DEFINITION MODULE ConvTypes;

```

```

  (* Common types used in the string conversion modules *)

```

```

TYPE

```

```

  ConvResults =      (* Values of this type are used to express the format of a string
  *)

```

```

  (strAllRight,      (* the string format is correct for the corresponding conversion
  *)

```



```

    strOutOfRange, (* the string is well-formed but the value cannot be represented
*)
    strWrongFormat, (* the string is in the wrong format for the conversion *)
    strEmpty        (* the given string is empty *) );

ScanClass = (* Values of this type are used to classify input to finite state
scanners *)
(padding,      (* a leading or padding character at this point in the scan - ignore it
*)
    valid,      (* a valid character at this point in the scan - accept it *)
    invalid,    (* an invalid character at this point in the scan - reject it *)
    terminator (* a terminating character at this point in the scan (not part of
token) *) );

ScanState = (* The type of lexical scanning control procedures *)
PROCEDURE (CHAR, VAR ScanClass, VAR ScanState);

END ConvTypes.

```

A8.3.2 WholeConv

DEFINITION MODULE WholeConv;

(* Low-level whole-number/string conversions *)

IMPORT

ConvTypes;

TYPE

ConvResults = ConvTypes.ConvResults; (* strAllRight, strOutOfRange,
strWrongFormat, strEmpty *)

PROCEDURE ScanInt (inputCh: **CHAR**; VAR chClass: ConvTypes.ScanClass;
VAR nextState: ConvTypes.ScanState);

(* Represents the start state of a finite state scanner for signed whole numbers -
assigns class of inputCh to chClass and a procedure representing the next state to
nextState. *)

PROCEDURE FormatInt (str: **ARRAY OF CHAR**): ConvResults;

(* Returns the format of the string value for conversion to **INTEGER**. *)

PROCEDURE ValueInt (str: **ARRAY OF CHAR**): **INTEGER**;

(* Returns the value corresponding to the signed whole number string value str if str
is well-formed; otherwise raises the WholeConv exception. *)

PROCEDURE LengthInt (int: **INTEGER**): **CARDINAL**;

(* Returns the number of characters in the string representation of int. *)

PROCEDURE ScanCard (inputCh: **CHAR**; VAR chClass: ConvTypes.ScanClass;
VAR nextState: ConvTypes.ScanState);

(* Represents the start state of a finite state scanner for unsigned whole numbers -
assigns class of inputCh to chClass and a procedure representing the next state to
nextState. *)

PROCEDURE FormatCard (str: **ARRAY OF CHAR**): ConvResults;

(* Returns the format of the string value for conversion to **CARDINAL**. *)

PROCEDURE ValueCard (str: **ARRAY OF CHAR**): **CARDINAL**;

(* Returns the value corresponding to the unsigned whole number string value str if str is well-formed; otherwise raises the WholeConv exception *)

PROCEDURE LengthCard (card: **CARDINAL**): **CARDINAL**;

(* Returns the number of characters in the string representation of card. *)

PROCEDURE IsWholeConvException (): **BOOLEAN**;

(* Returns **TRUE** if the current coroutine is in the exceptional execution state because of the raising of an exception in a routine from this module; otherwise returns **FALSE**. *)

END WholeConv.

A8.3.3 RealConv

DEFINITION MODULE RealConv;

(* Low-level **REAL**/string conversions *)

IMPORT

ConvTypes;

TYPE

ConvResults = ConvTypes.ConvResults; (* strAllRight, strOutOfRange, strWrongFormat, strEmpty *)

PROCEDURE ScanReal (inputCh: **CHAR**; **VAR** chClass: ConvTypes.ScanClass;
 VAR nextState: ConvTypes.ScanState);

(* Represents the start state of a finite state scanner for real numbers - assigns class of inputCh to chClass and a procedure representing the next state to nextState. *)

PROCEDURE FormatReal (str: **ARRAY OF CHAR**): ConvResults;

(* Returns the format of the string value for conversion to **REAL**. *)

PROCEDURE ValueReal (str: **ARRAY OF CHAR**): **REAL**;

(* Returns the value corresponding to the real number string value str if str is well-formed; otherwise raises the RealConv exception. *)

PROCEDURE LengthFloatReal (real: **REAL**; sigFigs: **CARDINAL**): **CARDINAL**;

(* Returns the number of characters in the floating-point string representation of real with sigFigs significant figures. *)

PROCEDURE LengthEngReal (real: **REAL**; sigFigs: **CARDINAL**): **CARDINAL**;

(* Returns the number of characters in the floating-point engineering string representation of real with sigFigs significant figures. *)

PROCEDURE LengthFixedReal (real: **REAL**; place: **INTEGER**): **CARDINAL**;

(* Returns the number of characters in the fixed-point string representation of real rounded to the given place relative to the decimal point. *)

PROCEDURE IsRConvException (): **BOOLEAN**;

```
(* Returns TRUE if the current coroutine is in the exceptional execution state
because of the raising of an exception in a routine from this module; otherwise
returns FALSE. *)
```

```
END RealConv.
```

A8.3.4 LongConv

```
DEFINITION MODULE LongConv;
```

```
(* Low-level LONGREAL/string conversions *)
```

```
IMPORT
```

```
ConvTypes;
```

```
TYPE
```

```
ConvResults = ConvTypes.ConvResults; (* strAllRight, strOutOfRange,
strWrongFormat, strEmpty *)
```

```
PROCEDURE ScanReal (inputCh: CHAR; VAR chClass: ConvTypes.ScanClass;
VAR nextState: ConvTypes.ScanState);
```

```
(* Represents the start state of a finite state scanner for real numbers -
assigns class of inputCh to chClass and a procedure representing the next state to
nextState. *)
```

```
PROCEDURE FormatReal (str: ARRAY OF CHAR): ConvResults;
```

```
(* Returns the format of the string value for conversion to LONGREAL. *)
```

```
PROCEDURE ValueReal (str: ARRAY OF CHAR): LONGREAL;
```

```
(* Returns the value corresponding to the real number string value str if str is
well-formed; otherwise raises the LongConv exception. *)
```

```
PROCEDURE LengthFloatReal (real: LONGREAL; sigFigs: CARDINAL): CARDINAL;
```

```
(* Returns the number of characters in the floating-point string
representation of real with sigFigs significant figures. *)
```

```
PROCEDURE LengthEngReal (real: LONGREAL; sigFigs: CARDINAL): CARDINAL;
```

```
(* Returns the number of characters in the floating-point engineering string
representation of real with sigFigs significant figures. *)
```

```
PROCEDURE LengthFixedReal (real: LONGREAL; place: INTEGER): CARDINAL;
```

```
(* Returns the number of characters in the fixed-point string representation of
real rounded to the given place relative to the decimal point. *)
```

```
PROCEDURE IsRConvException (): BOOLEAN;
```

```
(* Returns TRUE if the current coroutine is in the exceptional execution state
because of the raising of an exception in a routine from this module; otherwise
returns FALSE. *)
```

```
END LongConv.
```

A8.4 SysClock--The Date and Time

```
DEFINITION MODULE SysClock;
```

```

(* =====
Original specification and design of SysClock
Copyright © 1990-1991 by R. Sutcliffe
Assigned to the BSI for standards work
===== *)

(* Facilities for accessing a system clock that records the date and time of day *)

CONST
    maxSecondParts = <implementation-defined integral value>;

TYPE
    Month      = [1 .. 12];
    Day        = [1 .. 31];
    Hour       = [0 .. 23];
    Min        = [0 .. 59];
    Sec        = [0 .. 59];
    Fraction   = [0 .. maxSecondParts];
    UTCDiff    = [-780 .. 720];
    DateTime   =
        RECORD
            year:      CARDINAL;
            month:     Month;
            day:       Day;
            hour:      Hour;
            minute:    Min;
            second:    Sec;
            fractions: Fraction;    (* parts of a second *)
            zone:      UTCDiff;    (* Time zone differential factor which is the number
of minutes to add to local time to obtain UTC. *)
            summerTimeFlag: BOOLEAN; (* Interpretation of flag depends on local usage. *)
        END;

PROCEDURE CanGetClock (): BOOLEAN;
    (* Returns TRUE if a system clock can be read; FALSE otherwise *)

PROCEDURE CanSetClock (): BOOLEAN;
    (* Returns TRUE if a system clock can be set; FALSE otherwise *)

PROCEDURE IsValidDateTime (userData: DateTime): BOOLEAN;
    (* Returns TRUE if the value of userData represents a valid date and time; FALSE
otherwise *)

PROCEDURE GetClock (VAR userData: DateTime);
    (* If possible, assigns system date and time of day to userData *)

PROCEDURE SetClock (userData: DateTime);
    (* If possible, sets the system clock to the values of userData *)

END SysClock.

```

Contents

Appendix 9--ISO Mathematics Library Modules

A9.1 RealMath

DEFINITION MODULE RealMath;

 (* Mathematical functions for the type REAL *)

CONST

 pi = 3.1415926535897932384626433832795028841972;

 exp1 = 2.7182818284590452353602874713526624977572;

PROCEDURE sqrt (x: **REAL**): **REAL**;

 (* Returns the positive square root of x *)

PROCEDURE exp (x: **REAL**): **REAL**;

 (* Returns the exponential of x *)

PROCEDURE ln (x: **REAL**): **REAL**;

 (* Returns the natural logarithm of x *)

 (* The angle in all trigonometric functions is measured in radians *)

PROCEDURE sin (x: **REAL**): **REAL**;

 (* Returns the sine of x *)

PROCEDURE cos (x: **REAL**): **REAL**;

 (* Returns the cosine of x *)

PROCEDURE tan (x: **REAL**): **REAL**;

 (* Returns the tangent of x *)

PROCEDURE arcsin (x: **REAL**): **REAL**;

 (* Returns the arcsine of x *)

PROCEDURE arccos (x: **REAL**): **REAL**;

 (* Returns the arccosine of x *)

PROCEDURE arctan (x: **REAL**): **REAL**;

 (* Returns the arctangent of x *)

PROCEDURE power (base, exponent: **REAL**): **REAL**;

 (* Returns the value of the number base raised to the power exponent *)

PROCEDURE round (x: **REAL**): **INTEGER**;

 (* Returns the value of x rounded to the nearest integer *)

PROCEDURE IsRMathException (): **BOOLEAN**;

```
(* Returns TRUE if the current coroutine is in the exceptional execution state
because of the raising of an exception in a routine from this module; otherwise
returns FALSE. *)
```

```
END RealMath.
```

A9.2 LongMath

```
DEFINITION MODULE LongMath;
```

```
(* Mathematical functions for the type LONGREAL *)
```

```
CONST
```

```
pi    = 3.1415926535897932384626433832795028841972;
```

```
exp1  = 2.7182818284590452353602874713526624977572;
```

```
PROCEDURE sqrt (x: LONGREAL): LONGREAL;
```

```
(* Returns the positive square root of x *)
```

```
PROCEDURE exp (x: LONGREAL): LONGREAL;
```

```
(* Returns the exponential of x *)
```

```
PROCEDURE ln (x: LONGREAL): LONGREAL;
```

```
(* Returns the natural logarithm of x *)
```

```
(* The angle in all trigonometric functions is measured in radians *)
```

```
PROCEDURE sin (x: LONGREAL): LONGREAL;
```

```
(* Returns the sine of x *)
```

```
PROCEDURE cos (x: LONGREAL): LONGREAL;
```

```
(* Returns the cosine of x *)
```

```
PROCEDURE tan (x: LONGREAL): LONGREAL;
```

```
(* Returns the tangent of x *)
```

```
PROCEDURE arcsin (x: LONGREAL): LONGREAL;
```

```
(* Returns the arcsine of x *)
```

```
PROCEDURE arccos (x: LONGREAL): LONGREAL;
```

```
(* Returns the arccosine of x *)
```

```
PROCEDURE arctan (x: LONGREAL): LONGREAL;
```

```
(* Returns the arctangent of x *)
```

```
PROCEDURE power (base, exponent: LONGREAL): LONGREAL;
```

```
(* Returns the value of the number base raised to the power exponent *)
```

```
PROCEDURE round (x: LONGREAL): INTEGER;
```

```
(* Returns the value of x rounded to the nearest integer *)
```

```
PROCEDURE IsRMathException (): BOOLEAN;
```

```
(* Returns TRUE if the current coroutine is in the exceptional execution state
because of the raising of an exception in a routine from this module; otherwise
returns FALSE. *)
```

```
END LongMath.
```

A9.3 ComplexMath

```
DEFINITION MODULE ComplexMath;
```

```
(* =====
Original COMPLEX specification and
      design of ComplexMath
Copyright © 1990-1991 by R. Sutcliffe
Assigned to the BSI for standards work
=====*)
```

```
(* Mathematical functions for the type COMPLEX *)
```

```
CONST
```

```
i =    CMPLX (0.0, 1.0);
one =  CMPLX (1.0, 0.0);
zero = CMPLX (0.0, 0.0);
```

```
PROCEDURE abs (z: COMPLEX): REAL;
```

```
(* Returns the length of z *)
```

```
PROCEDURE arg (z: COMPLEX): REAL;
```

```
(* Returns the angle that z subtends to the positive real axis *)
```

```
PROCEDURE conj (z: COMPLEX): COMPLEX;
```

```
(* Returns the complex conjugate of z *)
```

```
PROCEDURE power (base: COMPLEX; exponent: REAL): COMPLEX;
```

```
(* Returns the value of the number base raised to the power exponent *)
```

```
PROCEDURE sqrt (z: COMPLEX): COMPLEX;
```

```
(* Returns the principal square root of z *)
```

```
PROCEDURE exp (z: COMPLEX): COMPLEX;
```

```
(* Returns the complex exponential of z *)
```

```
PROCEDURE ln (z: COMPLEX): COMPLEX;
```

```
(* Returns the principal value of the natural logarithm of z *)
```

```
PROCEDURE sin (z: COMPLEX): COMPLEX;
```

```
(* Returns the sine of z *)
```

```
PROCEDURE cos (z: COMPLEX): COMPLEX;
```

```
(* Returns the cosine of z *)
```



```

PROCEDURE tan (z: COMPLEX): COMPLEX;
    (* Returns the tangent of z *)

PROCEDURE arcsin (z: COMPLEX): COMPLEX;
    (* Returns the arcsine of z *)

PROCEDURE arccos (z: COMPLEX): COMPLEX;
    (* Returns the arccosine of z *)

PROCEDURE arctan (z: COMPLEX): COMPLEX;
    (* Returns the arctangent of z *)

PROCEDURE polarToComplex (abs, arg: REAL): COMPLEX;
    (* Returns the complex number with the specified polar coordinates *)

PROCEDURE scalarMult (scalar: REAL; z: COMPLEX): COMPLEX;
    (* Returns the scalar product of scalar with z *)

PROCEDURE IsCMathException (): BOOLEAN;
    (* Returns TRUE if the current coroutine is in the exceptional execution state
    because of the raising of an exception in a routine from this module; otherwise
    returns FALSE. *)

END ComplexMath.

```

A9.4 LongComplexMath

```

DEFINITION MODULE LongComplexMath;

(* =====
   Original COMPLEX specification and
       design of ComplexMath
   Copyright © 1990-1991 by R. Sutcliffe
   Assigned to the BSI for standards work
   =====*)

    (* Mathematical functions for the type LONGCOMPLEX *)

CONST
    i =      CMPLX (0.0, 1.0);
    one =    CMPLX (1.0, 0.0);
    zero =   CMPLX (0.0, 0.0);

PROCEDURE abs (z: LONGCOMPLEX): LONGREAL;
    (* Returns the length of z *)

PROCEDURE arg (z: LONGCOMPLEX): LONGREAL;
    (* Returns the angle that z subtends to the positive real axis *)

PROCEDURE conj (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the complex conjugate of z *)

```

```

PROCEDURE power (base: LONGCOMPLEX; exponent: LONGREAL): LONGCOMPLEX;
    (* Returns the value of the number base raised to the power exponent *)

PROCEDURE sqrt (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the principal square root of z *)

PROCEDURE exp (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the complex exponential of z *)

PROCEDURE ln (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the principal value of the natural logarithm of z *)

PROCEDURE sin (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the sine of z *)

PROCEDURE cos (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the cosine of z *)

PROCEDURE tan (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the tangent of z *)

PROCEDURE arcsin (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the arcsine of z *)

PROCEDURE arccos (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the arccosine of z *)

PROCEDURE arctan (z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the arctangent of z *)

PROCEDURE polarToComplex (abs, arg: LONGREAL): LONGCOMPLEX;
    (* Returns the complex number with the specified polar coordinates *)

PROCEDURE scalarMult (scalar: LONGREAL; z: LONGCOMPLEX): LONGCOMPLEX;
    (* Returns the scalar product of scalar with z *)

PROCEDURE IsCMathException (): BOOLEAN;
    (* Returns TRUE if the current coroutine is in the exceptional execution state
    because of the raising of an exception in a routine from this module; otherwise
    returns FALSE. *)

END LongComplexMath.

```

Appendix 10--ISO Process Support

A10.1 Processes

DEFINITION **MODULE** Processes;

(* This module allows concurrent algorithms to be expressed using processes. A process is a unit of a program that has the potential to run in parallel with other processes. *)

IMPORT SYSTEM;

TYPE

ProcessId; (* Used to identify processes *)
Parameter = SYSTEM.ADDRESS; (* Used to pass data between processes *)
Body = **PROC**; (* Used as the type of a process body *)
Urgency = **INTEGER**; (* Used by the internal scheduler *)
Sources = **CARDINAL**; (* Used to identify event sources *)
ProcessesExceptions = (* Exceptions raised by this module *)
(passiveProgram, processError);

(* The following procedures create processes and switch control between them. *)

PROCEDURE Create (procBody: Body; extraSpace: **CARDINAL**; procUrg: Urgency; procParams: Parameter; **VAR** procId: ProcessId);

(* Creates a new process with procBody as its body, and with urgency and parameters given by procUrg and procParams. At least as much workspace (in units of SYSTEM.LOC) as is specified by extraSpace is allocated to the process.

An identity for the new process is returned in procId.

The process is created in the passive state; it will not run until activated. *)

PROCEDURE Start (procBody: Body; extraSpace: **CARDINAL**; procUrg: Urgency; procParams: Parameter; **VAR** procId: ProcessId);

(* Creates a new process, with parameters as for Create.

The process is created in the ready state; it is eligible to run immediately. *)

PROCEDURE StopMe ();

(* Terminates the calling process.

The process must not be associated with a source of events. *)

PROCEDURE SuspendMe ();

(* Causes the calling process to enter the passive state. The procedure only returns when the calling process is again activated by another process. *)

PROCEDURE Activate (procId: ProcessId);

(* Causes the process identified by procId to enter the ready state, and thus to become eligible to run again. *)

```

PROCEDURE SuspendMeAndActivate (procId: ProcessId);
    (* Executes an atomic sequence of SuspendMe() and Activate(procId). *)

PROCEDURE Switch (procId: ProcessId; VAR info: Parameter);
    (* Causes the calling process to enter the passive state; the process
    identified by procId becomes the currently executing process. Info is used to pass
    parameter information from the calling to the activated process.
    On return, info will contain information from the process that chooses to switch
    back to this one (or will be NIL if Activate or SuspendMeAndActivate are used instead
    of Switch). *)

PROCEDURE Wait ();
    (* Causes the calling process to enter the waiting state. The procedure will
    return when the calling process is activated by another process, or when one of its
    associated eventSources has generated an event. *)

(* The following procedures allow the association of processes with
sources of external events. *)

PROCEDURE Attach (eventSource: Sources);
    (* Associates the specified eventSource with the calling process. *)

PROCEDURE Detach (eventSource: Sources);
    (* Dissociates the specified eventSource from the program. *)

PROCEDURE IsAttached (eventSource: Sources): BOOLEAN;
    (* Returns TRUE if and only if the specified eventSource is currently
    associated with one of the processes of the program. *)

PROCEDURE Handler (eventSource: Sources): ProcessId;
    (* Returns the identity of the process, if any, that is associated with the
    specified eventSource. *)

(* The following procedures allow processes to obtain their identity,
parameters, and urgency. *)

PROCEDURE Me (): ProcessId;
    (* Returns the identity of the calling process (as assigned when the process was
    first created). *)

PROCEDURE MyParam (): Parameter;
    (* Returns the value specified as procParams when the calling process was created.
    *)

PROCEDURE UrgencyOf (procId: ProcessId): Urgency;
    (* Returns the urgency established when the process identified by procId was first
    created. *)

(* The following procedure provides facilities for exception handlers. *)

PROCEDURE ProcessesException (): ProcessesExceptions;
    (* If the current coroutine is in the exceptional execution state because of the
    raising of a language exception, returns the corresponding enumeration value, and
    otherwise raises an exception. *)

```

```

PROCEDURE IsProcessesException (): BOOLEAN;
    (* Returns TRUE if the current coroutine is in the exceptional execution state
because of the raising of an exception in a routine from this module; otherwise
returns FALSE. *)

END Processes.

```

A10.2 Semaphores

```

DEFINITION MODULE Semaphores;

    (* Provides mutual exclusion facilities for use by processes. *)

TYPE
    SEMAPHORE;

PROCEDURE Create (VAR s: SEMAPHORE; initialCount: CARDINAL );
    (* Creates and returns s as the identity of a new semaphore that has its associated
count initialized to initialCount, and has no processes yet waiting on it. *)

PROCEDURE Destroy (VAR s: SEMAPHORE);
    (* Recovers the resources used to implement the semaphore s, provided that no
process is waiting for s to become free. *)

PROCEDURE Claim (s: SEMAPHORE);
    (* If the count associated with the semaphore s is non-zero, decrements this count
and allows the calling process to continue; otherwise suspends the calling process
until s is released. *)

PROCEDURE Release (s: SEMAPHORE);
    (* If there are any processes waiting on the semaphore s, allows one of them to
enter the ready state; otherwise increments the count associated with s. *)

PROCEDURE CondClaim (s: SEMAPHORE): BOOLEAN;
    (* Returns TRUE if the call Claim(s) would cause the calling process to be
suspended; in this case the count associated with s is not changed. Otherwise returns
TRUE and the associated count is decremented. *)

END Semaphores.

```

[Contents](#)

Appendix 11--Modula-2 and Pascal

This information is included for the benefit of those who are coming to the Modula-2 language from Pascal. Only the differences between the two languages are shown. Since there are many versions of Pascal, the information here may not be correct for a given version.

A11.1 Statement Syntax Differences

:
:

Modula-2	Pascal	
FOR i := S TO N [BY M]	for i := S [down]to N do	
DO statement sequence END;	<compound<&	and
Comment delimiters	(* *)	(* *) or { }
Quotation marks	' or "	' only
Character constants	literal or <octal& handled	n/a

[Contents](#)

Appendix 12--Generic Modula-2 Syntax

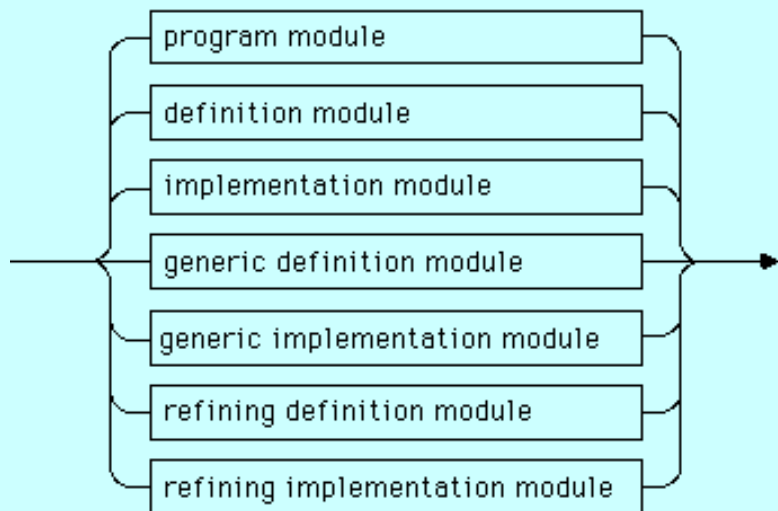
The term Generic Modula-2 refers to a set of extensions to the base language that allow for programmers to create and refine generic separate modules or *templates*.

A12.1 Keywords

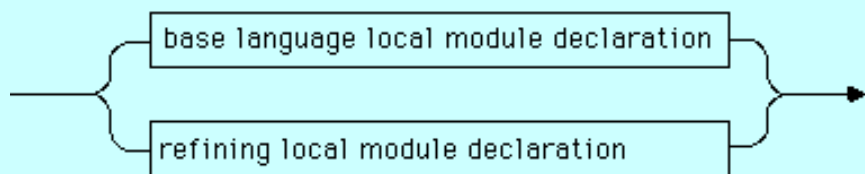
The word GENERIC is added to the list of keywords for the base language.

A12.2 Diagrams of Changes to Base Language Syntax

compilation module (see [A.2.2.1.1](#))



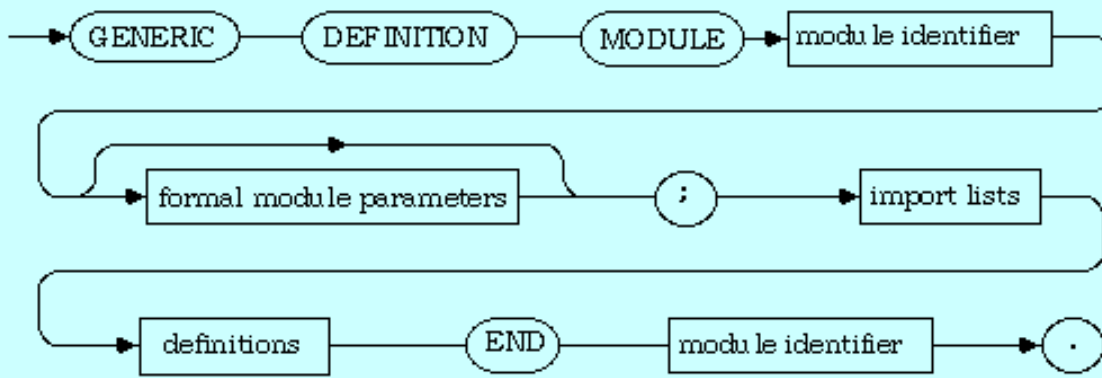
local module declaration (see [A.2.2.2.9](#))



A12.3 Generic Modula-2 Syntax Diagrams

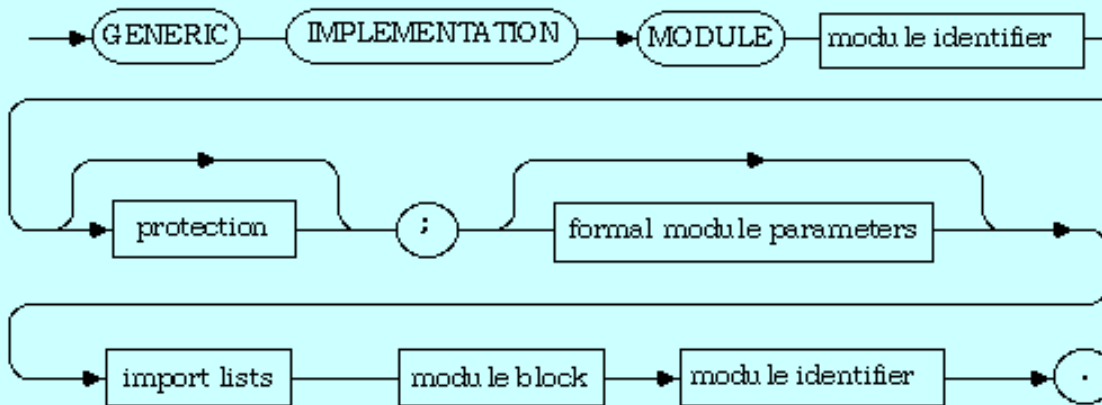
A12.3.1 Generic Definition Module

generic definition module



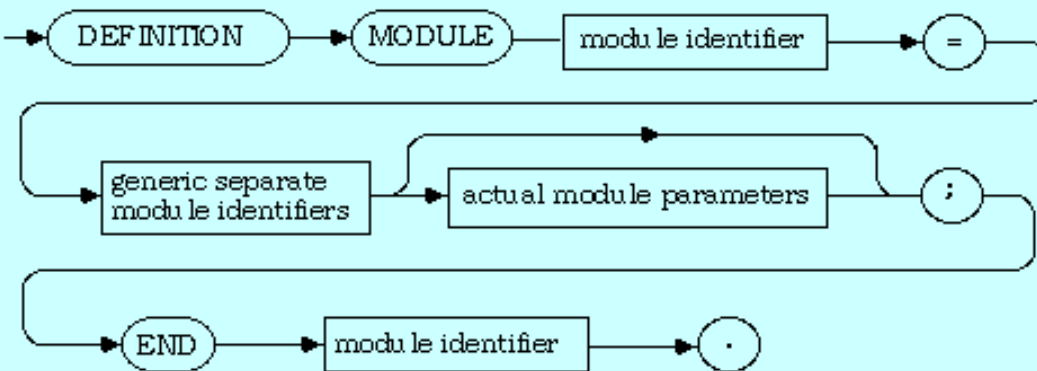
A12.3.2 Generic Implementation Module

generic implementation module



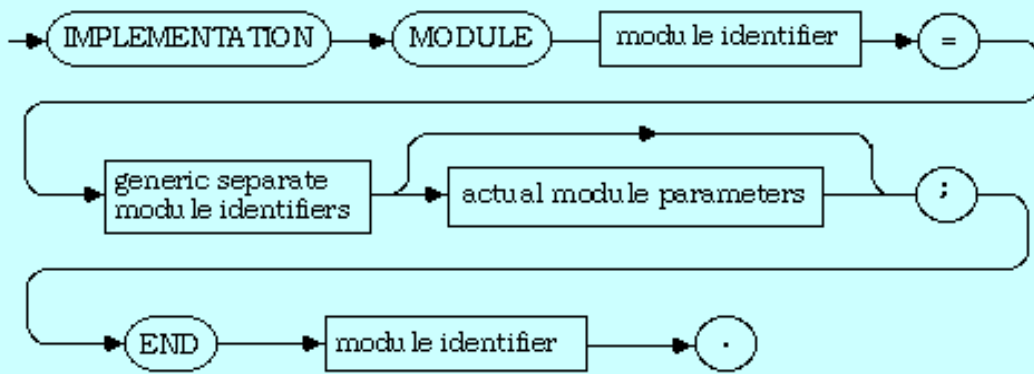
A12.3.3 Refining Definition Module

refining definition module



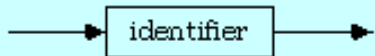
A12.3.4 Refining Implementation Module

refining implementation module

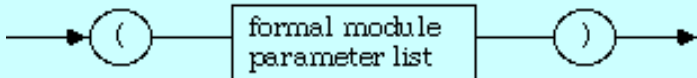


A12.3.5 Identifiers and Formal Parameters

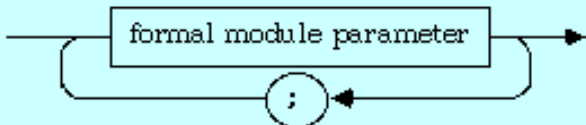
generic separate module identifier



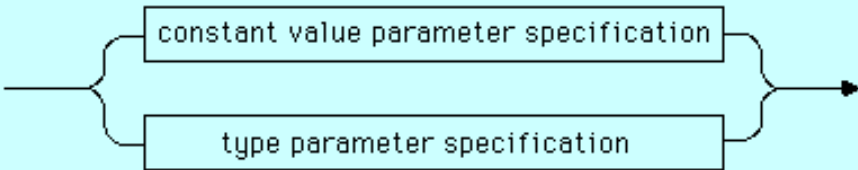
formal module parameters



formal module parameter list



formal module parameter



constant value parameter specification

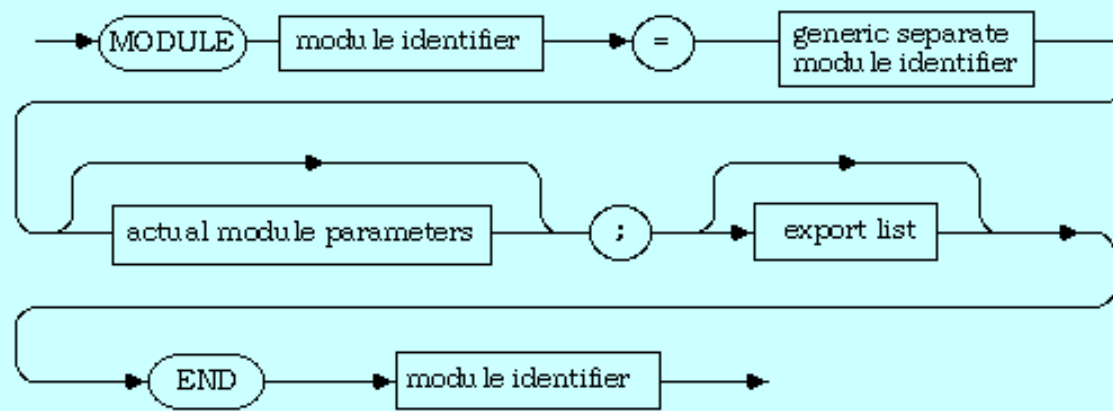


type parameter specification



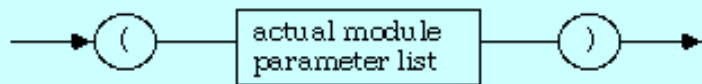
A12.3.6 Refining Local Module Declaration

refining local module declaration (see [A.2.2.2.9](#))

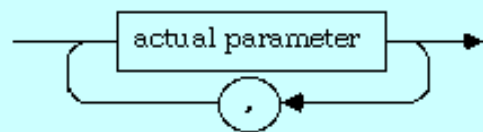


A12.3.7 Actual Parameters

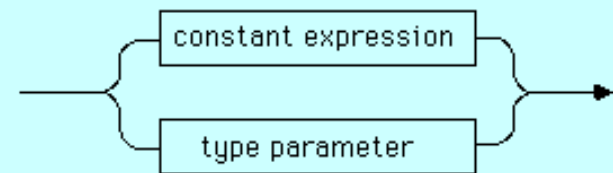
actual module parameters



formal module parameter list



actual parameter



A12.4 Changes To the Syntax of the Base Language in EBNF

keyword = base language keyword | "GENERIC" ;

compilation module = program module | definition module |
implementation module | generic definition module | generic
implementation module | refining definition module | refining
implementation module ;

local module declaration = Base Language local module
declaration | refining local module declaration ;

A12.5 The Syntax of Generic Modula-2 in EBNF

The concrete syntax in this section is taken from ISO/IEC IS 10514-2, the international standard for the generic extensions to Modula-2.

generic definition module =
"GENERIC", "DEFINITION", "MODULE", module identifier, [formal module parameters],
semicolon,
import lists, definitions,

```

    "END", module identifier, period ;
generic implementation module =
    "GENERIC", "IMPLEMENTATION", "MODULE", module identifier, [interrupt protection],
[formal module parameters], semicolon,
    import lists, module block,
    module identifier, period ;
refining definition module =
    "DEFINITION", "MODULE", module identifier, equals, generic separate module
identifier, [actual module parameters], semicolon,
    "END", module identifier, period ;
refining implementation module =
    "IMPLEMENTATION", "MODULE", module identifier, equals, generic separate module
identifier, [actual module parameters], semicolon,
    "END", module identifier, period ;
generic separate module identifier =
    identifier ;
formal module parameters =
    left parenthesis, formal module parameter list, right parenthesis ;
formal module parameter list =
    formal module parameter, {semicolon, formal module parameter} ;
formal module parameter =
    constant value parameter specification | type parameter specification

constant value parameter specification =
    identifier list, colon, formal type;
type parameter specification =
    identifier list, colon, "TYPE" ;
refining local module declaration =
    "MODULE", module identifier, equals, generic separate module identifier, [actual
module parameters], semicolon,
    [export list],
    "END", module identifier ;
actual module parameters =
    left parenthesis, actual module parameter list, right parenthesis ;
actual parameter list =
    actual parameter, {comma, actual parameter} ;
actual parameter =
    constant expression | type parameter ;

```

Appendix 13--Object Oriented Modula-2 Syntax

The term Object Oriented Modula-2 refers to a set of extensions to the base language that allow for programmers to create classes and instantiate objects of those classes. Material in this section is adapted from ISO/IEC 10514-3, the official standard for the object extensions to Modula-2.

A13.1 Keywords and Pervasive Identifiers

The following are added to the list of keywords for the base language.

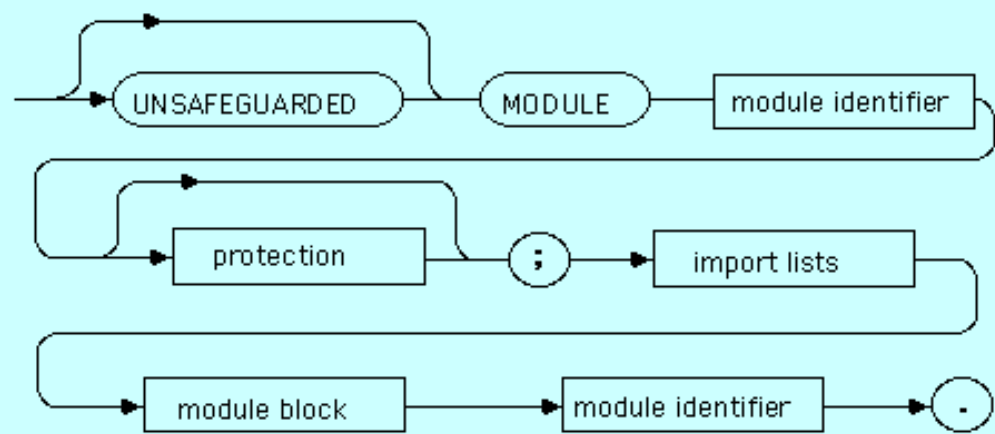
AS	ABSTRACT	CLASS	GUARD	INHERIT
OVERRIDE	READONLY	REVEAL	TRACED	UNSAFEGUARDED

The following are added to the list of pervasive identifiers for the base language.

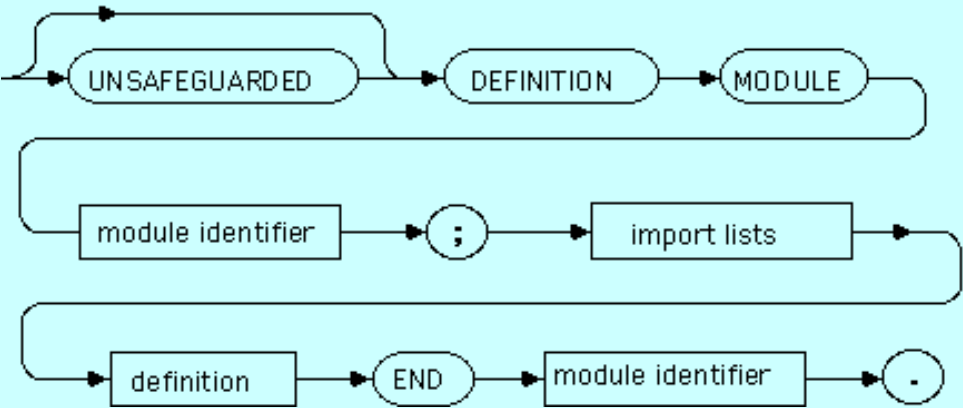
CREATE	DESTROY	EMPTY	ISMEMBER	SELF
--------	---------	-------	----------	------

A13.2 Diagrams of Changes to Base Language Syntax

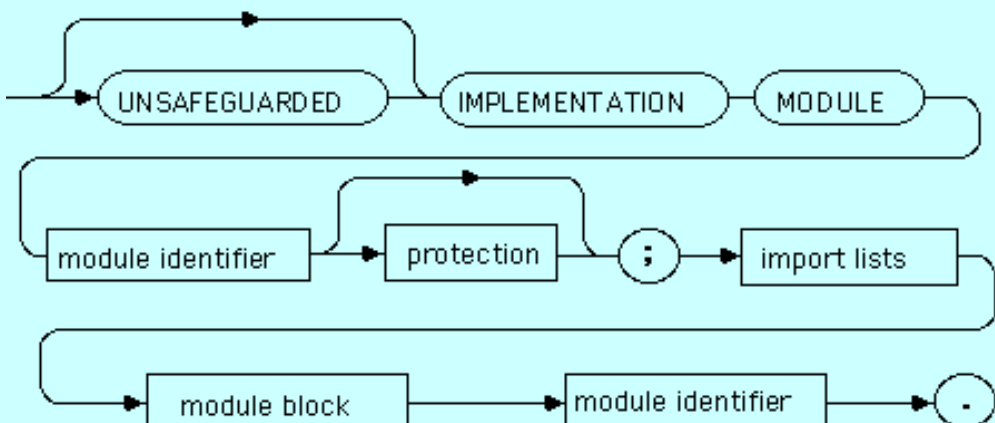
program module (see [A.2.2.1.2](#))



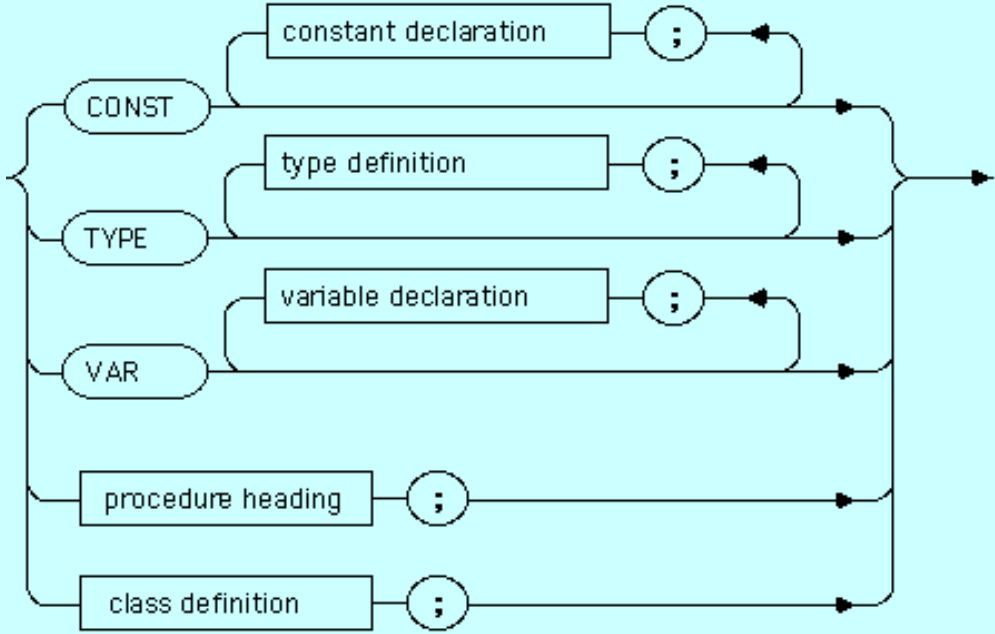
definition module (see [A.2.2.1.3](#))



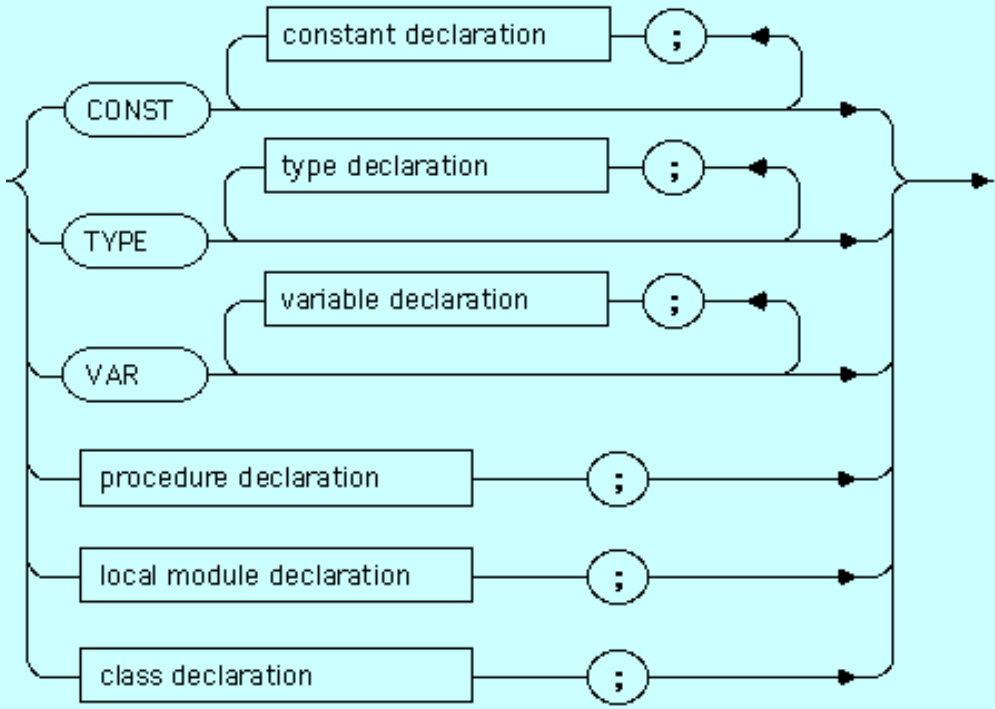
implementation module (see [A.2.2.1.4](#))



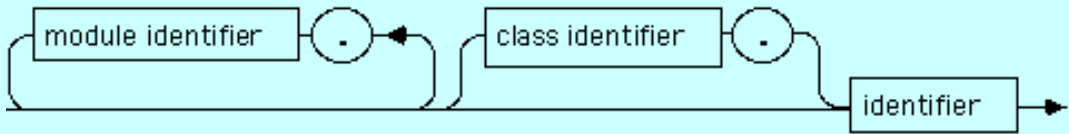
definition (see [A.2.2.2.2](#))



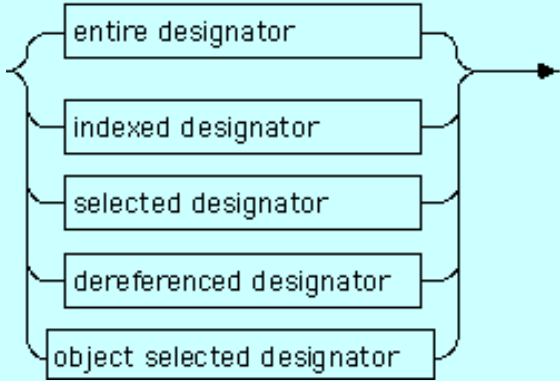
declaration (see [A.2.2.2.3](#))



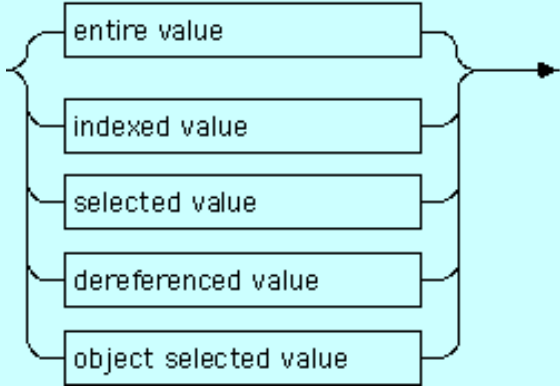
qualified identifier (see [A.2.2.2.1](#))



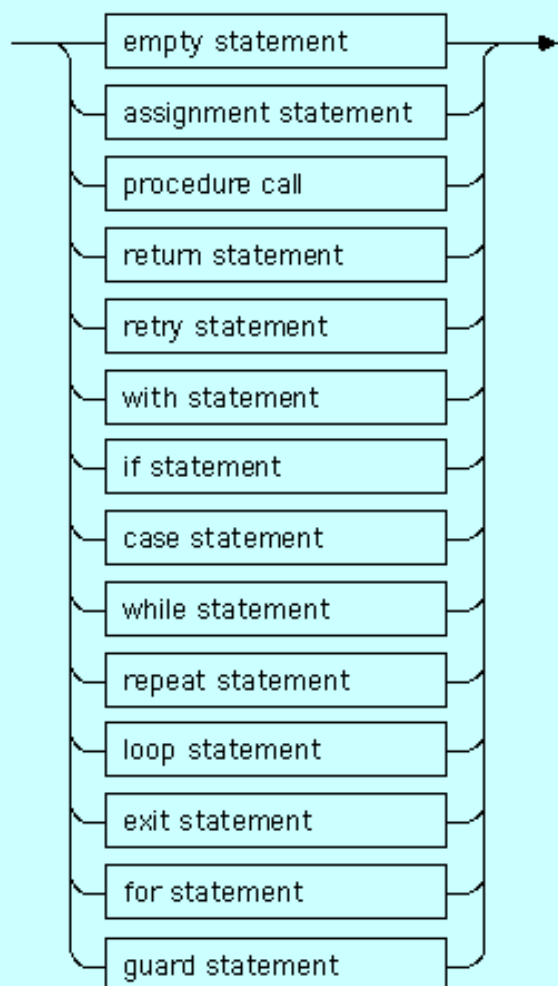
variable designator (see [A.2.2.6](#))



value designator (see [A.2.2.7.2](#))



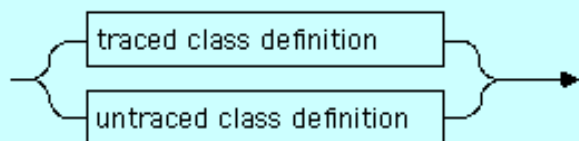
statement (see [A.2.2.5](#))



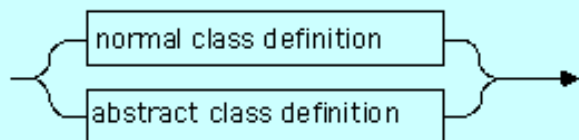
A13.3 Object Oriented Modula-2 Syntax Diagrams

A13.3.1 Class Definition

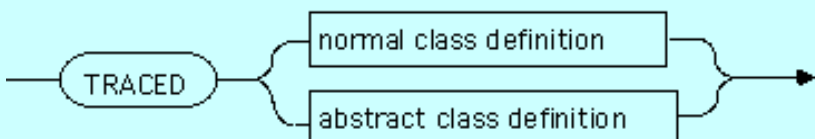
class definition



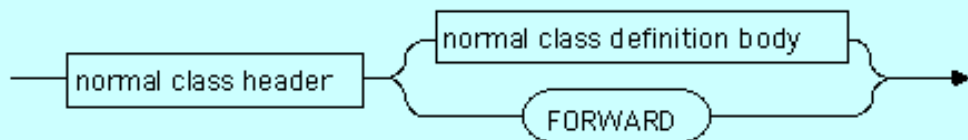
untraced class definition



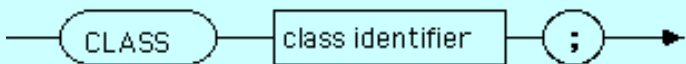
traced class definition



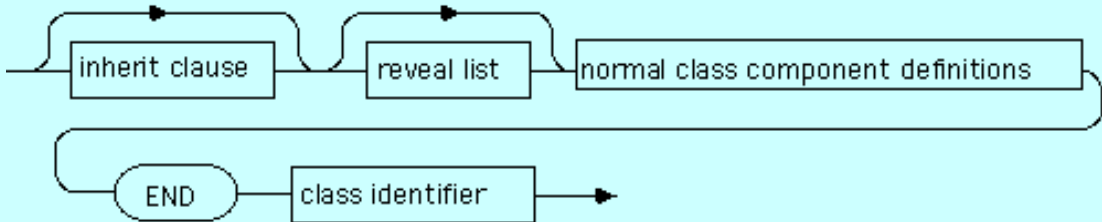
normal class definition



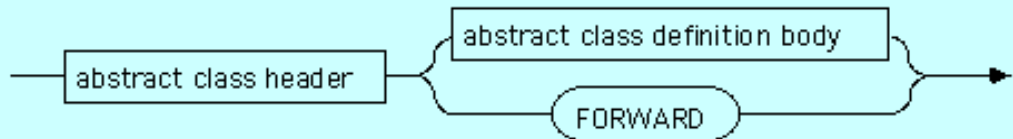
normal class header



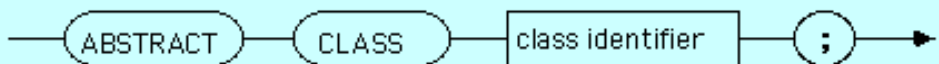
normal class definition body



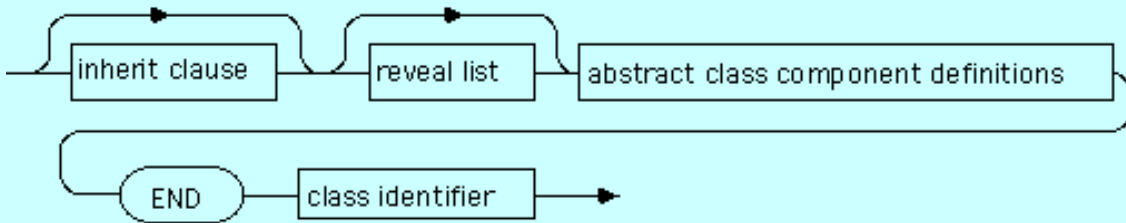
abstract class definition



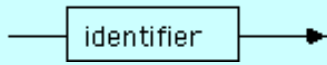
abstract class header



abstract class definition body



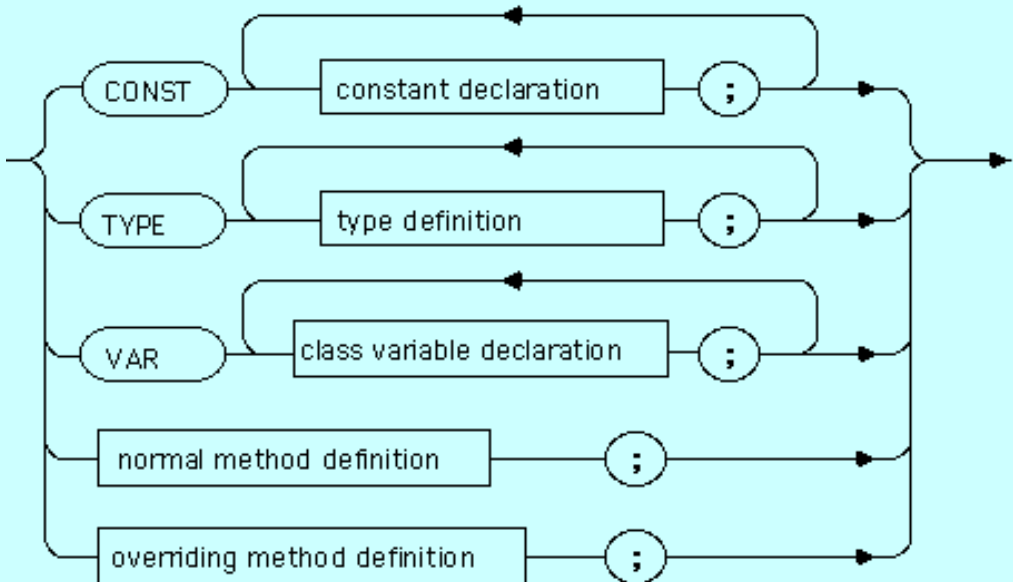
class identifier



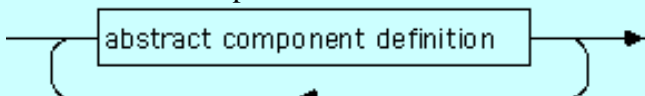
normal class component definitions



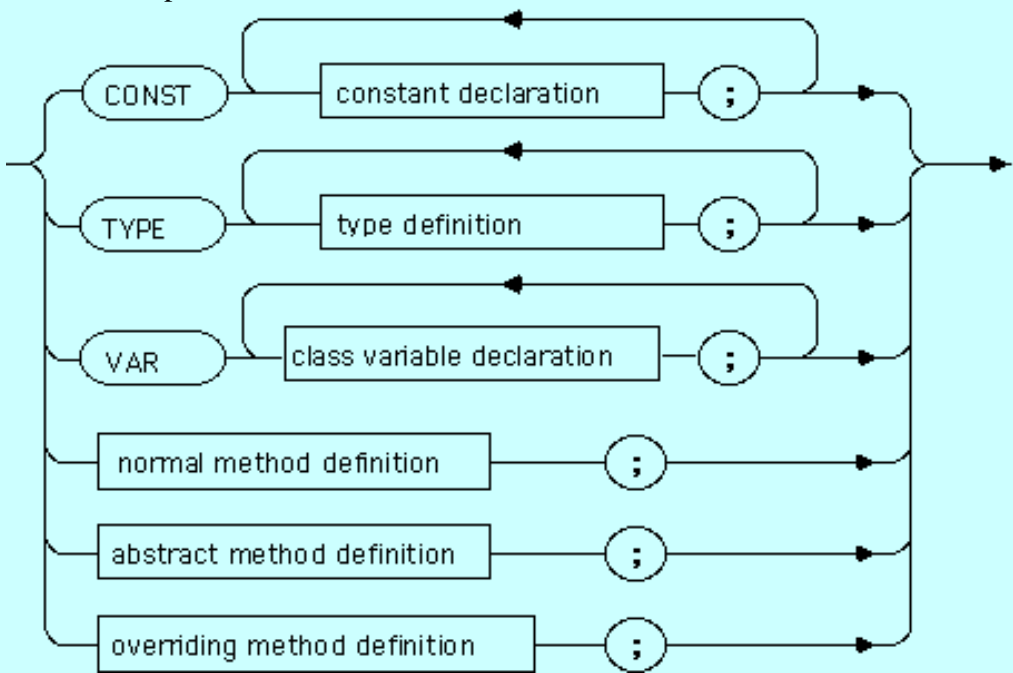
normal component definition



abstract class component definitions



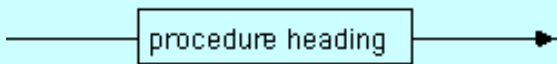
abstract component definition



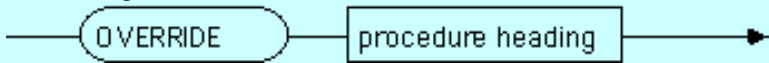
class variable declaration



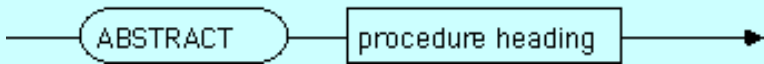
normal method definition



overriding method definition

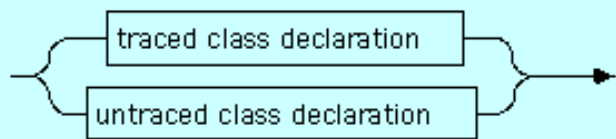


abstract method definition

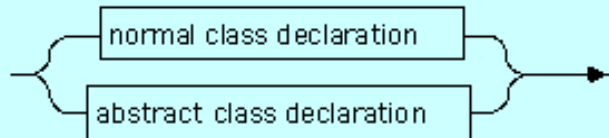


A13.3.2 Class Declaration

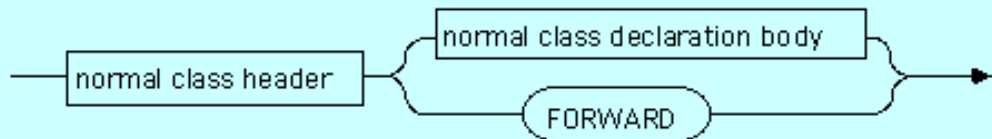
class declaration



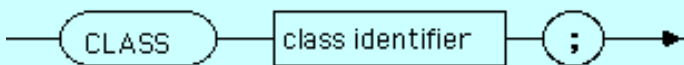
untraced class declaration



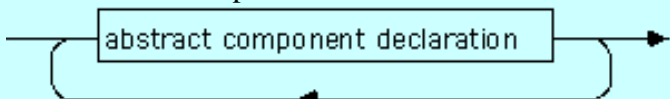
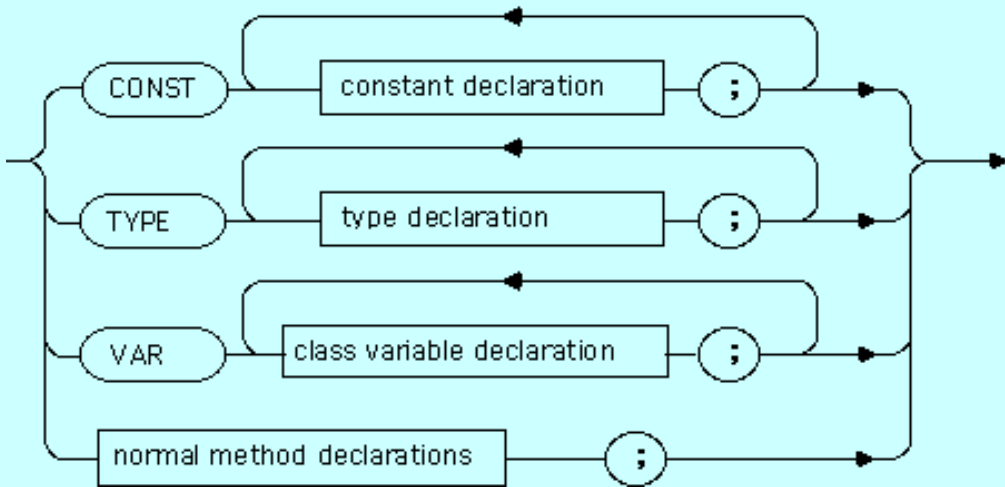
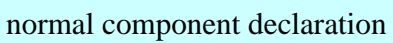
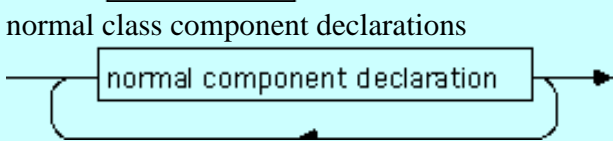
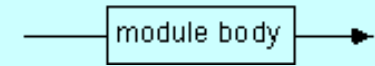
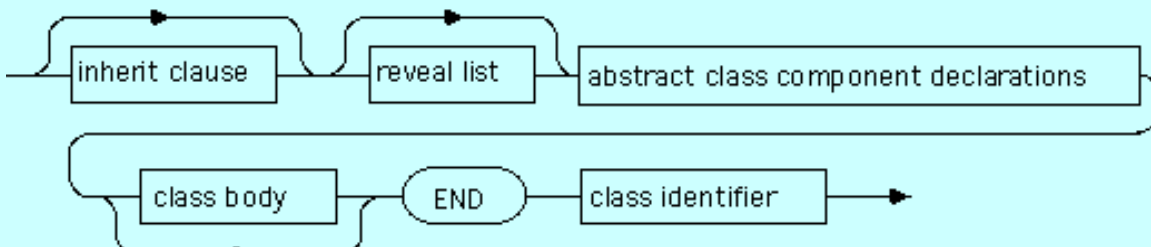
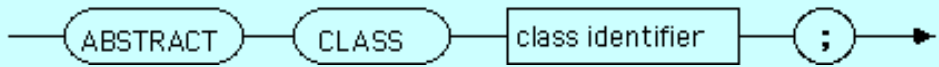
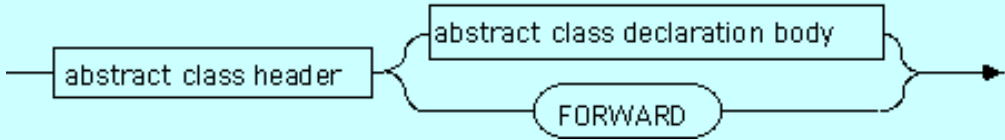
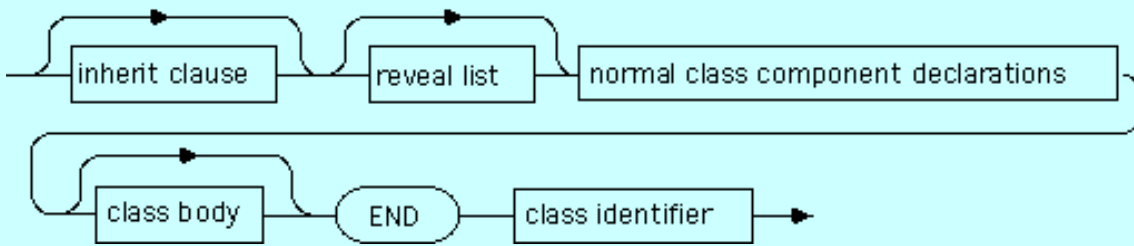
normal class declaration

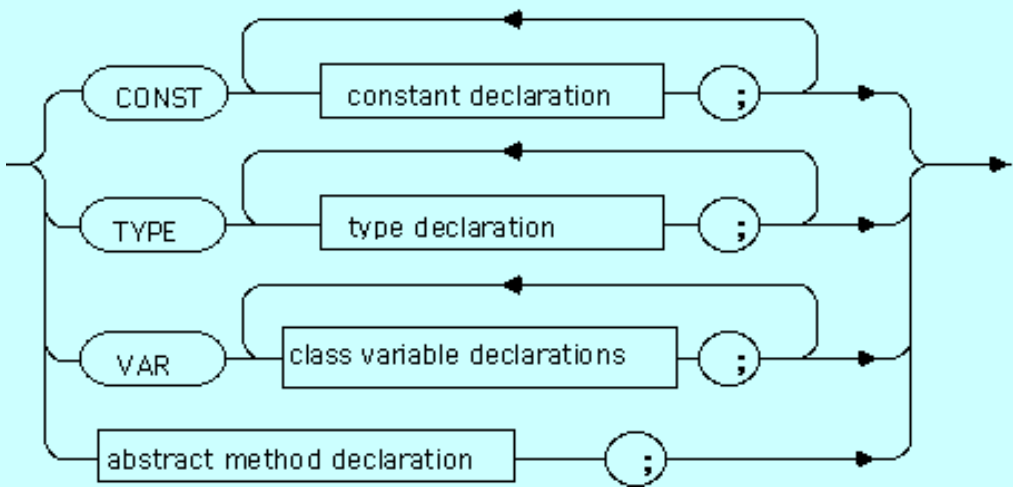


normal class header

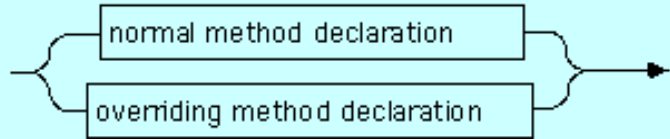


normal class declaration body

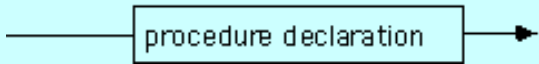




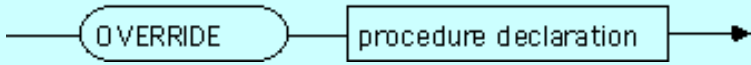
normal method declarations



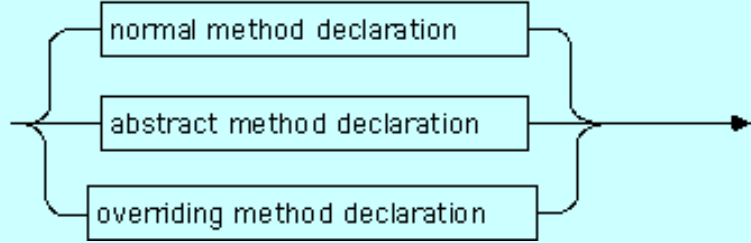
normal method declaration



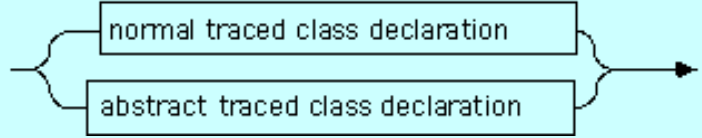
overriding method declaration



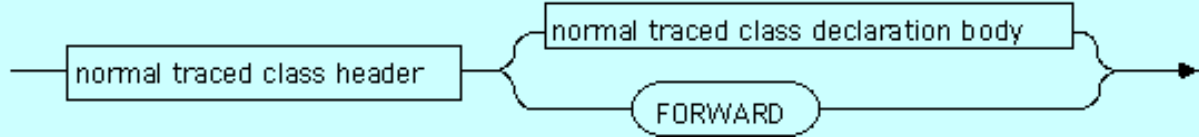
abstract method declarations



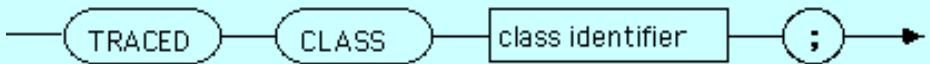
traced class declaration



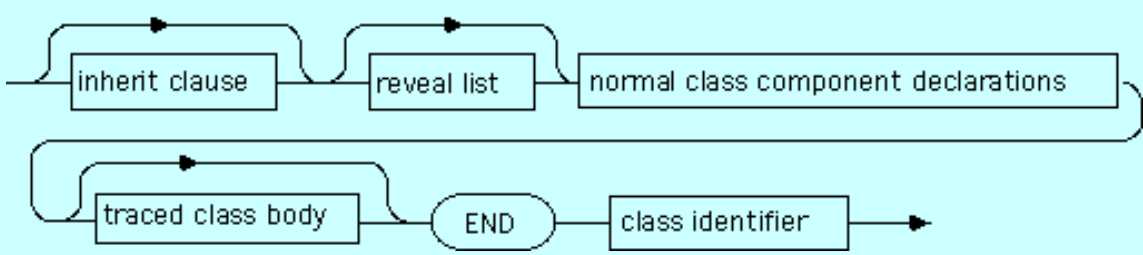
normal traced class declaration



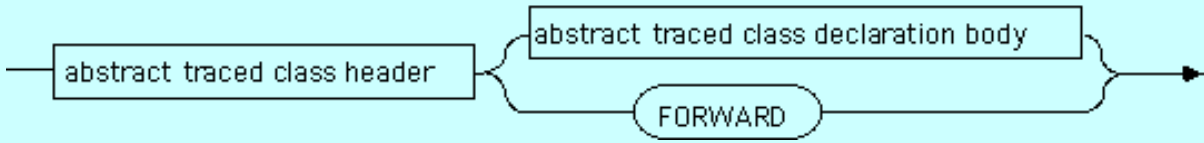
normal traced class header



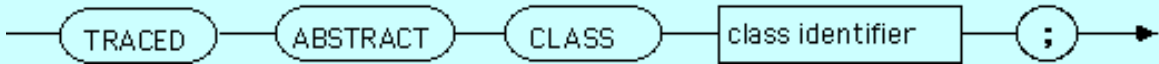
normal traced class declaration body



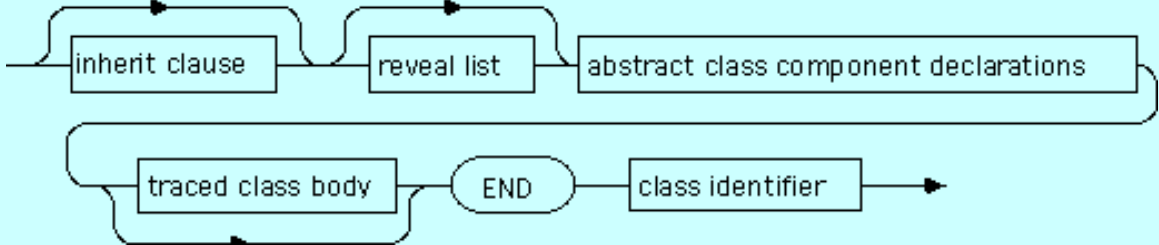
abstract traced class declaration



abstract traced class header



abstract traced class declaration body

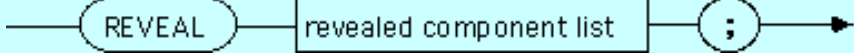


traced class body

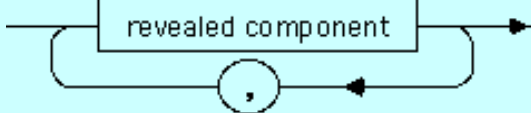


A13.3.3 Reveal List

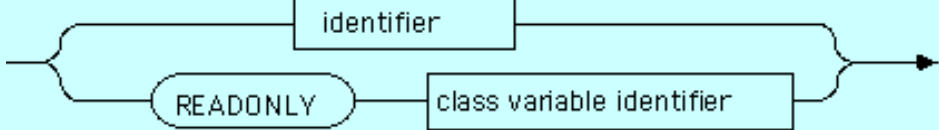
reveal list



revealed component list



revealed component

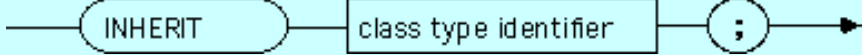


class variable identifier

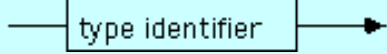


A13.3.4 Inherit Clause

inherit clause

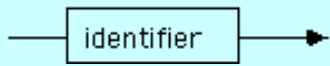


class type identifier



A13.3.5 Designators

object selected designator



A13.4 Changes To the Syntax of the Base Language in EBNF

See [Appendix 3.3](#) for the originals.

```
program module =  
    ["UNSAFEGUARDED"] "MODULE", module identifier, [protection], semicolon,  
    import lists, module block, module identifier, period ;  
definition module =  
    ["UNSAFEGUARDED"] "DEFINITION", "MODULE", module identifier, semicolon,  
    import lists, definitions, "END", module identifier, period ;  
implementation module =  
    ["UNSAFEGUARDED"] "IMPLEMENTATION", "MODULE", module identifier,  
    [protection], semicolon, import lists, module block,  
    module identifier, period ;  
definition =  
    "CONST", {constant declaration, semicolon} |  
    "TYPE", {type definition, semicolon} |  
    "VAR", {variable declaration, semicolon} |  
    procedure heading, semicolon ; |  
    class definition, semicolon ;  
declaration =  
    "CONST", {constant declaration, semicolon} |  
    "TYPE", {type declaration, semicolon} |  
    "VAR", {variable declaration, semicolon} |  
    procedure declaration, semicolon |  
    class declaration, semicolon |  
    local module declaration, semicolon ;  
qualified identifier =  
    {module identifier, period}, [class identifier, period],  
    identifier ;  
variable designator =  
    entire designator | indexed designator |  
    selected designator | dereferenced designator |  
    object selected designator ;  
value designator =  
    entire value | indexed value | selected value |  
    dereferenced value | object selected value ;  
statement =  
    empty statement | assignment statement | procedure call |  
    return statement | retry statement | with statement |  
    if statement | case statement | while statement |  
    repeat statement | loop statement | exit statement |  
    for statement | guard statement ;
```

A13.5 The Syntax of Object Oriented Modula-2 in EBNF

The concrete syntax in this section is taken from ISO/IEC IS 10514-3, the international standard for the object oriented extensions to Modula-2.

A13.5.1 Class Definition

```
class definition =  
    ( traced class definition | untraced class definition ) ;  
untraced class definition =  
    ( normal class definition | abstract class definition ) ;  
traced class definition =  
    "TRACED", ( normal class definition | abstract class definition ) ;  
  
normal class definition =  
    normal class header, ( normal class definition body | "FORWARD" ) ;  
normal class header =  
    "CLASS", class identifier, semicolon ;  
normal class definition body =  
    [ inherit clause ], [ reveal list ],  
    normal class component definitions, "END", class identifier ;  
  
abstract class definition =  
    abstract class header, ( abstract class definition body | "FORWARD" ) ;  
abstract class header =  
    "ABSTRACT", "CLASS", class identifier, semicolon ;  
abstract class definition body =  
    [ inherit clause ], [ reveal list ],  
    abstract class component definitions, "END", class identifier ;  
  
class identifier = identifier ;  
  
normal class component definitions = { normal component definition } ;  
normal component definition =  
    "CONST", { constant declaration, semicolon }      |  
    "TYPE", { type definition, semicolon }             |  
    "VAR", { class variable declaration, semicolon }   |  
    (normal method definition | overriding method definition),  
    semicolon ;  
  
abstract class component definitions =  
    {abstract component definition } ;  
abstract component definition =  
    "CONST", { constant declaration, semicolon }      |  
    "TYPE", { type definition, semicolon }             |  
    "VAR", { class variable declaration, semicolon }   |  
    (normal method definition | abstract method definition |  
    overriding method definition), semicolon ;  
  
class variable declaration = identifier list, colon, type denoter ;  
  
normal method definition = procedure heading ;  
overriding method definition = "OVERRIDE", procedure heading ;  
abstract method definition = "ABSTRACT", procedure heading ;
```

A13.5.2 Class Declaration

```

class declaration =
    ( traced class declaration | untraced class declaration ) ;
untraced class declaration =
    ( normal class declaration | abstract class declaration ) ;

normal class declaration =
    normal class header, ( normal class declaration body | "FORWARD" ) ;
normal class header =
    "CLASS", class identifier, semicolon ;
normal class declaration body =
    [ inherit clause ], [ reveal list ],
    normal class component declarations,
    [ class body ], "END", class identifier ;

abstract class declaration =
    abstract class header,
    ( abstract class declaration body | "FORWARD" ) ;
abstract class header =
    "ABSTRACT", "CLASS", class identifier, semicolon ;
abstract class declaration body =
    [ inherit clause ], [ reveal list ],
    abstract class component declarations,
    [ class body ], "END", class identifier ;

class body = module body;

normal class component declarations =
    { normal component declaration } ;
normal component declaration =
    "CONST", { constant declaration, semicolon }      |
    "TYPE", { type declaration, semicolon }            |
    "VAR", { class variable declaration, semicolon }   |
    normal method declarations , semicolon ;

abstract class component declarations =
    {abstract component declaration } ;
abstract component declaration =
    "CONST", { constant declaration, semicolon }      |
    "TYPE", { type declaration, semicolon }            |
    "VAR", { class variable declaration, semicolon }   |
    abstract method declarations , semicolon ;

normal method declarations =
    normal method declaration | overriding method declaration ;
normal method declaration = procedure declaration ;
overriding method declaration = "OVERRIDE", procedure declaration ;

abstract method declarations =
    normal method declaration | abstract method definition |
    overriding method declaration ;

traced class declaration =
    ( normal traced class declaration | abstract traced class declaration ) ;

```



```

normal traced class declaration =
    normal traced class header,
    ( normal traced class declaration body | "FORWARD" ) ;
normal traced class header =
    "TRACED", "CLASS", class identifier, semicolon ;
normal traced class declaration body =
    [ inherit clause ], [ reveal list ],
    normal class component declarations,
    [ traced class body ], "END", class identifier ;

abstract traced class declaration =
    abstract traced class header,
    ( abstract traced class declaration body | "FORWARD" ) ;
abstract traced class header=
    "TRACED", "ABSTRACT", "CLASS", class identifier, semicolon ;
abstract traced class declaration body =
    [ inherit clause ], [ reveal list ],
    abstract class component declarations,
    [ traced class body ], "END", class identifier ;

traced class body = "BEGIN", block body ;

```

A13.5.3 Reveal List

```

reveal list = "REVEAL" revealed component list, semicolon ;

revealed component list =
    revealed component, { comma, revealed component } ;
revealed component = identifier | "READONLY" class variable identifier ;
class variable identifier = identifier ;

```

A13.5.4 Inherit Clause

```

inherit clause = "INHERIT", class type identifier, semicolon ;
class type identifier = type identifier ;

```

A13.5.5 Designators

```

object selected designator =
    object variable designator, period,
    [ class identifier, period ], class variable identifier ;
object variable designator = variable designator ;

object selected value =
    object value designator, period, [ class identifier, period ], entity identifier ;
object value designator = value designator ;
entity identifier = identifier ;

```

A13.5.6 Guard Statement

```

guard statement =

```

```
"GUARD", guard selector, "AS", guarded list,  
["ELSE" statement sequence],  
"END" ;
```

guard selector = expression ;

guarded list =

guarded statement sequence {vertical bar, guarded statement sequence} ;

guarded statement sequence =

[[object denoter], colon, guarded class type, "DO", statement sequence] ;

guarded class type = class type identifier ;

object denoter = identifier ;

A13.6 Other Changes to the Base Language

Unless a program module or an implementation module is tagged as unsafeguarded in its header, it shall not contain:

- an import that directly or indirectly causes the import of an untraced class type,
- a declaration of an untraced class type,
- a call of SYSTEM.CAST with a variable of a traced class type or a structured type that contains a component of a traced class type as the second actual parameter,
- a call that passes a variable of a traced class type as a value actual parameter to an open or fixed ARRAY OF LOC,
- a call of SYSTEM.ADR with a class variable (attribute) as the actual parameter,
- a call that passes a class variable (attribute) as an actual variable parameter to an open or fixed ARRAY OF LOC.

Unless a definition module is tagged as unsafeguarded in its header, it shall not contain:

- an import that directly or indirectly causes the import of an untraced class type,
- a definition of an untraced class type,

It shall be an error to declare a pointer type or a variable of a pointer type with a bound type that is or contains a record with a variant of a traced class type. It shall be an error to declare a type or a variable that is or contains a record with a variant of a traced class type.

OOM-2 extensions to the base language define four new exceptions. Detection of the first three is mandatory, but detection of the last one is optional. They are:

1. emptyException

raised whenever an attempt is made to access an object via an empty reference

2. abstractException

raised whenever an attempt is made to call an abstract method

3. guardException

raised if there is no match on the list of selections (possibly due to an empty reference) and no ELSE

4. immutableException

an implementation may choose to raise this if there is an attempt to change an immutable entity.

A13.7 ISO Libraries Supporting Object Oriented Modula-2

A13.7.1 Modifications to Coroutines

The module COROUTINES in the base standard has one procedure added to provide support for OOM-2. It is

```
PROCEDURE DISPOSECOROUTINE (VAR cr: COROUTINE);  
  (* Declare that the coroutine identified by cr has reached the end of its lifetime.  
  *)
```

A13.7.2 The Pseudo-Module M2OOEXCEPTION

```
DEFINITION MODULE M2OOEXCEPTION;
```

```
(* Provides facilities for identifying exceptions of the extended language *)
```

```
TYPE
```

```
  M2OOExceptions =  
    (emptyException, abstractException, immutableException, guardException);
```

```
PROCEDURE M2OOException (): M2OOExceptions;
```

```
  (* If the current coroutine is in the exceptional execution state because  
  of the raising of an exception of the language extensions, returns the  
  corresponding enumeration value, and otherwise raises an exception. *)
```

```
PROCEDURE IsM2OOException (): BOOLEAN;
```

```
  (* If the current coroutine is in the exceptional execution state because  
  of the raising of an exception of the language extensions, returns  
  TRUE, and otherwise returns FALSE. *)
```

```
END M2OOEXCEPTION.
```

A13.7.2 GARBAGECOLLECTION

```
DEFINITION MODULE GARBAGECOLLECTION;
```

```
(* Provides facilities for controlling the garbage collector. *)
```

```
PROCEDURE IsCollectionEnabled (): BOOLEAN;
```

```
  (* If garbage collection is enabled then returns TRUE and otherwise returns FALSE.  
  *)
```

```
PROCEDURE CollectionEnable (on: BOOLEAN);
```

```
  (* If on is TRUE then enable garbage collection; otherwise if on is FALSE and  
  garbage  
  collection can be disabled then disable garbage collection. *)
```

```
PROCEDURE ForceCollection;
```

```
  (* If garbage collection can be forced then force it else do nothing. *)
```

```
END GARBAGECOLLECTION.
```

Appendix 14--Bibliography

- Adams, J. Mack, Gabrini, Philippe J & Kurtz, Barry L. *An Introduction to Computer Science with Modula-2* Lexington, MA D.C. Heath & Co 1988
- Beidler, John & Jackowitz, Paul *Modula-2* Boston Prindle Weber & Schmidt 1985
- Budgen, David *Software Development with Modula-2* Reading, MA Addison-Wesley 1989
- Chirlian, Paul M. *Introduction to Modula-2* Beaverton, Or. Matrix Publishers
- Christian, Kaare *A guide to Modula-2* New York Springer-Verlag 1986
- Cooling, J.E. *Modula-2 for Microcomputer Systems* Van Nostrand Reinhold 1988
- Cooper, Doug *Oh My! Modula-2!* New York Norton 1990
- Cornelius, Barry *Programming with TopSpeed Modula-2* Reading, MA Addison Wesley 1991
- Eisenbach, Susan & Sadler, Cristopher *Program Design with Modula-2* Reading, MA Addison-Wesley 1989
- Etling, Don *Modula-2 Programmer's Resource Book* Blue Ridge Summit, PA Tab Books 1988
- Feldman, Michael B. *Data Structures with Modula-2* Englewood Cliffs, NJ Prentice Hall 1988
- Ford, Gary & Wiener, Richard. *Modula-2: A Software Development Approach* New York Wiley 1985
- Gleaves, Richard *Modula-2 for Pascal Programmers* New York Springer-Verlag 1984
- Gough, K. John & Mohay, George M. *Modula-2: A Second Course In Programming* Englewood Cliffs, NJ Prentice Hall 1988
- Greenfield, Stuart B. *Invitation to Modula-2* Petrocelli Books 1985
- Harrison, Rachael *Abstract Data Types in Modula-2* New York Wiley 1989 Wiley
- Harter, Edward D *Modula-2 Programming: A First Course* Englewood Cliffs, NJ Prentice Hall 1990
- Harter, Edward D. *Modula-2 Programming. A First Course* Englewood Cliffs, NJ Prentice-Hall 1990
- Helman, Paul & Veroff, Robert *Walls and Mirrors: Intermediate Problem Solving and Data. Modula-2* Menlo Park, CA Benjamin Cummings 1988
- Hewitt, Jill A. & Frak, Raymond J. *Software Engineering in Modula-2: an object-oriented approach* London Macmillan 1989.
- Hille, R.F. *Data Abstraction and Program Development Using Modula-2* Sydney Prentice Hall 1989
- Hopper, Keith. *The Magic of Modula-2* Melbourne Prentice Hall 1991
- Jones, William C. Jr. *Data Structures Using Modula-2* New York Wiley 1988
- Jones, William C. Jr. *Modula-2 Problem Solving and Programming with Style* New York Harper & Row 1987
- Joyce, Edward J. *Modula-2: A Seafarer's Manual & Shipyard Guide* Reading, MA Addison-Wesley 1985
- Kaplan, Ian & Miller, Mike *Modula-2 Programming* Rochelle Park, NJ Hayden Book Co. 1986
- Kelly-Bootle, Stan *Modula-2 Primer* Howard W. Sams & Co. 1987
- King, K.N. *Modula-2: A Complete Guide* Lexington, MA D.C. Heath & Co 1988
- Knepley, Ed & Platt, Robert *Modula-2 Programming* Reston, VA Reston Pub. Co. 1985
- Koffman, Elliot B. *Problem Solving and Structured Programming in Modula-2* Reading, MA Addison-

Wesley 1988

Leestma, Sanford & Nyhoff, Larry *Programming & Problem-Solving in Modula-2* New York Macmillan 1989

Lins, C. (Charles) *The Modula-2 Software Component Library Volumes I-IV* New York Springer-Verlag 1989-

Mayer, Herbert G. *Programming in Modula-2. the Art & the Craft* New York Macmillan 1988

McCracken, Daniel D. & W. Salmon *A Second Course in Computer Science with Modula-2* New York Wiley 1987

Messer, P. A. & I. Marshall *Modula-2 Constructive Program Development* Oxford Blackwell Scientific Publications 1986

Metrowerks, Inc. Staff *Metrowerks Modula-2 Start Pak* New York Macmillan 1990

Moore, John B. & McKay, Kenneth N. *Modula-2 Text and Reference* Englewood Cliffs, NJ Prentice-Hall 1987

Novak, M.M. *Modula-2 in Science & Engineering* London McGraw 1990

Nyhoff, Larry & Leestma, Sanford *Data Structures & Advanced Programming in Modula-2* New York Macmillan 1990

Ogilvie, John W. L. *Modula-2 Programming* New York McGraw-Hill 1985

Pinson, Lewis Sincovec, Richard & Weiner, Richard *A First Course in Computer Science with Modula-2* New York Wiley 1987

Pomberger, Gustav. *Software Engineering and Modula-2* Englewood Cliffs, NJ Prentice Hall 1984

Rechenberg, P. & Mssenbck, H. (tr. O'Meara, John) *A Compiler Generator for Microcomputers* Englewood Cliffs, NJ Prentice Hall / Carl Hanser Verlag 1989

Riley, David D. *Data Abstraction and Structures: An Introduction To Computer Science II* Boston Boyd & Fraser Pub. Co. 1987

Riley, David D. *Using Modula-2: An Introduction To Computer Science I* Boston Boyd & Fraser Pub. Co. 1987

Sale, Arthur H. J. *Modula-2: Discipline & Design* Sydney Addison-Wesley 1986

Sawyer, Brian & Foster, Dennis. *Programming Expert Systems in Modula-2* New York Wiley 1986

Schildt, Herbert *Advanced Modula-2* Berkeley, CA Osborne McGraw-Hill 1987

Schildt, Herbert *Modula-2 Made Easy* Berkeley, CA Osborne McGraw-Hill 1986

Schipper, Andre; (tr. Howlett, Jack) *Concurrent programming: Illustrated With Examples in Portal, Ada, and Modula-2* Halsted Press 1989

Schnapp, Russell L. *Macintosh Graphics in Modula-2* Englewood Cliffs, NJ Prentice-Hall 1986

Sincovec, Richard F. & Richard S. Wiener. *Data Structures Using Modula-2* New York Wiley 1986

Sincovec, Richard F. & Wiener, Richard S. *Modula-2 Software Components* New York Wiley 1987

Stubbs, Daniel F. & Webre, Neil W. *Data Structures With Abstract Data Types and Modula-2* Monterey, CA Brooks/Cole Pub. Co. 1987

Sutcliffe, Richard J. *Introduction to Programming Using Modula-2* Columbus, OH Merrill 1987

Sutherland, Robert J. *The Professional Programmer's Guide to Modula-2* London Pitman 1988

Terry, Patrick D. *An Introduction To Programming with Modula-2* Reading, MA Addison-Wesley 1987

Thalmann, Daniel *Modula-2: An Introduction* New York Springer-Verlag 1985

Tremblay, Jean-Paul DeDourek, John M. & Daoust, David A. *Programming in Modula-2* New York McGraw-Hill 1989

Tucker, Allen B. Jr. *Computer science: A Second Course Using Modula-2* New York McGraw-Hill 1988

Ural, Saim & Ural, Suzan *Introduction to Programming with Modula-2* New York Harper & Row 1987

Walker, Billy K *Modula-2 Programming With Data Structures* Belmont, CA Wadsworth Pub. Co. 1986

Walker, Robert D. *Modula-2 Library Modules: A Programmer's Reference* Blue Ridge Summit, PA Tab Books 1988

Ward, Terry A. *Advanced Programming Techniques in Modula-2* Glenview, IL Scott Foresman 1987

Welsh, Jim & Elder, John *Introduction to Modula-2* Englewood Cliffs, NJ Prentice-Hall 1987

Wiatrowski, Claude A. & Wiener, Richard S. *From C to Modula-2--and Back - Bridging The Language Gap* New York Wiley

Wiener, Richard *Modula-2 Wizard's Programming Reference* New York Wiley 1986

Wiener, Richard & Ford, G. *Modula-2 A Software Development Approach* New York Wiley 1985

Wiener, Richard & Sinovec, R. F. *Software Engineering with Modula-2 and Ada* New York Wiley 1984

Wirth, Niklaus *Programming in Modula-2 (3rd corrected ed.)* New York Springer-Verlag 1985

Wirth, Niklaus *Algorithms and Data Structures (1986 edition)* Englewood Cliffs, NJ Prentice-Hall 1986

Wirth, Niklaus *Programming in Modula-2 (4th ed.)* New York Springer-Verlag 1990

Woodman, Mark et al *Portable Modula-2 Programming* Maidenhead, Berkshire UK McGraw-Hill 1989

[Contents](#)

Appendices

[Appendix 1--The Lexis of Modula-2](#)

[A1.1 Reserved Words \(Keywords\)](#)

[A1.2 Standard \(Pervasive\) Identifiers](#)

[A1.3 Standard Symbols](#)

[A1.4 Standard Operators](#)

[Appendix 2--Syntax Diagrams](#)

[A2.1 Lexis](#)

[A2.2 Syntax](#)

[Appendix 3--The Syntax of Modula-2](#)

[A3.1 A Notation to Describe Languages](#)

[A3.2 Some Examples of EBNF](#)

[A3.3 The Syntax of Modula-2 in EBNF](#)

[Appendix 4--Classical Library Modules](#)

[A4.1 High Level Input and Output](#)

[A4.2 Mathematical Functions](#)

[A4.3 SYSTEM and Other Low Level and System Access Modules](#)

[A4.4 Storage](#)

[A4.5 String Handling](#)

[A4.6 File I/O](#)

[A4.7 Character Information--ASCII](#)

[Appendix 5--ISO I/O Library](#)

[A5.1 An Overview of the ISO I/O Library](#)

[A5.2 I/O On Standard Channels](#)

[A5.3 Supplied Channels](#)

[A5.4 Specified Channels](#)

[A5.5 Channel Constants--IOConsts](#)

[A5.6 Device Independent Channel I/O--IOChan](#)

[A5.7 Device Drivers](#)

[A5.8 Device Module Constants--ChanConsts](#)

[A5.9 Linking Drivers to Channels--IOLink](#)

[Appendix 6--ISO Support Modules for This Text](#)

[A6.1 RedirStdIO](#)

[A6.2 Files](#)

[A6.3 Keyboard](#)

[A6.4 CharBuffer](#)

[A6.5 STerminal](#)

[A6.6 ACSCI](#)

[A6.7 SComplexIO](#)

[A6.8 SLongComplexIO](#)

[A6.9 ComplexIO](#)

[A6.10 LongComplexIO](#)

[A6.11 GraphPaper](#)

[A6.12 GraphWindow](#)

[Appendix 7--ISO Required System Modules](#)

[A7.1 SYSTEM](#)

[A7.2 COROUTINES](#)

[A7.3 EXCEPTIONS](#)

[A7.4 TERMINATION](#)

[A7.5 M2EXCEPTION](#)

[Appendix 8--ISO Utility and Information Modules](#)

[A8.1 Characters and Strings](#)

[A8.2 High Level String Conversion Modules](#)

[A8.3 Low Level String Conversion Modules](#)

[A8.4 SysClock--The Date and Time](#)

[Appendix 9--ISO Mathematics Library Module](#)

[A9.1 RealMath](#)

[A9.2 LongMath](#)

[A9.3 ComplexMath](#)

[A9.4 LongComplexMath](#)

[Appendix 10--ISO Process Support](#)

[A10.1 Processes](#)

[A10.2 Semaphores](#)

[Appendix 11--Modula-2 and Pascal](#)

[A11.1 Statement Syntax Differences](#)

[A11.2 Symbols](#)

[A11.3 Overall Structure](#)

[Appendix 12--Generic Modula-2 Syntax](#)

[A12.1 Keywords](#)

[A12.2 Diagrams of Changes to Base Language Syntax](#)

[A12.3 Generic Modula-2 Syntax Diagrams](#)

[A12.4 Changes To the Syntax of the Base Language in EBNF](#)

[A12.5 The Syntax of Generic Modula-2 in EBNF](#)

[Appendix 13--Object Oriented Modula-2 Syntax](#)

[A13.1 Keywords and Pervasive Identifiers](#)

[A13.2 Diagrams of Changes to Base Language Syntax](#)

[A13.3 Object Oriented Modula-2 Syntax Diagrams](#)

[A13.4 Changes To the Syntax of the Base Language in EBNF](#)

[A13.5 The Syntax of Object Oriented Modula-2 in EBNF](#)

[A13.6 Other Changes to the Base Language](#)

[A13.7 ISO Libraries Supporting Object Oriented Modula-2](#)

[Appendix 14--Bibliography](#)

[Answers to Questions and Selected Problems](#)

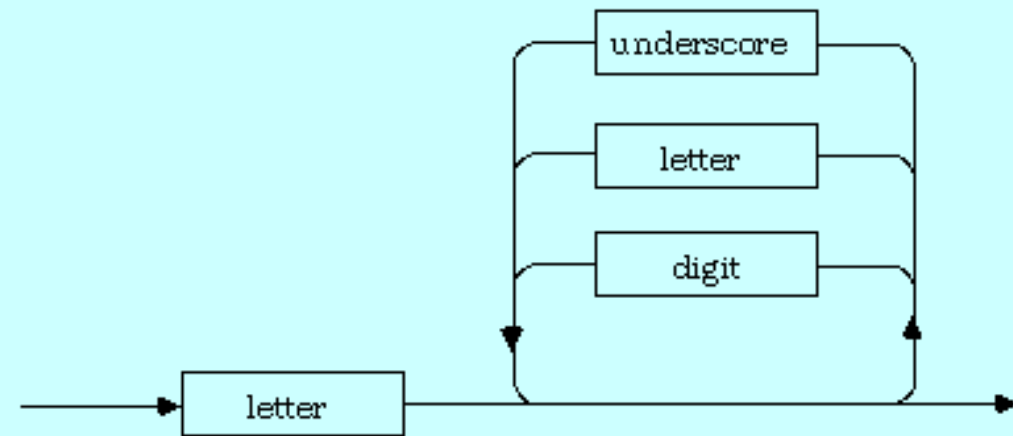
Contents

Appendix 2 Syntax Diagrams

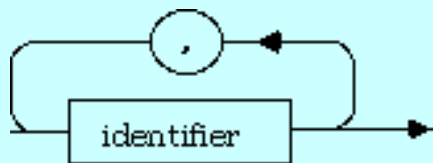
The syntax diagrams in this section are more complete and expand upon those given in the body of the text. They are intended as visual aids to understanding the correct structure of Modula-2 programming constructions, but they do not constitute an official or complete definition of the language. For that, see the Modula-2 Draft Standard.

A2.1 Lexis

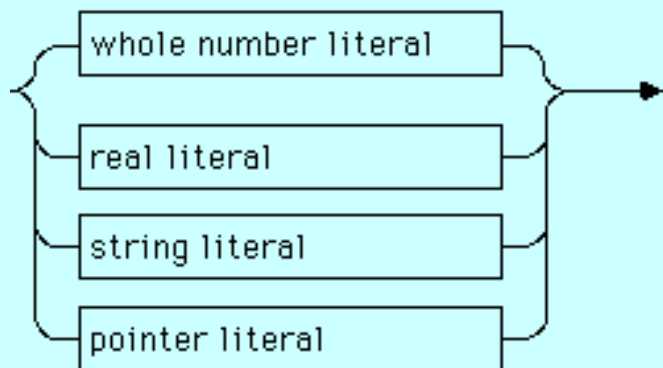
identifier



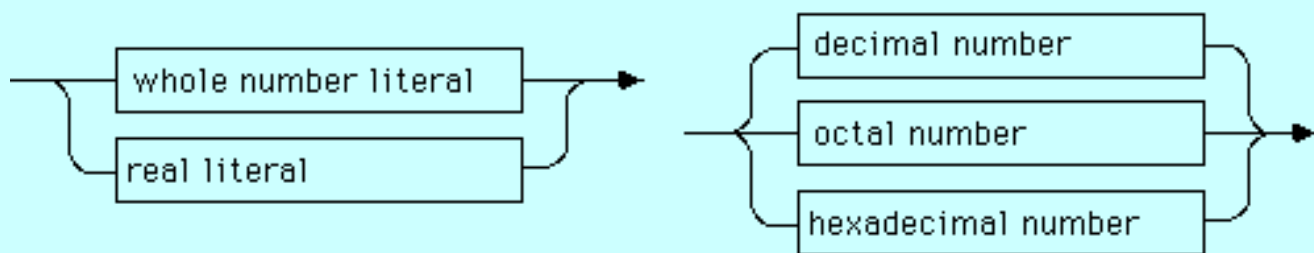
identifier list



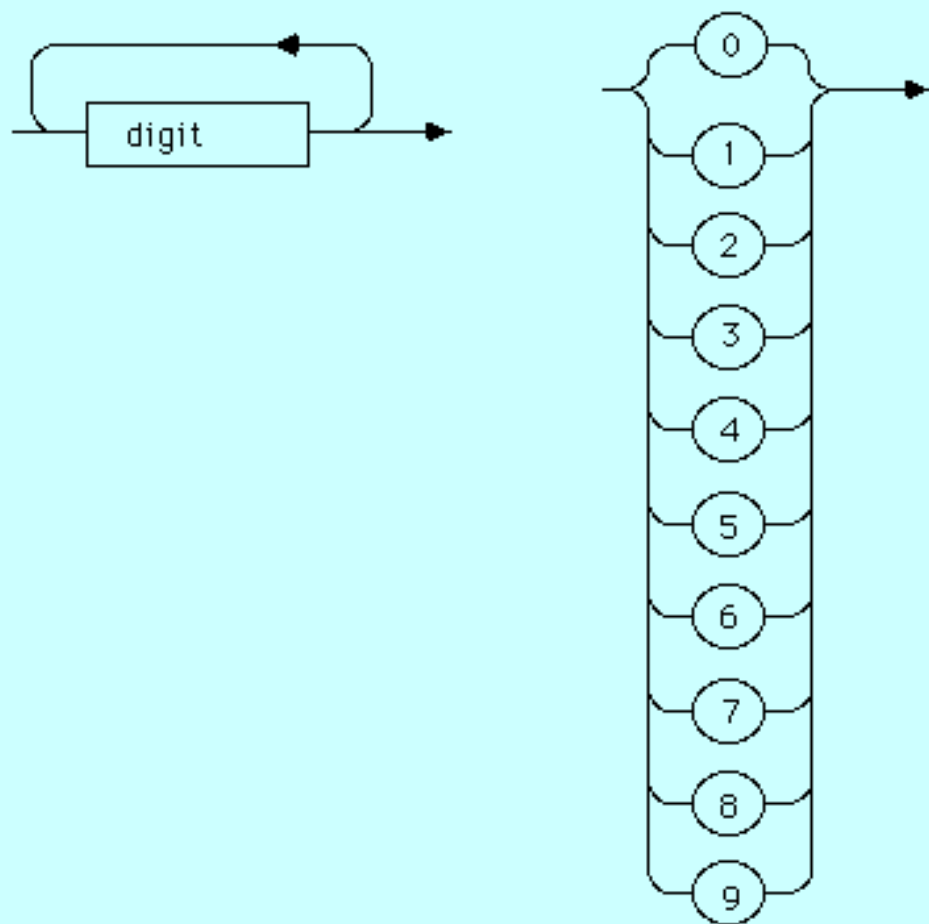
constant literal



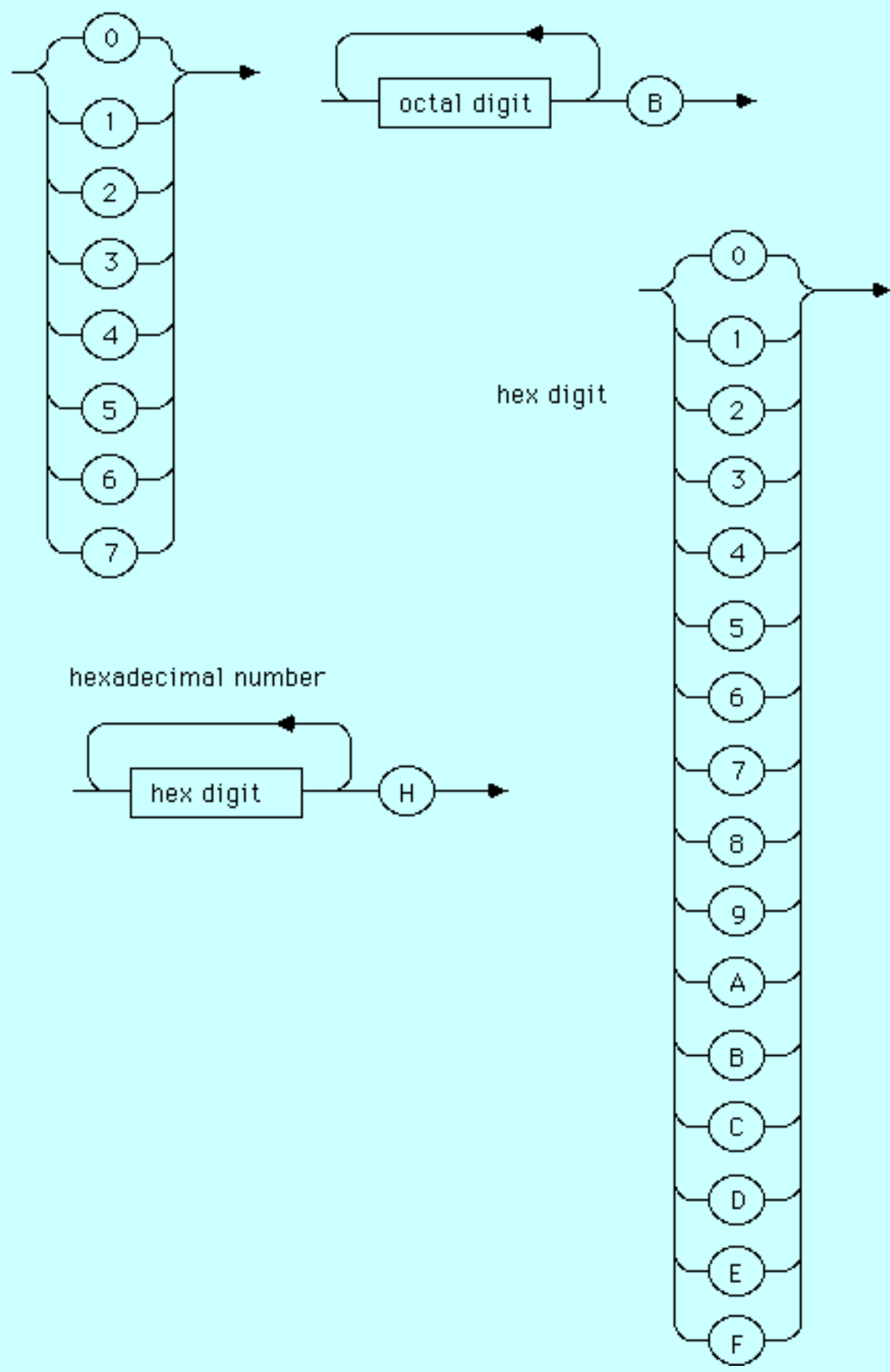
numeric literal whole number literal



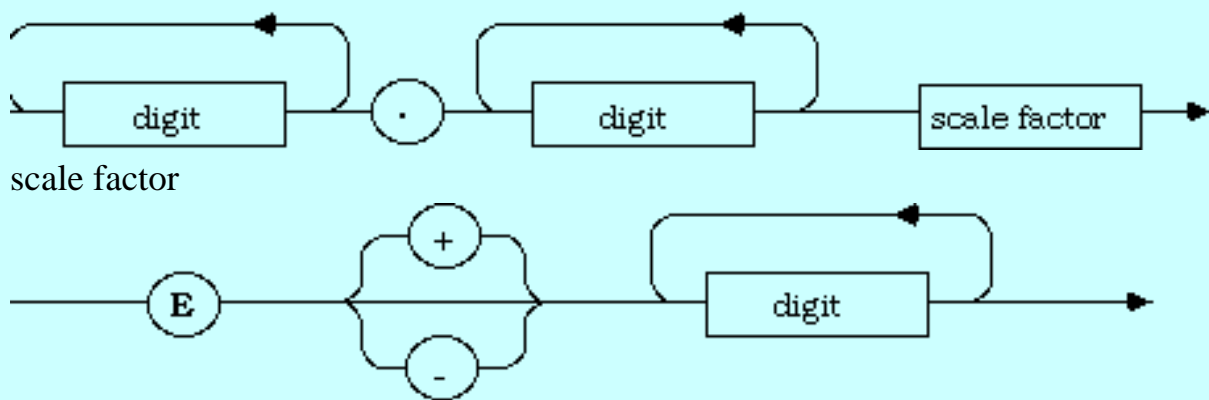
decimal number digit



octal digit octal number



real literal

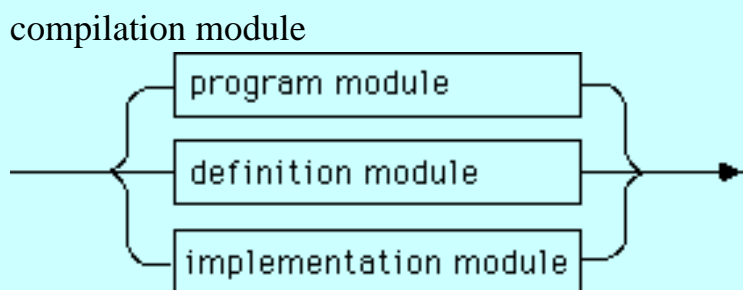


A2.2 Syntax

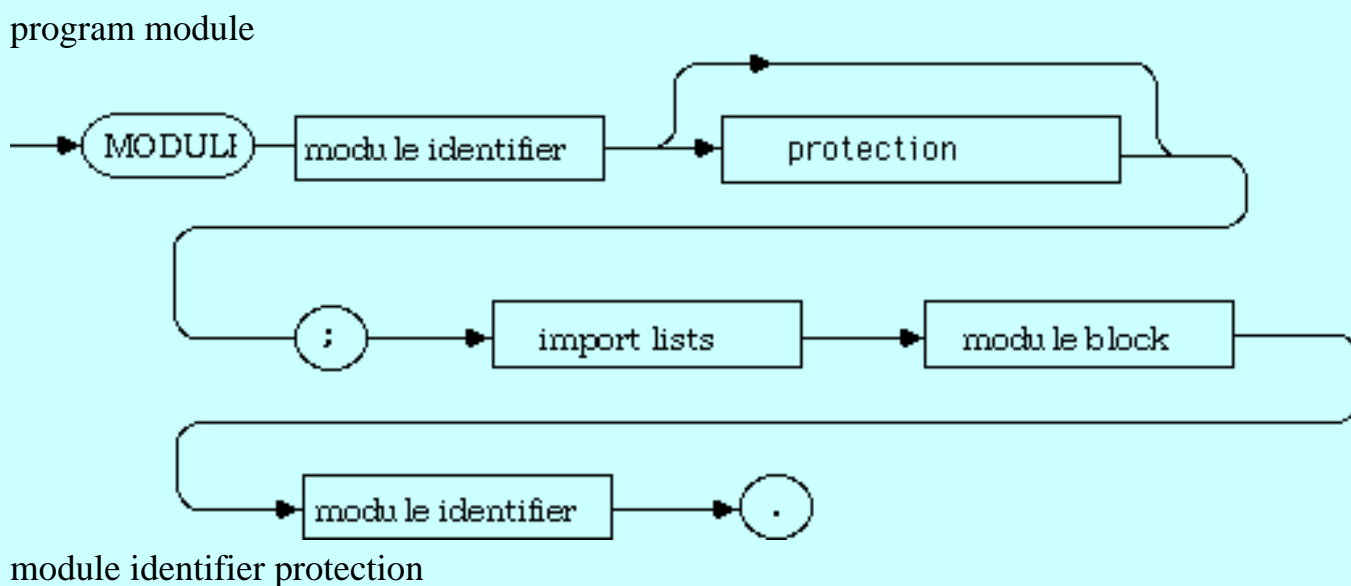
The numbering in this section (after the A2.2) corresponds to that of the Modula-2 ISO Standard Annex C.

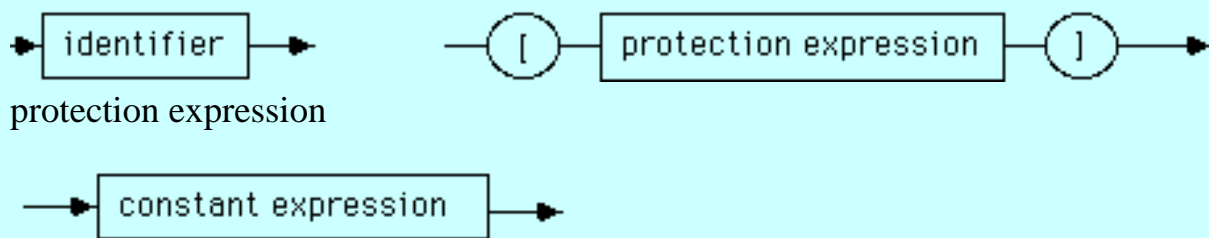
A2.2.1 Programs, Program Modules, and Separate Modules

A2.2.1.1 Programs and Compilation Modules

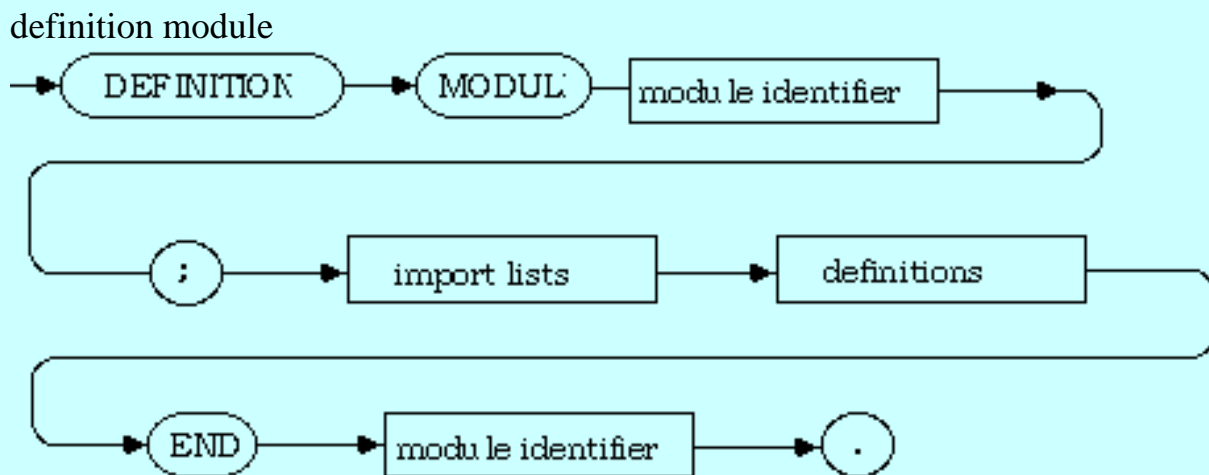


A2.2.1.2 Program Modules



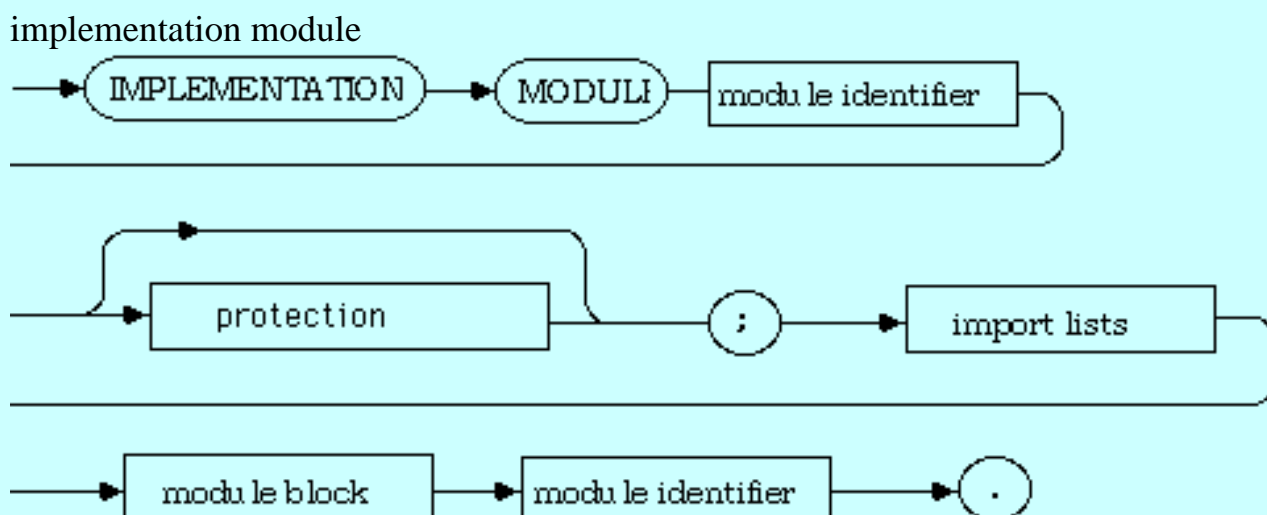


A2.2.1.3 Definition Module



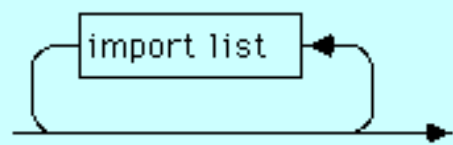
A2.2.1.4 Implementation Modules

A2.2.1.4.1 Sourced Implementation Modules

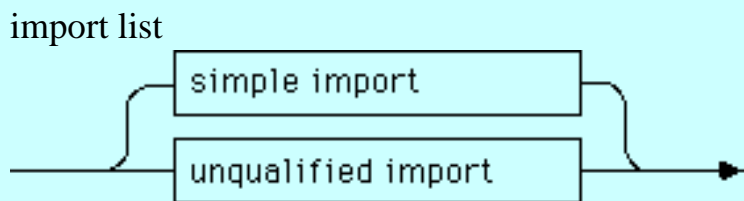


A2.2.1.5 Import Lists

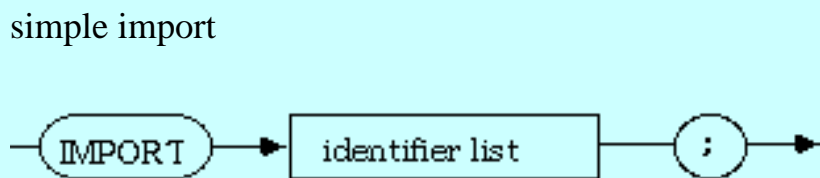
import lists



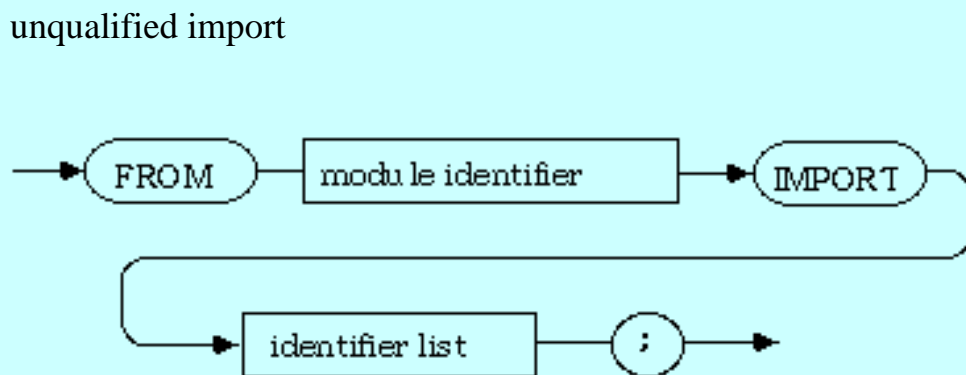
A2.2.1.5.1 Import List



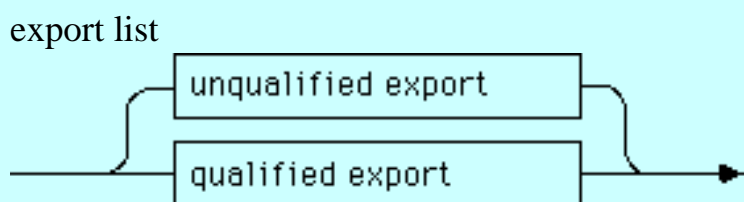
A2.2.1.5.2 Simple Import



A2.2.1.5.3 Unqualified Import

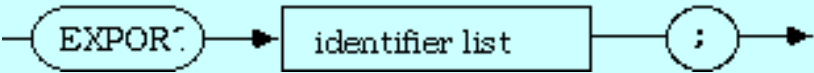


A2.2.1.6 Export Lists



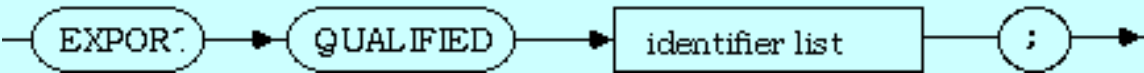
A2.2.1.6.1 Unqualified Exports

unqualified export



A2.2.1.6.2 Qualified Exports

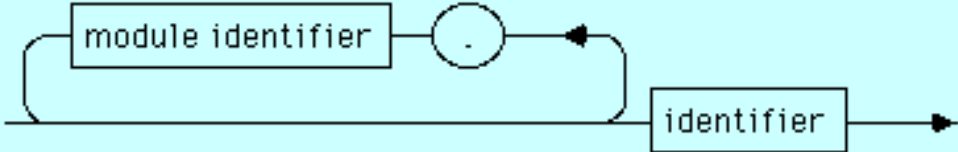
qualified export



A2.2.2 Definitions and Declarations

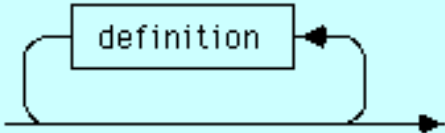
A2.2.2.1 Qualified Identifiers

qualified identifier

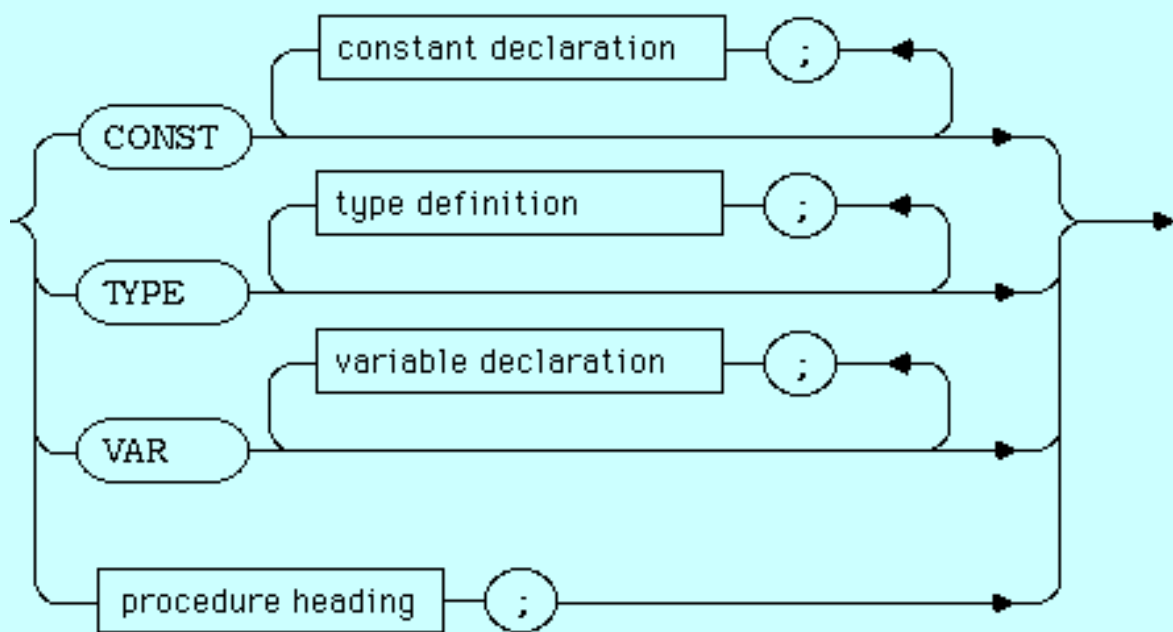


A2.2.2.2 Definitions

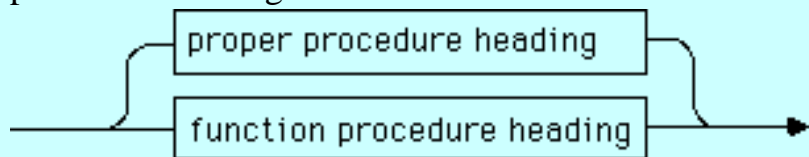
definitions



definition

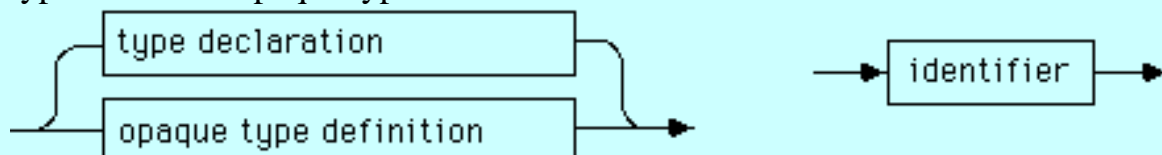


procedure heading



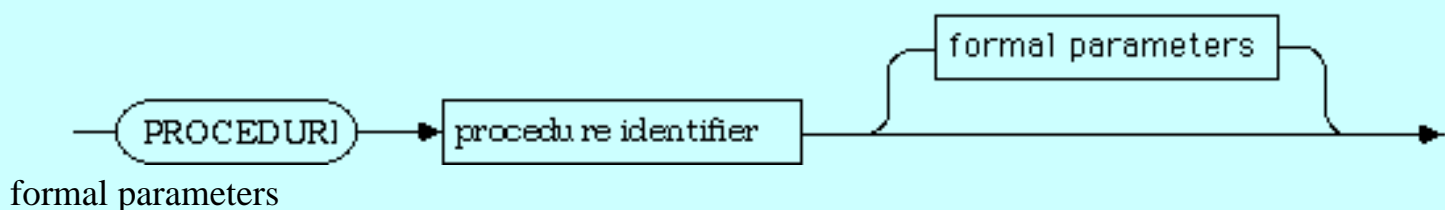
A2.2.2.2.1 Type Definitions

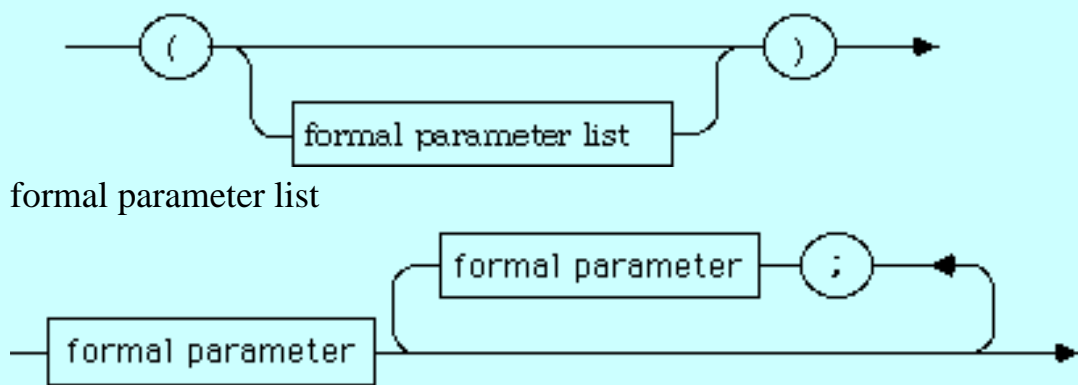
type definition opaque type definition



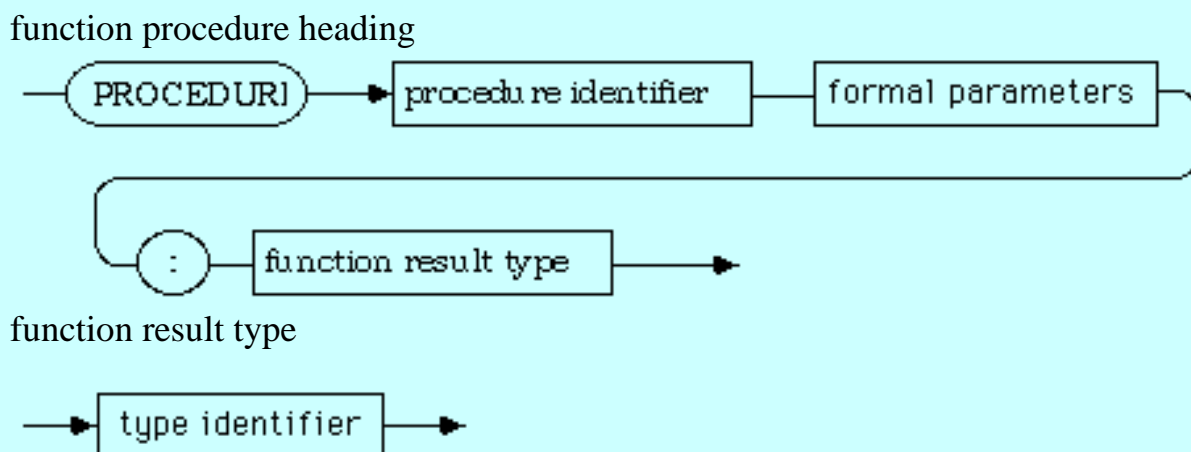
A2.2.2.2.2 Proper Procedure Headings

proper procedure heading

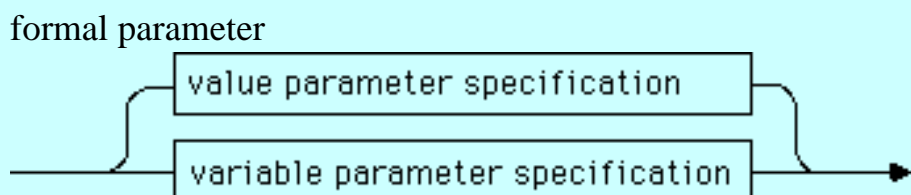




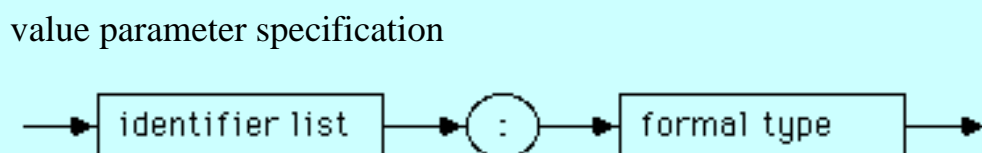
A2.2.2.2.3 Function Procedure Headings



A2.2.2.2.4 Formal Parameters

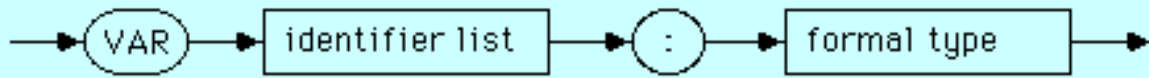


A2.2.2.2.4.1 Value Parameters



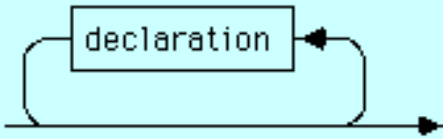
A2.2.2.2.4.2 Variable Parameters

variable parameter specification

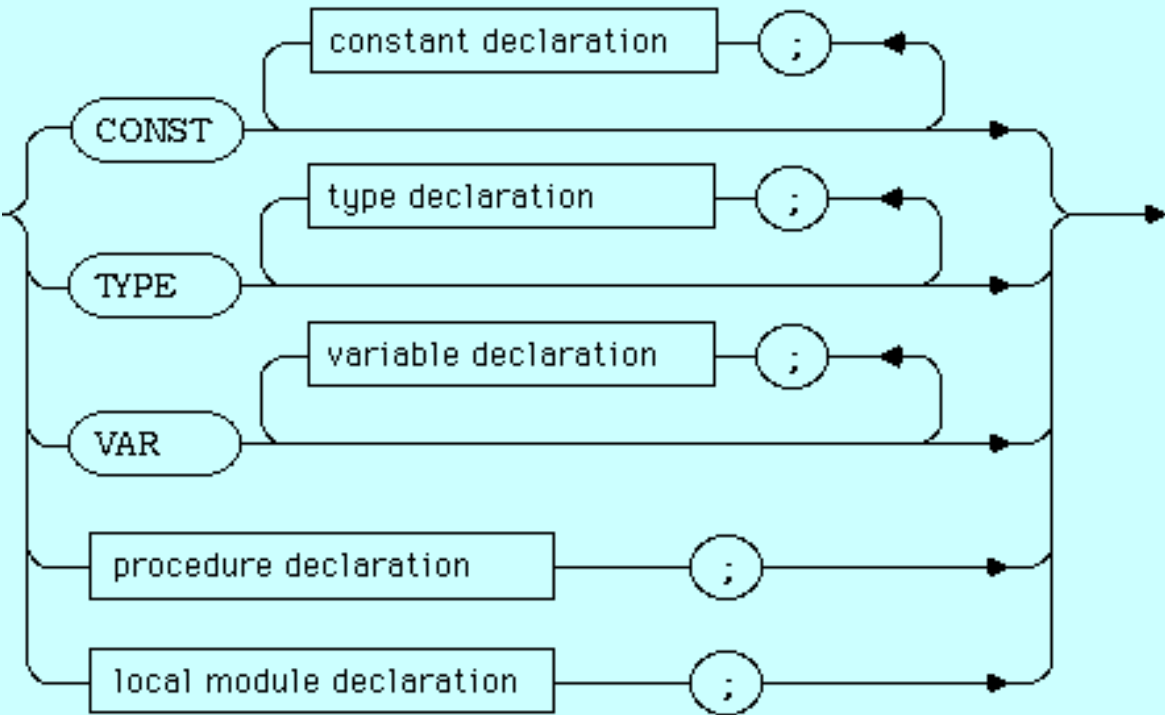


A2.2.2.3 Declarations

declarations

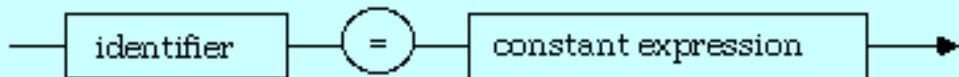


declaration



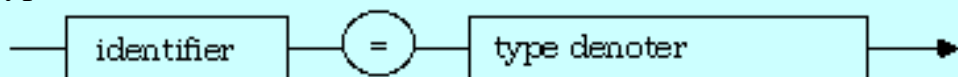
A2.2.2.4 Constant Declarations

constant declaration



A2.2.2.5 Type Declarations

type declaration

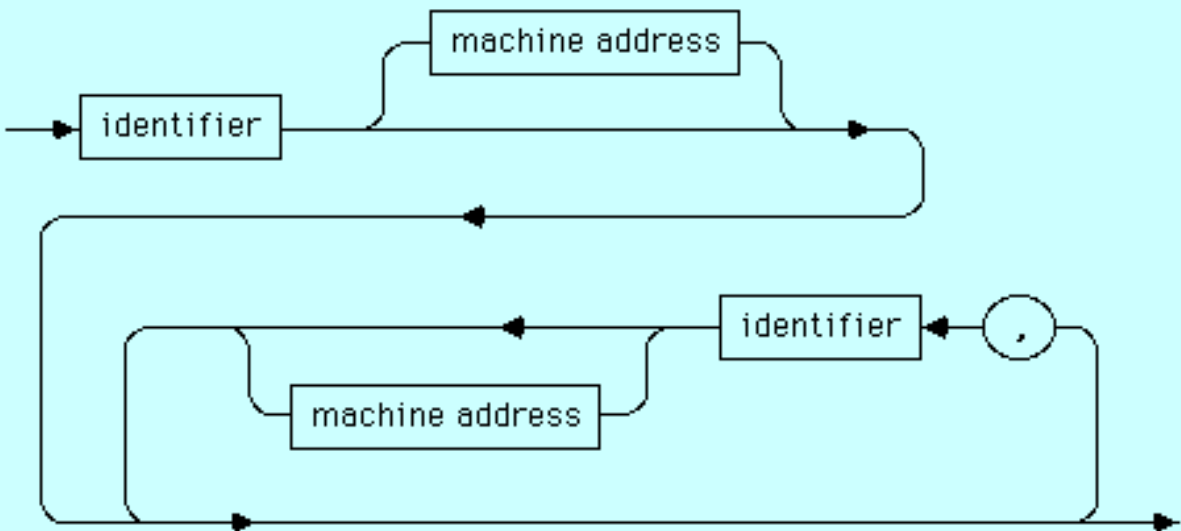


A2.2.2.6 Variable Declarations

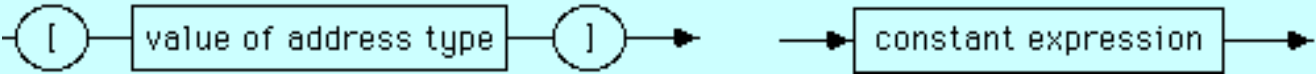
variable declaration



variable identifier list

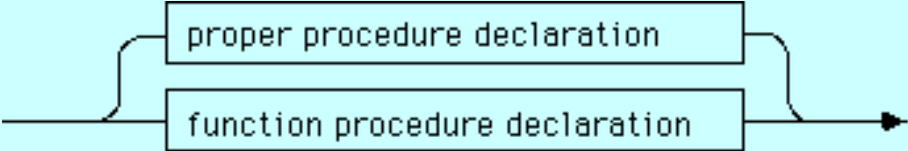


machine address value of address type



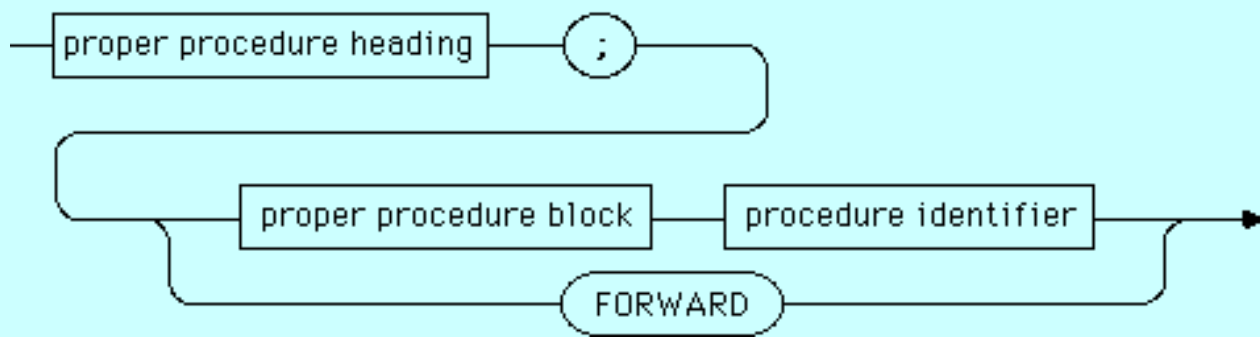
A2.2.2.7 Procedure Declarations

procedure declaration

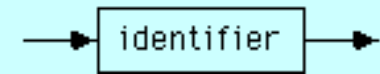


A2.2.2.8 Proper Procedure Declarations

proper procedure declarations

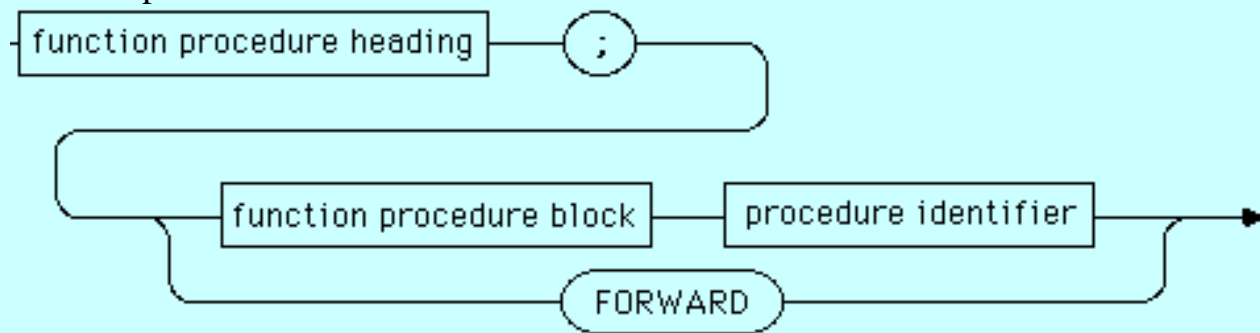


procedure identifier



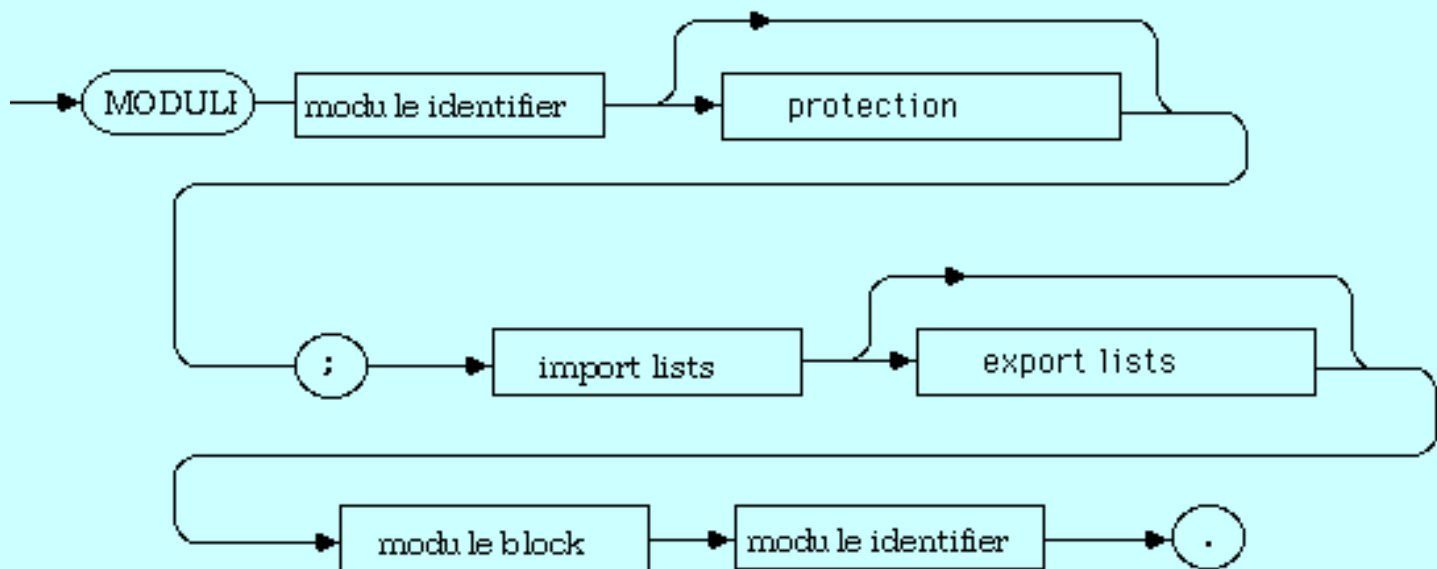
A2.2.2.8.1 Function Procedure Declarations

function procedure declaration



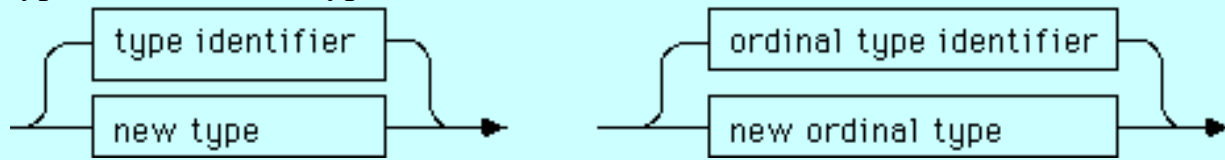
A2.2.2.9 Local Module Declarations

local module declaration



A2.2.3 Types

type denoter ordinal type denoter



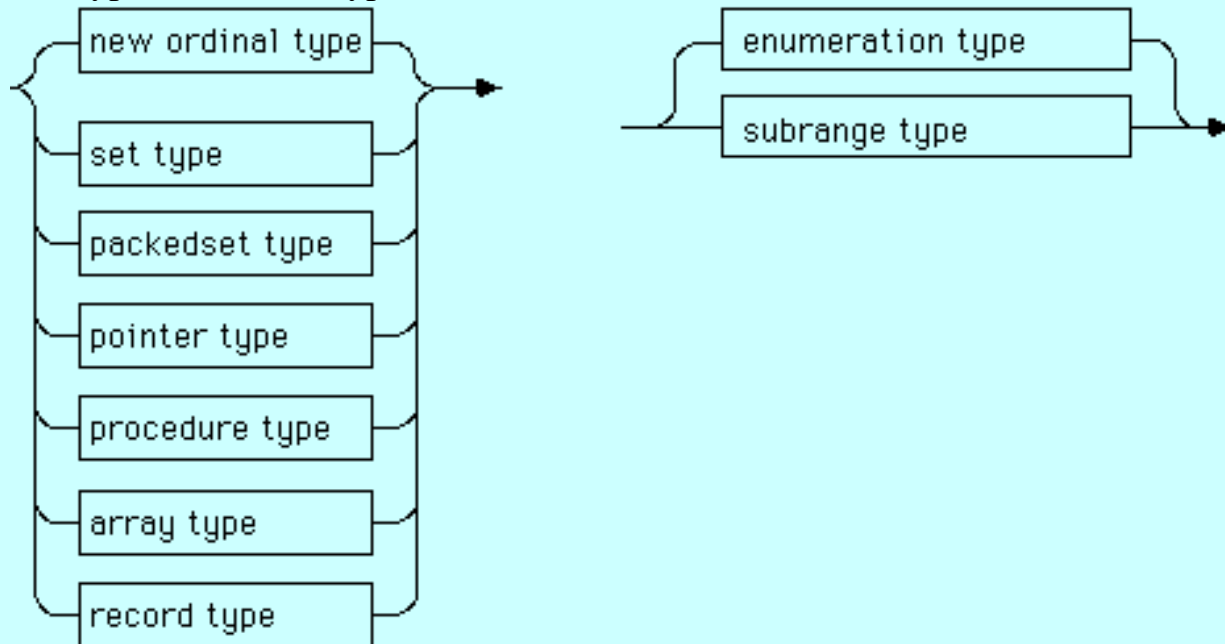
A2.2.3.1 Type Identifiers

type identifier ordinal type identifier



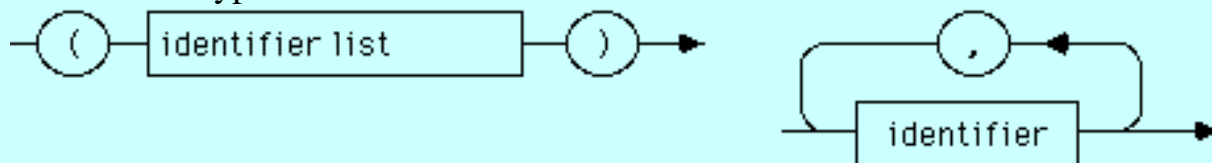
A2.2.3.2 New Types

new type new ordinal type



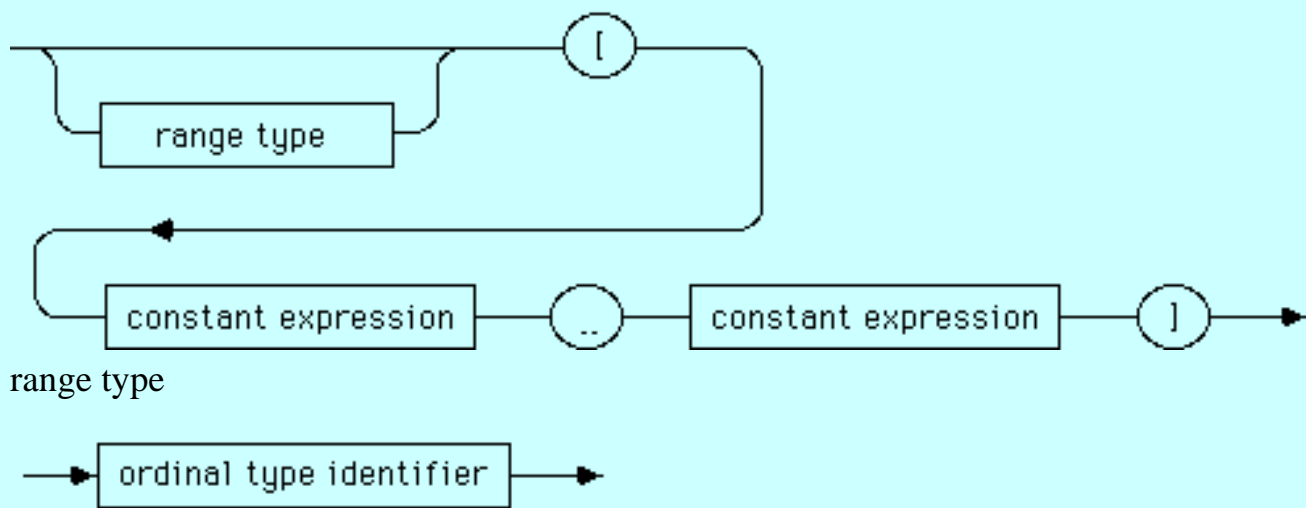
A2.2.3.2.1 Enumeration Types

enumeration type identifier list



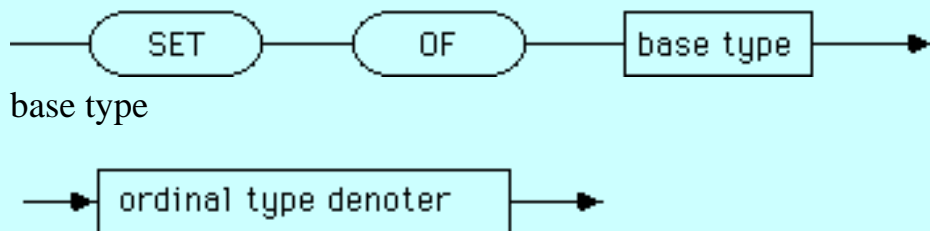
A2.2.3.2.2 Subrange Types

subrange type



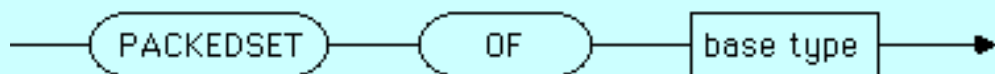
A2.2.3.2.3 Set Types

set type



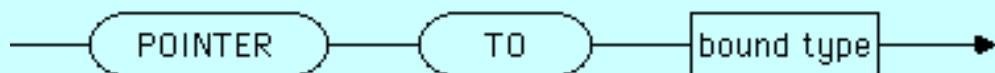
A2.2.3.2.4 Packedset Types

packedset type

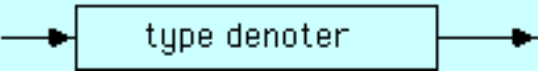


A2.2.3.2.5 Pointer Types

pointer type

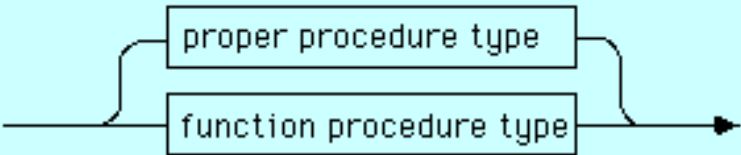


bound type

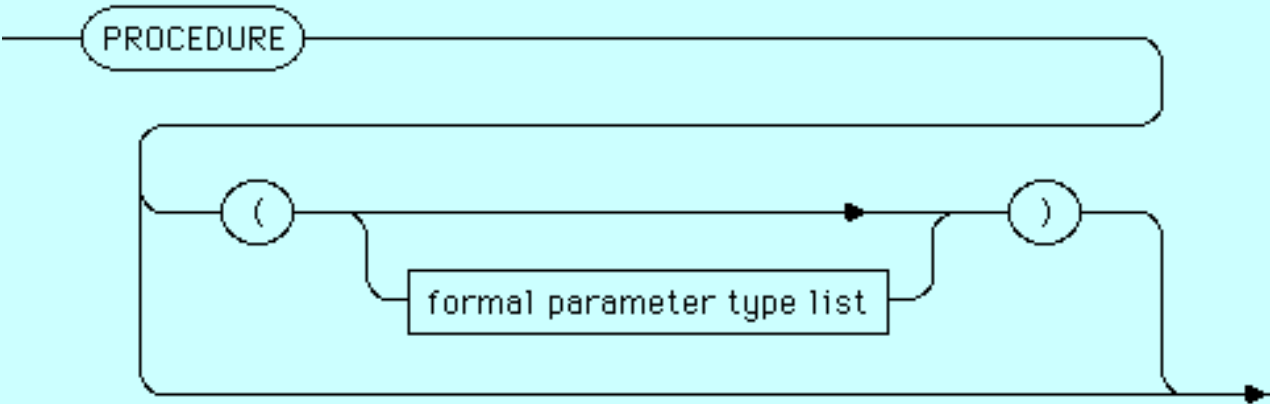


A2.2.3.2.6 Procedure Types

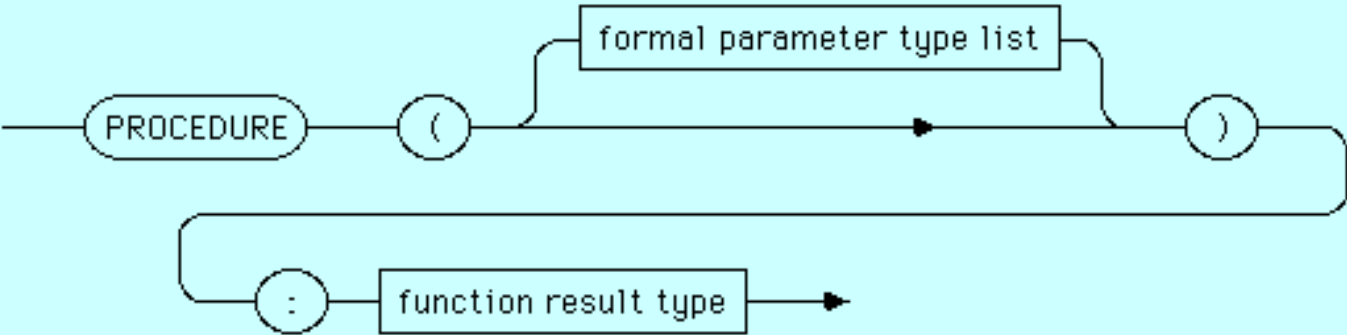
procedure type



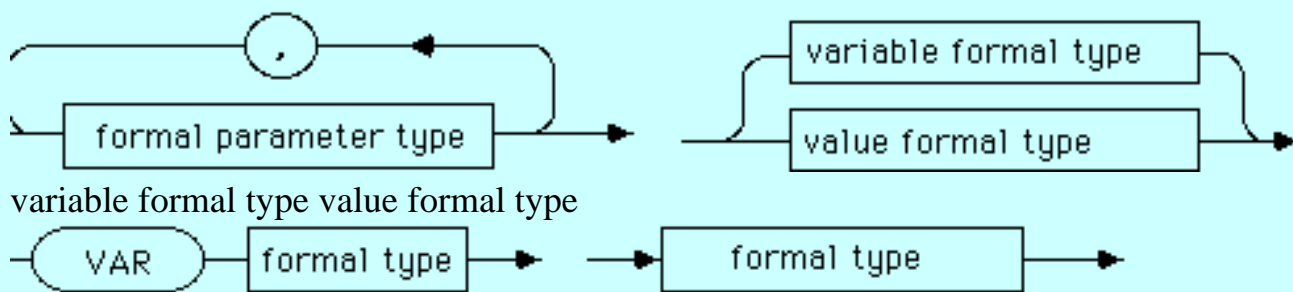
proper procedure type



function procedure type

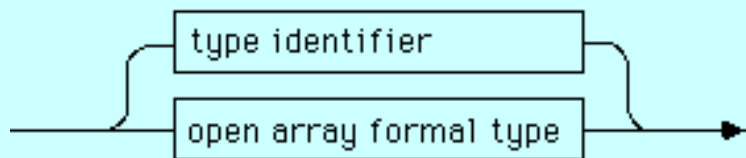


formal parameter type list formal parameter type

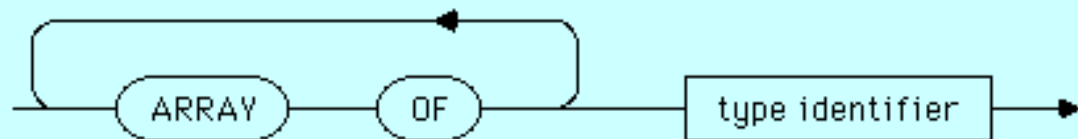


A2.2.3.2.7 Formal Types

formal type

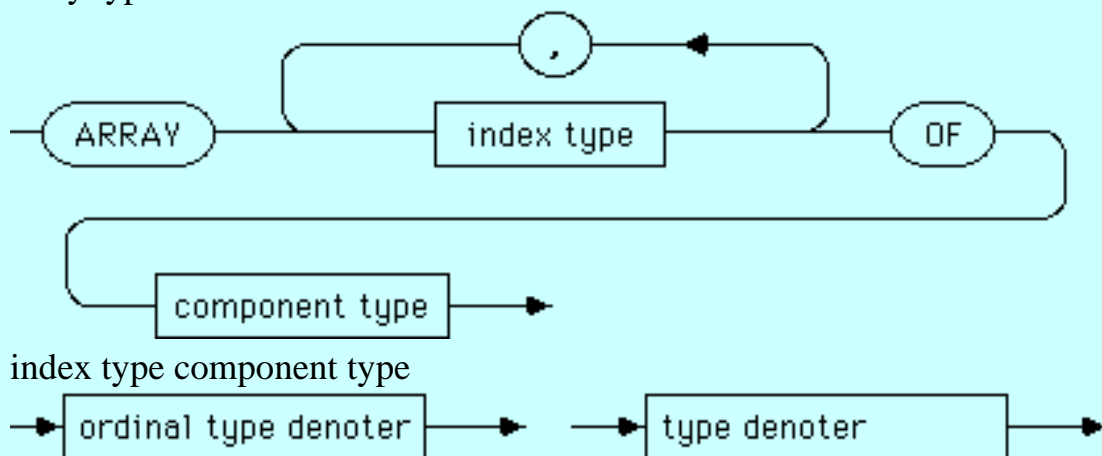


open array formal type



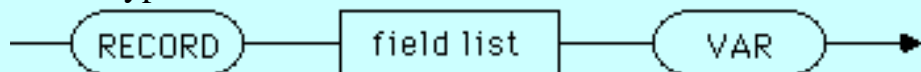
A2.2.3.2.8 Array Types

array type

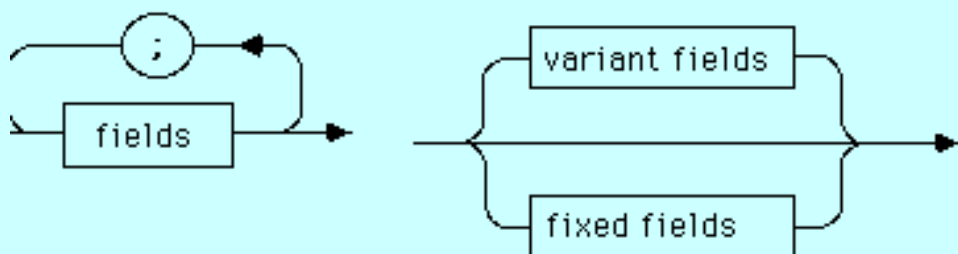


A2.2.3.2.9 Record Types

record type



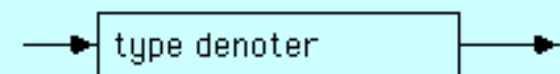
field list fields



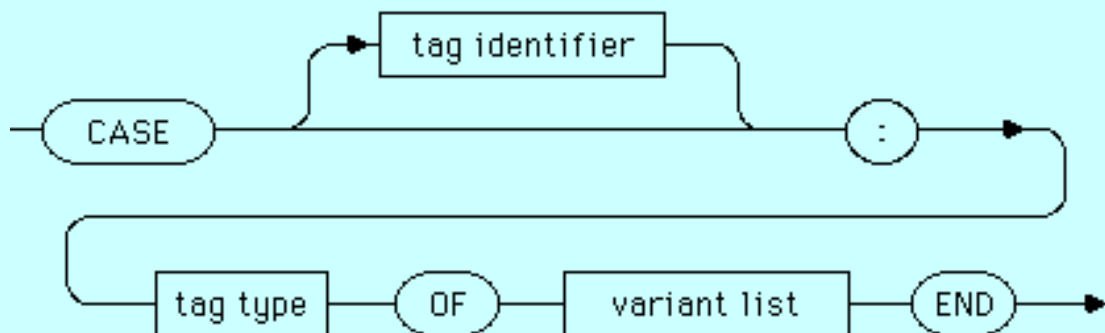
fixed fields



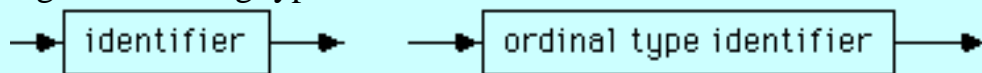
field type



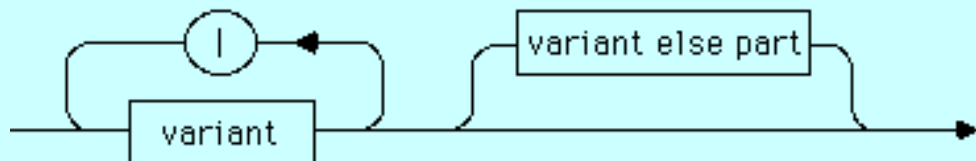
variant fields



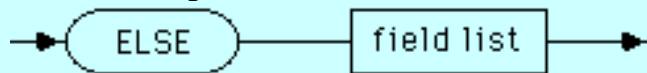
tag identifier tag type



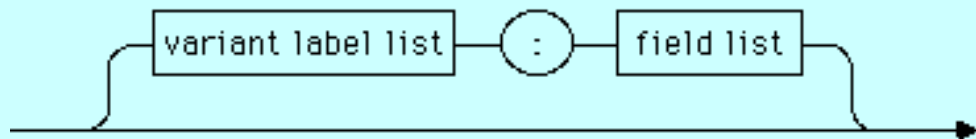
variant list



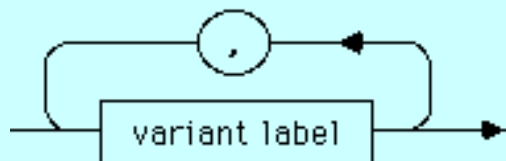
variant else part



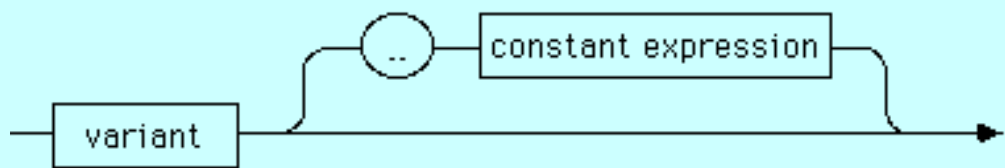
variant



variant label list



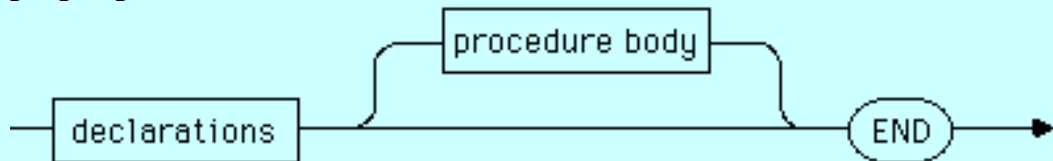
variant label



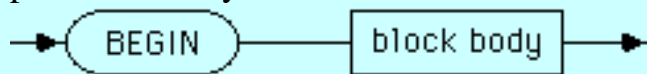
A2.2.4 Blocks

A2.2.4.1 Proper Procedure Blocks

proper procedure block

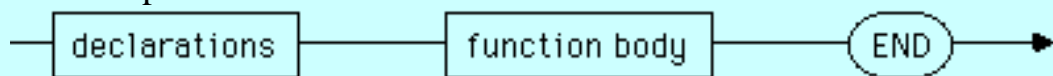


procedure body

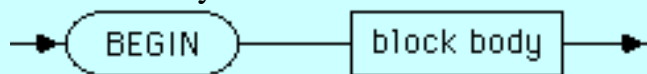


A2.2.4.2 Function Procedure Blocks

function procedure block

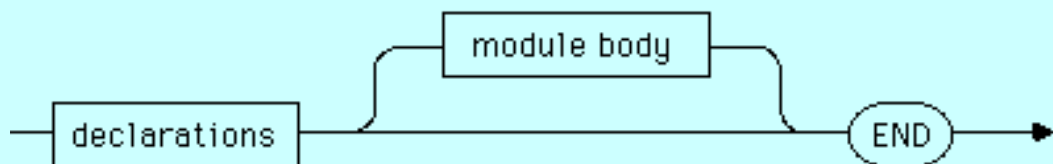


function body

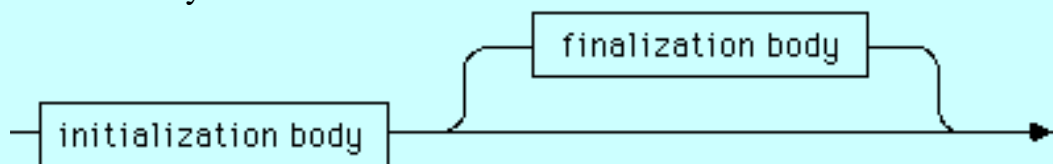


A2.2.4.3 Module Blocks

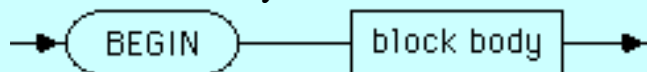
module block



module body



initialization body

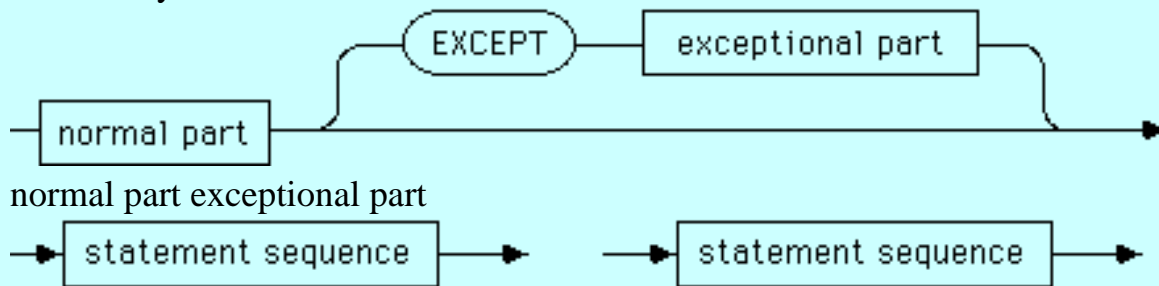


finalization body



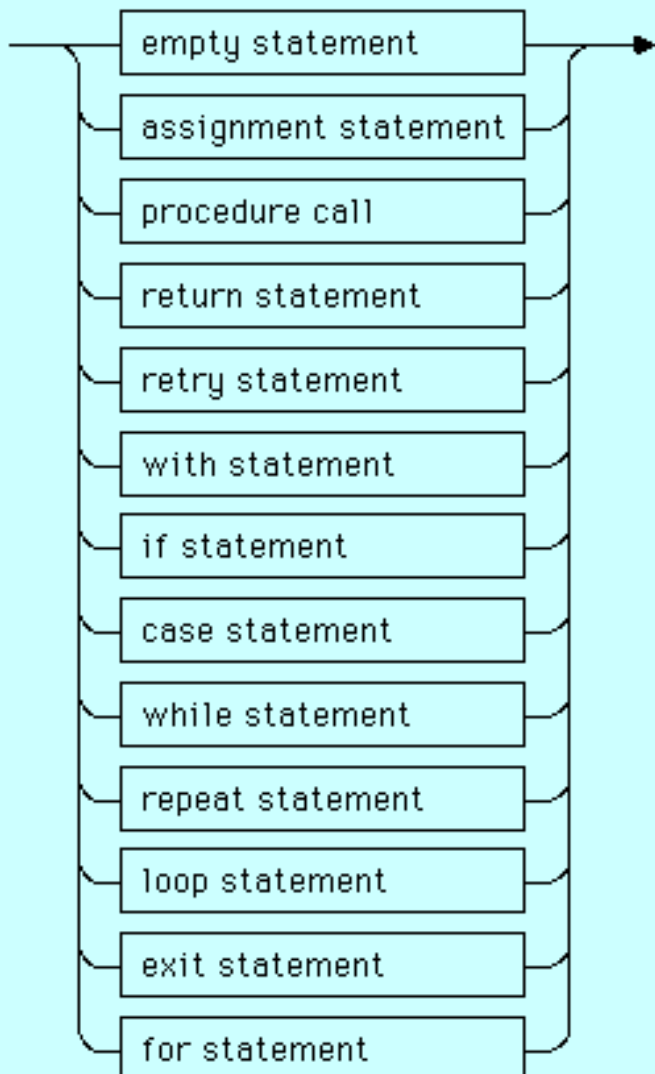
A2.2.4.4 Block Bodies and Exception Handling

block body



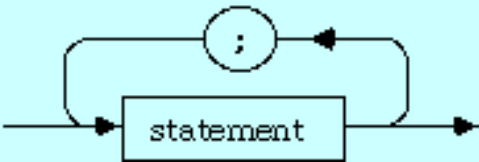
A2.2.5 Statements

statement



A2.2.5.1 Statement Sequences

statement sequence



A2.2.5.2 Empty Statements

empty statement

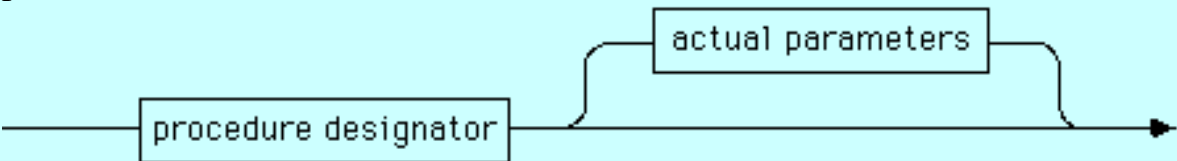
A2.2.5.3 Assignment Statements

assignment statement

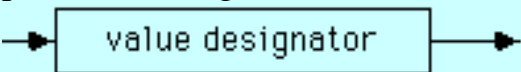


A2.2.5.4 Procedure Calls

procedure call

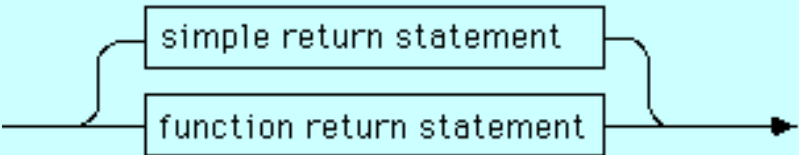


procedure designator



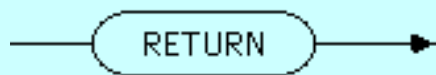
A2.2.5.5 Return Statements

return statement



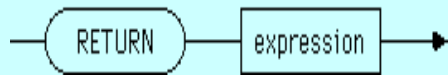
A2.2.5.5.1 Simple Return Statements

simple return statement



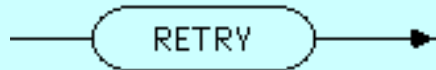
A2.2.5.5.2 Function Return Statements

function return statement



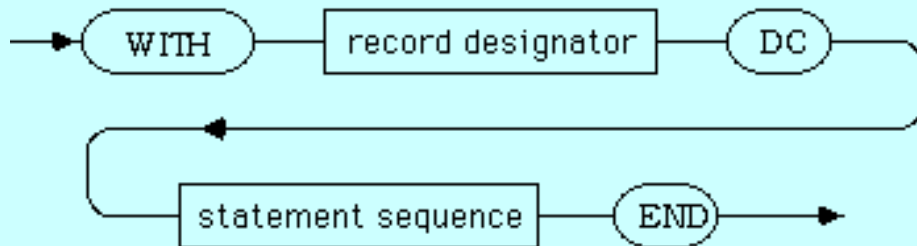
A2.2.5.6 Retry Statements

retry statement

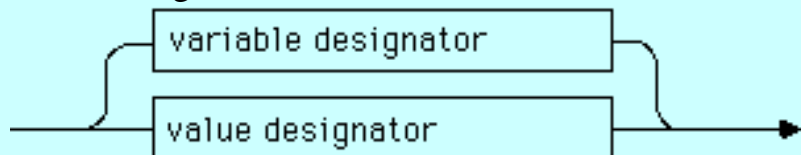


A2.2.5.7 With Statements

with statement

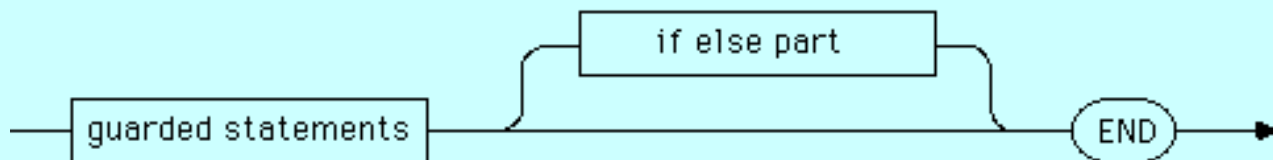


record designator

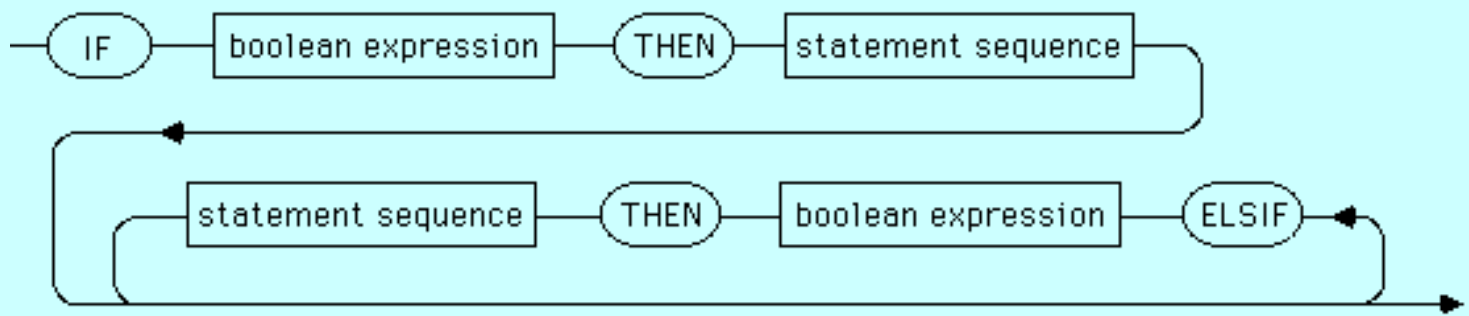


A2.2.5.8 If Statements

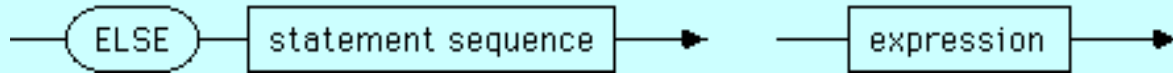
if statement



guarded statements

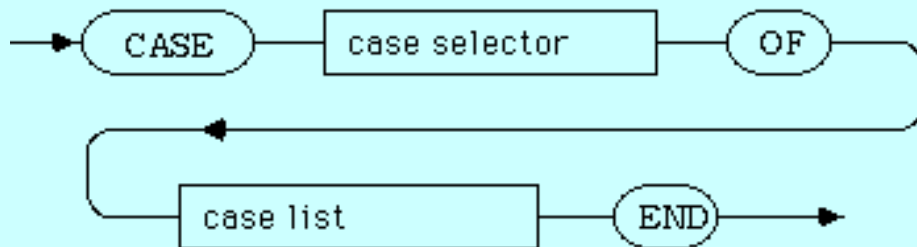


if else part boolean expression

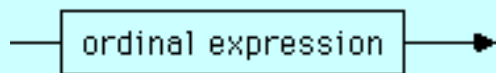


A2.2.5.9 Case Statements

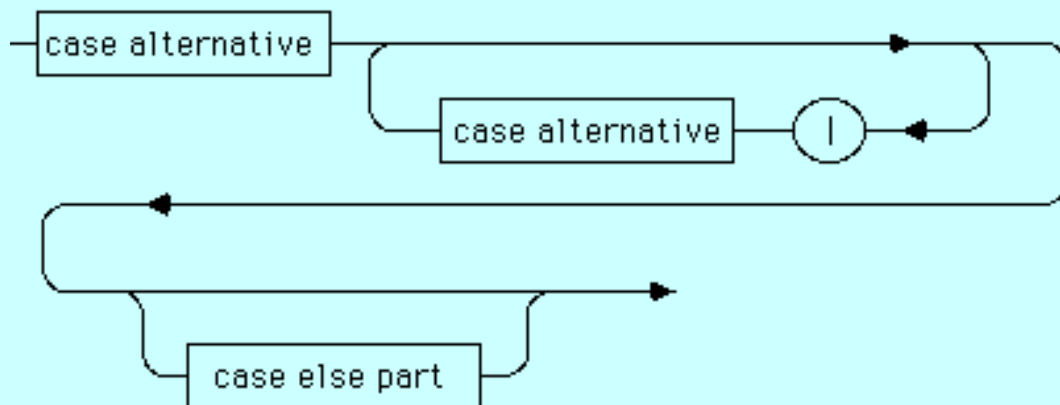
case statement



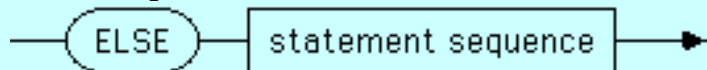
case selector



case list

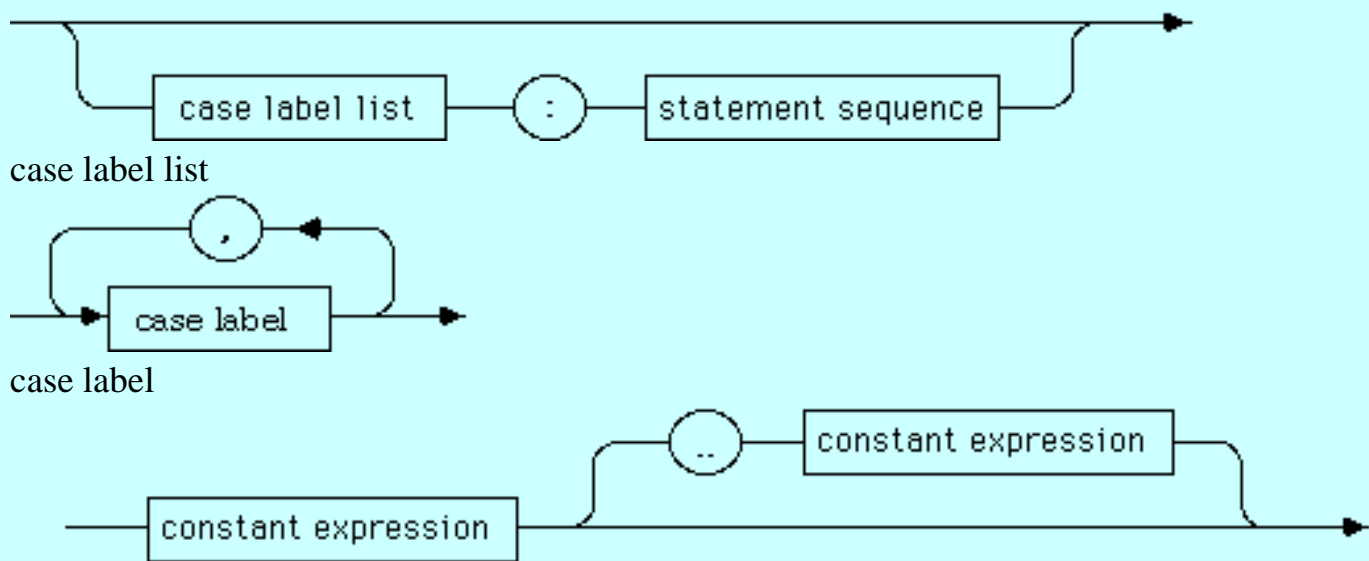


case else part

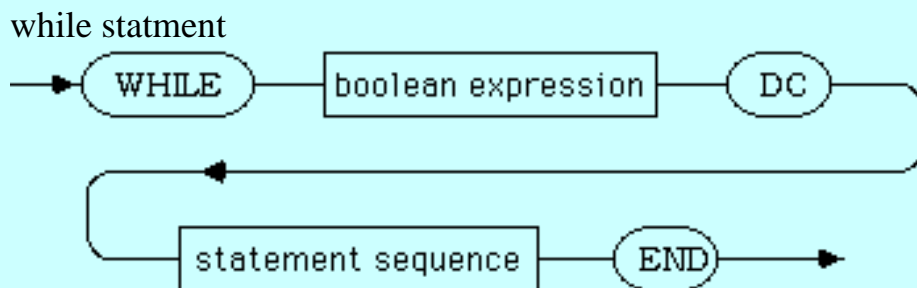


A2.2.5.9.1 Case Alternatives

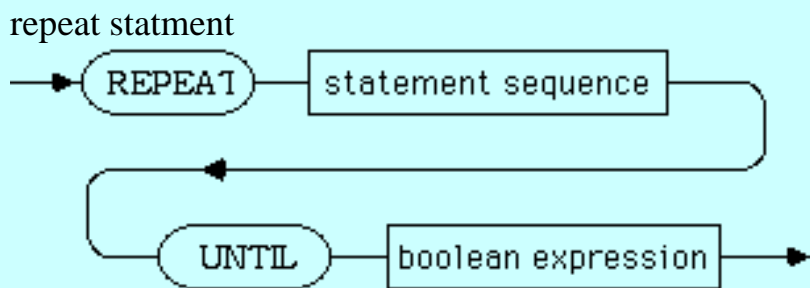
case alternative



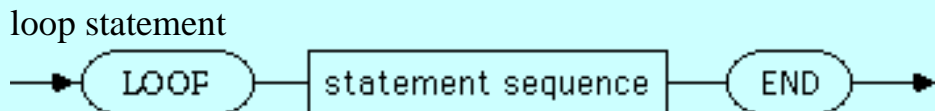
A2.2.5.10 While Statements



A2.2.5.11 Repeat Statements

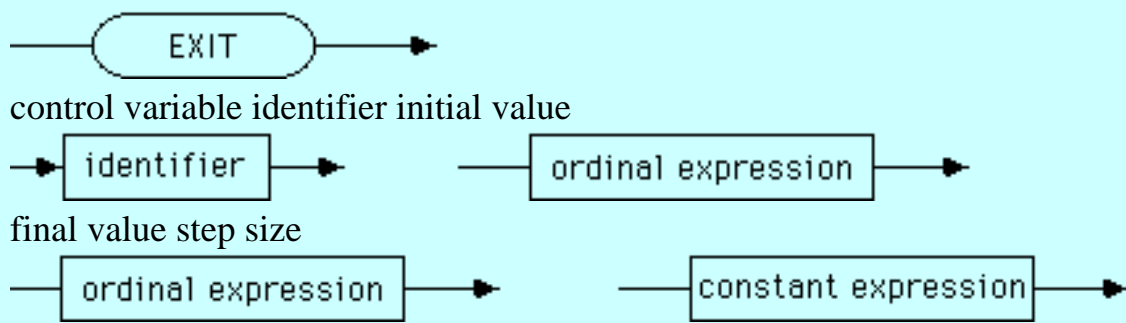


A2.2.5.12 Loop Statements

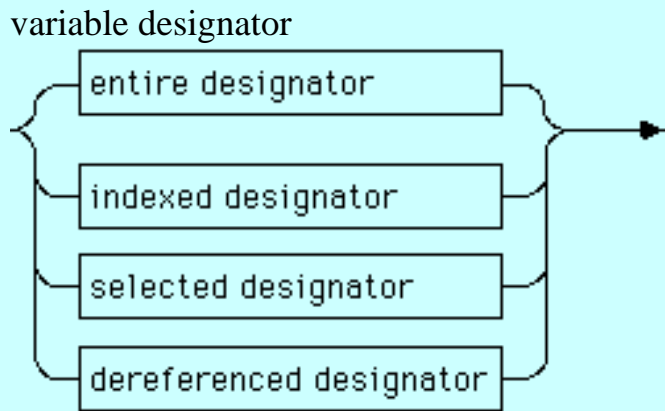


A2.2.5.13 Exit Statements

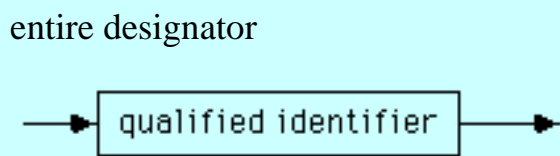
exit statement



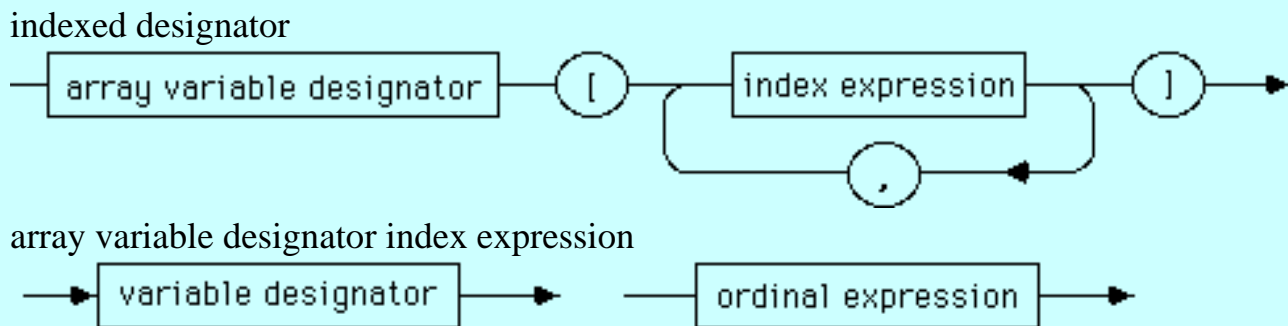
A2.2.6 Variable Designator



A2.2.6.1 Entire Designators

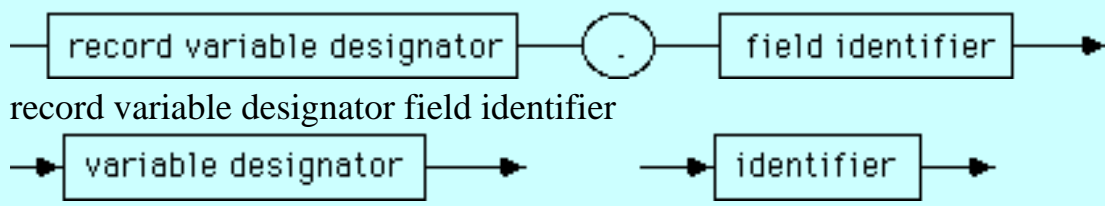


A2.2.6.2 Indexed Designators

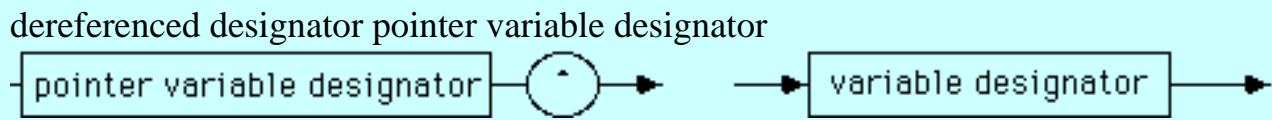


A2.2.6.3 Selected Designators

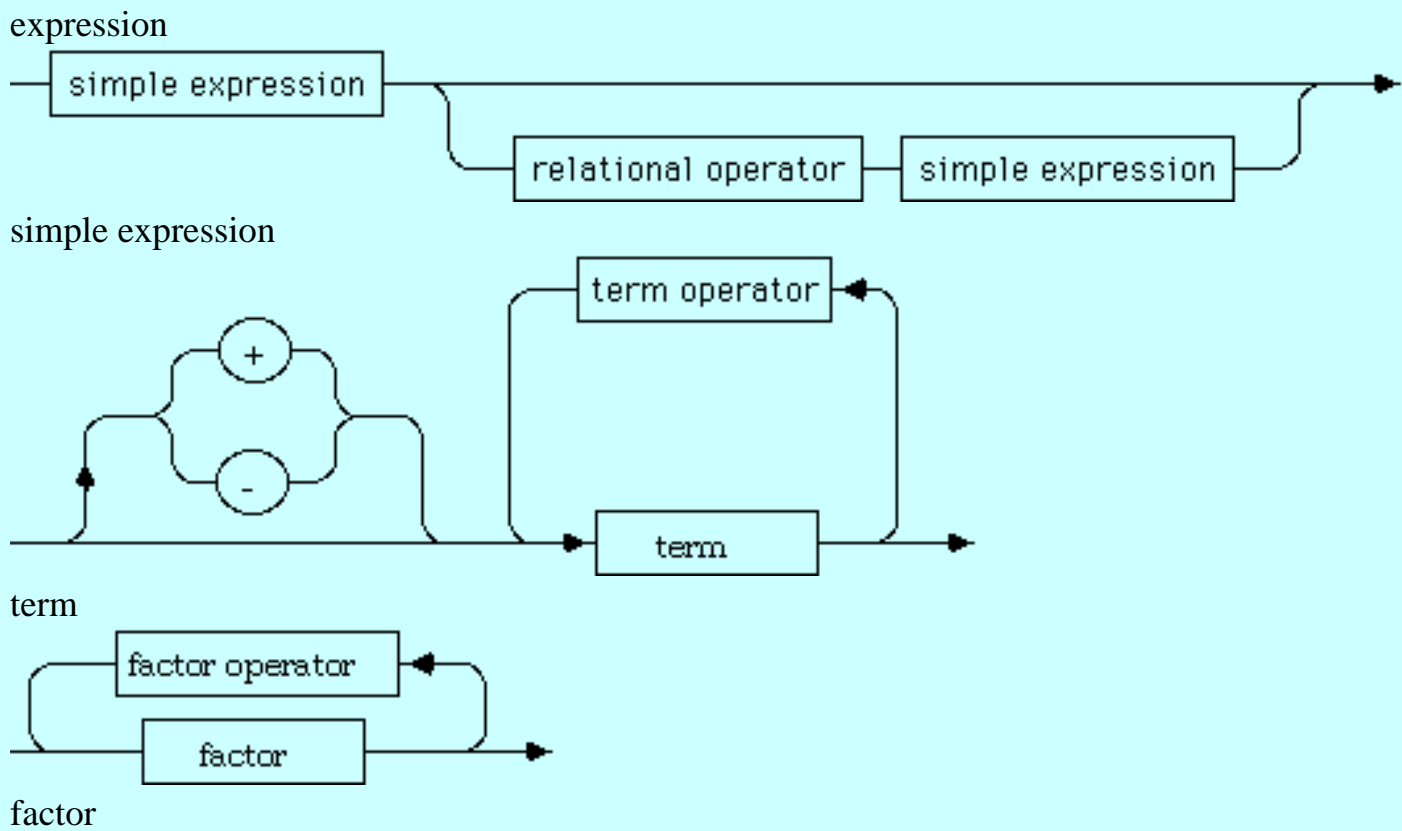
selected designator

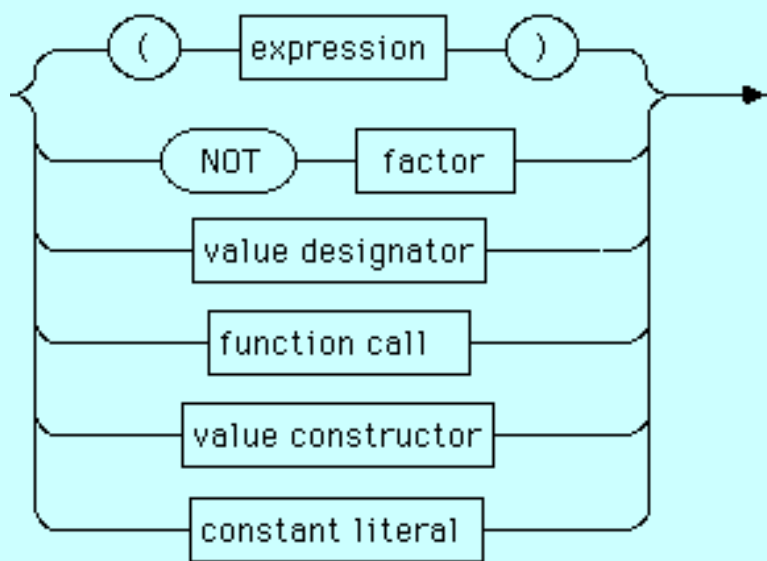


A2.2.6.4 Dereferenced Designators

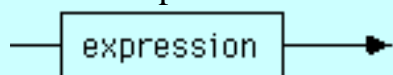


A2.2.7 Expressions



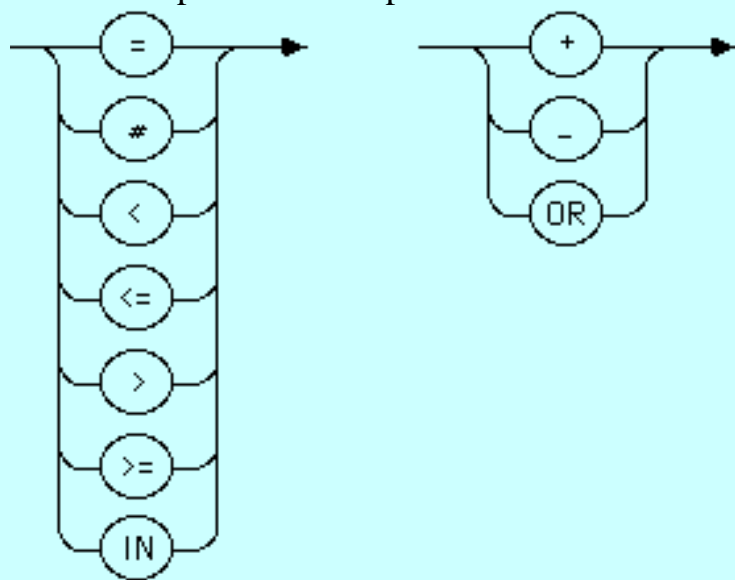


ordinal expression

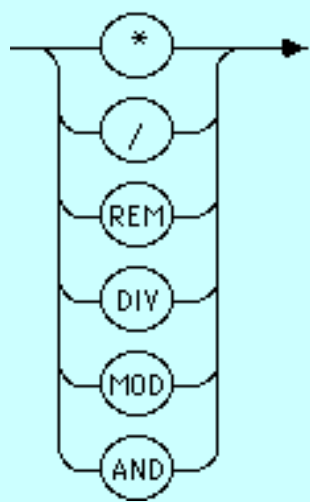


A2.2.7.1 Infix Expressions

relational operator term operator

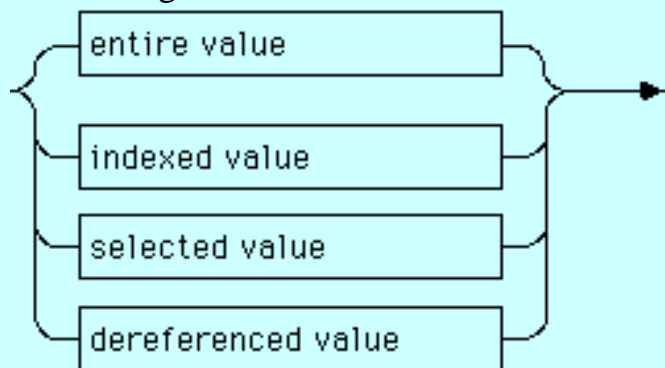


factor operator



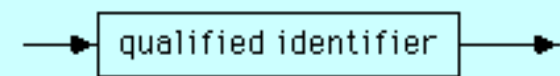
A2.2.7.2 Value Designators

value designator



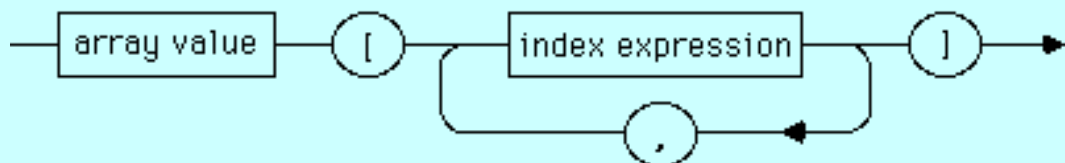
A2.2.7.2.1 Entire Value

entire value

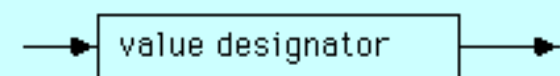


A2.2.7.2.2 Indexed Values

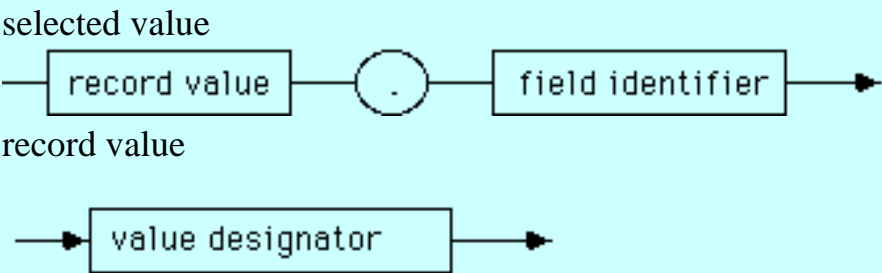
indexed value



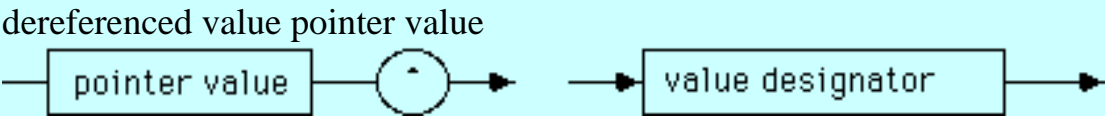
array value



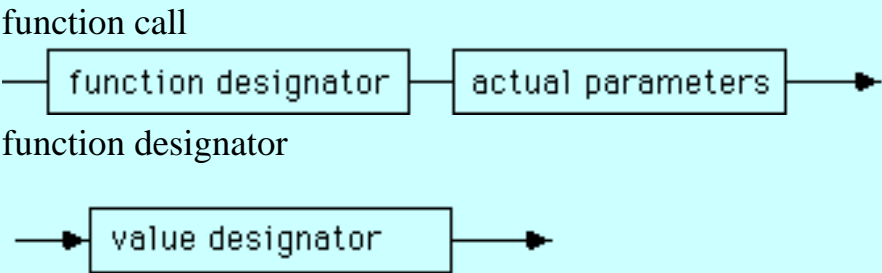
A2.2.7.2.3 Selected Values



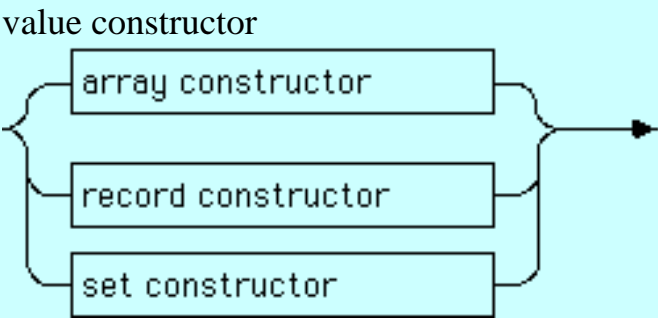
A2.2.7.2.4 Dereferenced Values



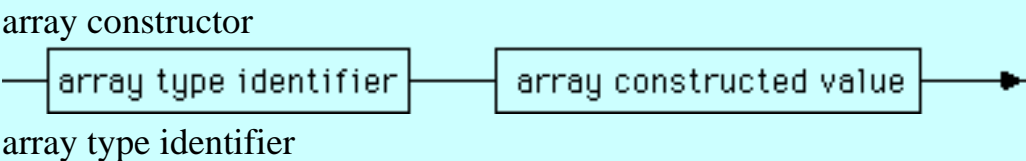
A2.2.7.3 Function Calls

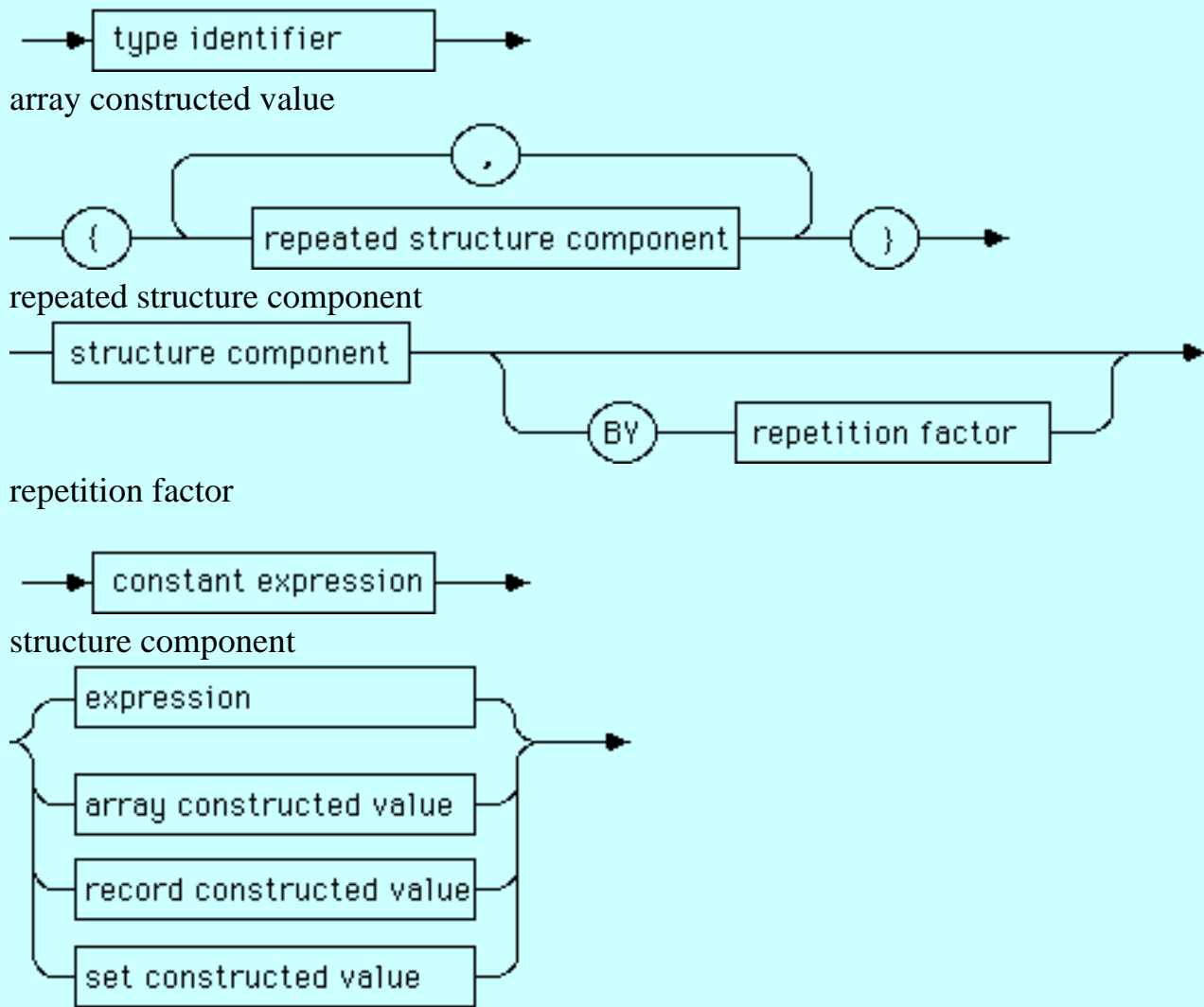


A2.2.7.4 Value Constructors

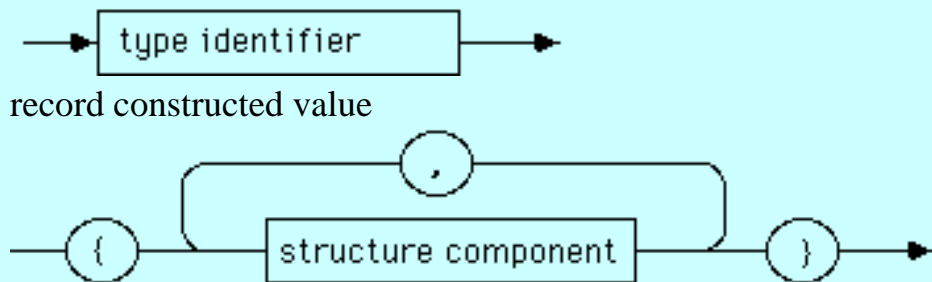
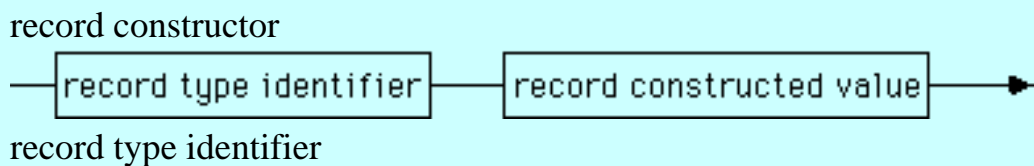


A2.2.7.4.1 Array Constructors



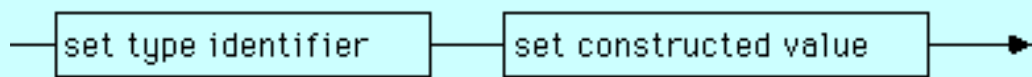


A2.2.7.4.2 Record Constructors



A2.2.7.4.3 Set Constructors

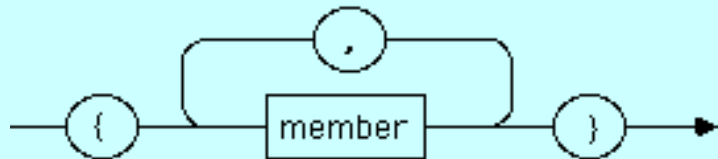
set constructor



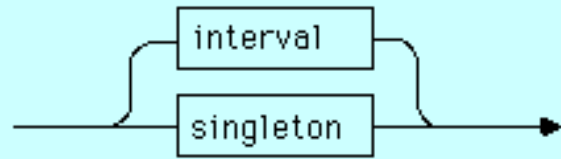
set type identifier



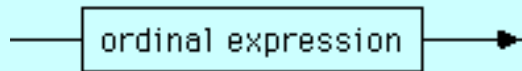
set constructed value member



interval

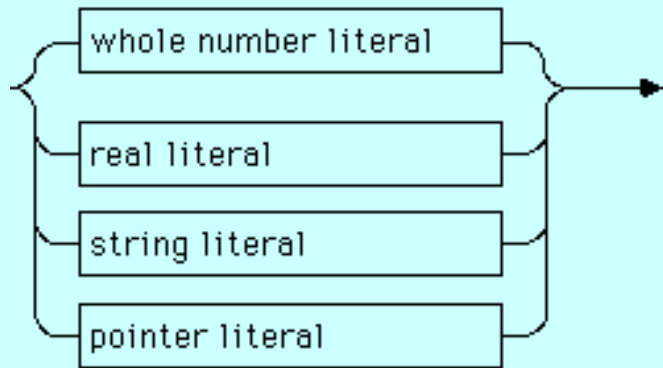


singleton



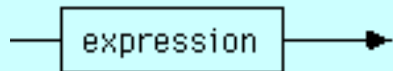
A2.2.7.5 Constant Literals

constant literal



A2.2.7.6 Constant Expressions

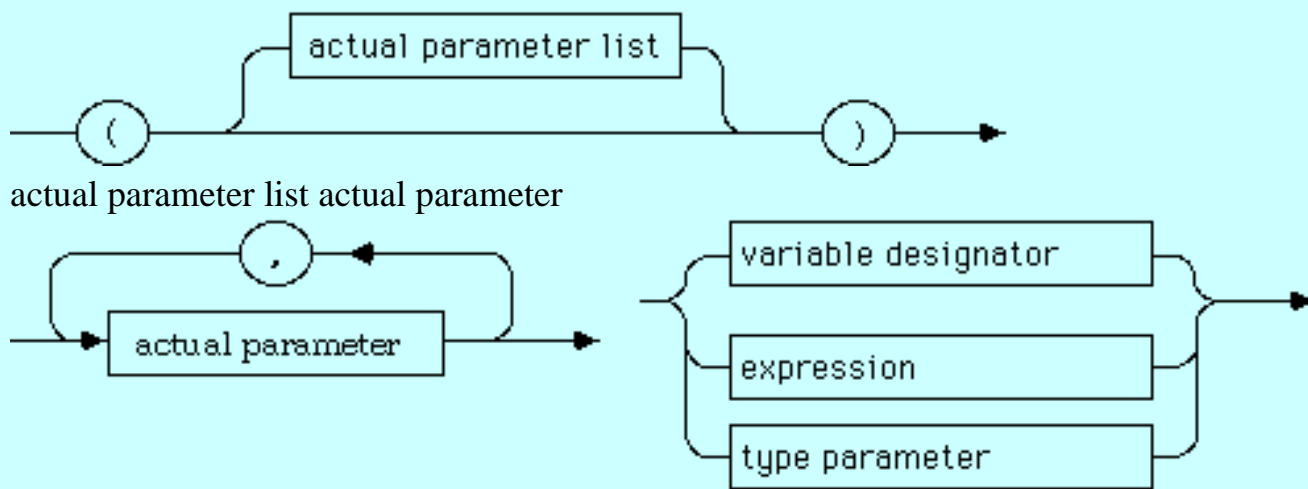
constant expression



A2.2.8 Parameters and Arguments Binding

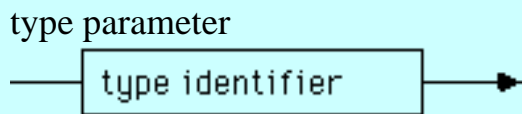
A2.2.8.1 Actual Parameters

actual parameters



A2.2.8.2 Special Parameters

A2.2.8.2.1 Type Parameters



[Contents](#)

ANSWER KEY

WARNINGS:

1. As of 2002 06 18, this is a third, corrected answer set.
 2. No guarantees are expressed or implied.
 3. Answers to only a few of the problems are provided, and none after chapter 12.
 4. To keep this document short, problem answers are very brief, and should be taken as coding suggestions only. They are not suitable as guides for style or commenting.
-

[Chapter 1](#)

[Chapter 2](#)

[Chapter 3](#)

[Chapter 4](#)

[Chapter 5](#)

[Chapter 6](#)

[Chapter 7](#)

[Chapter 8](#)

[Chapter 9](#)

[Chapter 10](#)

[Chapter 11](#)

[Chapter 12](#)

[Chapter 13](#)

[Chapter 14](#)

[Chapter 15](#)

[Contents](#)

CHAPTER 1

Questions

1. Top down design is the process of solving a problem by breaking it down to successively smaller steps whose interrelationship is clearly defined and then solving each of the steps. Bottom up design is focusing on the details, then tackling each one separately, then gradually welding the partial answers into a coherent whole that provides a solution to the whole problem.
2. Step 1: Write everything down. Step 2: Apprehend the problem. Step 3: Design a solution. Step 4: Execute the completed plan. Step 5: Scrutinize the results.
3. Since an abstraction can be roughly defined as deliberately not knowing details, one can consider driving a car an abstraction because one does not need to know how build an automobile or know the details of how the car works in order to drive one.
4. An abstraction is a task that is seen as an organic whole. Going back to the example of the logger, we see abstraction at work. Unexperienced people would detail the steps to take in order to perform the task. The professional logger however, will view it as a whole and not worry about individual steps. It is part of the professional's built in function.
5. Answers may vary, depending on whether one believes a human has a soul, and is more than merely a body.
6. Money is a medium of exchange that represents an underlying value; it has no value except by mutual agreement.
7. A paradigm is a way of looking at something by way of analogy or example. A model is a representation in more concrete or accessible form than the original. The word may also be also used of a scale model for some proposed project. A world view is a complete set of philosophic or religious presuppositions within which paradigms and individual abstractions are formed. A meme is a transmittable idea that is the basis of a social movement or a political philosophy.
8. Answers may vary.
9. Data abstractions. --data representation abstractions (data structures). --data manipulation abstractions (expression structure). Machine abstractions. --the computing apparatus itself (machine structures) --the instruction and manipulation of the machine (program structures).
10. Data is a conglomerate of facts that has been collected, and has no particular meaning. Information is data that has been assigned a particular meaning. It is useful data that can be understood and applied.
11. (a) compound (b) atomic (c) compound (d) compound (e) atomic (f) compound
12. (a) real (b) unsigned whole number (c) compound : pair of unsigned whole numbers (d) string (e) signed whole number (f) boolean (g) unsigned whole number (h) real (i) boolean (j) compound: set of whole numbers (k) depends on the type of y_2 and y_1
13. An ADT is a specified set of items with certain properties and operations in common.
14. There is a difference because of the limitation of the hardware. For some types, such as whole numbers, the abstract set is of infinite size. Abstract reals also have infinite precision. Neither property can be expressed in finite hardware.
15. A variable is a name whose value is subject to change during a course of a computation whereas a constant names a value that is fixed.
16. An expression is a combination of data items with various operators that are available for that data type. An expression is assigned a type according to the type of data produced when the expression is evaluated.
17. The major tasks of computing hardware are: (i) input (ii) memory (iii) processing (iv) control (v) output
18. Read Only Memory is permanently coded into a memory chip at the time of manufacturing. It never loses its data when power to the chip is lost. Random Access Memory contains data that is temporarily stored and can be changed. It loses its data if power is lost.
19. Hardware is the physical components, including the electronics, that make up the computer itself. Firmware are the

programs that are coded into ROM chips at time of manufacturing.

20. The virtual machine is the total environment presented to the user by the combination of hardware and software being employed at the moment.

21. The major task of the OS is to handle the disk drives and other I/O devices , and to provide an interface for programs.

22. Niklaus Wirth invented Pascal Modula-2, and Oberon.

23. COBOL is the language principally used for business applications.

24. A compiler is a program that takes code in text form text and translates it into machine language a single time for later execution. An interpreter is a program that translates into machine code as the program is run and every time it is run.

25. Sequence: one instruction following another in order. Selection: the choice among two or more alternative tasks depending on circumstances encountered when the solution is executed. Repetition/Iteration: a series of steps under the control of some condition (also known as a loop). There are 2 kinds<top-of-loop tested and bottom-of-loop tested.

Composition: letting the name of some code stand for the whole; one command containing a number of sequenced steps. Parallelism: The ability for code to be implemented on many processing devices simultaneously.

26. In top-of-loop testing, the test for exiting is made in the beginning of the loop. For example, a WHILE loop tests if the condition is met then proceeds if it is. However in a bottom-of-loop testing, the test is considered at the end of the loop, as in a REPEAT loop.

27. The advantage of writing programs in pseudocode before actual coding is as follows: (i) One need not pay particular attention at this stage to the specific grammatical details of the actual code in a particular notation. (ii) The pseudocode is general enough so that the solution can later be expressed in any one of several different actual coding notations. (iii) Writing in pseudocode forces the programmer to pay sufficient attention to detail to ensure that the solution is completely thought out. (iv) The pseudocode is easy to examine for possible efficiency improvements and for the elimination of logical errors.

28. An algorithm is a technique to solve a problem expressed as a series of steps or instructions.

29. The syntax of a programming notation consists of a set of legal symbols, together with the grammatical rules expressing how those symbols may be employed to write correct programs. Semantics on the other hand is the meaning of code, either in the context of the notation or in that of a program.

30. Syntax errors are errors caused by incorrect spelling, misplaced or missing punctuation (such as semicolons) or otherwise incorrect use of some part of the notation. It can be prevented by good proof reading.

Logical errors are caused by insufficient planning, fuzzy thinking, or poor program organization. They are also caused by a failure to express the meaning of the problem in a fashion that can be translated into a solution. These can be corrected by good planning, organization, and development.

Problems

31. (a) 522 (b) -0.8 (c) 47 (d) true (e) true (f) false (g) bad expression: mixed whole and boolean (h) bad expression: mixed whole and real (i) bad expression: mixed boolean and whole (j) 7.0

32.

```
start at zero
set a counter to zero
while counter < 20
    set number variable to counter times itself
    print counter
    increase counter by 1
end while
```

33.

```
read in number
if number mod 2 = 0
    even
else
    odd
```

34.

```
Pass01: 1, 1
Pass02: 2, 1
        2, 2
Pass03: 3, 1
        3, 2
        3, 3
.
.
.
Pass10: 10, 1
        .
        .
        .
        10, 10
```

35.

```
SumOfNSquares
    ask user for number
    read number
    set counter to 1
    set sum of square to 0
    while counter <= number
        add counter * counter to sum of square
        increase counter by 1
    end while
end SumOfNSquares
```

36.

```
SumOf10
    set counter to 1
    set sum to 0
    while counter <= 10
        read n
        add n to sum
        increase counter by 1
    end while
```

```
end SumOf10
```

37.

```
SumOfN
  ask user for number
  read number
  set counter to 1
  set sum to 0
  while counter <= number
    add counter to sum
    increase counter by 1
  end while
end SumOfN
```

38.

```
SumNSumofSquare
  ask user for number
  read number
  set counter to 1
  set sum to 0
  set sum of square to 0
  while counter <= number
    add counter to sum
    add counter * counter to sum of square
    increase counter by 1
  end while
end SumNSumofSquare
```

39.

```
Sort3
  set counter to 1
  if first > second
    set temp to second
    set second to first
    set first to temp
  end if
  if first > third
    set temp to third
    set third to first
    set first to temp
  end if
  if second > third
    set temp to third
    set third to second
    set second to temp
  end if
```

```
    end if
end Sort3
```

40.

```
compute average
  read n (* number of reals to do *)
  set count to 1
  set partial sum to 0
  while count <= n
    read currentReal
    add currentReal to partial sum
    add 1 to count
  end while
  set average to partial sum/ n
  write out average
end compute average
```

41.

```
FindSequence
  set total to number of items
  fetch first number
  set small to first number
  set large to first number
  set counter to 2
  while counter <= total
    fetch current number
    if current number < small
      set small to current
    end if
    if current number > small
      set large to current
    end if
    increase counter by 1
  end while
end Find Sequence
```

42.

```
LargeNSmall
  read n
  set count to 0
  set sum to 0
  fetch first number
  set largest to first
  set smallest to first
  while count <= n
```

```

    read current
    add current to sum
    if current > LargestNumber
        set LargestNumber to current
    end if
    if current < SmallestNumber
        set SmallestNumber to current
    end if
    increase count by 1
end while
set average to sum/n
print average
print LargestNumber
print SmallestNumber
end LargeNSmall

```

43.

```

MatrixAddition
    set row to 0
    set col to 0
    set sum to 0
    while row < 3
        while col < 5
            set sum to sum + matrix[row, col]
            increase col by 1
        end while
        increase row by 1
    end while
end MatrixAddition

```

44.

```

MatrixAddition
    set row to 0
    set col to 0
    set sum to 0
    read numRows
    read numcols
    while row < numRows
        while col < numcols
            set sum to sum + matrix[row, col]
            increase col by 1
        end while
        increase row by 1
    end while
end MatrixAddition

```

45.


```

Project
  set index to 1
  while index <= 12
    set b(index) to a(index) * 1.05
    increase index by 1
  end while

  (* or using a matrix more explicitly
  set index to 1
  while index <= 12
    set table(b, index) to table(a, index) * 1.05
    increase index by 1
  end while *)

end Project

```

46.

```

3Fields
  set row to 1
  set col to 1
  while row <= lastItemNumber
    set table(row, col+2) to table(row, col) - table(row, col+1)
    increase row by 1
  end while

  (* or, if the columns are labeled, use
  set table(row, netprofit) to table(row, reverse) - table(row, expense)
  without a column variable *)

end 3Fields

```

47.

```

3FieldsB
  set row to 1
  set col to 1
  while row <= lastItemNumber
    set table(row, col+1) to table(row, col) * 0.07
    set table(row, col+2) to table(row, col) + table(row, col+1)
    increase row by 1
  end while

  (* same option will apply for question 46 *)

end 3FieldsB

```

CHAPTER 2

Questions

1. (a) Editor: a program that produces a text version of a program in the high level notation. (b) Compiler: a program that translates the text version into machine-readable form. (c) Linker: a program that merges previously compiled code and the current code to generate an executable program. (d) Interpreter: a program similar to a compiler but that translates code into machine language every time the program is run.
2. Input and output functions are hardware dependent, so programs written for one kind of computer are unlikely to work on another. By having I/O in the library, only the library has to be changed to port code to another computer.
3. MODULE --> name --> IMPORT commands --> BEGIN --> END.
4. Answers may vary. Check for such characters as \$.
5. Answers may vary. Non-standard types may be found in SYSTEM.
6. Answers may vary.
7. Illegal identifiers: Reason: Result#1 # is not allowed Execute User space not allowed 12Dozen can't start with a digit MODULE reserved word Canada'sBest quote mark is not allowed
8. An identifier is a name for a value or an action. In Modula-2 it is a sequence of non-blank letters or digits beginning with a letter.
9. A statement is an instruction directing a computer to take some action. It usually is written out in a high level notation for subsequent translation by the computer into one or more low level instructions and their corresponding actions.
10. There are two "standards", the de facto standard of classical Modula-2 as defined by Professor Wirth's books and interpreted by implementors, and the de jure standard represented by the ISO document 10514, parts 1, 2, and 3.
11. Reserved Word: A special word or marker used to outline the structure of a program in the manner of punctuation. It must be entirely written in uppercase letters, and cannot be used for any other purpose. Standard Identifier: A name that is built-in to the notation. It must be written entirely in capital letters. It can be re-used, through this is unwise Standard Library Identifier: an identifier that names a particular library item in every standard implementation of Modula-2.
12. Literal: any entity whose name is an encoding of its value Constant: an identified value that remains the same throughout a program; that is, it is not subject to change or later re-assignment. Variable: a name for a memory location, the contents of which can be changed by a program.
13. Answers may vary.
14. Constants are declared only in the beginning of the program code. The "=" operator is used to assign names to constant values. Variables are assigned whenever needed in the program. The ":=" operator is used to assign values to variables.
15. Modula-2 expression: a combination of literals, constants, and variables using addition, subtraction,

multiplication, division, and/or such other operations as may be defined for the type of the entities being combined.

16. To say they are assignment but not expression compatible means that one can assign **CARDINAL** and **INTEGER** values to variables of types each other's as long as they fall in the same range, though it may cause overflow problems at run time. That they are not expression compatible, means that one cannot use the two different data types in one expression.

17. (a) `card := card + VAL(CARDINAL, int);` or `card := card + TRUNC (int);`

(b) `re := FLOAT (int);`

(c) `lre := LFLOAT (re);`

(d) `lre := LFLOAT (card);`

(e) `re := FLOAT (card);`

(f) `re := FLOAT(6);`

(g) This one probably cannot be fixed, unless -5 is replaced by 5 or we write `int := INT(-5)`, both of which seem trivial.

18. (a) 36

(b) 1

(c) error: division by 0

(d) -12

(e) -53

(f) -163.8

(g) error: real subtract whole; incompatible types

(h) 8.4

(i) error: divisor in MOD has to be positive

(j) 11.0

(k) 4.5 E+13

(l) 4.0 E+15

(m) 5.0 E-15

(n) 10.9

(o) error: divisor is a cardinal (2 different data types)

(p) error: mixing integer and cardinal types

(q) error: mixing real and longreal types.

19. Note: `_` will indicate spaces.

(a) `_ 4 _ -2`

(b) `answer _ _ -6`

(c) `_ _ 1`

`_ _ 2`

`_ _ 3`

`...infinite loop`

(d) `_ _ 0` then the program will not write any others because counter will be an integer and `WriteCard` does not handle integers, so the program crashes with an overflow error.

(e) `_ _ 3 _ _ 6 _ _ 12`

Problems

Note: Not all problems are shown. Most problems are left to students as labs.

```
(* Created
    July.07.1999
    Chapter 2, Question 21 *)

MODULE LbsToKg;

FROM STextIO IMPORT
    WriteString, WriteLn;
FROM SRealIO IMPORT
    WriteFixed;

VAR
    convert : REAL;

BEGIN
    (* Conversion: 1lb = 0.453592Kg *)
    convert = 4.0 * 0.453592;
    WriteString ("4 pounds is equivalent to");
    WriteFixed (convert, 3, 0);
END LbsToKg.
```

```
(* Created
    July.07.1999
    Chapter 2, Question #22 *)

MODULE mphToms;

FROM STextIO IMPORT
    WriteString, WriteLn;
FROM SRealIO IMPORT
    WriteFixed;

VAR
    convert : REAL;

BEGIN
    (* Conversion: 1mi = 1606.344m; 1hr = 3600s *)
    convert = 45.0 * 1606.344 * 3600;
    WriteString ("45 miles per hour equivalent to");
    WriteFixed (convert, 3, 0);
    WriteString (" meters per second");
```

END mphToms .

CHAPTER 3

Questions

1. A BOOLEAN variable can have either a TRUE or FALSE value.
2. Order of operations for Modula-2 Boolean expressions:
First NOT (~)
Second *, /, DIV, MOD, AND (&)
Third +, -, OR
Last the relational operators =, <>, <, <=
3. (a) FALSE (b) TRUE (c) TRUE (d) TRUE
4. (a) TRUE (b) TRUE (c) TRUE (d) FALSE (e) TRUE (f) FALSE
5. Change to:

```
IF finished
  THEN
    a := a + 1;
  END;
```

6.

```
REPEAT
  WriteString ("enter a cardinal here ==>");
  ReadCard (myCard)
  flag := ReadResult () = allRight;
  SkipLine
UNTIL ~flag;
```

7. The count will never equal 0 because decreasing an odd number by 2 will never result in an even number.
8. Unless count is a CARDINAL, nothing is wrong because the count will eventually be less than 0.
9. Variables used in the DEC (or INC) procedure have to be of type INTEGER or CARDINAL.
10. realCount = 0 has to be changed to realCount = 0.0.
11. STextIO:
ReadString: Any read into a compatible variable will be legal because all characters can be part of a string.
ReadChar: Any read will also be all right for the same reason, however only the first character will be read if there is a SkipLine following.
SWholeIO:
ReadCard;
ReadCard (x):
flag := ReadResult () = allRight;
SRealIO:
ReadReal;
ReadReal (y);
flag := ReadResult () = allRight;
12. In the import section...

```
FROM RealInOut IMPORT
```

```
    Done;  
IMPORT InOut;
```

```
In the code...  
    refer to the first as Done;  
    refer to the second as InOut.Done;
```

13. Answers may vary.
14. Answers left to the student.
15. (i) Include temporary print statements in the code.
(ii) Test a program with alternate data.
(iii) Hand check results for reasonability.
(iv) Check loops for efficiency and correctness.
(v) Use comments.
(vi) Watch for spelling and punctuation errors.

Problems

Note: Not all problems are shown. Most problems are left up students as labs.

```
(* Created  
   June 10, 1999  
   Chpater 3 Question 16  
   ERROR TRAPPING *)
```

```
MODULE Convert;
```

```
FROM STextIO IMPORT  
    WriteString, WriteLn, ReadChar, SkipLine;  
FROM SRealIO IMPORT  
    ReadReal, WriteFixed;  
FROM SWholeIO IMPORT  
    ReadCard;  
FROM SIOResult IMPORT  
    ReadResult, ReadResults;
```

```
VAR  
    option : CARDINAL;  
    toConvert, converted : REAL;  
    exit : CHAR;  
    ok : BOOLEAN;
```

```
CONST  
    inchFactor = 2.54;      (* conversion factor: lin. = 2.54cm *)
```

```
BEGIN  
    REPEAT  
        (* Ask to choose what they want to convert *)  
        REPEAT  
            WriteString ("Choose one of the following options:"); WriteLn;  
            WriteString ("1)    Convert inches to centimeters."); WriteLn;  
            WriteString ("2)    Convert centimeters to inches."); WriteLn;
```

```

WriteLn;
WriteString ("Choice: ");
ReadCard (option);
ok := ReadResult() = allRight;

IF NOT ok THEN
    WriteString ("Not an option!! Retry");
    WriteLn;
END;

SkipLine;
UNTIL ok;

(* Get number to convert *)
ok := FALSE;

WHILE NOT ok DO
    WriteString ("Enter the amount to convert: ");
    ReadReal (toConvert);
    ok := ReadResult() = allRight;

    IF NOT ok THEN
        WriteString ("Enter a proper number");
        WriteLn;
    END;

    SkipLine;
END;

(* Do calculations and display*)
IF option = 1 THEN
    converted := inchFactor * toConvert;
    WriteFixed (toConvert, 2, 1);
    WriteString (" inch(es) is equivalent to");
    WriteFixed (converted, 2, 0);
    WriteString (" centimeters.");
ELSE
    converted := toConvert/inchFactor;
    WriteFixed (toConvert, 2, 1);
    WriteString (" centimeters is equivalent to");
    WriteFixed (converted, 2, 0);
    WriteString (" inch(es).");
END;

(* For exiting *)
ok := FALSE;
WriteLn;
WriteString ("Enter x to exit or any key run program again ");
ReadChar (exit);
SkipLine;

IF (exit = 'x') OR (exit = 'X') THEN
    ok := TRUE;

```



```

    END;

    WriteLn; WriteLn;
UNTIL ok;
END Convert.

(*      Created
      June.10.1999
      Chapter 3 Question 19
      NO ERROR TRAPPING  *)

MODULE Sort3;

FROM STextIO IMPORT
    WriteString, WriteLn, SkipLine;

FROM SWholeIO IMPORT
    ReadCard, WriteCard;

CONST
    fieldLength = 6;

VAR
    num1, num2, num3, temp : CARDINAL;

BEGIN
    (* information *)
    WriteString ("This program sorts two cardinal numbers from smallest to largest.");
    WriteLn;
    (* collect the numbers from the user *)
    WriteString ("Enter the first number please. ==> ");
    ReadCard (num1);
    SkipLine;
    WriteLn;
    WriteString ("Enter the second number please. ==> ");
    ReadCard (num2);
    SkipLine;
    WriteLn;
    WriteString ("Enter the third number please. ==> ");
    ReadCard (num3);
    SkipLine;
    WriteLn;
    WriteString ("From least to greatest, the numbers are: ");

    (* Sort them first *)
    IF num1 > num2 THEN
        temp := num2;
        num2 := num1;
        num1 := temp;
    END;

    IF num1 > num3 THEN
        temp := num3;

```

```
    num3 := num1;  
    num1 := temp;  
END;
```

```
IF num2 > num3 THEN  
    temp := num2;  
    num2 := num3;  
    num3 := temp;
```

```
END;
```

```
    (* Write them out *)
```

```
WriteCard (num1, fieldLength);
```

```
WriteString (" ", " ");
```

```
WriteCard (num2, fieldLength);
```

```
WriteString (" ", " ");
```

```
WriteCard (num3, fieldLength);
```

```
WriteLn;
```

```
SkipLine;
```

```
END Sort3.
```

CHAPTER 4

Questions

1. A procedure is a unit of code designed to perform a sub-task of some main task. Its code is written out only once in some module. It is better to use a procedure when you have to perform a sub-task more than once.
2. First type: no data used or produced.
Second type: data used but none produced.
Third type: data produced only.
Fourth type: data used and produced.
3. Invocation of a procedure is the occurrence of its name somewhere in a program. The invocation instructs the processor to execute the code of the procedure.
4. A procedure that calls itself is recursive. Recursion should be used when the original problem is naturally recursive.
5. When procedure value parameters are declared, say `PROCEDURE Write (ch : CHAR)`, a memory location is set aside for that parameter by the compiler. When the procedure is called, `Write (somevar)`, the parameter is copied into the memory location set aside for that particular procedure, creating a new and independent copy of the value. In using variable parameters, for example, the one in `PROCEDURE Read (VAR ch : CHAR)`, the parameter name is attached to the same memory location as that named by the actual parameter at the time the call is made. These constitute one memory location for that parameter so that changes using the local name are automatically reflected in the global one. The actual parameter must be a variable. A variable parameter can be differentiated from a value parameter by the `VAR` heading declared before a parameter.
6. This is true because `INTEGER`s are assignment compatible with `CARDINAL`s, as long as they are in the same range. When `WriteCard` is called with an `INTEGER` as its parameter, it will assign that parameter to the memory location because of their compatibility. However, using `ReadCard` with an `INTEGER` is forbidden because of expression compatibility concerns. Because `ReadCard` is a variable parameter, it tries to couple the `INTEGER` parameter to a `CARDINAL` memory location, which is forbidden. That is why the actual type must exactly match the formal type of a variable parameter.
7. A function procedure returns a type, the name of the returned type being placed after the closing parenthesis of the formal parameter list. A specific `RETURN [value]` statement has to be executed somewhere in the body of the function procedure. When calling a function procedure, a parameter list has to be included even if it is empty. Finally a call to a function procedure, has to be in a form of an expression rather than just a command line (ie. `x := SomeProc()` rather than `SomeProc()`).
8. `RETURN [value]` returns the value of the specified type from a function procedure. A bare `RETURN` in a proper procedure terminates the action prematurely, returning to the caller.
9. `ABS`: a built-in function procedure.
`FLOAT`: a built-in function procedure.
`ReadReal`: an imported proper procedure.
`MAX`: a built-in function procedure.
`ReadResult` an imported function procedure.
`Read`: an imported proper procedure.
`CAP` : a built-in function procedure.

MIN : a built-in function procedure.

exp : imported function procedure.

ln : imported procedure.

10. Each recursive call of Fac has to supply a new and independent value to start the next code cycle. This can only be done with value parameters. A variable parameter would modify all the data held in each of the recursive calls.

11. In simple interest, the interest rate is applied to only the original principal amount when computing interest. In compound interest, the rate is applied to both the original principal and whatever interest has accumulated.

12. The net present value is the amount equal to a series of future payments. The net future value is the predicted value of what the principal will eventually become.

13. When one makes mortgage payments against a debt, one is paying off the present value by a series of future payments. When one receives an annuity, the present value of a deposit is being paid out by a series of future payments. The only difference between the two is the direction the money goes; the computation is identical.

Problems

Note: Not all problems are shown. Most problems are left up to students as labs.

```
(* modified
   June 11 1999
   Chapter 4 Question 15
   ERROR TRAPPING INCLUDED  *)

MODULE Areas;

(* Written by R.J. Sutcliffe *)
(* to illustrate procedures *)
(* using P1 Modula-2 for the Macintosh computer *)
(* last revision 1993 02 25 *)

FROM STextIO IMPORT
  WriteString, ReadChar, WriteLn, SkipLine;
FROM SRealIO IMPORT
  WriteFixed, ReadReal;
FROM RealMath IMPORT
  sqrt;
FROM SIOResult IMPORT
  ReadResult, ReadResults;

VAR
  dimension, height, mArea, mPerim : REAL;
  which, ans : CHAR;
  again : BOOLEAN;

(*****)
PROCEDURE GetNum (VAR theNum : REAL);
```

```

VAR
    readOK : BOOLEAN;

BEGIN
    REPEAT
        WriteString ("Type the number here ==> ");
        ReadReal (theNum);
        readOK := (ReadResult() = allRight);
        IF NOT readOK
            THEN
                WriteLn;
                WriteString ("error in input number; try again.");
                WriteLn;
            END;
        SkipLine;
    UNTIL readOK;
END GetNum;

PROCEDURE CalcSquare (side : REAL; VAR area, perim: REAL);

BEGIN
    area := side * side;
    perim := 4.0 * side;
END CalcSquare;

PROCEDURE CalcCircle (radius : REAL; VAR area, perim: REAL);

CONST
    pi = 3.141592653;
    twopi = 2.0 * pi;

BEGIN
    area :=pi * radius * radius;
    perim := twopi * radius;
END CalcCircle;

PROCEDURE CalcTri (height : REAL; base : REAL; VAR area, perim : REAL);

PROCEDURE hypot (a : REAL; b : REAL) : REAL;

VAR
    c : REAL;

BEGIN
    c := a * a + b * b;
    RETURN sqrt(c);
END hypot;

```

```

BEGIN
    area := 0.5 * base * height;
    perim := base + 2.0 * (hypot(0.5 * base, height));
END CalcTri;

PROCEDURE CalcRect (height : REAL; base : REAL; VAR area, perim : REAL );
BEGIN
    area := base * height;
    perim := 2.0 * base + 2.0 * height;
END CalcRect;

(***** )

BEGIN      (* main program *)
    WriteString ("This program calculates areas and perimeters of");
    WriteLn;
    WriteString ("squares from a side length ");
    WriteString ("or circles from the radius.");
    REPEAT
        WriteLn;
        (* Present menu, give user the choice. *)
        WriteString ("Do you want to work with ");
        WriteString ("a circle or a square? ");
        WriteLn;
        WriteString ('Type a "C", "R", "S", or "T" here ==> ');
        ReadChar (which);
        SkipLine;
        which := CAP ( which);
        WriteLn;
        again := TRUE;

        (* now obtain a set of data *)
        IF (which = 'C') OR (which = 'S')
            THEN
                WriteString ("What is the dimension of the figure ?");
                WriteLn;
                GetNum (dimension);
            ELSIF (which = 'R') OR (which = 'T') THEN
                WriteString ("What is the height?"); WriteLn;
                GetNum (height);
                WriteString ("What is the base?"); WriteLn;
                GetNum (dimension);
            ELSE
                WriteString ("Not an option...quitting"); WriteLn;
                again := FALSE;
            END;
    END;

```

```

(* Do only if correct option selected *)
IF again
  THEN

    IF which = 'C'
      THEN      (* Now go do one. "S" is the default *)
        CalcCircle (dimension, mArea, mPerim);
      ELSIF which = "R" THEN
        CalcRect (height, dimension, mArea, mPerim);
      ELSIF which = "S" THEN
        CalcSquare (dimension, mArea, mPerim);
      ELSE
        CalcTri (height, dimension, mArea, mPerim);
      END;      (* if *)

```

```

  WriteString ("The area is ");
  WriteFixed (mArea, 2, 0);
  WriteString (" square units and ");
  WriteLn;
  WriteString ("the perimeter is ");
  WriteFixed (mPerim, 2, 0);
  WriteString (" units.");
  WriteLn;
  WriteLn;

```

```

END;

WriteString ("Do another (Y/N) ==>");
ReadChar (ans);
SkipLine;
again := (CAP (ans) = "Y");

```

```

UNTIL NOT again;

```

```

END Areas.

```

```

(*      Translated into ISO Modula-2
        June.11.1999
        Chapter 4 Question 27
        ERROR TRAPPING INCLUDED *)

```

```

MODULE Change;

```

```

FROM STextIO IMPORT
  WriteString, WriteLn, ReadChar, SkipLine;
FROM SWholeIO IMPORT
  WriteCard, ReadCard;
FROM SIOResult IMPORT
  ReadResult, ReadResults;

```

```

PROCEDURE CoinChange (VAR amountLeft : CARDINAL; value : CARDINAL);
(* Calculates the number of coins of the given value *)

VAR
    numberOfCoins : CARDINAL;

BEGIN
    numberOfCoins := amountLeft DIV value;
    amountLeft := amountLeft MOD value;
    WriteCard (numberOfCoins, 0);
    WriteLn;
END CoinChange;

(* main program *)

VAR
    count, total, leftover, cost, coinValue : CARDINAL;
    again : CHAR;
    ok : BOOLEAN;

BEGIN
    WriteString ("This program will dispense change for a dollar for some");
    WriteLn;
    WriteString ("purchased item costing less than a dollar.");
    WriteLn; WriteLn; WriteLn;

    REPEAT    (* give the option to redo program *)

        (* initalize vars *)
        total := 0;
        REPEAT    (* keep getting till their done giving *)

            REPEAT    (* make sure that all are correct types *)
                cost := 0;
                WriteString ("Enter the cost of the item (in cents please): ");
                WriteLn;
                ReadCard (cost);
                ok := ReadResult() = allRight;
                SkipLine;
            UNTIL ok;

            total := cost + total;
        UNTIL (cost = 0) OR (total > 100);

        (* amount was greater than 100 *)
        IF total > 100 THEN
            WriteString ("Sorry, you've exceeded the limit.");

```



```

        WriteLn;
ELSE
    leftover := 100 - total;
    WriteLn; WriteLn;

    FOR count := 1 TO 4 DO      (* 4 types of coins, so a loop till 4 *)
        WriteString ("The number of  ");

        IF count = 1 THEN
            WriteString ("quarter ");
            coinValue := 25;
        ELSIF count = 2 THEN
            WriteString ("dimes ");
            coinValue := 10;
        ELSIF count = 3 THEN
            WriteString ("nickels ");
            coinValue := 5;
        ELSE
            WriteString ("pennies ");
            coinValue := 1;
        END;

        WriteString ("is: ");
        CoinChange (leftover, coinValue);
        WriteLn;
    END;      (* end FOR *)

END;      (* end IF *)

```

```

WriteLn; WriteLn;
WriteString ("Do you want to go again (Y/N)? ");
ReadChar (again);
SkipLine;
WriteLn; WriteLn; WriteLn;
UNTIL CAP (again) # 'Y';

```

```

WriteString ("See you around... ");
WriteLn;
END Change.

```

CHAPTER 5

Questions

1. An ADT is a data type whose representation details are hidden from the user of entities of that type, whereas transparent data type have their representation details visible to the user of entities of that type.
2. The specification of types must include the allowable values and all valid operations.
3. Built-in ordinal types have many pre-specified operations (possibly including +, -, *, /) whereas user-defined ordinal types have only INC, DEC, MAX, MIN, VAL, and ORD. Any other operators must be defined as procedures.
4. When WriteString is called, it expects a variable of the same type (array of char). Here however, the item used is of type MonthNames and therefore will not work with WriteString. The type MonthNames is an ADT of its own.
5. The value Saturday is needed to prevent a run time error when the value Friday is incremented.
6. Advantages:
 - (i) The subrange is a different abstraction than the original type. Use of the subrange may therefore make a program clearer.
 - (ii) Some systems may store the values of a subrange more efficiently than those of the original type.
 - (iii) Error trapping. An error is generated if any attempt is made to assign something to a variable that is not in the subrange.
7. (a) is backwards (b) uses two different types<a character and a number. The specification for (c) requires prior definition of an enumeration with values a and e.
8. An iterated repetition involves a pre-specified pattern in an arithmetic sequence whereas a more general repetition can use any sequence of numbers, even altering the sequencing on the fly.
9. The FOR loop can only count up or down in constant increments, whereas the WHILE construction places no limitations on manipulating the loop counter variable inside the loop or on using it afterwards.
10. (a) the assignment operator is incorrect.
(b) the FOR loop increments by 1 by default. Therefore, the counter will never get to 1.
(c) "count" should be a CHAR variable; perhaps it's name should be changed
(e) watch out for a possible overflow.
(f) 1.5 is not an ORDINAL type.
(g) 1.5 is not an ORDINAL type.
(h) OK, but misleading, as count is evaluated only of the start and so is one.
11. The reason is that the subrange has a maximum of 10 and when the count reaches 10, it increments one more because the condition on the WHILE has not yet been met. At this point, the range will have been exceeded.
12. (a) 5
(b) adding 2 different data types.
(c) Thursday
(d) may compile but cardValue will not be correct. (cardVar will be taking in an integer).
(e) only allowed to INC using variables.
13. Inside the loop, the control loop variable is being threatened. Outside, it is undefined, so there is no point incrementing it (logical error).
14. The outer control loop variable is being threatened.
15. (a) 33 (b) 9 (c) 892
(d) 68 (e) will not do anything because the start value is already beyond the stop value.
16. Note: _ are used as spaces and cr is used as a carriage return.
(a) cccchhhhooorrrrrddd

- (b) _____5 cr
(c) _1_1_1_2_1_3_2_2_2_3_3_3
17. A two dimensional array is called a matrix.
18. (i) matrix = ARRAY [1..m], [1..n] OF CARDINAL
(ii)

```
FOR outer := 1 TO m
DO
  FOR inner := 1 TO n
  DO
    matrix [outer, inner] := 0;
  END;
END;
```

Problems

NOTE: Not all problems are shown. Most problems are left up to students as labs.

```
(* Modified
   June.11.1999
   Chapter 5 Question 19  *)
```

```
MODULE LetterCount;
```

```
(* Written by R.J. Sutcliffe *)
(* to illustrate the use of the for loop *)
(* using ISO Modula-2 *)
(* last revision 1991 02 27 *)
```

```
FROM STextIO IMPORT
  ReadString, ReadChar, WriteChar, WriteString, WriteLn, SkipLine;
FROM SWholeIO IMPORT
  WriteCard;
FROM SIOResult IMPORT
  ReadResults, ReadResult;
```

```
CONST
  space = CHR (32);
  alpha = CHR (65);
  cr = CHR (13);
  period = ".";
  min = 33; (* Set limits of printable ASCII characters *)
  max = 126;
  maxOnLine = 5;
```

```
TYPE
  LetArray = ARRAY CHAR OF CARDINAL;
```

```
VAR
  letterCount, wordCount, numOnLine, avPerWord : CARDINAL;
```

```
lets : LetArray;      (* Examples:  lets ['A'], lets [",","] *)
ch, last : CHAR;
lastResult : ReadResults;
userDone : BOOLEAN;
```

```
BEGIN
```

```
  FOR ch := CHR (min) TO CHR (max) (* initialize totals to zero *)
  DO
    lets [ch] := 0;
  END;    (* for *)
  userDone := FALSE;
```

```
  WriteString ("Please type in text you want analyzed.");
  WriteString (" End with period at start of line.");
  WriteLn;
```

```
  wordCount := 0;
  letterCount := 0;
  last := space; (* now leading space won't be seen as words *)
```

```
  REPEAT    (* main loop to read text by characters *)
    ReadChar (ch);      (* reads whatever is next *)
    lastResult := ReadResult ();
    IF lastResult = endOfLine  (* translate end of line state *)
    THEN  (* into 'carriage return character read' *)
      ch := cr;
      SkipLine;
    END;
    IF (lastResult = allRight) AND (ch > space) AND (ch # period)
      (* no control characters counted *)
    THEN
      (* see if it's in the alphabetic range *)
      IF (ch > CHR(64)) AND (ch < CHR(91))
      THEN
        INC (lets [VAL(CHAR, ORD(ch) + 32)]);
        INC (letterCount);
      ELSIF (ch > CHR(96)) AND (ch < CHR(123)) THEN
        INC (lets [ch]);
        INC (letterCount);
      END;
      ELSIF ( (ch = space) AND (last # space) ) OR (ch = cr) AND (last # cr) THEN
        (* two consec. is just one word *)
        INC (wordCount);
      ELSIF (ch = period) AND (last = cr) THEN
        (* end of input *)
        userDone := TRUE;
        SkipLine;
      END;
      last := ch; (* reset last for next time *)
  UNTIL (lastResult = endOfInput) OR userDone;
```

```

(* now tell user the results, several to a line *)
numOnLine := 0;
WriteLn;
FOR ch := CHR (min) TO CHR (max)
DO
    WriteChar (ch);      (* put out character *)
    WriteCard (lets [ch], 5);    (* # of times it was there *)
    WriteString ("      "); (* leave some space; make columns *)
    INC (numOnLine );
    IF numOnLine MOD maxOnLine = 0
    THEN
        WriteLn;
    END;
END;      (* for *)
WriteLn;
WriteLn;
WriteString ("# of words = ");
WriteCard (wordCount, 0);
WriteLn;
WriteString ("# of letters = ");
WriteCard (letterCount, 0);
WriteLn;
IF wordCount # 0
THEN
    avPerWord := TRUNC ((FLOAT (letterCount) / FLOAT (wordCount)) + 0.5);
    WriteString ("# of letters/word (nearest whole number) = ");
    WriteCard (avPerWord, 0);
    WriteLn;
END;

WriteString ("Press a key to end ==>");
ReadChar (ch);
END LetterCount.

(* Created June 14, 1999
   Chapter 5 Question 22
   Canned Program *)

MODULE PrintASCII;

FROM STextIO IMPORT
    WriteString, WriteLn, WriteChar, SkipLine;
FROM SWholeIO IMPORT
    WriteCard;

CONST
    start = 32;  (* start at CHR(32) *)
    end = 127;   (* end at CHR(127) to see non used character *)
VAR
    count, colcount, howmany : CARDINAL;

```

```

BEGIN
  WriteString ("This program writes out the ASCII character values with its");
  WriteLn;
  WriteString ("corresponding ordinal values.");
  WriteLn;
  WriteLn;
  WriteLn;

  (* initialize vars *)
  howmany := 0;
  colcount := 1;
  count := start;

  (* print them out nicely in a table *)
  WHILE count <= end DO
    WriteChar(CHR(count));
    WriteCard(count, 9);
    WriteString ("          ");    (* 8 spaces for formatting *)

    (* make a table of 2 rows *)
    IF colcount = 2 THEN
      colcount := 0;
      DEC (count, 2*48);    (* go twice back because it incs later *)

      IF (count # 31) THEN  (* make sure it doesn't write same *)
        INC (count, 1);
      ELSE
        count := 128; (* make sure it gets out *)
      END;

      WriteLn;
    END;

    (* inc counters *)
    INC (count, 48);
    INC (colcount);
    INC (howmany);
  END;  (* end WHILE *)

  WriteLn;
  WriteLn;
  WriteCard (howmany, 0);
  SkipLine;
END PrintASCII.

```

CHAPTER 6

Questions

1. A module is a container to hold the entities that constitute all or part of an executable program. More formally, it is an enclosing device to delimit the visibility and use of entities, whether these be internally defined or imported.
2. Modules can contain types, constants, variables, procedures, and other modules.
3. Some of the differences between procedures and modules include the ability to export and import, scope of variables, and permanence. Modules use EXPORT and IMPORT to make any variable, procedure, type or constant visible outside the module in a surrounding one or visible inside the module if it is defined in the one around it. All objects declared in a module remain active even when the module is not in use and are invisible to any other modules in the rest of the program. The modules taken thus far do not use parameters. Conversely, variables in a procedure exist only when the procedure is in use (and are therefore invisible outside of it) and all variables in a module or procedure are visible (changeable) inside every procedure contained in it. A module exists as long as the surrounding module (or procedure) exists but a procedure is called into existence only when a reference is made to it and then vanishes after the procedure finishes.
4. When the form FROM SomeLibrary IMPORT is used, the imports are available unqualified in the importing environment. When the module is imported as a whole (i.e.. IMPORT Terminal) all identifiers are available to the importing environment.
5. The word "standard" could refer to the ISO standard that all vendors have to comply with if they want to have an "ISO compliant stamp" on their libraries. Some vendors of older products might use it to mean "classic" Modula-2 complying with PIM-3. It could also mean the "standard I/O channel". In this definition, the standard I/O is normally the keyboard/screen devices.
6. Input and output vary from computer to computer. A Macintosh may handle input and output differently from IBM compatible computers. Thus, standardization can only be achieved at a relatively high level.
7. Answers may vary, but the ISO facilities are in the appendix.
8. Answers may vary.
9. Answers may vary.
10. The ISO modules operate on common channels that are defined and opened at the lower level. If the standard channel is redirected, I/O using it is redirected regardless of what module employs the channel. In classical Modula-2, RealInOut, though expressed as a separate module, is supposed to be thought of as a part of InOut. Consequently, the redirection facilities of InOut apply to it also.
11. Module hierarchy refers to the fact that some library modules import other library modules. These in turn may be imported by some other modules thus forming a "chain of dependence". An example of this kind of chain starts with STextIO and the hierarchy of modules imported to it:
STextIO <= TextIO <= Terminal (non standard)
The lowest modules in such a chain are those which are connected most intimately to the specific operating system. Routines that import from these, low-level Modules are less portable than those that import only from say, STextIO.
12. A compilation unit is a module that can be compiled. In Modula-2, there are 3 kinds (implementation, definition, and program module) and all resulting code is brought together into one executable when the program is linked together.
13. Definition: does not contain a body< it only has syntax, such as procedure parameter lists. It contains the interface for clients to import. No code is produced.
Implementation: the parameter lists of their procedure should exactly repeat the ones in the corresponding definition

modules. It cannot contain an export list. It must implement all procedures in the corresponding definition, and may have a body.

Program: similar to an implementation module, but cannot be imported into another program.

14. Answers may vary.

15. Supplied Library: The collection of modules provided to the programmer by the vendors. These include the Standard Library Modules, Nonstandard Utility Modules (modules or data types provided by the vendor as ready tools the programmer can use) and System Specific Modules, which use facilities available only to a particular type or brand of computer/operating system.

User Library: This is a set of library modules that a programmer has created as tools to use with other library or program modules.

Program Library: The set of modules that are part of a program itself. A programmer may use this to hide certain types and make them invisible to other modules.

16. Using modules means there will be only one source for that imported module. If an identifier, type, or variable is at fault, it is unnecessary to correct the mistake at every point it is declared but only at the source. Moreover, names can be re-used across module boundaries without causing side-effects.

17. Module decoupling is the separation of the definition and implementation. It allows the design of the program and its library support to be conducted in one phase of the job, and all the code implementation to be postponed until later.

18. The modules that need to be changed and compiled are the IMPLEMENTATION and program modules. Changing the DEFINITION module most likely means new syntax has been created and the IMPLEMENTATION module needs to be updated to take advantage of the new changes. The program module will then need to be updated because of the new key compilation of the DEFINITION module has generated. If the program module is not compiled, the linker will refuse to link the object code because the IMPLEMENTATION code will incorporate the new key of the DEFINITION at compilation time.

19. Only the program needs to be re-linked. Nothing need be recompiled other than the changed implementation. The reason the DEFINITION module is not recompiled is that a key is not generated when the IMPLEMENTATION module is compiled. The program module will need the new object code generated by the IMPLEMENTATION module in order for the changes to take effect.

20. Not using implementation details of a data type will actually prevent client programs from knowing any details about the representation, and force them to access items of the abstract type solely through the provided procedures. This helps preserve the principle of information hiding.

21. Precondition checking: this method relies on checking for potential error conditions before taking the action that might cause the error to arise. It places the responsibility for doing the checking on the client program.

Postcondition checking: In this method, the operation is attempted, and its success is checked afterward (perhaps for several tries) by examining the value of the appropriate error variable.

Automatic Error Handling: This method invoke some procedure that automatically handles errors whenever they take place, without any boolean or enumerated type having to be set or checked.

22. $\text{cotangent} := 1 / \text{RealMath.tan}(x)$;

$\text{secant} := 1 / \text{RealMath.cos}(x)$;

$\text{cosecant} := 1 / \text{RealMath.sin}(x)$;

Problems

Note: Not all problems are shown. Most problems are left up to students as labs.

(* Created
June 15 1999

Chapter 6 Question 25

NO ERROR TRAPPING *)

```

MODULE WriteTwo;
FROM TextIO IMPORT
    ReadString, ReadChar, WriteString, WriteLn, WriteChar, SkipLine;
FROM StreamFile IMPORT
    ChanId, Open, write, old, Close, OpenResults;
IMPORT TextIO;

TYPE
    String = ARRAY [0..255] OF CHAR;  (* 256 characters at max *)

VAR
    StringIn : String;
    sink : ChanId;                    (* destination to the file to write *)
    res : OpenResults;                (* give the results of file status *)
    pos, count : CARDINAL;           (* position in string *)

BEGIN
    WriteString ("This program will read in a string and print the string ");
    WriteLn;
    WriteString ("on the screen and in a file with double lettering");
    WriteLn;
    WriteLn;
    WriteLn;
    WriteLn;
    WriteString ("Enter the string: ");
    WriteLn;
    ReadString (StringIn);
    SkipLine;    (* don't need to check for error since all chars are valid *)
    Open (sink, "test.txt", write+old, res);
    IF res = opened THEN
        pos := 0;
        WHILE StringIn [pos] # CHR(0) DO
            count := 0;

            (* write the text out twice *)
            WHILE count < 2 DO
                WriteChar (StringIn[pos]);
                TextIO.WriteChar (sink, StringIn[pos]);
                INC (count);
            END;

            INC (pos);
        END;    (* end WHILE *)

        TextIO.WriteLn (sink);
        Close(sink);
    ELSE

```

```
    WriteString ("Channel could not be opened. ")  
END;
```

```
WriteLn;
```

```
WriteLn;
```

```
WriteString ("The file created is called 'test.txt'.  Press ENTER to quit");
```

```
SkipLine;
```

```
END WriteTwo.
```

CHAPTER 7

Questions

1. A Modula-2 string is an ARRAY [0..n-1] OF CHAR, zero based, and terminated (usually by the null character).
2. An implied abstract type has its basic structure visible, but can be treated abstractly. Modula-2 allows an array of char to be thought of and used in most cases as an abstract string.
3. (a) E = "dogs chasecats "
(b) m = 5
(c) found = FALSE, position = undefined;
(d) i = less
(e) cogs
(f) str1 = docatsg
(g) found = TRUE, position = 8
(h) found = TRUE, position = 4
4. Silent truncation is the discarding of characters in a string without the user knowing. This arises, for example, when two strings are concatenated and the result does not have enough room for both strings. In the case of Concat, the first string is placed in the result and the second string (whatever can fit), is then placed in the rest of the array.
5. Length returns the number of characters in the string whereas HIGH returns the highest index used when the actual parameter array is assigned to the open formal parameter array. The latter depends on the string type, not its contents.
6. (a) greater; "r" > "i"
(b) greater; lower case letters have higher ASCII values than upper case letters.
(c) greater; same as (b)
(d) less; "%" < "p"
(e) less; "1" < "0"
7. An unused (uninitialized) string will contain characters that are in the memory location at that moment. The user will not know what that will be and it will therefore affect the output of the string.
8. See http://searchSecurity.techtarget.com/sDefinition/0,,sid14_gci213893,00.html
9. The mean is the ordinary average. More formally, the mean of a group of data is their sum divided by the number of items in the group.
Variance is the mean of the squares of the differences between data items and the mean.
If the data is a sample, we divide by (n-1) rather than n when calculating variance.
Standard deviation is the square root of variance.
10. The median is the middle observation of an ordered distribution. One needs a facility to find whether the data set size is even or odd and to sort the data.
11. The mode is the value that occurs the most in a data set. One needs a facility to keep track of how many occurrences of each data item there are.
12. Separating low and high level functions makes code easier to debug. All procedures in the modules are small and therefore easy to manage.
13. Random numbers generated from a formula produce the same sequence every time. They are better defined as pseudo-random numbers. To make the numbers more random, a seed can be used and a new seed produced every time a random number is called for.
14. A matrix is a rectangular arrangement of symbols in a two dimensional array. They are used for such tasks as equation solving, expressing tables, representing abstract graphs and as abstractions in and of themselves.
15. A vector is an entity that has magnitude and direction. In computing science, it can be represented as a one-dimensional array. They are used to represent velocities, forces, displacements, and accelerations.
16. The ADT Vectors include the following procedures not in the ADT Points:
PROCEDURE assignP (ABS, ARG: REAL) : Vector;

```

PROCEDURE add (u, v : Vector) : Vector;
PROCEDURE sub (u, v : Vector) : Vector;
PROCEDURE dotProduct (u, v : Vector) : REAL;
The ADT Points include the following procedures not in the ADT Vectors:
PROCEDURE polarToRect (abs, arg : REAL) : Point;
PROCEDURE reflectX (p: Point) : Point;
PROCEDURE reflectY (p: Point) : Point;
PROCEDURE reflect0 (p: Point) : Point;
PROCEDURE reflect45 (p: Point) : Point;
PROCEDURE rotate (p : Point; rotAngle : REAL) : Point;
PROCEDURE translate (p : Point; deltaX, deltaY : REAL) : Point;

```

Problems

Note: Not all problems are shown. Most problems are left up to students as labs.

```

(*    Translated to ISO standard Modula-2
    June 15 1999
    Chapter 7 Question 18
    NO ERROR TRAPPING    *)

```

```

DEFINITION MODULE StringFunc;

```

```

TYPE
    STRING = ARRAY [0..80] OF CHAR;

```

```

CONST
    nul = CHR(0);

```

```

VAR
    strDone : BOOLEAN;

```

```

PROCEDURE Insert (VAR str, substr : ARRAY OF CHAR; pos : CARDINAL);
(* inserts a string into another string *)

```

```

PROCEDURE Delete (VAR str : ARRAY OF CHAR; pos, len : CARDINAL);
(* deletes n characters from a specified index from a string *)

```

```

PROCEDURE ConCat (VAR str1, str2, result : ARRAY OF CHAR);
(* concatenates 2 strings and puts it in the variable result *)

```

```

PROCEDURE Length (str : ARRAY OF CHAR) : CARDINAL;
(* returns the length of a string *)

```

```

END StringFunc.

```

```

IMPLEMENTATION MODULE StringFunc;

```

```

PROCEDURE Insert (VAR str, substr : ARRAY OF CHAR; pos : CARDINAL);

```

```

VAR
    len, count, subcount : CARDINAL;

```

BEGIN

```
(* see if we have room to fit the entire substring into the string if
   so then strDone = TRUE and set 'len' to the length of the substring
   otherwise strDone = FALSE and set 'len' to the max size it can be,
   that is the HIGH of the string minus the string's length. *)
```

```
IF Length (str) + Length (substr) <= HIGH (str)
```

```
THEN
```

```
    strDone := TRUE;
```

```
    len := Length (substr);
```

```
ELSE
```

```
    strDone := FALSE;
```

```
    len := HIGH (str) - Length (str)
```

```
END;
```

```
(* make room within the string to insert the substring *)
```

```
FOR count := Length (str) TO pos BY -1
```

```
DO
```

```
    str [count + len] := str [count]
```

```
END;
```

```
(* insert the substring into the string at the position 'pos' *)
```

```
subcount := 0; (* init the substring counter to 0 *)
```

```
FOR count := pos TO pos + len
```

```
DO
```

```
    (* if this character isn't the string terminator then insert it *)
```

```
    IF substr [subcount] # nul
```

```
    THEN
```

```
        str [count] := substr [subcount];
```

```
        INC (subcount)
```

```
    END
```

```
END;
```

```
(* if the new string doesn't fill the string set a null terminator at end *)
```

```
IF Length (str) <= HIGH (str)
```

```
THEN
```

```
    str [Length (str)] := nul (* mark end of the new string *)
```

```
END
```

END Insert;

```
(*-----*)
```

PROCEDURE Delete (VAR str : ARRAY OF CHAR; pos, len : CARDINAL);

VAR

```
    count, count2 : CARDINAL;
```

BEGIN

```
(* if there is something beyond the deleted part of the string
   move it down, and set strDone to TRUE; otherwise delete the
   remainder of the string and set strDone to FALSE. *)
```

```

IF len + pos < Length (str)
THEN
    strDone := TRUE;

    (* move remainder of the string down to overwrite the deleted part *)
    count2 := pos;  (* init to start of area to delete *)
    FOR count := pos + len TO Length (str)
        DO
            str [count2] := str [count];
            INC (count2);
        END;

    str [Length (str)] := nul  (* mark end of new string *)
ELSE
    strDone := FALSE;
    str [pos] := nul;
END

```

END Delete;

(*-----*)

```

PROCEDURE ConCat (VAR str1, str2, result : ARRAY OF CHAR);

```

```

VAR

```

```

    count, count2 : CARDINAL;

```

```

BEGIN

```

```

    (* initialize variables *)
    FOR count := 0 TO Length (str1)
        DO
            result [count] := str1 [count]  (* set result equal to string1 *)
        END;
    count2 := 0;  (* init string2 counter to 0 *)

```

```

    (* check to see if both strings will fit into the result,
       if so then set strDone to TRUE, otherwise to FALSE.      *)
    IF Length (str1) + Length (str2) <= HIGH (result)
    THEN
        strDone := TRUE
    ELSE
        strDone := FALSE
    END;

```

```

    (* insert string2 onto the end of result until you run out of room *)
    FOR count := Length (result) TO HIGH (result)
        DO
            result [count] := str2 [count2];
            INC (count2)
        END;

```

```

    (* put a string terminator at the end of the new string if it isn't full *)
    IF Length (result) < HIGH (result)

```

```

    THEN
        result [Length (result)] := nul
    END

END ConCat;

PROCEDURE Length (str : ARRAY OF CHAR) : CARDINAL;

VAR
    count : CARDINAL;

BEGIN

    count := 0;  (* init count (* number of characters in string *) to 0 *)

    (* check to see if there is anything in the string *)
    IF str [count] # nul
    THEN
        strDone := TRUE;  (* if yes then set strDone to TRUE, find its length *)

        (* loop through the string to the end of it *)
        REPEAT
            INC (count)
        UNTIL (count > HIGH (str)) OR (str [count] = nul)
        ELSE
            strDone := FALSE  (* if not then set strDone to TRUE and return 0 *)
        END;

    RETURN count

END Length;

(*=====*)
(*
BEGIN

    strDone := TRUE;  (* init to TRUE so that it is initialized before a *)
(* procedure from StringFunc is called *)
*)
END StringFunc.

MODULE StringTest;

FROM STextIO IMPORT
    WriteLn, WriteString, ReadChar, ReadString, SkipLine;

FROM SWholeIO IMPORT
    WriteCard, ReadCard;

FROM StringFunc IMPORT
    STRING, Insert, Delete, ConCat, Length;

(*=====*)

```

```

VAR
  str1, str2, str3 : STRING;
  len, pos : CARDINAL;
  ch : CHAR;

(*=====*)

BEGIN  (* Main Body *)

  (* Start up *)

  REPEAT

    (* Introduction *)

    WriteLn;
    WriteString ("This little program tests the String module that we just");
    WriteString ("created.");
    WriteLn;
    WriteLn;

    (* Calculations *)

    (* Insertion *)
    WriteString ("Enter a string please ==> ");
    ReadString (str1);
    SkipLine;
    WriteLn;
    WriteLn;
    WriteString ("      Insertion");
    WriteLn;
    WriteString ("Enter a string to insert into the previous ==> ");
    ReadString (str2);
    SkipLine;
    WriteLn;
    WriteString ("At what position do you want it inserted ==> ");
    ReadCard (pos);
    SkipLine;
    WriteLn;
    WriteLn;
    Insert (str1, str2, pos);
    WriteString (str1);
    WriteLn;
    WriteLn;

    (* Deletion *)
    WriteString ("      Deletion");
    WriteLn;
    WriteString ("Where do you want to start deleting from the first string ");
    WriteString ("==> ");
    ReadCard (pos);

```



```

SkipLine;
WriteLn;
WriteString ("How much do you want to delete ==> ");
ReadCard (len);
SkipLine;
WriteLn;
WriteLn;
Delete (str1, pos, len);
WriteString (str1);
WriteLn;
WriteLn;

    (* ConCatination *)
WriteString ("      Concatination");
WriteLn;
WriteString ("What do you want to add to the first string ==> ");
ReadString (str2);
SkipLine;
WriteLn;
WriteLn;
ConCat (str1, str2, str3);
WriteString (str3);
WriteLn;
WriteLn;

    (* Length *)
WriteString ("      Length ");
WriteLn;
len := Length (str1);
WriteString ("The length of the first string was ");
WriteCard (len,0);

(* Conclusion *)

WriteLn;
WriteLn;
WriteString("Press 'Q' to quit, any other key to continue. ");
ReadChar (ch);
UNTIL CAP (ch) = 'Q';

WriteLn;

END StringTest.

```

```

(* Translated to ISO standard Modula-2
June.15.1999
Chpater 7 Question 20
NO ERROR TRAPPING

```

MODIFICATIONS: Created a pascal string type (str255)

since
Created a procedure to write the pascal string type
there was no library for the original created.
Added a WriteCard to show the lenght of the string *)

[illegible]

```
(* <><><><><><><><><><><><><><><><><><><><><><> *)  
  
(* main program *)  
  
BEGIN  
    WriteString ('We will be testing the conversion from Modula-2 strings to');  
    WriteString (' Pascal strings');  
    WriteLn;  
    WriteLn;  
    WriteLn;  
    WriteString ('Please input a Modula-2 string now.');    WriteLn;  
    ReadString (ModStr);  
    SkipLine;  
    WriteLn;  
    WriteLn;  
    WriteLn;  
    StrModToPas (ModStr, PasStr);      (* convert *)  
    WriteString ("The pascal string seen as M2 string:");  
    WriteLn;  
    WriteString (PasStr);  
    WriteLn;  
    WriteString ('The Pascal string is:');  
    WriteLn;  
    WritePascal (PasStr);  
    WriteLn;  
  
    (* this line added to show the length in the first element of the array--ry *)  
    WriteString ("String length PasStr[0]: ");  
    WriteLn;  
    WriteCard (ORD(PasStr[0]), 1);  
  
    WriteLn;  
    WriteLn;  
    WriteLn;  
    WriteString ('Press any key to return to the desktop.');    ReadChar (ch);  
    SkipLine;  
END ModToPas.
```

CHAPTER 8

Questions

1. An implementation restriction is a numerical limitation on the user or complexity of data types or of programming structures. For example, a manufacturer will specify a restriction like how deeply nested a loop, selection, parentheses or procedure can be.
2. Implementation defined refers to information the ISO standard requires the manufacturer to specify, for example, the maximum and minimum values of the built-in scalar type CHAR. Implementation dependent refers to behaviour that depends on the underlying machine and is potentially different on different machines. The code `MyProc(a, function(a))` where `a` is a variable parameter, depends for its meaning on the order of parameter evaluations and is therefore erroneous.
3. Computer languages or individual language constructions are called low level if they require for their correct use a knowledge of machine language, if they name or manipulate the machine's memory locations directly, or involve a knowledge or use of details of that machine's hardware construction or the operating system. To the degree that the use of a language is independent of and isolated from any of these specific details, it is said to be high level. Modula-2 is a high level language with some low level facilities.
4. High level constructs tend to use English or near English words for commands. Low level constructs tend to have more cryptic abbreviations that have meaning only in the context of a particular machine, type of machine, or operating system. Low level constructs also tend to be more detailed and usually accomplish a single step at a time unlike in a high level construct where they tend to be more general and many steps are accomplished in one command. Most language constructs in Modula-2 are high level. However, `SYSTEM.LOC`, and all procedures using this type, are low level.

5.

	Binary	Octal	Hexadecimal
(a)	1111011	173	7B
(b)	11111111	377	F
(c)	10000000000	2000	400
(d)	1000000000000	10000	1000
(e)	10111011100	2734	5DC
(f)	1000001010011	10123	1053

6.

	Decimal	Hexadecimal	Octal
(a)	181	B5	265
(b)	119	77	167
(c)	203	CB	313
(d)	240	F0	360
(e)	170	AA	252

7.

	Binary	Decimal
(a)	1010000010000001	41089
(b)	1111011100	988
(c)	1100000000000000	49152
(d)	10000000000000	4096
(e)	1100000000	768
(f)	111110001111	3983
(g)	1101000000000000	53248
(h)	1111100000000000	63488

8

	Binary	Hexadecimal
(a)	0000000000001010	A
(b)	0000000001100100	64
(c)	0000001111101000	3E8
(d)	0010011100010000	2710
(e)	0000000100000000	100
(f)	0000010000000000	400
(g)	1111111111111111	FFFF
(h)	0001100001011010	1472
(i)	0010100000001101	280D

9. Bit: a single memory component that stores a one or a zero.

Byte: A sequence of eight bits as a single memory location that can store a number from zero through 255.

Nibble: half a byte (four bits). 2 parts: high (upper four) & low (lower four).

"K": 1024 bytes, or 1Kilobyte.

10. Answers may vary.

11. Answers may vary. Generally, a page of memory is 256K. A sector is 256K of data storage, and a block is 2 sectors. However, the first two these terms have fallen into disuse, and allocation blocks commonly vary in sizes depending on the operating system and hard drive size.

12. A data location is a grouping of two or more adjacent bytes (memory locations) collectively employed as a single data storage unit. It is abstracted as SYSTEM.LOC, and is best thought of as the smallest addressible storage unit.

13. An address is a unique value identifying a particular storage location. ISO standard Modula-2 has procedures that manipulate addresses: ADDADR, SUBADR, and DIFADR.

14. PROCEDURE SYSTEM.ADR returns the address of a variable v.

15. someVariable[B7ED] : CARDINAL (declared under the VAR heading).

16. The library import FROM SYSTEM IMPORT flags a program as non-portable.

17. A safe conversion converts a one type to another by using the standard identifier VAL and its shortcuts. An unsafe conversion forcibly re-interprets the bit pattern of one type as though it were of another without conversion and is done with the SYSTEM.CAST.

18. (a) octal digits for the character value are followed by a C e.g. 0C

(b) octal numeric literals are followed by a B e.g. 177777B

(c) hexadecimal digits for the character value are followed by an H e.g. 789H

19. A file is a source or a sink for data.

20. A logical file is an abstract structuring of data storage as viewed by the programmer and/or user of the program. It is the high level, or conceptual view of a file. A program file is a specific data collection as seen and manipulated by a program. It is often (but not always) represented by a variable, perhaps of type "file" or ChanId. At this middle level of abstraction, a file can be regarded as residing within the machine's memory, and as existing only as long as the program that employs it is active. A physical file is a recording of a logical file. It takes the form of a magnetic image on a disk or tape surface, in which form it exists independently of any particular program. The details of this recording provide the lowest level view of a file.

21. A sequential file is a file that is organized as a stream with writing allowed only at the end. A random-access file is a file that may have any of the data elements read or written directly using an indexing scheme without having to start at the beginning of the file.

22. TextIO, WholeIO, RealIO, LongIO, RawIO, and IOResult are the high level Modula-2 I/O modules. The S versions are specializations of these.

23. StreamFile, SeqFile, RndFile, and TermFile are the ISO Modula-2 drivers. They are all considered as middle level.

24. SRawIO (high), RawIO, IOConsts, StreamFile, SeqFile, RndFile, and TermFile (middle) are used in the ISO suite to do input and output of binary data. SRawIO is a high level module and the rest of the modules mentioned are middle level.

25. A sequence is a function from the positive whole numbers into some other collection of objects. A stream is a sequence of data items of the same type.

26. Streams have the following properties: (i) The number of elements in a stream is not known ahead of time. (ii) A stream has an origin and a destination, which are also not necessarily known ahead of time but which may default to some standard place or device. (iii) Writing is only done at the end of a stream, and deleting is the only way of modifying a stream entirely. (iv) Any element of a stream can be read, provided that reading starts at the beginning of the stream and proceeds through the elements one at a time in order until the desired one is reached. (v) A pointer or a window may be maintained which points to the last element read so that the next item can be read at a later time without starting from the beginning. (vi) Streams have modes which only allow them to be written to or read from, depending on the mode. However, some streams may allow procedures to act on their status altering the mode.

27. Streams are all of the same type. The sequence shown is a mixture of type CARDINAL/INTEGER and of type REAL.

28. A text stream is a stream whose items are of the type CHAR, and is also known as a legible stream. A raw stream is a stream of binary items.

29. A channel is an abstract medium through which a stream flows.

30. A sequential file has all its items the same type. A text file is a stream of type CHAR.

31. 1. Declare a file variable to logically identify the file within the program

or declare a channel variable to identify the logical/physical connection.*

2. Use the file's actual name (a string) to look it up (or create it) on the external device (say, a disk*).

3. Identify the logical file variable with the actual disk file (open the file)*.

4. Connect a program I/O stream to the previously opened file.

5. Read from or write to the stream, and hence the logical, and so the physical file*.

6. When finished, disconnect the stream from the program.

7. Close the file, ensuring that the data is secure on the disk*.

* are needed in an ISO system.

Answers may vary on other systems.

- 32. A restricted stream reads only from the beginning and writes only to the current position. A rewindable sequential stream has the properties of a restricted stream but has the capability of rewinding the read or write position back to the start of the file. For the restricted stream, Modula-2 uses StreamFile and for the rewindable sequential stream it uses SeqFile.
- 33. Closing a file makes sure that it gets written to the disk and secured. The writing that happens before the close command only writes to the memory location, not actually to the disk.
- 34. A buffer is a temporary storage area that is used to store information being transmitted to or from an external location (including a physical file).
- 35. Answers may vary. Some limit the characters that may be used, and all have limits on the length of the name.
- 36. The procedures InChan and OutChan return the current channel for standard I/O, possibly as redirected by the a default channel that is specified by the programmer (i.e. a file). StdInChan and StdOutChan on the other had specify the standard channel created on startup, normally the keyboard and screen.
- 37. StdInChan.NullChan can be used for testing purposes without employing (and possibly damaging) a file with a specified channel. IOChan.InvalidChan is the channel returned by failed attempts to open a channel and serves as a flag to check such attempts.
- 38. ISO Modula-2 has the module IOChan for device independent I/O.
- 39. An echo is a copy of an input character to an output device.
- 40. Character mode input has the echo flag is set for input. For line mode input, the channel is opened without the echo flag.
- 41. There can be any number of channels in character mode.
- 42. A generic procedure is a procedure that is capable of acting on any data regardless of type.

Problems

Note: Not all problems are shown. Most problems are left up to students as labs.

```
(*    Created
      June 18 1999
      Chapter 8 Question 43
      ERROR TRAPPING INCLUDED *)
```

```
MODULE CardToBin;

FROM STextIO IMPORT
  WriteString, WriteLn, WriteChar, SkipLine, ReadChar;
FROM SWholeIO IMPORT
  WriteCard, ReadCard;
FROM Strings IMPORT
  Length;
FROM SIOResult IMPORT
  ReadResult, ReadResults;

TYPE
  String = ARRAY [0..255] OF CHAR;

VAR
  binary : String;
```

```
ok : BOOLEAN;
input : CARDINAL;
quit : CHAR;
```

```
PROCEDURE ConvertCard (card : CARDINAL; VAR binary : ARRAY OF CHAR);
(* pre: none
   post: finds the binary equivalent and puts it in an array of char *)
```

```
VAR
    quo, rem, count, alt: CARDINAL;
```

```
BEGIN
    alt := card;
    quo := alt;
    count := 0;

    WHILE quo > 0 DO
        card := quo;
        quo := card DIV 2;
        rem := card MOD 2;

        IF rem = 0 THEN
            binary[count] := '0';
        ELSE
            binary[count] := '1';
        END;

        INC (count);
    END; (* end WHILE *)
```

```
    binary[count] := CHR(0);
END ConvertCard;
```

```
PROCEDURE WriteBinary (binary : ARRAY OF CHAR);
(* pre: string must contain the binary string
   post: writes out the binary string *)
```

```
VAR
    count : INTEGER;

BEGIN
    count := Length (binary);
```

```
    WHILE count >= 0 DO
        WriteChar (binary[count]);
        DEC (count);
    END; (* end WHILE *)
```

```
END WriteBinary;
```



```

(* Start main program *)

BEGIN
    REPEAT                                (* do again *)
        WriteString ("This program will convert a cardinal to a binary number.");
        WriteLn;

        REPEAT
            WriteLn;
            WriteString ("Enter the CARDINAL you want to convert: ");
            ReadCard (input);
            ok := ReadResult () = allRight;

            IF NOT ok THEN
                WriteLn;
                WriteString ("Error in input");
            END;

            SkipLine;
        UNTIL ok;

        WriteLn;
        WriteLn;

        (* convert the cardinal *)
        ConvertCard (input, binary);
        WriteString ("The binary conversion is:");

        (* write the binary *)
        WriteBinary(binary);
        WriteLn;
        WriteLn;

        (* ask to quit *)
        WriteString ("Enter 'Q' to quit: ");
        ReadChar (quit);
        SkipLine;
    UNTIL quit = 'Q';
END CardToBin.

(*    Created
    June.18.1999
    Chapter 8 Question 45
    NO ERROR TRAPPING

    This program uses SYSTEM.CAST to CAST the REAL into a CARDINAL
    then the binary is found.  *)

```

```

MODULE RealPrinter;

FROM STextIO IMPORT
    WriteString, WriteLn, WriteChar, SkipLine, ReadChar;
FROM SRealIO IMPORT
    WriteReal, ReadReal;
FROM Strings IMPORT
    Length;
FROM SYSTEM IMPORT
    CAST;
FROM SIOResult IMPORT
    ReadResult, ReadResults;

TYPE
    String = ARRAY [0..255] OF CHAR;

VAR
    binary : String;
    ok : BOOLEAN;
    realInput : REAL;
    input : CARDINAL;
    quit : CHAR;

PROCEDURE ConvertCard (card : CARDINAL; VAR binary : ARRAY OF CHAR);
(*  pre: none
   post: finds the binary equivalent and puts it in an array of char *)

VAR
    quo, rem, count, alt: CARDINAL;

BEGIN
    alt := card;
    quo := alt;
    count := 0;

    WHILE quo > 0 DO
        card := quo;
        quo := card DIV 2;
        rem := card MOD 2;

        IF rem = 0 THEN
            binary[count] := '0';
        ELSE
            binary[count] := '1';
        END;

        INC (count);
    END; (* end WHILE *)

```

```

    binary[count] := CHR(0);
END ConvertCard;

PROCEDURE WriteBinary (binary : ARRAY OF CHAR);
(* pre: string must contain the binary string
   post: writes out the binary string *)

VAR
    count : INTEGER;

BEGIN
    count := Length (binary);

    WHILE count >= 0 DO
        WriteChar (binary[count]);
        DEC (count);
    END;

END WriteBinary;

(* Start main program *)

BEGIN
    REPEAT                                (* do again *)
        WriteString ("This program will convert a REAL into a binary number.");
        WriteLn;

        REPEAT
            WriteLn;
            WriteString ("Enter the REAL you want to convert: ");
            ReadReal (realInput);
            ok := ReadResult () = allRight;

            IF NOT ok THEN
                WriteLn;
                WriteString ("Error in input");
            END;

            SkipLine;
        UNTIL ok;

        WriteLn;
        WriteLn;
        (* do an unsafe conversion into a CARDINAL *)
        input := CAST (CARDINAL, realInput);
        (* convert the cardinal *)
        ConvertCard (input, binary);
        WriteString ("The binary conversion is:");
        (* write the binary *)

```

```
WriteBinary(binary);
WriteLn;
WriteLn;

(* ask to quit *)
WriteString ("Enter 'Q' to quit: ");
ReadChar (quit);
SkipLine;
UNTIL quit = 'Q';
END RealPrinter.
```

CHAPTER 9

Questions

1. A Modula-2 set is a collection of items of the same scalar non-real type without regard to order, whereas an abstract set is a collection of items of any type.
2. A subrange of a scalar type is a sequence of consecutive values of the host type, unlike a subset which is a collection of values (in no particular order) taken from a set.
3. TYPE month30 = SET OF [jan, mar, may, jul, oct, dec]; month31 = SET OF [apr, jun, aug, sep, nov];
4. Sets in older versions of Modula-2 were of limited use because most of them followed the suggestion of Wirth which limited the ranges of a set (usually to a maximum of 16 or 32 elements). Some implemented the language to limit sets to cardinal ranges starting at zero. Instead of making a maximum allowable, the ISO standard requires a SET OF CHAR to be permitted.
5. All the driver modules export flags that are sets. They are compatible because they are defined elsewhere, imported, then re-exported.
6. Union: creates a new set that contains all the elements present in either of the two original sets. Intersection: creates a new set that contains only those elements common to the original pair of sets. Difference: creates a new set consisting of all elements of one set that are not in the other.
Symmetric Set Difference: creates a new set whose elements are in either of the original sets, but not in both. INCL: same as set union, but only one element is inserted into the set. EXCL: same as set difference, but only one element is removed from the set.
7. A BITSET is a set of [0..bitsperbitset-1] with membership in the set dependent on whether the bit position at that number contains a one or a zero. A PACKEDSET is similar, but the maximum bit number is user-defined. Operations that are used on these types are SHIFT, which shift all the bits of the pattern n positions to the left or right, and ROTATE, which rotates all the bits of the pattern n positions to the left or right.
8. Shifting 1 bit to the left equals multiplication by 2. Shifting one bit right is division by 2. Shifting n bits is multiplication or division by the nth power of 2.
9. (a) {1,3,5}
(b) {1,4,5}
(c) {4}
(d) {'a','2'}
(e) {1,10}
(f) {'m'}
(g) {5,7,9}
(h) {4,8}
(i) {'y'}
(j) {2,3,4,a,b,c}
10. (a) TRUE
(b) FALSE
(c) TRUE
11. In Modula-2, a record is a data abstraction designed to allow for aggregates of various types of related data named by a single identifier. In mathematics a record is an item taken from a cross product of two or more sets, producing tuples.
- 12.

```
TYPE
  String = ARRAY [0..50] OF CHAR;
  Time =
    RECORD
      Year, Month, Day : CARDINAL;
```

```

    END;
traveller =
    RECORD
        name : String;
        addr : String;
        airline : String;
        flightnum : CARDINAL;
        arrival : Time;
        departure : Time;
        luggage : BOOLEAN;
    END;

```

Mathematically:

(namestring) * (addstring) * (CARDINAL) * (timestring) * (timestring) * (BOOLEAN)

13.

```

PROCEDURE Filltraveller (VAR rectype : traveller);
BEGIN
    WITH rectype DO
        name := 'Jack';
        addr := '1600 Pensylvania Ave.';
        airline := 'Air Canada';
        flightnum := 654321;
        WITH arrival DO
            Year := 1999;
            Month := 12;
            Day := 31;
        END;
        WITH departure DO
            Year := 2000;
            Month := 1;
            Day := 1;
        END;
        luggage := TRUE;
    END;
END Filltraveller;

```

14.

```

student =
    RECORD
        name : string;
        sid : CARDINAL;
        age : [0..100];
        owing : REAL;
    END;

```

15.

```

PROCEDURE FillStudent (VAR sturec : student);

BEGIN

```

```

WITH sturec DO
  WriteString ("Enter student's name: ");
  ReadString (name);
  SkipLine;
  WriteLn;
  WriteString ("Enter SID#: ");
  ReadCard (sid);
  SkipLine;
  WriteLn;
  WriteString ("Enter age: ");
  ReadCard (age);
  SkipLine;
  WriteLn;
  WriteString ("Enter amount owing: ");
  ReadReal (owe);
  SkipLine;
  WriteLn;
END;
END FillStudent;

```

16.

```

TYPE
  Address =
    RECORD
      num : CARDINAL;
      street : string;
      postal : CARDINAL;
      city : string;
    END;
  Customer =
    RECORD
      name : string;
      addr : Address;
      amount : REAL;
    END;

PROCEDURE FillCustomer (VAR customer : Customer);
BEGIN
  WITH customer DO
    name := 'Steve';
    WITH addr DO
      num := 1492;
      street := 'Glover Rd.'
      postal := 654321;
      city := 'Langley';
    END;
    amount := 0.00
  END;
END FillCustomer;

```

17. An array should be used when grouping items of the same types and only a simple structure is needed A record on the

other had should be used when there is a diverse group of data that need to be combined into one abstraction. An array could be used to hold the number of runs that were scored in an inning of a baseball game where one would have an array of 1 to n. A record could be used to store patient information in a clinic stating the name, address, birthday, etc.

18. A qualified identifier has the individual fields of a record preceded by the record name itself. It is similar to a qualified import. The former are unqualified by WITH, the latter by FROM.

19. The first and simplest way is using regular ASCII characters. The record however can take up a lot of storage space. A second way of storing records is as binary information. The same idea is applied for this technique, but using a raw format to write instead of an ASCII format. The third is to use a random access technique. Here since memory size is similar to disk storage size, one can find a particular item by searching for a particular item or else, use the position markers and end-of-file markers.

20. Random access has the ability to calculate and set a position marker to read and write anywhere in the file. ISO Modula-2 uses the module RndFile to implement the random access model.

21. A sequential file should be used when a file does not need large amounts of insertions in different parts of the file, or when few changes are needed for the file in the span of its life. If the file is subject to frequent change, and requires insertions, then a random access file would be better.

22. A file position variable is a variable that marks a location in the file, whether it be at the beginning, the middle, or the end of the file. Manipulation of this variable is done automatically by procedures that read or write, and manually by StartPos, CurrentPos, EndPos, NewPos, and SetPos.

23. The end-of-file marker is important, because it tell the program where the end of the file is and it does not permit writing after that position.

24. OpenOld will open an existing file and set the read/write position to the start of the file. Using OpenClean will truncate the file to zero length.

25. After reading the file, the position marker moves to the end of the last read. If one were to write immediately after reading, the writing will be done in the incorrect position. To overcome this problem, one needs to reset the position marker to the position where the last record was read

26. Reading/writing to a position beyond the limits of a file produces a run time error.

27. Answer left to student.

28. Answer left to student.

29. Answer left to student.

Problems

Note: Not all problems are shown. Most problems are left up to students as labs.

```
(*      Created
      June 17 1999
      Chapter 9 Question 31  *)
```

```
MODULE LetterCheck;
```

```
FROM STextIO IMPORT
```

```
  WriteString, WriteLn, WriteChar, ReadString, ReadChar, SkipLine;
```

```
FROM SIOResult IMPORT
```

```
  ReadResult, ReadResults;
```

```
TYPE
```

```
  CharSet = SET OF CHAR;
```

```
  CharType = (const, vowel);
```

```
  CharArray = ARRAY [65..90] OF BOOLEAN;
```

```
VAR
```

```
  VowelSet : CharSet;
```

```
  CharList : CharArray;
```



```

    type : CharType;
    ch : CHAR;

PROCEDURE Init (array : CharArray);
(*  pre: none
    post: initializes an array of boolean to FALSE;  *)

VAR
    count : CARDINAL;

BEGIN
    FOR count := 65 TO 90 DO
        array[count] := FALSE;
    END;
END Init;

PROCEDURE WriteType (list : CharArray; type : CharType);
(*  pre: none
    post: writes out the type indicated *)

VAR
    count, format : CARDINAL;

BEGIN
    format := 0;

    FOR count := 65 TO 90 DO
        IF list[count] THEN

            IF type = vowel THEN

                IF VAL(CHAR, count) IN VowelSet THEN
                    WriteChar (VAL(CHAR, count));
                    WriteString ("      ");
                    INC (format);  (* for formatting purposes *)
                END;  (* end IF *)

            ELSE

                IF NOT ( VAL(CHAR, count) IN VowelSet) THEN
                    WriteChar (VAL(CHAR, count));
                    WriteString ("      ");
                    INC (format);  (* for formatting purposes *)
                END;  (* end IF *)

            END;  (* end IF *)

        END;  (* end IF *)

        IF format = 5 THEN
            WriteLn;
            format := 0;
        END;

    END;  (* end FOR *)

```

```

END WriteType;

BEGIN
    (* set the vowels *)
    VowelSet := CharSet {'A', 'E', 'I', 'O', 'U'};
    WriteString ("Enter a string and I will show you which are consonats and which ");
    WriteLn;
    WriteString ("vowels.");
    WriteLn; WriteLn; WriteLn;
    WriteString ("Enter the string: ");
    (* initialize the string *)
    Init (CharList);
    (* check and mark of vowels used *)

    REPEAT
        ReadChar(ch);
        IF (ORD(CAP(ch)) >= 65) AND (ORD(CAP(ch)) <= 90) THEN
            CharList[ORD(CAP(ch))] := TRUE;
        END;
    UNTIL ReadResult () # allRight;

    SkipLine;
    WriteLn; WriteLn;
    (* write out the results *)
    WriteString ("*****VOWELS*****");
    WriteLn;
    type := vowel;
    WriteType (CharList, type);
    WriteLn; WriteLn;
    WriteString ("*****CONSONANTS*****");
    WriteLn;
    type := const;
    WriteType (CharList, type);
    WriteLn; WriteLn; WriteLn;
    (* let user read results before killing program *)
    WriteString ("Press ENTER to continue");
    SkipLine;
END LetterCheck.

(*    Created
    June.17.1999
    Chapter 9 Question 37
    NO ERROR TRAPPING

    This program collects information about individuals and stores them in array
    of records. The student may elaborate on this an store the record to a file
    NOTE: no error checking is done on the inputs. *)

MODULE Collection;

FROM STextIO IMPORT
    WriteString, WriteLn, ReadString, ReadChar, WriteChar, SkipLine;
FROM SRealIO IMPORT

```

```

    ReadReal, WriteFixed;
FROM SWholeIO IMPORT
    ReadCard;

CONST
    max = 3;

TYPE
    String = ARRAY [0..30] OF CHAR;
    sexType = (M, F);
    List = ARRAY [1..max] OF CARDINAL;
    person =
        RECORD
            name : String;
            height : REAL;
            mass : REAL;
            sex : sexType;
            hair : String;
            eye : String;
            church : String;
        END;

VAR
    quit : BOOLEAN;
    option, howmany : CARDINAL;
    Person : person;
    list : List;

PROCEDURE FillPerson (VAR recType : person);
(*  pre: none
    post: fills a with given information *)

VAR
    tempnum : REAL;
    tempsex : CHAR;

BEGIN
    WITH recType DO
        WriteString ("Enter the name: ");
        ReadString (name);
        SkipLine;
        WriteLn;
        WriteString ("Enter the hight (in cm): "); (* since we use metric in canada *)
        ReadReal (height);
        SkipLine;
        WriteLn;
        WriteString ("Enter the weight (in kgs): "); (* since we use metric in canada *)
        ReadReal (mass);
        SkipLine;
        WriteLn;
        WriteString ("Enter the sex: ");
        ReadChar (sex);
        SkipLine;

```

```

    IF (tempsex = 'M') OR (tempsex = 'm') THEN
        sex := M;
    ELSE
        sex := F;
    END;

    WriteLn;
    WriteString ("Enter the hair colour: ");
    ReadString (hair);
    SkipLine;
    WriteLn;
    WriteString ("Enter the eye colour: ");
    ReadString (eye);
    SkipLine;
    WriteLn;
    WriteString ("Enter the church affiliation: ");
    ReadString (church);
    SkipLine;
    WriteLn;
END (* end WITH *)
END FillPerson;

PROCEDURE menu (VAR option : CARDINAL);

BEGIN
    WriteString ("*****");
    WriteLn;
    WriteString ("Select an option");
    WriteLn; WriteLn;
    WriteString ("1. Add an individual to the list");
    WriteLn;
    WriteString ("2. Display the list");
    WriteLn;
    WriteString ("3. Exit");
    WriteLn;
    WriteString ("*****");
    WriteLn; WriteLn;
    WriteString ("Option desired: ");
    WriteLn;
    ReadCard (option);
    SkipLine;
END menu;

PROCEDURE DisplayRec (list : List; recType : person);

VAR
    count : CARDINAL;
BEGIN
    FOR count := 1 TO howmany DO
        WITH recType DO
            WriteString ("Name: ");
            WriteString (name); WriteLn;
            WriteString ("Height: ");

```

```

WriteFixed (height, 2, 1);WriteLn;
WriteString ("Weight: ");
WriteFixed (mass, 2, 1);WriteLn;
WriteString ("Sex: ");

IF sex = M THEN
    WriteChar ('M');
ELSE
    WriteChar ('F');
END;

WriteLn;
WriteString ("Hair Colour: ");
WriteString (hair);WriteLn;
WriteString ("Eye Colour: ");
WriteString (eye);WriteLn;
WriteString ("Church Affiliaiton: ");
WriteString (church);WriteLn;

```

```

END; (* end WITH *)

```

```

    WriteLn; WriteLn;
END; (* end FOR *)

```

```

END DisplayRec;

```

```

(* start main program *)

```

```

BEGIN
    WriteString ("This program will take input from the user and store it in a list.");
    WriteLn; WriteLn; WriteLn;
    howmany := 0;

    REPEAT
        menu(option);

        IF option = 1 THEN
            FillPerson (Person);
            INC (howmany);
        ELSIF option = 2 THEN
            DisplayRec (list, Person);
            WriteString ("Press ENTER to continue");
            WriteLn;
            SkipLine;
        ELSE
            quit := TRUE;
        END;

    UNTIL quit;

END Collection.

```

CHAPTER 10

Questions

1. A block is a sequence of declarations followed by a block body. It is a component of program modules, implementation modules, local modules, and procedure declarations. In Modula-2, blocks have names.
2. A procedure block includes a parameter list and the module blocks seen thus far does not.
3. A module may have a body if it is a program, implementation, or local module or, though a program module that neither has nor imports a module block body will not do anything. A definition module, however, must not have a body.
4. Scope is the portion of a program where an identifier is visible.
5. A procedure's parameters and local variables are in its scope, but its own name is in the surrounding scope and visible inside the procedure by inheritance.
6. A procedure's name might not be visible inside itself if the name is cut off by its reuse as a parameter or local variable.
7. The term local means that the entity is in the scope of the module or procedure defining it. Global on the other hand means that the entity is defined externally to a procedure and inherited by it.
8. Using too many global variables can result in difficulty keeping track of them. Failure to keep track of global variables may result in writing code that modifies the value of a variable that is important to the correct functioning of some other part of the program.
9. An assignment to the *z* in the innermost procedure has no effect on the ones of the same name that are external to the procedure at either level. If the assignment is made inside another procedure declared at the same level as the outermost *z*, its value is affected unless the said procedure has a *z* of its own. If the assignment takes place in a third-level nested procedure without a *z* of its own, the one altered is that in the immediately surrounding procedure (the second one nested).
10. Visibility is the ability for an entity to be used in a statement in some portion of a program.
11. A side effect is the modification, by a procedure, of some variable global to it. Normally this is done by using a variable parameter or by assignment on return from a function procedure.
12. Side effects are good when the intent of the side effect is to manipulate the global variable. It is bad when it changes a global variable that it is not intended to change. That is, all such changes should be planned and documented.
13. It is common to use names such as "count" or "counter" as loop control variables. As long as the same name is used in a different scope, this isn't a problem.
14. Visibility of entities declared in procedures is inherited hierarchically and automatically inward with increasing levels, but visibility of entities declared in modules must be explicitly and manually controlled using **IMPORT** and **EXPORT**.
15. The outermost level is referred to as the main (program) module. In other words, the outermost scope is local to the main program.
16. Two items with the same name can be imported into the same scope as long as one of them is imported qualified.
17. If the item being exported has the same name as another entity in that scope, it has to be EXPORTed as **QUALIFIED**.
18. An item that is EXPORTed does not need a reference to the module it was exported from. An item EXPORTed as **QUALIFIED** needs to be qualified with the module name. For example, consider the following modules:

```
MODULE Outer;  
  
  MODULE Inner1;  
    EXPORT QUALIFIED item1  
    VAR  
      item1 : CARDINAL;  
  END Inner1;  
  
  MODULE Inner2;  
    EXPORT item2;
```

```

VAR
    item2 : CARDINAL;
END Inner2;

END Outer;

```

When item1 from Inner1 is to be used, it needs to be referenced because the name item1 is used in the MODULE Outer, thus it will be called as: Inner1.item1, whereas item2 can be used as is because of the manner of its export.

19. Program modules cannot have an EXPORT list, because they are at the outermost scope level. They have nowhere to EXPORT the variables to, as the (abstract) scope of library modules surrounding them cannot be manipulated except by adding another exporting library module.

20. A standard identifier is a built-in identifier of a pre-defined constant, variable, procedure, type, or module. It may be re-used in the program. A reserved word is a special word or symbol used as part of the framework or punctuation of a module. It cannot be used other than for its intended purposes.

21. A dynamic module is a module that resides in a procedure. It only exists when the procedure is invoked.

22. If a standard identifier name is cut off by its use in a procedure, it will still be available in an enclosed dynamic module.

23. A Fibonacci sequence is a sequence in which after the first two numbers, all subsequent numbers in the sequence are generated by adding the previous two.

24. A type that is exported from a definition module, but whose details are hidden in the implementation module is called an opaque type. Types whose details are visible (normal types) are called transparent types.

25. The type of a procedure is the form or pattern of all its parameter types and/or return types taken together. The presence of this facility allows for procedure variables.

26.

```

(a)      TYPE
           writeToFile : PROCEDURE (VAR ChanId; ARRAY OF CHAR; FlagSet; VAR
OpenResults);
(b)      TYPE
           writeToStd : PROCEDURE (ARRAY OF CHAR);
(c)      TYPE
           writeRealtoChan : PROCEDURE (IOChan.ChanId, REAL; INTEGER; CARDINAL);
(d)      TYPE
           AltAppend : PROCEDURE (ARRAY OF CHAR; VAR ARRAY OF CHAR);
(d)      TYPE
           NatLog : PROCEDURE (LONGREAL) : LONGREAL;

```

27. One should use a LOOP structure a loop structure when (a) there must be more than one exit point from a loop, (b) the natural exit point is in the middle of the sequence, or (c) when one must exit from a deeply nested loop or selection structure.

28. EXIT and RETURN are unconditional transfers of control.

29. HALT and RETURN (in the outermost block) will terminate a program.

30. A program could terminate if an exception is raised.

31. A termination event can be detected in a FINALLY clause.

32. One can detect if the program is in an exceptional execution state by using EXCEPTIONS.IsExceptionalExecution.

33. An exception is a violation of the run-time meaning of a program that when detected automatically alters the normal flow of control in the procedure or module body where it occurs, immediately transferring control to an exception handler for that procedure or module body, if one exists.

34. An exception is trapped by an exception handler or EXCEPT clause that may be attached to any block.

35. The first strategy is to let the control run off the end of the handler with the exception still raised. The second is to issue a RETURN command and clear the exception. This returns the state to normal execution and control passes back out to the caller of the procedure in which the exception occurred. Third, one can use a RETRY command which is similar to the RETURN except instead of giving control out to the caller of the procedure, it gives control to the beginning of the same procedure that initially raised the exception.

36. To declare and detect user-defined exceptions, one must define the exception type (usually an enumeration) and register it as the source of the exception with EXCEPTIONS. Procedures such as IsMyException and MyException should be defined to detect and discriminate one's own exceptions, which can be raised with RAISE, then handled in an EXCEPT clause.
37. When any exception is raised, a variable of type ExceptionSource must be passed, which only the module defining and raising the exception has access to.
38. Every block may contain one exception handler, and a FINALLY clause one more.
39. The body of a module may have two, one for itself and one for its FINALLY clause.
40. Note: _ is used to denote spaces
- ```
-----1-----2-----3-----4
-----7-----2-----3-----4
```
41. Point1: inner1C, inner1D.  
Point2: inner2E, inner2F, inner3H.  
Point3: inner3G, inner3H.  
Point4: outA, outB, Inner2.inner2E, Inner2.inner3H.

## Problems

Note: Not all problems are shown. Most problems are left up to students as labs.

```
(* Created
 June 21 1999
 Chapter 10 Question 42

 This program shows what happens when a procedure changes a variable global
 to it. This program should fill in the database at once. *)

MODULE BadGlobal;

FROM STextIO IMPORT
 WriteString, WriteLn, ReadString, ReadChar, SkipLine;
FROM SWholeIO IMPORT
 WriteCard, ReadCard;
FROM SRealIO IMPORT
 ReadReal, WriteFixed;

CONST
 start = 1;
 end = 3;

TYPE
 StudentRec =
 RECORD
 test : ARRAY [1..5] OF REAL;
 name : ARRAY [0..20] OF CHAR;
 final : REAL;
 END;

 StuRecType = ARRAY [start..end] OF StudentRec;

VAR
 count, option : CARDINAL;
 avg : REAL;
 Student : StuRecType;
```



```
full : BOOLEAN;
```

```
PROCEDURE Menu (VAR option : CARDINAL);
(* displays and gets menu option *)
```

```
BEGIN
 WriteString ("1. Populate Database");
 WriteLn;
 WriteString ("2. Display Final Grades");
 WriteLn;
 WriteString ("3. Exit");
 WriteLn; WriteLn;
 WriteString ("Enter an option: ");
 ReadCard (option);
 SkipLine;
 WriteLn;
END Menu;
```

```
(* bad procedure count is a global variable *)
PROCEDURE Fill(VAR student : StudentRec);
(* populate student database one student at a time *)
```

```
BEGIN
 WITH student DO
 WriteString ("Enter the student's name: ");
 ReadString (name);
 SkipLine;
 count := 1;

 (* enter the 5 grades *)
 WHILE count <= 5 DO
 WriteString ("Enter grade for test");
 WriteCard (count, 0);
 WriteString (": ");
 ReadReal (test[count]);
 SkipLine;
 WriteLn;
 INC (count);
 END;

 END;
END Fill;
```

```
PROCEDURE Display (VAR student : StuRecType);
(* displays the students and their final grade *)
```

```
VAR
 countout, countin : CARDINAL;
```

```
BEGIN
 WriteString ("---Grades---");
 WriteLn;

 FOR countout := start TO end DO
```

```

WITH student[countout] DO
 WriteString (name);
 WriteLn;
 final := 0.0;

 FOR countin := 1 TO 5 DO
 final := final + test[countin];
 END; (* end FOR *)

 final := final / 50.0 * 100.0;
 WriteString ("Final Grade:");
 WriteFixed (final, 2, 0);
 WriteString (" %");
 WriteLn; WriteLn;
END; (* end WITH *)
END; (* end FOR *)

END Display;

BEGIN
 WriteString ("This program will fill a database with data consisting ");
 WriteLn;
 WriteString ("of the student's name and 5 test scores.");
 WriteLn;

 REPEAT
 Menu (option);

 IF option = 1 THEN
 count := start;

 (* do x students continuous no stopping *)
 WHILE count <= end DO
 Fill(Student[count]);
 INC (count);
 END;

 ELSIF option = 2 THEN
 Display (Student);
 END;

 UNTIL option = 3;

 END BadGlobal.

 (* Created
 June.21.1999
 Chapter 10 Question 43

 This program demonstrates bad variable parameter use *)

MODULE BadVarUse;

FROM STextIO IMPORT
 WriteString, WriteLn, ReadString, ReadChar, SkipLine;

```

```

FROM SWholeIO IMPORT
 WriteCard, ReadCard;
FROM SRealIO IMPORT
 ReadReal, WriteFixed;
FROM SIOResult IMPORT
 ReadResult, ReadResults;

CONST
 start = 1;
 end = 3;

TYPE
 StudentRec =
 RECORD
 test : ARRAY [1..5] OF REAL;
 name : ARRAY [0..20] OF CHAR;
 final : REAL;
 END;
 StuRecType = ARRAY [start..end] OF StudentRec;

VAR
 count, option : CARDINAL;
 avg : REAL;
 Student : StuRecType;
 full : BOOLEAN;

PROCEDURE Menu (VAR option : CARDINAL);
(* displays and gets menu option *)

BEGIN
 WriteString ("1. Populate Database");
 WriteLn;
 WriteString ("2. Display Final Grades");
 WriteLn;
 WriteString ("3. Display Final Grades Curved");
 WriteLn;
 WriteString ("4. Exit");
 WriteLn; WriteLn;
 WriteString ("Enter an option: ");
 ReadCard (option);
 SkipLine;
 WriteLn;
END Menu;

PROCEDURE Fill(VAR student : StudentRec);
(* populate student database one student at a time *)

VAR
 count : CARDINAL;

BEGIN
 WITH student DO
 WriteString ("Enter the student's name: ");
 ReadString (name);

```

```

SkipLine;
count := 1;
final := 0.0;
FOR count := 1 TO 5 DO
 WriteString ("Enter grade for test");
 WriteCard (count, 0);
 WriteString (": ");
 ReadReal (test[count]);
 SkipLine;
 WriteLn;
 (* calculate the final grade as grades are coming in *)
 final := final + test[count];
END;
final := final / 50.0 * 100.0;
END;
END Fill;

PROCEDURE Display (VAR student : StuRecType);
(* displays the students and their final grade *)

VAR
 countout, countin : CARDINAL;

BEGIN
 WriteString ("---Grades---");
 WriteLn;
 FOR countout := start TO end DO
 WITH student[countout] DO
 WriteString (name);
 WriteLn;
 WriteString ("Final Grade:");
 WriteFixed (final, 2, 0);
 WriteString (" %");
 WriteLn; WriteLn;
 END;
 END;
END Display;

(* bad procedure it is only suppose to show the "curved" mark, not change the actual
*)
PROCEDURE Curve (VAR student : StuRecType);
(* displays the grades curved--somewhat at least gives the option of how much % each
student gets. *)

VAR
 countout, countin : CARDINAL;
 rate : REAL;

BEGIN
 WriteString ("By how much % do you want to give them? ");
 ReadReal (rate);
 SkipLine;
 WriteLn;

```

```

WriteString ("--Curved Grades--");
WriteLn;
FOR countout := start TO end DO
 WITH student[countout] DO
 WriteString (name);
 WriteLn;
 final := final + rate;
 WriteString ("Curved Grade:");
 WriteFixed (final, 2, 0);
 WriteString (" %");
 WriteLn; WriteLn;
 END;
END;
END Curve;

BEGIN
 WriteString ("This program will fill a database with data consisting ");
 WriteLn;
 WriteString ("of the student's name and 5 test scores.");
 WriteLn;

 REPEAT
 Menu (option);
 IF option = 1 THEN
 count := start;
 (* do "end" continuous *)

 WHILE count <= end DO
 Fill(Student[count]);
 INC (count);
 END;

 ELSIF option = 2 THEN
 Display (Student);
 ELSIF option = 3 THEN
 Curve (Student);
 END;
 UNTIL option = 4;

 END BadVarUse.

```

# CHAPTER 11

## Questions

1. Backtracking is the recursive use of trial-and-error steps through some kind of pattern, retracing the logical path back to the last success and then following a different branch.
2. The IF and CASE statements are the Modula-2 selection constructs.
3. One should opt for the CASE statement when: (i) the decision involves only the value of a single variable, (ii) there are several (but not very many) adjacent alternative values, (iii) the majority of the alternatives do NOT fall into the ELSE category.
4. CARDINAL has too many values, unless there is an optimizing compiler to reduce the size of the code.
5. The range in one case of a CASE statement cannot be interlaced with ranges in other cases.
6. Range checking can be turned off when error trapping code has been thoroughly tested.
7. Answers may vary.
8. In Modula-2, a pragma is a directive to the compiler that is included in the source file it is compiling. It gives the programmer a chance to give directions to the compiler while the code is being compiled. It is written in pragma brackets: "<\*" and "\*>".
9. Answers may vary.
10. Some strategies for reducing running time include: (i) Control Run-Time Checking (ii) Compile programs for specified environments (iii) Fine-tune loops (iv) link code segments or (v) use an optimized linker (the latter two are system dependent).
11. Answers may vary.
12. A program library is a collection of the code of one or more modules into a single file.
13. A structure constructor has the form typeName{values}. It is able to construct specific sets, arrays, and records using literals.
- 14.

```
TYPE
 realArray = ARRAY [1..10] OF REAL;
VAR
 reels : realArray;
BEGIN
 reels := realArray{0.0 BY 10};
```

15. The case separator is denoted by the "|" (vertical bar) character and is used to separate cases in a CASE statement. The statement separator is the ";" character and is used to denote the end of a statement in a block of code. Example for case and statement separators:

```
CASE num OF
 1:
 <statement1<statement2<statement3<statement1>;
END;
```

16. If none of cases match the value of case variable then an error will be generated at execution time. To avoid this

problem, an ELSE clause can be attached to the CASE statement to handle any other values that are not handled by a case.

17. A variant record field is a record field that gives the option to select different fields from a CASE clause embedded in the record depending on the value of a tag variable. One would declare a variant field by using a case statement in a record with the expression as a record field and the cases as an optional record fields.

## 18. TYPE

```
Classification = (single, married, divorce);
Date =
 RECORD
 year, month, day : CARDINAL;
 END;

Employee =
 RECORD
 lastname : nameString;
 firstname : nameString;
 sex : SexType; (* enumerated data type of M or F *)
 dob : Date;
 CASE marital : Classification OF
 single :
 nextofKin : nameString; |
 married :
 marriageDate : Date;
 spouse : nameString;
 kids : CARDINAL; |
 divorce:
 divorceDate : Date;
 nextofKin : nameString;
 END; (* end CASE *)
 CASE executive : BOOLEAN OF
 TRUE:
 salary : REAL;
 car : REAL;
 rank : RankString; |
 FALSE:
 hourly : REAL;
 rank : RankString;
 END; (* end CASE *)
 END (* end RECORD *)
```

## 19. CONST

```
defSingle = Employee{ "", BY 2, M, Date{0 BY 3}, single, "", TRUE, 0 BY 2, "" };
and there are 5 more to make a complete set.
```

20. Using a variant record can improve the simplify the structure of the RECORD. However, it can also make many nested CASEs that could make the programmer confused when debugging. If most of the fields in the records are identical, a variant form may be best.

21. Unrolling a loop is writing two or more repetitions of the steps in each iteration of the loop. This is done because some control structures are more efficient than repetition.

22. More efficient programs run faster, saving time and money. However, most efficiency improvements can be made with minimal changes to code, and determining the most efficient code technique may not be practical in any reasonable amount of time.

## Problems

Note: Not all problems are shown. Most problems are left up to students as labs.

```
(* Created
 June.22.1999
 Chapter 11, Question#26

 No error checking has been used for this program...be careful when inputting
 data!! *)

MODULE DataBase;

FROM STextIO IMPORT
 WriteString, ReadString, WriteLn, ReadChar, SkipLine;
FROM SWholeIO IMPORT
 ReadCard, WriteCard;
FROM SRealIO IMPORT
 ReadReal, WriteFixed;
FROM SeqFile IMPORT (* use regular text so we can see what the file contains *)
 OpenWrite, OpenRead, ChanId, Close, OpenResults, read, write, old;
FROM Strings IMPORT
 Compare, CompareResults, Delete;
IMPORT TextIO;
IMPORT RealIO;
IMPORT WholeIO;

CONST
 min = 1;
 max = 10;

TYPE
 Name = ARRAY [0 .. 20] OF CHAR;
 MonthType = ARRAY [0..9] OF CHAR;
 Classification = (student, faculty, staff);
 Year = (freshman, sophomore, junior, senior);
 Rank = (instructor, assistant, associate, professor);
 Job = (secretary, maintenance, janitor);
 Date =
 RECORD
 year : CARDINAL;
 month : MonthType;
 day : [1 .. 31];
 END; (* of the record Date *)
 Person =
 RECORD
```



```

 lastname, firstname : Name;
 birthdate : Date;
 (* use previous declarations for these. *)
 male : BOOLEAN;
 CASE status : Classification OF (* variant part here *)
 student:
 idnumber : CARDINAL;
 year : Year |
 faculty:
 position : Rank;
 pay : REAL |
 staff:
 occupation : Job;
 END; (* case *)
 married : BOOLEAN;
END; (* of the record Person *)
RecType = ARRAY [min..max] OF Person;

```

VAR

```

 RecArray : RecType;
 option, current : CARDINAL;

```

```

PROCEDURE Menu (VAR option : CARDINAL);
(* pre: none
 post: changes the value of option *)

```

BEGIN

```

 WriteString ("*****");
 WriteLn;
 WriteString ("1. Insert Record ");
 WriteLn;
 WriteString ("2. Edit Record ");
 WriteLn;
 WriteString ("3. Display Info ");
 WriteLn;
 WriteString ("4. Save Record");
 WriteLn;
 WriteString ("5. Get Record File ");
 WriteLn;
 WriteString ("6. Exit ");
 WriteLn;
 WriteString ("*****");
 WriteLn; WriteLn;
 WriteString ("Enter an option: ");
 ReadCard (option);
 SkipLine;
 WriteLn; WriteLn; WriteLn;

```

END Menu;

```

PROCEDURE Insert (VAR RecArray : RecType; currentRecNum : CARDINAL);

```

```

(* pre: none
 post: Adds a record into the array *)

VAR
 tempcard : CARDINAL;
 tempreal : REAL;
 tempchar : CHAR;
 str : Name;
 str1 : MonthType;

BEGIN
 WITH RecArray[currentRecNum]
 DO
 WriteString ("Name:");
 WriteLn; WriteLn;
 WriteString ("Last Name: ");
 ReadString (lastname);
 SkipLine;
 WriteLn;
 WriteString ("First Name : ");
 ReadString (str);
 SkipLine;
 firstname := str;
 WriteLn;WriteLn;

 WITH birthdate DO
 WriteString ("Date of Birth: ");
 WriteLn; WriteLn;
 WriteString ("Month: ");
 ReadString (month);
 SkipLine;
 WriteLn;
 WriteString ("Day: ");
 ReadCard (tempcard);
 SkipLine;
 day := tempcard;
 WriteLn;
 WriteString ("Year: ");
 ReadCard (year);
 SkipLine;
 END;
 WriteLn; WriteLn;

 WriteString ("Male or Female (m/f): ");
 ReadChar (tempchar);
 SkipLine;

 IF CAP(tempchar) = 'M' THEN
 male := TRUE;
 ELSE

```

```

 male := FALSE;
END; (* IF *)

WriteLn; WriteLn;

WriteString ("Classification: ");
WriteLn;
WriteString ("1. student");
WriteLn;
WriteString ("2. faculty");
WriteLn;
WriteString ("3. staff");
WriteLn;
WriteString ("Choice: ");
ReadCard (tempcard);
SkipLine;
status := VAL (Classification, tempcard - 1);

CASE status OF
 student:
 WriteString ("Give i.d. number, please. ");
 ReadCard (idnumber);
 SkipLine;
 WriteLn;
 WriteString ("and enter the year 1 .. 4 ");
 WriteString ("of studies. ");
 ReadCard (tempcard);
 SkipLine;
 year := VAL(Year, tempcard - 1) |
 faculty:
 WriteString ("Enter the rank of the faculty member ");
 WriteLn;
 WriteString ("by number. A '1' for instructor, ");
 WriteLn;
 WriteString ("a '2' for assistant, a '3' for associate, ");
 WriteLn;
 WriteString ("or a '4' for a full professor. ");
 ReadCard (tempcard);
 SkipLine;
 position := VAL (Rank, tempcard - 1);
 WriteLn;
 WriteString ("How much is this faculty member paid? ");
 WriteLn;
 WriteString ("Answer using decimal point, please. ");
 ReadReal (pay);
 SkipLine; |
 staff:
 WriteString ("Please enter a '1' for a secretary, ");
 WriteLn;
 WriteString ("a '2' for a maintenance employee, ");

```

```

 WriteLn;
 WriteString ("or a '3' for a janitor. ");
 ReadCard (tempcard);
 SkipLine;
 WriteLn;
 occupation := VAL (Job, tempcard - 1); (* no bar here *)
END; (* CASE *)

```

```

WriteLn; WriteLn;

```

```

WriteString ("Married (y/n)? ");
ReadChar (tempchar);
SkipLine;

```

```

IF CAP(tempchar) = 'Y' THEN
 married := TRUE;
ELSE
 married := FALSE;
END; (* IF *)

```

```

END; (* WITH *)

```

```

END Insert;

```

```

PROCEDURE Display (record : RecType; currentRecNum : CARDINAL);
(* pre: none
 post: changes a displayed record *)

```

```

VAR
 count : CARDINAL;

```

```

BEGIN
 IF currentRecNum = 0 THEN
 WriteString ("No records available");
 WriteLn; WriteLn;
 ELSE

 FOR count := 1 TO currentRecNum DO
 WriteCard (count, 1);
 WriteString (" ");

 WITH record[count] DO
 WriteString (lastname);
 WriteString (" ");
 WriteString (firstname);
 END;

 WriteLn;
 END; (* end FOR *)

 END; (* end IF *)

```

```

 WriteLn; WriteLn;
END Display;

PROCEDURE DisplayFull (RecArray : RecType; recnum : CARDINAL);
(* pre: none
 post: displays full record information *)

VAR
 class : Classification;

BEGIN
 WITH RecArray[recnum] DO
 WriteString ("Name: ");
 WriteString (firstname);
 WriteString (" ");
 WriteString (lastname);
 WriteLn;

 WITH birthdate DO
 WriteString ("Date of Birth:");
 WriteString (" Month: ");
 WriteString (month);
 WriteString (" Day:");
 WriteCard (ORD(day), 0);
 WriteString (" Year:");
 WriteCard (year, 0);
 WriteLn;
 END; (* end WITH *)

 WriteString ("Gender: ");

 IF male THEN
 WriteString ("Male");
 ELSE
 WriteString ("Female");
 END;

 WriteLn;
 WriteString ("Classification: ");

 CASE status OF
 student:
 WriteString ("Student");
 WriteLn;
 WriteString ("Id Number: ");
 WriteCard (idnumber, 1);
 WriteLn;
 WriteString ("Year: ");

 IF year = freshman THEN
 WriteString ("Freshman");

```

```
ELSIF year = sophomore THEN
 WriteString ("Sophomore");
ELSIF year = junior THEN
 WriteString ("Junior");
ELSE
 WriteString ("Senior");
END; |
```

faculty:

```
WriteString ("Faculty");
WriteLn;
WriteString ("Rank: ");

IF position = instructor THEN
 WriteString ("Instructor");
ELSIF position = assistant THEN
 WriteString ("Assistant");
ELSIF position = associate THEN
 WriteString ("Associate");
ELSE
 WriteString ("Professor");
END;
```

```
WriteLn;
WriteString ("Salary: $");
WriteFixed (pay, 2, 1); |
```

staff:

```
WriteString ("Staff");
WriteLn;
WriteString ("Job: ");

IF occupation = secretary THEN
 WriteString ("Secretary");
ELSIF occupation = maintenance THEN
 WriteString ("Maintenance");
ELSE
 WriteString ("Janitor");
END;
```

```
END; (* end CASE *)
```

```
WriteLn;
WriteString ("Married: ");
```

```
IF married THEN
 WriteString ("Yes");
ELSE
 WriteString ("No");
END;
```

```
WriteLn; WriteLn;
```

```
END; (* end WITH *)
```

```
END DisplayFull;
```

```
PROCEDURE SaveRec (record : RecType; numtosave : CARDINAL) : BOOLEAN;
```

```
(* pre: none
```

```
post: saves the array of records to a file specified by user *)
```

```
VAR
```

```
file : ChanId;
```

```
filename : ARRAY [0..12] OF CHAR;
```

```
res : OpenResults;
```

```
count : CARDINAL;
```

```
BEGIN
```

```
WriteString ("Enter filename: ");
```

```
ReadString (filename);
```

```
SkipLine;
```

```
WriteLn;
```

```
OpenWrite (file, filename, read+write+old, res);
```

```
IF res = opened THEN
```

```
FOR count := 1 TO numtosave DO
```

```
WITH record[count] DO
```

```
(* save name *)
```

```
TextIO.WriteString (file, lastname);
```

```
TextIO.WriteString (file, " ");
```

```
TextIO.WriteString (file, firstname);
```

```
TextIO.WriteLine (file);
```

```
(* save birthdate *)
```

```
WITH birthdate DO
```

```
TextIO.WriteString (file, month);
```

```
TextIO.WriteString (file, " ");
```

```
WholeIO.WriteCard (file, ORD(day), 1);
```

```
TextIO.WriteString (file, " ");
```

```
WholeIO.WriteCard (file, year, 1);
```

```
TextIO.WriteLine (file);
```

```
END; (* end WITH *)
```

```
(* save gender *)
```

```
WholeIO.WriteCard (file, ORD(male), 1);
```

```
TextIO.WriteLine (file);
```

```
(* save classification *)
```

```
WholeIO.WriteCard (file, ORD(status), 1);
```

```
TextIO.WriteString (file, " ");
```

```
CASE status OF
```

```
student:
```

```
WholeIO.WriteCard (file, idnumber, 1);
```

```
TextIO.WriteString (file, " ");
```

```

 WholeIO.WriteCard (file, ORD(year), 1); |
 faculty:
 WholeIO.WriteCard (file, ORD(position), 1);
 TextIO.WriteString (file, " ");
 RealIO.WriteFixed (file, pay, 2, 1); |
 staff:
 WholeIO.WriteCard (file, ORD(occupation), 1);
 END; (* end CASE *)

```

```

 TextIO.WriteLine (file);
 (* save marital status *)
 WholeIO.WriteCard (file, ORD(married), 1);
 TextIO.WriteLine (file);
 TextIO.WriteLine (file);

```

```

 END; (* end WITH *)

```

```

END; (* end FOR *)

```

```

TextIO.WriteString (file, "<>");

```

```

Close (file);

```

```

RETURN TRUE;

```

```

ELSE

```

```

 Close (file);

```

```

 RETURN FALSE;

```

```

END; (* end IF *)

```

```

END SaveRec;

```

```

PROCEDURE GetRec (VAR record : RecType; VAR numofRecs : CARDINAL) : BOOLEAN;

```

```

(* pre: none

```

```

 post: retrieves a record from a file and puts it in an array *)

```

```

VAR

```

```

 done : BOOLEAN;

```

```

 count, tempcard : CARDINAL;

```

```

 file : ChanId;

```

```

 res : OpenResults;

```

```

 filename : ARRAY [0..13] OF CHAR;

```

```

BEGIN

```

```

 WriteString ("Enter name of record file to retrieve: ");

```

```

 ReadString (filename);

```

```

 SkipLine;

```

```

 (* initialize variables *)

```

```

 numofRecs := 0;

```

```

 done := FALSE;

```

```

 count := 1;

```

```

 (* open file *)

```

```

 OpenRead (file, filename, read+write+old, res);

```

```

 IF res = opened THEN

```

```

 WHILE NOT done DO

```

```

 WITH record[count] DO

```



```

(* get name *)
TextIO.ReadToken (file, lastname);
(* check for end of file *)
IF Compare (lastname, "<>") # equal THEN
 TextIO.ReadString (file, firstname);
 TextIO.SkipLine (file);
 Delete (firstname, 0, 1); (* to counter the extra space effect *)

 (* get birthdate *)
 WITH birthdate DO
 TextIO.ReadToken (file, month);
 WholeIO.ReadCard (file, tempcard);
 day := tempcard;
 WholeIO.ReadCard (file, year);
 TextIO.SkipLine (file);
 END; (* end WITH *)

 (* get gender *)
 WholeIO.ReadCard (file, tempcard);
 male := VAL(BOOLEAN, tempcard);
 TextIO.SkipLine (file);

 (* get classification *)
 WholeIO.ReadCard (file, tempcard);
 status := VAL(Classification, tempcard);

 CASE status OF
 student:
 WholeIO.ReadCard (file, idnumber);
 WholeIO.ReadCard (file, tempcard);
 year := VAL(Year, tempcard); |
 faculty:
 WholeIO.ReadCard (file, tempcard);
 position := VAL(Rank, tempcard);
 RealIO.ReadReal (file, pay); |
 staff:
 WholeIO.ReadCard (file, tempcard);
 occupation := VAL(Job, tempcard);
 END; (* end CASE *)

 TextIO.SkipLine (file);

 (* get marital status *)
 WholeIO.ReadCard (file, tempcard);
 married := VAL(BOOLEAN, tempcard);
 TextIO.SkipLine (file);
 TextIO.SkipLine (file);
 INC (count);
 INC (numofRecs);
ELSE

```

```

 done := TRUE;
 END;

 END; (* end WITH *)
END; (* end WHILE *)
Close (file);
RETURN TRUE;
ELSE
 Close (file);
 RETURN FALSE;
END; (* end IF *)
END GetRec;

(* begin main block *)
BEGIN
 current := 0;
 REPEAT
 Menu (option);

 IF option = 1 THEN
 INC (current);
 Insert(RecArray, current);
 ELSIF option = 2 THEN
 Display (RecArray, current);

 IF current # 0 THEN
 WriteString ("Choose a record to edit: ");
 ReadCard (option);
 SkipLine;
 Insert (RecArray, option);
 option := 2;
 END;

 ELSIF option = 3 THEN
 Display (RecArray, current);

 IF current # 0 THEN
 WriteString ("Choose a record to display full info: ");
 ReadCard (option);
 SkipLine;
 DisplayFull(RecArray, option);
 option := 3;
 END;

 ELSIF option = 4 THEN

 IF SaveRec(RecArray, current) THEN
 WriteString ("Save successful");
 ELSE
 WriteString ("An error has occurred couldn't save");
 END;

```

```

ELSIF option = 5 THEN

 IF GetRec(RecArray, current) THEN
 WriteCard (current, 1);
 WriteString ("record(s) retrieved");
 ELSE
 WriteString ("Error occured when retrieved");
 END;

END (* End IF *);

WriteLn;
UNTIL option = 6;
END DataBase.

(* Created
June.21.1999
Chapter 11 Question 27 *)

MODULE NumberCheck;

FROM STextIO IMPORT
 WriteString, WriteLn, ReadChar, SkipLine;
FROM SWholeIO IMPORT
 ReadInt, WriteInt;
FROM SIOResult IMPORT
 ReadResult, ReadResults;

VAR
 number, count, sum : INTEGER;
 exit : CHAR;

BEGIN
 REPEAT
 (* get input *)
 WriteString ("Enter a whole number to be evaluated: ");
 ReadInt (number);

 IF ReadResult () # allRight THEN
 WriteString ("Not the right a valid number");
 END;

 SkipLine;
 WriteLn;
 (* initialize *)
 count := 1;
 sum := 0;

 (* evaluate if it's a perfect, deficitent, or abundant *)
 WHILE count < number DO

```

```

 IF number MOD count = 0 THEN
 sum := sum + count;
 END;

 INC (count);
END;

(* print *)
WriteString ("The number");
WriteInt (number, 0);
WriteString (" is ");
count := sum - number;

CASE count OF
 0:
 WriteString ("PERFECT");|
 1..MAX(INTEGER):
 WriteString ("ABUNDANT");
 ELSE
 WriteString ("DEFICIENT");
 END;

(* redo *)
WriteLn;
WriteString ("Do you want to do another (y/n)? ");
ReadChar (exit);
SkipLine;
UNTIL CAP(exit) = 'N';
END NumberCheck.

```

# CHAPTER 12

## Questions

1. A pointer is a variable that identifies a memory location and holds the address of some other entity. It "points" to the other entity.
2. The "^" character is known as the dereferencing operator.
3. A use of point employs the value of a memory location. Using point^ references the actual value. It has been dereferenced.
4. All pointers are assignment compatible with the type ADDRESS and with the type of NIL.
5. Pointers are more easily moved around than actual values. Moving a data value may take significantly more steps than changing the value of a pointer.
6. Some pointers should be guarded, because they could hold the value NIL. Referencing an unguarded pointer when it holds the value NIL causes an error.
7. Procedures that can do pointer arithmetic are (from SYSTEM): ADDADR, SUBADR, DIFADR.
8. A dangling pointer is a pointer that has been left pointing to a memory location that has been deallocated.
9. Passing a pointer as a value parameter makes a copy of the pointer and therefore will be pointing to the same memory location as the original pointer. Since it is pointing to the same memory location, any changes to the value dereferenced will be reflected in the original memory, replicating the effect of a variable procedure.
10. A stack is an area of memory where procedure activation records are placed. Each time a procedure is called, an activation record is placed on top of the last activation record. Every time a procedure calls itself, another copy of its activation record is created, and they have to be removed in reverse order they were placed. The stack's ability to grow is dependent on how much memory is allocated to the stack and how deep the current chain of procedure calls is. A stack pointer is a marker that delimits the top end of the currently allocated stack. It is not available for program manipulation.
11. An activation record is the contents of the memory assigned to a procedure for its parameters and variables when it is invoked.
12. Since a new activation record is created for each procedure call for that specific procedure, each level of activation yields a new set of variables for independent manipulation. Once a recursive call is finished it can give a value back to the previous activation record and the values are passed down the chain to the original procedure. The amount of recursion that can be handled depends on how much memory is allocated to the stack.
13. The use of activation records is automatic, not under programmer control and so not truly dynamic.
14. A heap is the region of memory above the stack and in which program-controlled dynamic allocation and deallocation of memory can take place.
15. NEW and DISPOSE are used in Modula-2 to manage the heap.
16. These two procedures are relatively low level and their implementations may vary by machine, so

they call ALLOCATE and DEALLOCATE, which must be visible. Their other magical property is that they have more than one possible syntax.

17. The two library procedures are ALLOCATE and DEALLOCATE from Storage. Any procedures with these names can be used, however, so the programmer can define ones other than those in Storage and make them visible.

18. A function procedure identifier cannot be dereferenced.

19. One can have the record contain another pointer; thus every allocation of such a record generates another pointer.

20. Two applications of dynamic memory include a dynamic database and an open ended queue.

21. A linked list consists of a sequence of dynamically allocated data items; each element of which includes a pointer to the next item on the list. The first item is the head, the last the tail. One needs at least a head pointer pointing to the head of the list.

22. Two extensions to the linked list ADT are append and delete. In append an item is linked to the last item of the linked list. In delete, an item is removed from one of the places in the list.

23.

```
DEFINITION MODULE LinkListADT;

TYPE
 ItemPointerType = POINTER TO ItemData;
 ItemData =
 RECORD
 number : CARDINAL;
 toPoint : ItemPointerType;
 END;

PROCEDURE Create (VAR List : ItemPointerType);

PROCEDURE Dispose (VAR List : ItemPointerType);

PROCEDURE AppendFirst (item : ItemData);

PROCEDURE Append (item : ItemData);

PROCEDURE Delete (index : CARDINAL);

END LinkListADT.
```

24. An opaque data type is a type that is defined in a definition module, but whose details are hidden in the implementation module. They have to be declared as another opaque type or as a pointer type.

25. This limitation was made because it was thought that a compiler could only set aside memory for opaque types if they were all the same size and it was also thought that this implies they must be pointers.

26. When keeping track of variant dynamic records, one must reserve only the amount necessary to store

each variant field. More house keeping is needed to ensure that the right amount of space is allotted for each variant.

27. The varying amounts of memory needed for variants of a dynamic record are handled using the alternate syntax of NEW.

28. Static memory allocation is determined when the program starts to run. The maximum amount of memory for the largest possible variant must be set aside.

29. Sub-record tags do affect the memory allocation, whether static or dynamic.

30. One would use handle^^ to point to the integer.

31. The built-in identifier SIZE will return the size of the type point^.

### Problems

Note: Not all problems are shown. Most problems are left up to students as labs.

```
(* Created
 June 22 1999
 Chapter 12 Question 34 *)
```

```
MODULE TestVariantSize;
```

```
FROM STextIO IMPORT
 WriteString, WriteLn, SkipLine;
```

```
FROM SWholeIO IMPORT
 WriteCard;
```

```
FROM Storage IMPORT
 ALLOCATE, DEALLOCATE;
```

```
FROM SYSTEM IMPORT
 TSIZE;
```

```
TYPE
```

```
 EnumDatType = (red, green, blue);
```

```
 Item = POINTER TO SomeItem;
```

```
 SomeItem =
```

```
 RECORD
```

```
 int : INTEGER;
```

```
 card : CARDINAL;
```

```
 real : REAL;
```

```
 CASE bool : BOOLEAN OF
```

```
 TRUE:
```

```
 colour : EnumDatType; |
```

```
 FALSE:
```

```
 char : CHAR;
```

```
 str : ARRAY [0..19] OF CHAR;
```

```
 END; (* end CASE *)
```

```
 CASE color : EnumDatType OF
```

```

 red:
 array1 : ARRAY [0..9] OF CARDINAL; |
 green:
 array2 : ARRAY [0..9] OF INTEGER; |
 blue:
 array3 : ARRAY [0..9] OF Item;
END; (* end CASE *)
END; (* end RECORD *)

```

VAR

```

 test1, test2, test3 : Item;

```

BEGIN

```

 NEW (test1, blue);
 WriteString ("The size of test1^ is :");
 WriteCard (SIZE(test1), 0);
 WriteLn;
 WriteString ("The tsize of test1^ is :");
 WriteCard (TSIZE(SomeItem, blue), 0);
 WriteLn;
 NEW (test2, TRUE);
 WriteString ("The size of test2^ is :");
 WriteCard (SIZE(test2), 0);
 WriteLn;
 WriteString ("The tsize of test2^ is :");
 WriteCard (TSIZE(SomeItem, TRUE, blue), 0);
 WriteLn;
 NEW (test3, FALSE);
 WriteString ("The size of test3^ is :");
 WriteCard (SIZE(test3), 0);
 WriteLn;
 WriteString ("The tsize of test3^ is :");
 WriteCard (TSIZE(SomeItem, FALSE, green), 0);
 WriteLn;
 SkipLine;
END TestVariantSize.

```



# CHAPTER 13

## Questions

1. The more complex code written for a binary search results in searches that are much faster for large data sets.
2. In general, simple sorts have a performance proportional to the square of the number of items, whereas advanced sorts have a performance proportional to  $n \log_2(n)$ .
3. The big - O notation signifies the proportionality mentioned above. Literally, it is "order of".
4. "In place" sorting refers to sorting back into the same data space as the unsorted items were presented to the sorter. Usually, this means that an array of unsorted items is presented, and the same array is returned, now sorted. This is in contrast to sorting, say, from one array to another.
5. None of the sorts other than Shell's are named after a person.
6. Left as an exercise for the student.
7. Answers may reference sorting railroad boxcars by load, date, number, origin, destination, or name of railroad. Similar considerations could be applied to airline passengers, taxpayers, customers, suppliers, students at a school, employees, etc.
8. Such conditions prevent an invalid array reference. Without it, the count would go outside the appropriate range of the array.
9. When answering this question, you should try to give a theoretical reason, then run the code and see what happens.
10. The issue here is the need to copy data. Any method that does so will suffer a performance hit as the size of the data item increases. If a value parameter is used for the array (because you're not sorting in place) data is copied when the procedure is entered.
11. Left as an exercise for the student.
12. Left as an exercise for the student.

## Problems

Note: At this stage, students should not need any problem solutions.

# CHAPTER 14

## Questions

1. Semi generic code is relatively free from implementation considerations, such as what data type will be used in an actual list. Some manual effort is still needed to re-code the module to take a new data type, but this may be as little as one line near the beginning if the design is well done.
2. Before implementing a list ADT, one must decide what data is to be listed, whether the list is a generic one (re-usable), whether the list is to be unidirectional or bidirectional, sorted or unsorted, and how many pointers to data in the list will be kept. There are some lesser issues such as how much data will be kept on the list (number of items, for instance).
3. A queue is a first-in-first-out structure and a stack is a last-in-first-out structure.
4. A lookup table or, for short, a table is a finite set of ordered pairs  $\{(x, f(x))\}$ , that is, it is a function on a finite domain (the first column) to some range (the second column).
5. How one implements a data structure is independent from the abstraction itself. The table could in some systems be implemented as an array, provided different array components could be of different types.
6. A tree is a structure in which all nodes may have more than one successor and all nodes but one (the root) have one predecessor.
7. In a binary tree, the maximum number of successors of a node is two.
8. A root has no predecessor.
9. leaves have no successors.
10. Interior nodes have a predecessor and one or more successors. To put it another way, they are the root of a subtree, but not a root of the entire tree.
11. A node in a binary tree has degree zero, one, or two.
12. A full binary tree with eight levels has 255 occupied nodes.
13. The mathematical sense of "full" simply means that all nodes on each used level are occupied. The memory sense is that there is no more place to put another node.
14. Searching a linked list requires examination of up to all  $n$  items, whereas, searching a binary tree requires at most  $n \log_2(n)$  comparisons.
15. Even with two pointers, a list is still linear, and searching is proportional to the number of items.
16. The letters ISAM mean Indexed Sequential Access Method.
17. Hashing refers to mapping data items into an index space. The function is not one-to-one (or there would be no point) so several items in the data space could have the same index value.
18. Tree traversal can be pre-order (root-left-right); in-order (left-root-right); or post-order (left-right-root).
19. Assuming that one works left to right as with a binary tree, the root of the ternary node could be processed before any one of the three children, or after the last, a total of four possibilities.

20-21. Try it and see.

22-25. Left as an exercise.

### **Problems**

Note: At this stage, students should not need any problem solutions.

# CHAPTER 15

## Questions

1. The "B" in B-tree stands for "balanced".
2. I won't exhaust the differences here, but note that a binary tree node has at most two children, and the number of possible children on a given level is predetermined. In a B-tree, the number of children of a node varies more widely, and so does the number of nodes on a level. A binary tree can get out of balance; a B-tree cannot.
- 3-5. Left as an exercise.
6. A heap is a sequence of data nodes  $n_1, n_2, n_3, n_4, \dots, n_{\max}$  in which  $n_i \leq n_{2i}$  and  $n_i \leq n_{2i+1}$  for all  $i$  with  $1 \leq i \leq \max/2$ . In a heap every parent is greater than its child. A reverse heap has the opposite relationship between parents and children.
7. "Generic" means without respect to implementation. In particular, a generic data structure such as a heap can be constructed without worrying about the kind of data in the heap.
8. Left as an exercise.
9. An ARRAY OF LOC is, in a sense the most generic form of data of them all, because any other data is compatible with it when passed as a parameter. On the other hand, there are some technical limitations to this (procedure types), and type checking (why you use a high level language) is lost.
- 10-13. Left as an exercise.
14. This is fairly straightforward, but you will need a parameter of the type ADDRESS to serve as a pointer to the temporary location. After all, the procedure does not know the type and so cannot declare the temporary itself.
15. Left as an exercise.
16. Two that do are Java and Modula-2. A third is left as an exercise.
17. Defragmentation is the gathering together of chunks of memory so that unallocated memory is all in one contiguous chunk. Garbage collection refers to deallocating (and so making available again) chunks of memory no longer being referred to by any pointer.
18. Left as an exercise.

## Problems

Note: At this stage, students should not need any problem solutions.

# Copyright

---

Under its original name, "A First Course in Programming Using Modula-2" was Copyrighted 1984, 1986, by Richard J. Sutcliffe and [Arjay Enterprises](#). Copyright was assigned to Charles Merrill publishing for the second edition and the title was changed to "Introduction to Programming Using Modula-2" in 1987. *Under the present title, "Modula-2: Abstractions for Data and Programming Structures" the entire contents are copyrighted 1989-2000 by Richard J. Sutcliffe and [Arjay Enterprises](#). No substantial portion of this text may be reproduced in any form without the express written consent of the author. No portion of this text may be stored or reproduced in any fashion unless accompanied by the copyright and shareware information. These materials may be shared locally provided the content is unaltered, except that a local homepage logo may be added to the main index frame.*

## Second Edition

Certain parts of the Preface and some of the examples in Chapter 3 were part of an article by the author for the March 1985 issue of *Call A.P.P.L.E.* Selections from the interchapter material appeared in the author's column "Anodidacticus" in the October and November 1984 issues of *CompuTech* magazine (formerly *Compuwest*) and are reprinted with permission. Early versions of portions of chapters 5 and 9 appeared in an abbreviated form in the author's column "The Dialectical Apple" in *Apple Orchard* magazine August/September 1984. "Nellie Hacker" is a fictional character who has appeared in the author's articles in *Compuwest*, and in his column "The Northern Spy" and other articles in *Call A.P.P.L.E.* and in his newspaper columns and fiction available from [Arjay Enterprises](#). "Nellie and the Pirates" appeared in *Call A.P.P.L.E.* September 1983.

## Third Edition

The contents of the definition modules and other materials from the ISO-standard Modula-2 are copyright 1992 - 1995 by BSI, D.J. Andrews, B.J. Cornelius, R. B. Henry, R. Sutcliffe, D.P. Ward, and M. Woodman and are used with permission. With the publication of the standard in 1996, this copyright passed to ISO.

## 1997 Edition

All the syntax and other materials from the proposed ISO-standard for Generic Modula-2 are copyright 1994 - 1997 by R. Sutcliffe. With the publication of the proposed standard in 1997, this copyright passed to ISO. Some of the examples in the chapter on Generic Modula-2 are taken from papers the author has published, in some cases jointly with Kees Pronk.

## 1999-2000 Edition

All the syntax and other materials from the proposed ISO-standard for Object Oriented Modula-2 are copyright by ISO. Some of the examples in the chapter on Object Oriented Modula-2 are taken from papers the author has published jointly with Kees Pronk, Albert Weidemann, and Martin Schoenhacker. Contents of modules for complex number IO and Graphics are copyright 1995-2000 by R. Sutcliffe, but may be freely used with acknowledgment as long as any modifications or ports to other platforms are submitted to the author.

2002-2003 Edition

For this edition a number of errors in the text and answers were corrected, and some small clarifications introduced, but no structural changes were made.

---

## [Contents](#)

## Mirrors

---

[Main Site](#) - Arjay Books

[Trinity Western University](#) - British Columbia, Canada

[Trinity Western University Computing Science](#) - British Columbia, Canada

The latter for internal use by TWU students only (too slow for outside)

[University of Brighton](#) - UK

Other sites are under negotiation - contact [Rick Sutcliffe](#) if you would like to mirror this site

---

[Contents](#)

# Index of Definitions

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

## A

[absolute value](#)  
[Section 4.6](#)  
[Section 7.11](#)  
[abstract data type](#)  
[Section 1.6](#)  
[Section 5.1](#)  
[activation record](#)  
[actual parameter list](#)  
[address](#)  
[ADDRESS](#)  
[advanced sorts](#)  
[algorithm](#)  
[ancestors](#)  
[argument](#)  
[arithmetic sequence](#)  
[array](#)  
[assignment operator](#)  
[atomic](#)  
[Section 1.6](#)  
[Section 9.1](#)

## B

[batch style](#)  
[binary coded decimal \(BCD\)](#)  
[binary search](#)  
[binary stream](#)  
[binary tree](#)

## F

[FALSE](#)  
[Fibonacci Sequence](#)  
[file](#)  
[finite state machine](#)  
[firmware](#)  
[flag](#)  
[formal parameter list](#)  
[format string](#)  
[fragmented](#)  
[function procedure](#)  
[full, tree](#)

## G

[garbage](#)  
[generic](#)  
[generic separate module](#)  
[geometric](#)  
[global](#)  
[global module](#)  
[Greatest Common Divisor](#)

## H

[handle](#)  
[hardware](#)  
[hash function](#)

## M

[markers](#)  
[mask](#)  
[matrix, m by n](#)  
[mean](#)  
[merging](#)  
[modes](#)  
[module](#)  
[Section 2.2](#)  
[Section 6.1](#)  
[module decoupling](#)  
[modulus](#)  
[mutual recursion](#)

## N

[NEW](#)  
[nil-type](#)  
[normal program](#)  
[termination](#)

## O

[object-oriented](#)  
[Object oriented design](#)  
[opaque type](#)  
[open](#)  
[open array](#)  
[operands](#)

## S

[safe conversions](#)  
[scope](#)  
[scope of visibility](#)  
[sector](#)  
[selector](#)  
[semantics](#)  
[sentinel value](#)  
[sentinel variable](#)  
[sequence](#)  
[sequential](#)  
[sequential](#)  
[set](#)  
[Shell sort](#)  
[side effect](#)  
[simple return](#)  
[simple sorts](#)  
[single character mode](#)  
[sink](#)  
[software](#)  
[solution](#)  
[source](#)  
[stack](#)  
[Section 13.4](#)  
[Section 14.4](#)  
[stack pointer](#)  
[standard console](#)  
[standard identifier](#)  
[Standard Library Item](#)  
[statement](#)  
[static](#)



|                                              |                                                  |                                              |                                                 |
|----------------------------------------------|--------------------------------------------------|----------------------------------------------|-------------------------------------------------|
| <a href="#"><u>bit</u></a>                   | <a href="#"><u>hash table</u></a>                | <a href="#"><u>operating system</u></a>      | <a href="#"><u>static memory allocation</u></a> |
| <a href="#"><u>BITSET</u></a>                | <a href="#"><u>hashing</u></a>                   | <a href="#"><u>operators</u></a>             | <a href="#"><u>stream</u></a>                   |
| <a href="#"><u>block</u></a>                 | <a href="#"><u>head</u></a>                      | <a href="#"><u>ORD</u></a>                   | <a href="#"><u>stream</u></a>                   |
| <a href="#"><u>Section 8.2</u></a>           | <a href="#"><u>head pointer</u></a>              | <a href="#"><u>outermost level</u></a>       | <a href="#"><u>string</u></a>                   |
| <a href="#"><u>Section 10.2</u></a>          | <a href="#"><u>heap</u></a>                      | <a href="#"><u>overflow</u></a>              | <a href="#"><u>structured</u></a>               |
| <a href="#"><u>BOOLEAN</u></a>               | <a href="#"><u>Section 12.4</u></a>              | <a href="#"><u>overloaded</u></a>            | <a href="#"><u>section 1.6</u></a>              |
| <a href="#"><u>bottom-of-loop tested</u></a> | <a href="#"><u>Section 15.3</u></a>              | <a href="#"><u>Section 1.7</u></a>           | <a href="#"><u>Section 9.1</u></a>              |
| <a href="#"><u>B-tree</u></a>                | <a href="#"><u>HIGH</u></a>                      | <a href="#"><u>Section 9.3</u></a>           | <a href="#"><u>S-type</u></a>                   |
| <a href="#"><u>buffer</u></a>                | <a href="#"><u>high level</u></a>                |                                              | <a href="#"><u>subrange</u></a>                 |
| <a href="#"><u>byte</u></a>                  | <a href="#"><u>high nibble</u></a>               | <b>P</b>                                     | <a href="#"><u>sum, vector</u></a>              |
| <b>C</b>                                     | <b>I</b>                                         | <a href="#"><u>page</u></a>                  | <a href="#"><u>syntax</u></a>                   |
| <a href="#"><u>carriage return</u></a>       | <a href="#"><u>identifier</u></a>                | <a href="#"><u>parallelism</u></a>           | <a href="#"><u>system module</u></a>            |
| <a href="#"><u>channel</u></a>               | <a href="#"><u>imaginary number</u></a>          | <a href="#"><u>parallel processing</u></a>   | <b>T</b>                                        |
| <a href="#"><u>Section 6.3</u></a>           | <a href="#"><u>implementation</u></a>            | <a href="#"><u>parameter list</u></a>        | <a href="#"><u>table</u></a>                    |
| <a href="#"><u>Section 8.5</u></a>           | <a href="#"><u>implementation defined</u></a>    | <a href="#"><u>parent</u></a>                | <a href="#"><u>tag</u></a>                      |
| <a href="#"><u>child</u></a>                 | <a href="#"><u>implementation</u></a>            | <a href="#"><u>physical file</u></a>         | <a href="#"><u>tail</u></a>                     |
| <a href="#"><u>circular, list</u></a>        | <a href="#"><u>dependent</u></a>                 | <a href="#"><u>picture</u></a>               | <a href="#"><u>terminated</u></a>               |
| <a href="#"><u>closed</u></a>                | <a href="#"><u>implementation</u></a>            | <a href="#"><u>pointer</u></a>               | <a href="#"><u>text file</u></a>                |
| <a href="#"><u>closure</u></a>               | <a href="#"><u>dependent</u></a>                 | <a href="#"><u>points to</u></a>             | <a href="#"><u>text stream</u></a>              |
| <a href="#"><u>coefficient</u></a>           | <a href="#"><u>implementation</u></a>            | <a href="#"><u>position marker</u></a>       | <a href="#"><u>top down</u></a>                 |
| <a href="#"><u>column</u></a>                | <a href="#"><u>restriction</u></a>               | <a href="#"><u>post- order traversal</u></a> | <a href="#"><u>top-of-loop tested</u></a>       |
| <a href="#"><u>comb sort</u></a>             | <a href="#"><u>implied abstract type</u></a>     | <a href="#"><u>pragma</u></a>                | <a href="#"><u>traced</u></a>                   |
| <a href="#"><u>comment</u></a>               | <a href="#"><u>IN</u></a>                        | <a href="#"><u>pragma delimiters</u></a>     | <a href="#"><u>transparent</u></a>              |
| <a href="#"><u>common ratio</u></a>          | <a href="#"><u>in use</u></a>                    | <a href="#"><u>predicates</u></a>            | <a href="#"><u>transparent types</u></a>        |
| <a href="#"><u>compilation unit</u></a>      | <a href="#"><u>INCL</u></a>                      | <a href="#"><u>pre-order traversal</u></a>   | <a href="#"><u>tree</u></a>                     |
| <a href="#"><u>compiled</u></a>              | <a href="#"><u>indexed</u></a>                   | <a href="#"><u>problem</u></a>               | <a href="#"><u>TRUE</u></a>                     |
| <a href="#"><u>compiler</u></a>              | <a href="#"><u>indexed sequential access</u></a> | <a href="#"><u>procedure</u></a>             | <a href="#"><u>two-dimensional matrix</u></a>   |
| <a href="#"><u>complex conjugate</u></a>     | <a href="#"><u>method</u></a>                    | <a href="#"><u>program</u></a>               | <a href="#"><u>type</u></a>                     |
| <a href="#"><u>complex number</u></a>        | <a href="#"><u>indices</u></a>                   | <a href="#"><u>program file</u></a>          | <a href="#"><u>type, expression</u></a>         |
| <a href="#"><u>compound</u></a>              | <a href="#"><u>indirect recursion</u></a>        | <a href="#"><u>program finalization</u></a>  |                                                 |
| <a href="#"><u>computer program</u></a>      | <a href="#"><u>indirectly referenced</u></a>     | <a href="#"><u>program library</u></a>       | <b>U</b>                                        |
| <a href="#"><u>concatenation</u></a>         | <a href="#"><u>initialization</u></a>            | <a href="#"><u>programmer</u></a>            | <a href="#"><u>underflow</u></a>                |
| <a href="#"><u>connectives</u></a>           | <a href="#"><u>initializing</u></a>              | <a href="#"><u>programming</u></a>           | <a href="#"><u>undiscriminated union</u></a>    |
| <a href="#"><u>constant</u></a>              | <a href="#"><u>in-order traversal</u></a>        | <a href="#"><u>notation</u></a>              | <a href="#"><u>unqualified</u></a>              |
| <a href="#"><u>constant, Modula-2</u></a>    | <a href="#"><u>interactive</u></a>               | <a href="#"><u>pseudo-module</u></a>         | <a href="#"><u>unsafe conversions</u></a>       |
| <a href="#"><u>constructor</u></a>           | <a href="#"><u>interpreted</u></a>               | <a href="#"><u>pseudo-random</u></a>         | <a href="#"><u>unstructured</u></a>             |
| <a href="#"><u>Section 9.2</u></a>           | <a href="#"><u>invocation</u></a>                | <b>Q</b>                                     | <a href="#"><u>Section 1.6</u></a>              |
| <a href="#"><u>Section 11.6</u></a>          | <a href="#"><u>iteration</u></a>                 |                                              |                                                 |

[counting loop](#)  
[coupled](#)  
[covering](#)

[Section 3.7](#)  
[Section 5.4](#)  
[iterative](#)

[qualified identifier](#)  
[queue](#)  
[quicksort](#)

[Section 9.1](#)  
[untraced](#)

## D

[data location](#)  
[decimal](#)  
[declaration part](#)  
[declaring](#)  
[decoupled](#)  
[default](#)  
[degree \(of tree\)](#)  
[degree \(of polynomial\)](#)  
[depth](#)  
[dereferencing](#)  
[dereferencing](#)  
[operator](#)  
[descendents](#)  
[determinant](#)  
[device drivers](#)  
[device independent](#)  
[discriminator](#)  
[DISPOSE](#)  
[divisor](#)  
[doubly dereferenced](#)  
[dynamic](#)  
[dynamic memory](#)  
[allocation](#)

## K

[Knight's Tour](#)  
[k-sorted](#)  
[k-sorts](#)

## L

[language](#)  
[Section 1.5](#)  
[Section 1.8](#)  
[leaf](#)  
[legible stream](#)  
[length](#)  
[level](#)  
[level one](#)  
[level n](#)  
[level zero](#)  
[LOC](#)  
[local](#)  
[local module](#)  
[logical file](#)  
[lookup table](#)  
[linear](#)  
[linear equation](#)  
[line mode](#)  
[linked list](#)  
[literal](#)  
[initial definition](#)  
[revised definition](#)  
[literal string](#)  
[low level](#)  
[low nibble](#)

## R

[raising an exception](#)  
[random-access](#)  
[random number](#)  
[raw stream](#)  
[read](#)  
[read/write](#)  
[REAL](#)  
[record](#)  
[recursive](#)  
[reference semantics](#)  
[reference variable](#)  
[refining separate](#)  
[module](#)  
[relatively prime](#)  
[reserved word](#)  
[row](#)

## V

[VAL](#)  
[value parameter](#)  
[value semantics](#)  
[variable](#)  
[variable, Modula-2](#)  
[variable parameter](#)  
[variance](#)  
[vector](#)  
[virtual machine](#)  
[visible](#)

## W

[word](#)  
[write](#)

## Z

[zero-based](#)  
[Z-type](#)

## E

[echoing](#)  
[End-Of-File](#)  
[enumerated type](#)  
[evaluating the](#)  
[expression](#)  
[exception](#)  
[exception, Modula-2](#)  
[exception handler,](#)

[Modula-2](#)  
[exceptional](#)  
[termination](#)  
[EXCL](#)  
[expression](#)  
[expression, Modula-2](#)  
[expression compatible](#)  
[expression](#)  
[incompatible](#)

---

## [Contents](#)

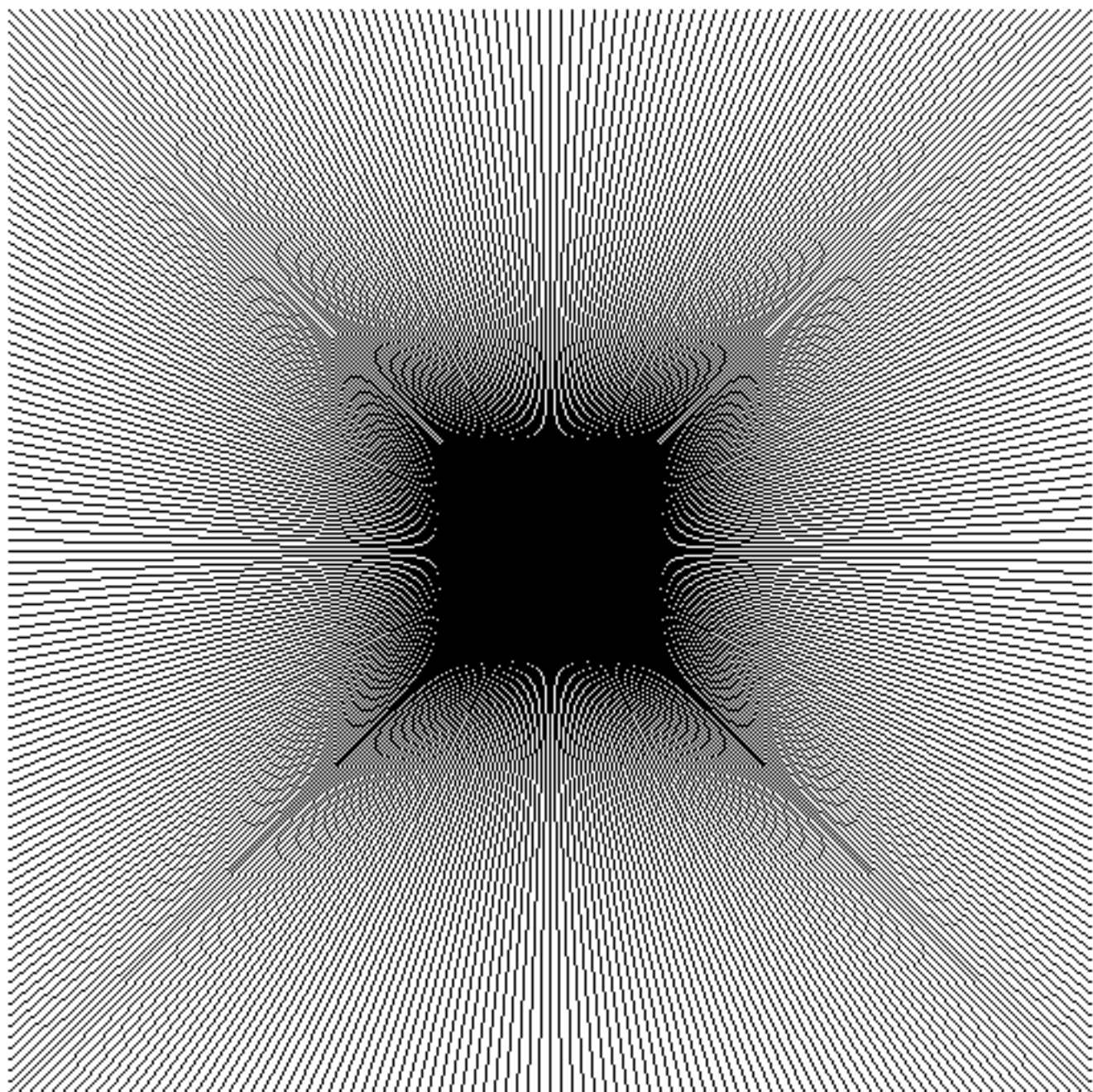


Figure 18.12

# Index of Definitions

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

## A

[absolute value](#)  
    [Section 4.6](#)  
    [Section 7.11](#)  
[abstract data type](#)  
    [Section 1.6](#)  
    [Section 5.1](#)  
[activation record](#)  
[actual parameter list](#)  
[address](#)  
[ADDRESS](#)  
[advanced sorts](#)  
[algorithm](#)  
[ancestors](#)  
[argument](#)  
[arithmetic sequence](#)  
[array](#)  
[assignment operator](#)  
[atomic](#)  
    [Section 1.6](#)  
    [Section 9.1](#)

## B

[batch style](#)  
[binary coded decimal \(BCD\)](#)  
[binary search](#)

## F

[FALSE](#)  
[Fibonacci Sequence](#)  
[file](#)  
[finite state machine](#)  
[firmware](#)  
[flag](#)  
[formal parameter list](#)  
[format string](#)  
[fragmented](#)  
[function procedure](#)  
[full, tree](#)

## G

[garbage](#)  
[generic](#)  
[generic separate module](#)  
[geometric](#)  
[global](#)  
[global module](#)  
[Greatest Common Divisor](#)

## H

[handle](#)

## M

[markers](#)  
[mask](#)  
[matrix, m by n](#)  
[mean](#)  
[merging](#)  
[modes](#)  
[module](#)  
    [Section 2.2](#)  
    [Section 6.1](#)  
[module decoupling](#)  
[modulus](#)  
[mutual recursion](#)

## N

[NEW](#)  
[nil-type](#)  
[normal program termination](#)

## O

[object-oriented](#)  
[Object oriented design](#)  
[opaque type](#)  
[open](#)

## S

[safe conversions](#)  
[scope](#)  
[scope of visibility](#)  
[sector](#)  
[selector](#)  
[semantics](#)  
[sentinel value](#)  
[sentinel variable](#)  
[sequence](#)  
[sequential](#)  
[sequential](#)  
[set](#)  
[Shell sort](#)  
[side effect](#)  
[simple return](#)  
[simple sorts](#)  
[single character mode](#)  
[sink](#)  
[software](#)  
[solution](#)  
[source](#)  
[stack](#)  
    [Section 13.4](#)  
    [Section 14.4](#)  
[stack pointer](#)  
[standard console](#)  
[standard identifier](#)

[binary stream](#)

[binary tree](#)

[bit](#)

[BITSET](#)

[block](#)

[Section 8.2](#)

[Section 10.2](#)

[BOOLEAN](#)

[bottom-of-loop tested](#)

[B-tree](#)

[buffer](#)

[byte](#)

## C

[carriage return](#)

[channel](#)

[Section 6.3](#)

[Section 8.5](#)

[child](#)

[circular, list](#)

[closed](#)

[closure](#)

[coefficient](#)

[column](#)

[comb sort](#)

[comment](#)

[common ratio](#)

[compilation unit](#)

[compiled](#)

[compiler](#)

[complex conjugate](#)

[complex number](#)

[compound](#)

[computer program](#)

[concatenation](#)

[connectives](#)

[constant](#)

[hardware](#)

[hash function](#)

[hash table](#)

[hashing](#)

[head](#)

[head pointer](#)

[heap](#)

[Section 12.4](#)

[Section 15.3](#)

[HIGH](#)

[high level](#)

[high nibble](#)

## I

[identifier](#)

[imaginary number](#)

[implementation](#)

[implementation defined](#)

[implementation](#)

[dependent](#)

[implementation](#)

[dependent](#)

[implementation](#)

[restriction](#)

[implied abstract type](#)

[IN](#)

[in use](#)

[INCL](#)

[indexed](#)

[indexed sequential access](#)

[method](#)

[indices](#)

[indirect recursion](#)

[indirectly referenced](#)

[initialization](#)

[initializing](#)

[in-order traversal](#)

[open array](#)

[operands](#)

[operating system](#)

[operators](#)

[ORD](#)

[outermost level](#)

[overflow](#)

[overloaded](#)

[Section 1.7](#)

[Section 9.3](#)

## P

[page](#)

[parallelism](#)

[parallel processing](#)

[parameter list](#)

[parent](#)

[physical file](#)

[picture](#)

[pointer](#)

[points to](#)

[position marker](#)

[post- order traversal](#)

[pragma](#)

[pragma delimiters](#)

[predicates](#)

[pre-order traversal](#)

[problem](#)

[procedure](#)

[program](#)

[program file](#)

[program finalization](#)

[program library](#)

[programmer](#)

[programming](#)

[notation](#)

[pseudo-module](#)

[Standard Library Item](#)

[statement](#)

[static](#)

[static memory allocation](#)

[stream](#)

[stream](#)

[string](#)

[structured](#)

[section 1.6](#)

[Section 9.1](#)

[S-type](#)

[subrange](#)

[sum, vector](#)

[syntax](#)

[system module](#)

## T

[table](#)

[tag](#)

[tail](#)

[terminated](#)

[text file](#)

[text stream](#)

[top down](#)

[top-of-loop tested](#)

[traced](#)

[transparent](#)

[transparent types](#)

[tree](#)

[TRUE](#)

[two-dimensional matrix](#)

[type](#)

[type, expression](#)

## U

[constant, Modula-2](#)

[constructor](#)

[Section 9.2](#)

[Section 11.6](#)

[counting loop](#)

[coupled](#)

[covering](#)

## D

[data location](#)

[decimal](#)

[declaration part](#)

[declaring](#)

[decoupled](#)

[default](#)

[degree \(of tree\)](#)

[degree \(of polynomial\)](#)

[depth](#)

[dereferencing](#)

[dereferencing](#)

[operator](#)

[descendents](#)

[determinant](#)

[device drivers](#)

[device independent](#)

[discriminator](#)

[DISPOSE](#)

[divisor](#)

[doubly dereferenced](#)

[dynamic](#)

[dynamic memory](#)

[allocation](#)

## E

[echoing](#)

[interactive](#)

[interpreted](#)

[invocation](#)

[iteration](#)

[Section 3.7](#)

[Section 5.4](#)

[iterative](#)

## K

[Knight's Tour](#)

[k-sorted](#)

[k-sorts](#)

## L

[language](#)

[Section 1.5](#)

[Section 1.8](#)

[leaf](#)

[legible stream](#)

[length](#)

[level](#)

[level one](#)

[level n](#)

[level zero](#)

[LOC](#)

[local](#)

[local module](#)

[logical file](#)

[lookup table](#)

[linear](#)

[linear equation](#)

[line mode](#)

[linked list](#)

[literal](#)

[initial definition](#)

[revised definition](#)

[pseudo-random](#)

## Q

[qualified identifier](#)

[queue](#)

[quicksort](#)

## R

[raising an exception](#)

[random-access](#)

[random number](#)

[raw stream](#)

[read](#)

[read/write](#)

[REAL](#)

[record](#)

[recursive](#)

[reference semantics](#)

[reference variable](#)

[refining separate](#)

[module](#)

[relatively prime](#)

[reserved word](#)

[row](#)

[underflow](#)

[undiscriminated union](#)

[unqualified](#)

[unsafe conversions](#)

[unstructured](#)

[Section 1.6](#)

[Section 9.1](#)

[untraced](#)

## V

[VAL](#)

[value parameter](#)

[value semantics](#)

[variable](#)

[variable, Modula-2](#)

[variable parameter](#)

[variance](#)

[vector](#)

[virtual machine](#)

[visible](#)

## W

[word](#)

[write](#)

## Z

[zero-based](#)

[Z-type](#)

[End-Of-File](#)  
[enumerated type](#)  
[evaluating the](#)  
[expression](#)  
[exception](#)  
[exception, Modula-2](#)  
[exception handler,](#)  
[Modula-2](#)  
[exceptional](#)  
[termination](#)  
[EXCL](#)  
[expression](#)  
[expression, Modula-2](#)  
[expression compatible](#)  
[expression](#)  
[incompatible](#)

[literal string](#)  
[low level](#)  
[low nibble](#)

---

## [Contents](#)