

The graphic on the left side of the page is a vertical rectangle. The left portion of this rectangle features a series of diagonal stripes, alternating between a light gray and a dark gray/black color. The right portion of the rectangle is a solid black field. Overlaid on the black field is the text 'Metrowerks CodeWarrior™ CD' in a white, sans-serif font. The text is oriented vertically, reading from bottom to top.

Metrowerks CodeWarrior™ CD

CodeWarrior Programming Practice: Pascal

Because of last-minute changes to CodeWarrior, some information in this manual may be out of date. Please read all the Release Notes files that come with CodeWarrior for the latest information.

Metrowerks CodeWarrior Copyright ©1993-1995 by Metrowerks Inc. and its Licensors. All rights reserved.

Documentation stored on the compact disc may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo and Software at Work are registered trademarks of Metrowerks Inc. CodeWarrior, PowerPlant, and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks or registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISC ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

Canada and International

Metrowerks Inc.
1500 du College, suite 300
St. Laurent, QC
H4L 5G6 Canada

voice: (514) 747-5999
fax: (514) 747-2822

U.S.A.

Metrowerks Corporation
Suite 310
The MCC Building
3925 West Braker Lane
Austin, TX 78759-5321

voice: 512-305-0400
fax: 512-305-0440

Metrowerks Mail Order

voice: (800) 377-5416 or (419) 281-1802
fax: (419) 281-6883

World Wide Web site (Internet): <http://www.metrowerks.com>

Registration information (Internet): register@metrowerks.com

Technical support (Internet): support@metrowerks.com

Sales, marketing, & licensing (Internet): sales@metrowerks.com

AppleLink: METROWERKS

America OnLine: goto: METROWERKS

CompuServe: goto: METROWERKS

eWorld: goto: METROWERKS

Table of Contents

Chapter 1 An Overview

1.1	Preview.....	14
1.2	Introduction to Programming Practice: Pascal.....	15
	A Global View.....	15
	The Road Ahead.....	15
	Signs Along the Road.....	16
1.3	Hardware: Computers and Peripherals.....	17
	The CPU.....	18
	Other Components and Packaging.....	19
1.4	The World of Programming.....	20
	What is Programming?.....	20
	“Real world” and “Abstract World”.....	20
1.5	Pascal.....	21
	An example Pascal program.....	22
	Running Programs: Compiling, Linking, Executing.....	23
1.6	Communicating to Computers.....	24
	Typing.....	24
1.7	Chapter 1 Review.....	27

Chapter 2 Computing: A Short Survey of Some Applications

2.1	Preview.....	30
2.2	Software and Applications.....	30
2.3	Application Software.....	33
	Editor Application.....	34
	Typing Applications.....	35
	Calculator Applications.....	36
	Retrieve Application: A Tiny Database.....	38
	Planner-Calendar Application.....	39
	Bar Plot Application.....	41
	Drill Application.....	42
	SSS: Small and Simple Spreadsheet.....	43
2.4	Chapter 2 Review.....	47

Chapter 3 Programming Language: Pascal

3.1	Preview.....	50
-----	--------------	----

3.2	Languages.....	50
	Syntax and Semantics.....	50
	Syntax Diagrams.....	53
3.3	Pascal Programs.....	56
	Program Format.....	56
	Program Presentation.....	59
	More Pascal Programming: Data and Actions.....	60
	Data Items.....	61
	Actions: Arithmetic Operations.....	63
3.4	More Example Programs.....	66
3.5	Chapter 3 Review.....	69
3.6	Chapter 3 Problems.....	70
3.7	Chapter 3 Programming Project.....	74
	Getting Acquainted.....	74

Chapter 4 Data and Actions

4.1	Preview.....	76
4.2	Programming: Data and Actions.....	76
	Declarations: Syntax Diagrams from the Bottom.....	76
	Simple Input and Output in Pascal.....	80
4.3	More Programs: A Top View of Pascal.....	83
4.4	Programming Style.....	85
4.5	Layout of Programs.....	86
4.6	More Programs: Continued.....	89
	Actions: Pre-Defined Standard Functions in Pascal....	89
	Libraries: Using Units in Pascal.....	92
4.7	A Foretaste of Procedures.....	94
4.8	Chapter 4 Review.....	96
4.9	Chapter 4 Problems.....	96
4.10	Chapter 4 Programming Projects.....	97
	Generate Conversion Tables.....	97
	Observing Errors.....	97
	Demilitarize Time.....	98
	TipTable.....	99
	STT: Sales Tax Table.....	100
	SSP: Simple Side Plot.....	101

Chapter 5 The Four Fundamental Forms in Pascal

5.1	Preview.....	105
-----	--------------	-----

5.2	The Sequence Form in Pascal.....	105
5.3	Conditions in Pascal.....	107
5.4	Repetition Form: The WHILE statement.....	108
	Tracing Loops.....	110
5.5	WHILES, REALS, and Errors.....	114
5.6	Selection Forms in Pascal.....	117
5.7	More Selections: Combinations of Selection Forms....	119
	Confusion in Choices.....	120
	More Nesting of Choices.....	120
	Alternative Ways to Code Selections.....	121
5.8	Select Form: Handling Many Branches.....	124
5.9	Awkward Nests: General Nesting.....	127
	Mixed Nests: Repetition and Selections.....	128
5.10	Subprograms: Using Subprograms as Black Boxes....	130
	The ShortSort Library.....	134
	Notation for Defined Procedures.....	137
	Procedures vs. Functions.....	140
5.11	Binary Logic Library: BitLib.....	145
5.12	Chapter 5 Review.....	149
5.13	Chapter 5 Problems.....	150
5.14	Chapter 5 Programming Problems.....	152
	Sequence Problems.....	153
	Selection Problems.....	153
	Loop Problems.....	153
	Subprogram Problems.....	153
	Debugging Problems.....	154
	Selection Programs.....	156
	Procedures and Repetitions.....	157
	Josephus' Problem.....	158
5.15	Chapter 5 Programming Projects.....	159
	Project A: Change Change.....	159
	Project B: Payroll.....	160
	Project C: Quadratic Roots.....	160
	Project D: Digital Circuits.....	160
	Project E: Roll Your Own.....	161
	CRN: Convert Roman Numbers.....	161
	GPR: Growing Pay Roll.....	161
	MWM: Many Ways to Mid.....	162
	DFP: Data Flow Programming.....	163

Chapter 6 Pascal with Bigger Blocks

6.1	Preview.....	169
-----	--------------	-----

6.2	Conglomerations.....	169
	Mixed and Nested Forms.....	169
6.3	More Data.....	171
	External.....	171
	CASE Form.....	173
6.4	More Repetition Forms.....	175
6.5	The For Loop.....	177
6.6	Character Type.....	182
	in Pascal.....	182
	Representation of Characters.....	185
6.7	Boolean Type in Pascal.....	190
6.8	More Types.....	193
	Big Types in Pascal.....	193
	Even Bigger Types.....	193
	Smaller Types.....	194
6.9	Programmer Defined Types.....	195
	Enumerated Types.....	195
	Subrange Types In Pascal.....	198
6.10	Strings in Pascal.....	198
6.11	Simple Files in Pascal.....	201
	More Files: Input and Output.....	205
6.12	Modification of Programs.....	207
6.13	Programming Style.....	209
	Documentation.....	209
	Further Guidelines for Identifiers.....	212
	A Bad Style Horror Story.....	213
	Criticism of the MeanMean Program.....	215
6.14	Errors in Programming.....	216
	Syntactic Errors.....	216
	Execution Errors.....	217
	Logical Errors.....	217
	Other Errors.....	217
6.15	Debugging, Testing, Proving.....	217
6.16	Chapter 6 Review.....	219
6.17	Chapter 6 Problems.....	219
6.18	Chapter 6 Programming Problems.....	221
6.19	Chapter 6 Programming Projects.....	224
	Plotting Programs.....	224
	Modify Calculator.....	226
	Statistics (Rainfall).....	227
	Project Plotup.....	227
	BSD: Big Statistics Data.....	228

BTD: Big Text Data.....	229
SMC: Small Monthly Calendar.....	230

Chapter 7 Better Blocks: Procedures and Libraries

7.1	Preview.....	233
7.2	Procedures in Pascal.....	233
	Use and Definition.....	233
7.3	Syntax of Subprogram Forms.....	235
	Procedures in Pascal.....	235
	More Examples of Pascal Procedures.....	239
	Power Procedure.....	242
7.4	Passing Parameters.....	242
	In, Out, In and Out, and Neither.....	242
	BigChange.....	242
	BigPay.....	245
	A Miscellany of procedures.....	248
	A Second Miscellany of procedures.....	250
7.5	Procedures with Char, Boolean and Other Types.....	252
	Generalized Item Types.....	255
7.6	Procedures with User-Defined types.....	256
7.7	More on Passing Parameters.....	259
7.8	Nested Procedures.....	261
7.9	Functions in Pascal.....	264
	Many Functions.....	266
7.10	SubPrograms: Variations on a theme.....	269
7.11	Recursion in Pascal.....	272
7.12	Libraries in Pascal.....	275
	Units.....	275
	UtilityLib: a custom-made utilities Library.....	278
	Other Libraries: DateLib, BitLib, CharLib.....	281
	The Interaction of Many Libraries.....	288
7.13	Function and Procedure Types.....	291
7.14	Top-Down Development.....	296
	Pay Again.....	296
7.15	Chapter 7 Review.....	299
7.16	Chapter 7 Problems.....	300
7.17	Chapter 7 Programming Problems.....	302
	Random Projects.....	302
	DateLib.....	305
	Create Libraries.....	306

	FinanceLib.....	307
	Change Again: Done Properly with Procedures.....	308
	MeanLib.....	308
7.18	Chapter 7 Programming Projects.....	310
	DMT: DeMilitarizeTime Lab with Procedures.....	310
	SLL: Small Library Project.....	311

Chapter 8 Pascal Data Structures

8.1	Preview.....	314
8.2	Arrays in Pascal.....	314
	ChangeMaker and Variance.....	317
	Parallel Arrays: Part Inventory.....	319
	A Tiny Data Base: Retrieving Strings from Arrays....	321
	Arrays as Parameters.....	323
	IntArrayLib: Integer Array Library.....	325
8.3	Two Dimensional Arrays in Pascal.....	328
	Arrays of Arrays—Two Dimensional Arrays.....	328
8.4	N-Dimensional Arrays.....	331
	More Dimensions.....	331
8.5	Records in Pascal.....	333
	ComplexLib: A Library for Complex Numbers.....	337
	Records of Records: Nested Records.....	339
	Arrays of Records in Pascal.....	342
	Records of Arrays.....	345
	Matrix Library.....	347
8.6	Sets in Pascal.....	352
	More Sets (Optional).....	355
8.7	Dynamic Variables and Pointers in Pascal.....	359
8.8	Chapter 8 Review.....	361
8.9	Chapter 8 Problems.....	361
8.10	Chapter 8 Programming Projects.....	363
	Gas Project.....	363
	Library Projects.....	364
	BSL: Big Stat Lab.....	364

Chapter 9 Algorithms to Run With

9.1	Preview.....	368
9.2	Sorting Algorithms.....	369
	A Context for Sorting.....	369
	Count Sort.....	373
	Bubble Sort.....	375

	Select Sort.....	376
9.3	Improving sorts (Optional).....	377
	Recursive Sort: Merge Sort.....	378
	Another Merge Sort: the Von Neumann Sort.....	380
9.4	Searching.....	387
	Binary Search.....	387
9.5	Implementing Stacks and Queues.....	391
	StackLib: Stack as an Abstract Data Type.....	391
	Dynamic Stacks.....	395
	QueueLib.....	397
	Big Cardinals.....	400
	SetLib: Stack of Strings.....	404
9.6	Trees.....	407
9.7	Chapter 9 Review.....	412
9.8	Chapter 9 Problems.....	412
9.9	Chapter 9 Programming Projects.....	415
	Queue Abstract Data Type.....	415
	PES: Performance Evaluation of Sorts.....	416
	VS: Visual Sorts.....	418

Chapter 10 The Seven Step Method

10.1	Method: Part II.....	425
	The Seven Step Method.....	425
10.2	Design Stage: Acme Payroll System.....	426
	1. Problem Definition.....	426
	Problem Definition Application.....	426
	2. Solution Design.....	427
	Solution Design Application.....	427
	3. Solution Refinement.....	428
	Solution Refinement Application.....	428
	4. Testing Strategy Development.....	429
	Testing Strategy Development Application.....	429
10.3	Implementation Stage: Acme Payroll System.....	431
	5. Program Coding and Testing.....	431
	Case Study: The Acme Payroll System.....	432
	6. Documentation Completion.....	444
	Documentation Completion Application.....	444
	7. Program Maintenance.....	445
	Program Maintenance Application.....	445
10.4	An Advanced Case Study: Building a Text Index.....	445
	Design Stage.....	446
	1. Problem Definition.....	446
	2. Solution Design.....	446
	3. Solution Refinement.....	447

Binary Search Tree Unit.....	449
Queues Unit.....	451
Implementation Stage.....	452
4. Testing Strategy Development.....	452
5. Program Coding and Testing.....	453
6. Documentation.....	463
7. Program Maintenance.....	463
10.5 Chapter 10 Review.....	464
10.6 Chapter 10 Programming Problems.....	464
1. Editor Application.....	464
2. Typing.....	470
3. Calculator Applications.....	472
10.7 Chapter 10 Programming Projects.....	473
10.8 Level 1 — Getting Started.....	473
1-1. General.....	474
A Guessing Game.....	474
1-2. Business.....	474
Computing a Customer's Change.....	474
1-3. Scientific.....	475
A Bouncing Ball.....	475
10.9 Level 2 — Getting Organized with Procedures.....	477
2-1. General.....	477
Your Age in Days.....	477
2-2 Business.....	477
What's the Cost of My Mortgage?.....	477
2-3 Scientific.....	478
Solving the Quadratic Equation.....	478
10.10 Level 3 — Getting Fancier with Parameters.....	479
3-1. General.....	479
Count the Word Occurrences in a Text.....	479
Processing Personnel Data.....	480
3-3. Scientific.....	481
Plotting a Function.....	481
10.11 Level 4 — Getting Your Wings with Units.....	483
4-1 General.....	483
The Kwic Index.....	483
4-2 Business.....	484
Information Retrieval.....	484
4-3 Scientific.....	486
Complex Algebra.....	486

Preface

The purpose behind Programming Practice: Pascal is to teach you how to apply the theories, tools, and concepts mentioned in the Principles book to problem-solving using the Pascal programming language. We suggest that you either read the Principles book before beginning this book, or read the two books concurrently.

While using this book to learn Pascal, you may find that there are too many examples and that some topics have been over-explained. You are encouraged to select only the topics and examples that best suit your needs and work habits.

Chapter 1 An Overview

This chapter provides an introduction to the Programming Practice: Pascal, an overview of current computing devices and systems, and a brief introduction of Pascal and programming. Being an overview, this chapter covers a wide range of subjects, and details. Do not be overwhelmed! Many of the topics will be discussed in more detail in other chapters. This chapter can be read quickly.

Chapter Overview

1.1	Preview.....	14
1.2	Introduction to Programming Practice: Pascal.....	15
	A Global View.....	15
	The Road Ahead.....	15
	Signs Along the Road.....	16
1.3	Hardware: Computers and Peripherals.....	17
	The CPU.....	18
	Other Components and Packaging.....	19
1.4	The World of Programming.....	20
	What is Programming?.....	20
	“Real world” and “Abstract World”.....	20
1.5	Pascal.....	21
	An example Pascal program.....	22
	Running Programs: Compiling, Linking, Executing.....	23
1.6	Communicating to Computers.....	24
	Typing.....	24
1.7	Chapter 1 Review.....	27

1.1 Preview

This book intends to teach you how to program a computer using the Pascal programming language, using the concepts and topics discussed in the Principles of Programming.

Although the journey through the Principles of Programming is echoed throughout this book, there are some additional obstacles and terrain that must be dealt with when learning the Pascal programming language. This chapter provides you with a guide to these obstacles and outlines the new terrain by introducing the Programming Practice: Pascal in three subsections:

- *A Global View* describes the layout of the text book's material in very general detail. This sub-section also offers an idea of the way in which the Programming Practice: Pascal should be used with its companion book: the Principles of Programming.
- *The Road Ahead* gives a very brief synopsis of each chapter so that you can visualize how all the chapters fit together. This provides a route that will take you from an introduction to computers, programs, and the compiling process (the last section in this chapter) to being able to write large useful programs in Pascal.
- *Road Signs* describes the common pattern on which each of the chapters are based, tells you how to read the road signs, and gives a brief explanation of some of the visual aids that will be used throughout this book.

Apart from introducing the Programming Practice: Pascal, this chapter also introduces the physical devices from which the computer system is constituted. The *hardware*— as it is commonly referred to— is the part of the computer system that changes most rapidly. As new technology is developed, becoming faster, cheaper and smaller, it replaces the old. For this reason, we will concentrate on the smaller micro-computers and mini-computers, which are more powerful than the “monster” computers of just a few years ago.

Programming is the creative process of designing and realizing software. Programs are written using programming languages of which there are many. In this book, we emphasize the high-level programming language called Pascal, but most of the concepts we'll introduce also apply to other languages.

Communication between people and computers at a detailed level is often done using a keyboard. This chapter includes a brief discussion of the practical skill of touch typing, which is useful for anyone wishing to communicate with computers. A sample program which teaches typing is shown in Chapter 2.

In summary, this chapter provides an introduction to the many practical aspects of computing today, and it does so by concentrating on some typical aspects. It emphasizes micro-computers over larger computers, but most of the material described applies equally to all computers. It aims to introduce you to the practice of programming.

1.2 Introduction to Programming Practice: Pascal

A Global View

The goal of this book is to teach you how to program computers using the Pascal programming language. However, learning to write computer programs is very much like learning any skill, you must first understand the underlying principles upon which the craft is based and then practice, practice, practice.

For example, you can read all the books ever published about riding a bicycle but, until you actually get on one and start pedaling on your own, you will not know how to ride a bicycle. The same applies to computer programming! Here, we provide the second book, the Programming Practice: Pascal, which assumes that you have already familiarized yourself with the Principles of Programming (referred to as Principles): the first book which describes the method behind problem solving using a computer.

As in the Principles book, Programming Practice: Pascal emphasizes the adage, “it’s best to learn through examples.” Each chapter uses example programs, many taken from the Principles book, to teach the same point. For this reason, you may find the number of examples overwhelming and you should feel free to pay attention to only the examples which you feel best describe the point that is being illustrated.

The Road Ahead

In this section, we give a brief summary of what is in each chapter of the Programming Practice: Pascal. A synopsis of each chapter is presented so that you can visualize the road ahead.

- **Chapter 2, Computing: A short survey of some applications**, this chapter introduces software by giving some examples of packaged programs, applications, which perform specific tasks.
- **Chapter 3, Programming Language: Pascal**, in this chapter we further introduce the Pascal programming language by comparing it to natural languages and by using syntax diagrams to help our understanding. Some extremely simple Pascal program examples are given as illustrations of the syntax presented. Although it will take a few more chapters to introduce all of the Pascal syntax, some other examples of complete Pascal programs are also given in this chapter.
- **Chapter 4, Data and Actions**, this chapter continues with the introduction to the Pascal programming language that was started in the last two chapters. Its main concerns are the basic components of the language with some emphasis on the way in which the language is

written (i.e. the syntax or grammar) but also on the precise meaning (i.e. the semantics) of what is written.

- **Chapter 5, The Four Fundamental Forms in Pascal**, the goal of this chapter is to introduce the four fundamental forms: Sequence, Selection, Repetition, and Invocation, that are necessary and sufficient to create any algorithm. At the end of the chapter you will have the necessary tools to develop your own programs.
- **Chapter 6, Pascal with Bigger Blocks**, this chapter continues the presentation of the programming language Pascal by introducing other forms, deeper nests, different data types and more details. These complements are not as fundamental or important as the topics of the previous chapter, but are useful and convenient in the development of clear and correct programs. Remember, bigger is not always better.
- **Chapter 7, Better Blocks: Procedures and Libraries**, this chapter presents the creation and use of Pascal subprograms, i.e. the Pascal procedures and functions. Subprograms are often part of libraries, and the chapter introduces the Pascal units that are used to implement libraries.
- **Chapter 8, Pascal Data Structures**, in this chapter we consider the three structured data types: arrays, records, and sets, and how they are used in Pascal. The main concepts introduced in Chapter 8 of the Principles book and “Abstract Data Type” (or ADT), will be developed further. ADTs will also be used to create three libraries: IntArrayLib, ComplexLib, and a Matrix Library
- **Chapter 9, Algorithms to Run With**, the primary purpose of this chapter is to provide more extensive examples of the data structures introduced in chapter 8. Algorithms to sort and search data structures will be discussed, along with the various ways of implementing stacks, queues, and trees. Also, this chapter will further develop the concept of an “Abstract Data Type” (or ADT) and create the StackLib, QueueLib, and SetLib libraries.
- **Chapter 10, The Seven Step Method**, this chapter implements solutions as computer programs, and then demonstrates how to verify the correctness of solutions through testing. To reach this solution, the seven step problem-solving method– introduced in the Principles book– will be reviewed and discussed. Although we’ll concentrate on the implementation in Pascal of an already designed program, we cannot ignore the design.

Signs Along the Road

Although each chapter constitutes a stage in our learning journey, introducing new topics and conventions while expanding on others, the same basic pattern is followed:

1. Each chapter begins with a *Preview* that gives a summary of the material that will be presented in the chapter. This will give you an idea of what to expect and introduce you to the major concepts in the chapter.
2. Next, the actual material of the chapter is provided, where topics concerning the Pascal language is divided into sections. Since the emphasis through each chapter is that we learn best by example, each section has many sample programs illustrating the Pascal statements being discussed. Most of these examples are taken from the Principles book and references are made when appropriate.
3. Following this, a *Review* of the material contained in the chapter is presented. This will serve to remind you of what you have learned and nudge you to go back and reread anything that you have forgotten.
4. Finally, each chapter ends with a set of *Problems* and *Programming Projects* to solve. These are the most important parts of each chapter. Some chapters contain Programming Problems as well. In any case, programming is an intensely practical skill that can only be acquired by practice. Remember: "Practice makes perfect!"

The Principles of Programming, because it is a book which explains a great many theories and concepts, includes *signs* to visually illustrate important points and concepts. However, because the Principles of Programming: Pascal teaches a programming language, it does not include these signs as frequently. If you are not familiar with these signs they are as follows:

Note: This is a tip or note box. Inside this box, important information and tips can be found.

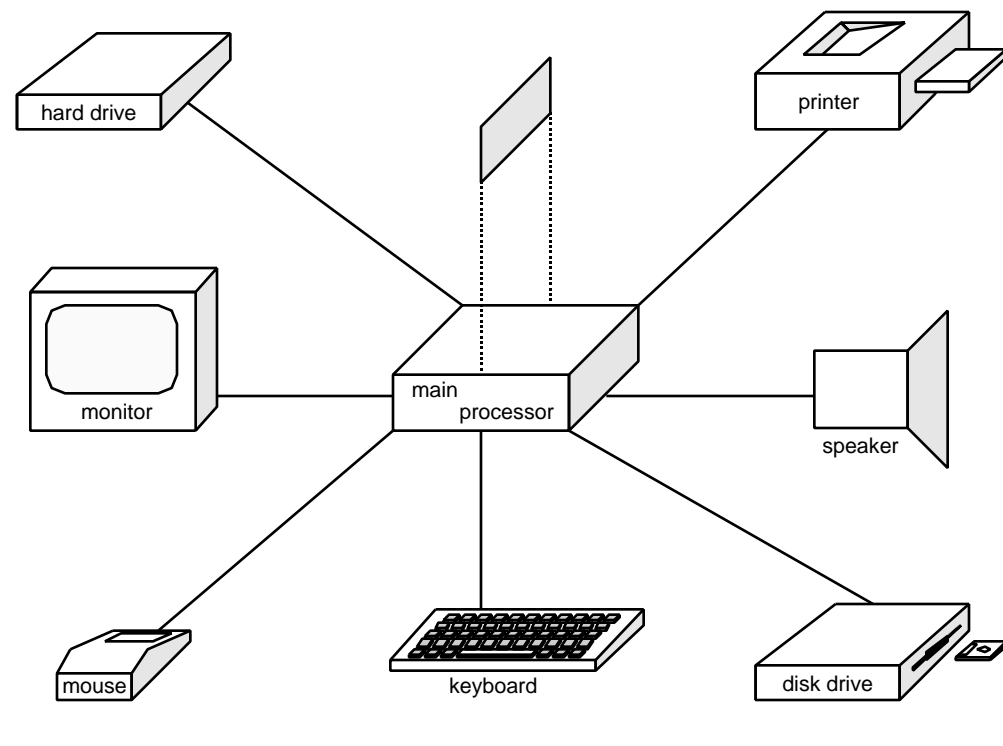
This is a note in margin which directs you to useful information.

In addition to tip boxes, when a paragraph briefly touches upon a subject which is covered in more detail in another chapter, a reference is provided in the left margin.

1.3 Hardware: Computers and Peripherals

Computers, which range in size from monster to micro, are all constructed from the same kind of components. Here, we will illustrate these components by describing micro-computers, realizing that today's "micro" is capable of much more than the "monster" of just a few dozen years ago. Actually micro-computers are, nowadays, much more interesting than the larger computers; they are more convenient to use, more flexible and more commonly available.

All micro-computers are built from only a few basic components, some of which are shown in Figure 1.1. The main component (shown in the center of the figure) is an electronic "black box" unit that contains the Central Processing Unit, some control circuitry, some internal memory, a power supply, and connectors of various sorts to allow other units to plug in.

Figure 1.1 A computer's basic components

The CPU

At the heart of a computer system (inside the black box in Figure 1.1) is the central processing unit, or CPU, which controls the behavior of the system's other components, and performs the operations on the data. Connected to the CPU are "peripherals", which perform a wide variety of functions. These pieces of hardware include printers, memory devices and communication units that can link one computer to another over standard telephone lines. In this chapter, we will briefly describe some of the many computers and peripherals, concentrating only on the most common ones.

The inner building blocks of the CPU are registers consisting of 8, 16 or 32 *bits* (binary digits), the larger ones usually corresponding to faster machines. When the computer is being used, the registers are mainly used to store the data being manipulated. The programs being executed are stored in the internal memory, as well as the data they manipulate. The amount of data and programs that can be stored in memory is measured in units of Kbytes, where a Kbyte, or kilobyte or simply K, is about one thousand bytes (actually 2^{10} or 1024), and where a *byte* is 8 bits and is the equivalent of a single character, for example a letter, on a written page. Very roughly, one Kbyte is sufficient to store a page of text. A Megabyte, or Meg, is one million bytes (actually 1,049,376 or 2^{20}). The internal memory ranges from 640K, to 16 Megs and larger.

Other Components and Packaging

Input/Output or “I/O” is usually done through a keyboard and a TV-like display or monitor. One commonly used display screen shows 24 lines (rows) each with 80 characters. Each character consists of a matrix of dots or pixels, usually 8 pixels high by 8 wide, making the screen 640 pixels wide by 200 high. Larger configurations, found in higher quality screens, are 1,000 pixels square.

Programs and data may be stored externally in *auxiliary memory*, which often takes the form of one or several disk drives. In the disk drives, the storage is on thin portable “floppy” magnetic disks, 5½ inches or 3½ inches in diameter. The disks are able to store from 720K to beyond a Meg. That’s sufficient space to store the text of an entire book. Examples of programs will be provided in Chapter 2.

The packaging of these basic components to form a complete computer system takes many forms. The organization ranges from all components being integrated into one box, to the other extreme where all are separate components that have to be plugged together.

The peripheral devices are connected to the main processor and perform many different functions, usually involving inputs and outputs. They often take the form of “cards”, plastic boards of electronic components. These cards plug into slots on the main processor unit.

The family of input peripheral devices include in particular a rectangular keypad for entering digits, a regular typewriter-like keyboard, function keys for specifying special operations and a “wand” for reading bar-codes.

For output, the range of peripheral devices includes mainly screens to display the output, and printers to provide “hard copy” on paper. A marker (cursor) on the display screen, which shows the current working point, can be moved by a mouse, pen, trackball or joystick. A mouse can be used to point, click, or drag objects on the screen. Printers produce text and graphics of varying quality, ranging from dot-matrix impact-type (where the image is produced by a type-head striking the paper as in a typewriter), to ink-jets, to higher quality laser printers and even typesetting machines. Plotters, which manipulate pens of varying colors are often also used to output high quality drawings.

Where there is a requirement for a large amount of auxiliary memory, this is available on “hard disk” drives, which may provide from 40 Megs to hundreds of Megs, and even Gigabytes (one billion bytes).

Communication between computers over telephone lines is possible with a device known as a modem (modulator-demodulator). Modems communicate usually at speeds ranging normally from 2400 bauds (bits per second) to 9600 bauds, and beyond. Computers can also be interconnected in *networks*, be they local area networks (LANs) or wide area networks (WANs), through simple twisted pair wires or coaxial cables.

Many new types of peripheral have recently become available. These include for instance speech recognition input devices, speech output devices, visual input scanners.

1.4 The World of Programming

What is Programming?

The subject of this book is programming, and as you progress, you will discover that programming is the art of instructing the computer to perform specific actions. In other words, to get the CPU and its components to work together and achieve a goal, such as, for example, solving a problem. This is done by providing the computer with a set of instructions written in a program language.

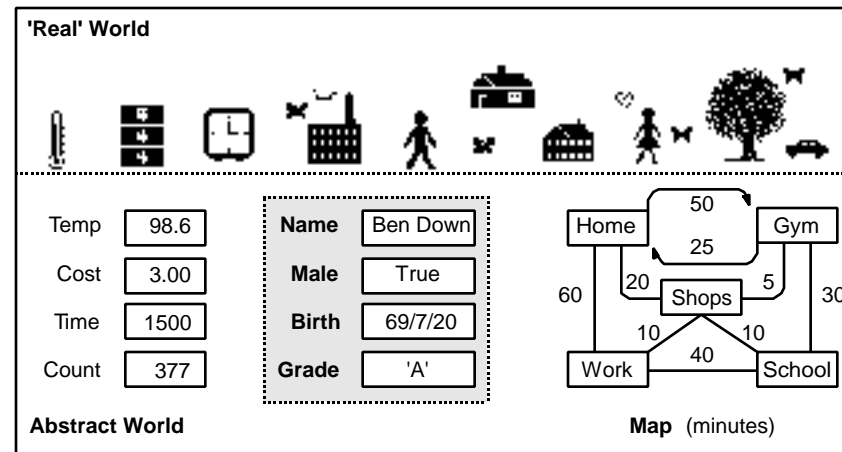
Why do the instructions have to be written in a programming language? The reason is simple. The English language was invented for humans to communicate with each other, and since the computer cannot understand human languages, the computer cannot understand English instructions. To bridge this gap, programming languages were invented so that humans could use computers to solve problems.

As the Principles book points out, programming languages are notations used for communicating algorithms between people and computers. There are hundreds of programming languages and, to give you a taste of these, Figures 2.20 to 2.25 of the Principles book show an example of the same program expressed in six different programming languages.

“Real world” and “Abstract World”

One view of programming, which was mentioned in Chapter 8 of the Principles book, is as a way of dealing with the “real” world by using a computer to build an abstract analogy of it. It is said to be “abstract” because it has been abstracted—taking only the parts that are essential to solve the problem at hand—from the real world original.

The real world is said to be “modeled” by the abstract world. The real world contains items of data such as time, money, grades, etc. whereas the abstract world treats these data as “black” boxes containing numbers and other symbols. The real world allows actions on these objects, including counting, measuring, sharing and sorting, which may also be modeled in the abstract world. A person is not a number. A map is not reality; much is left out—this is the process of abstraction, illustrated by Figure 1.2.

Figure 1.2 Reality vs. Abstraction

Communication in the real world is between people, and is possible even if there are some errors of spelling, punctuation, grammar or some imprecision. This is possible because we all have enough knowledge about the real world to correct the errors as we go. Generally, this error correction happens so naturally and fast that we don't even notice it.

Communication of a program in the abstract world, between users and computers, is impossible if there is any error of spelling, punctuation, etc. Computers and programming languages, are at present, rather intolerant of errors and have very little automatic error correcting ability; however, they can often pinpoint the errors for the users to correct. The stages involved in successfully programming an abstract model of the real world is shown in the next section.

1.5 Pascal

Pascal is a programming language developed by Niklaus Wirth around 1970. It is a rather simple language of moderate size and complexity, but powerful, compact, reliable, and efficient. Its clarity, simplicity, and structure make it most suitable as the first language to be learned. In fact, one of Wirth's major design goals was to produce a language that would be suitable for the teaching of programming.

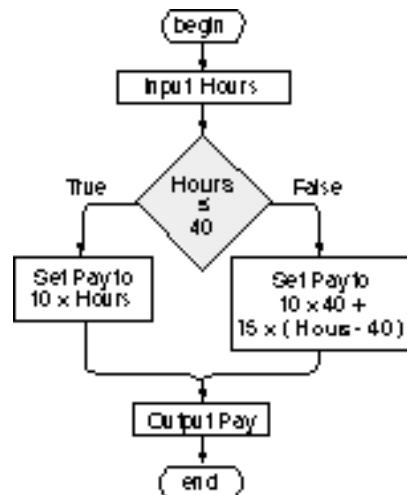
It is not a good idea to skip the fundamental concepts that were introduced in the Principles book, and begin by learning the language, for, as good as Pascal is, it is unnecessarily restrictive to think in terms of any one programming language.

An example Pascal program

In Chapter 2 of the Principles book you've seen a number of simple pay algorithms. Although we don't want to anticipate on the presentation of the

various features of the Pascal programming languages, we'll show you now the Pascal program corresponding to the Pay algorithm found at the left of Figure 2.18 of the Principles book. The spreadsheet Pascal program we've just seen was mostly based on invocations, which made it extremely simple. This simple pay program is a better example of what Pascal statements look like. We'll reproduce here in Figure 1.3 the original pay algorithm.

Figure 1.3 A simple pay algorithm



The corresponding Pascal program is given in Figure 1.4. Try and match the various parts of the flowchart with the various parts of the program. This should not be too difficult as we have been careful to use indentation to show more clearly the various parts of the program (and help you match the IFs with the ELSEs). Here again we don't expect you to grasp all the details, but this should help you become familiar with the form of Pascal programs, a skill you'll acquire in the following chapters.

Figure 1.4 The Simple Pay Pascal program

```

PROGRAM SimplePay(INPUT, OUTPUT);
{ Simple pay in Pascal }
VAR
  hours, pay: INTEGER;

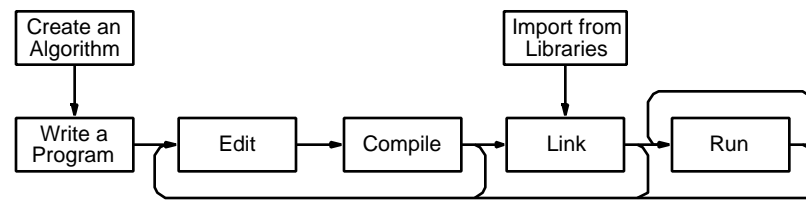
BEGIN
  Read(hours);
  IF hours <= 40 THEN
    pay := 10 * hours
  ELSE
    pay := 10 * 40 + 15 * (hours - 40);
  Writeln('Gross pay is ', pay:6);
END.

```

Running Programs: Compiling, Linking, Executing

As illustrated in the Principles book, the complete creation of a program takes a number of steps. In fact, our problem solving method has seven steps. But even the Program Coding and Testing step is itself made of a number of smaller steps or stages. The diagram in Figure 1.5 shows the sequence of stages (Editing, Compiling, Linking, and Executing), and what is involved at each stage. These stages must be followed in order to get a computer to run a program properly.

Figure 1.5 Stages of the Program Coding step



Editing: this might be the hard part of the Program Coding step. The algorithm is coded in some programming language, in this case Pascal. Through the use of an editor application program (see Chapter 2), the program is put into some computer readable form. This form of the program is the *source code*.

Compiling: this is the process of translating a source program into a lower-level machine language program, the *object program*. This translation process is performed by another application program, a *compiler*. There is a different compiler for each programming language and for each machine on which the program is to run. Part of the compiling process involves checking the source code for syntax errors.

Linking: this is the process of connecting a program to other separately compiled programs such as modules from various Libraries. If all the parts that comprise the complete program are available, the result of linking will be an *executable program*. If some parts are missing, error messages will be produced to explain why the link did not complete properly.

Executing: this is the process of running the executable program on the computer. An executable program can be executed any number of times without additional compiling or linking.

Any errors detected during these stages must be corrected, and the stages tried again in the sequence shown. If there are no errors or other changes, then to rerun the program requires only executing the executable program at the last stage.

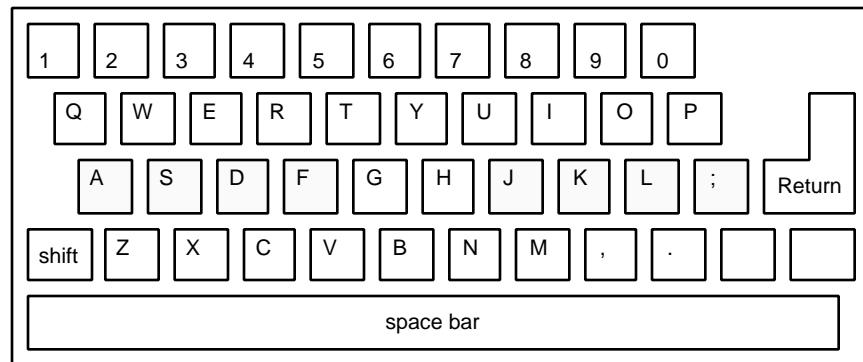
1.6 Communicating to Computers

Typing

Typewriting is a skill that is very useful for communicating to computers, to bosses, to colleagues, to professors, etc. Without this skill, the computer user is likely to be quite handicapped. It is possible to “hunt and peck” with two fingers, but that method is very tedious, slow, tiring, and inefficient. If you intend to use computers a lot, it would be a good investment to spend some time and effort on attaining proper skills, and possibly overcoming some bad habits which may be deeply ingrained.

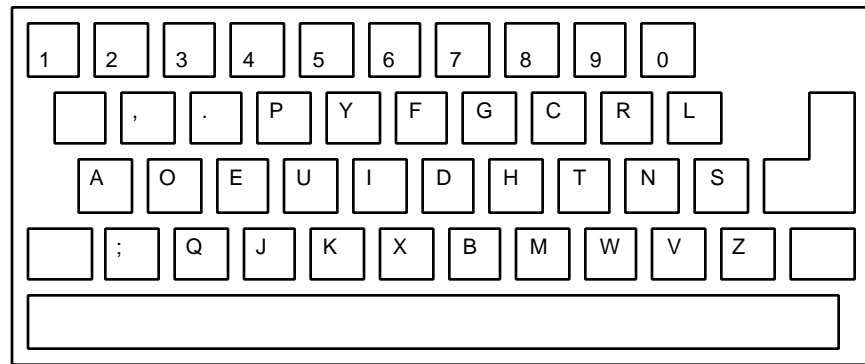
Touch typing is actually very easy to learn, especially with computers. It just takes practice which can be done while writing necessary material such as: programs, term papers, letters, etc. In the past, with mechanical typewriters, it was difficult to correct typing errors, so speed had to be sacrificed for accuracy. With computers it is very easy to correct errors, either instantly when they are made or at a later time.

Figure 1.6 QWERTY keyboard layout

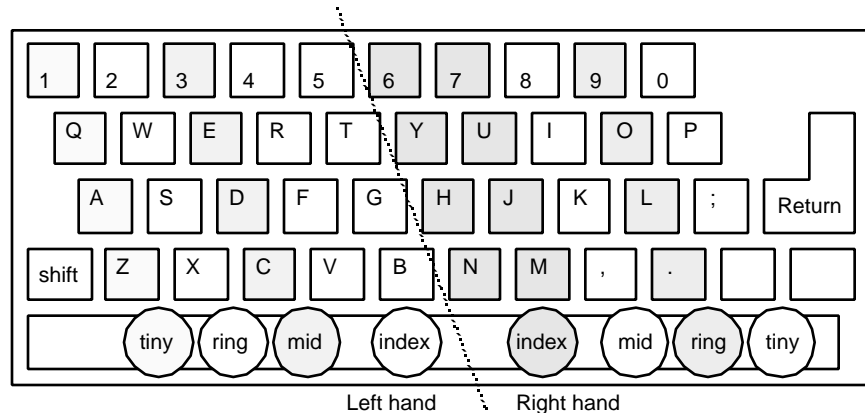


The most common keyboard is called QWERTY, named after the first row of keys, as shown in Figure 1.6. Another layout of keys, called Dvorak, is more efficient but not common. It is shown in Figure 1.7.

The normal “home row” position on the QWERTY layout has fingers on the keys “asdf” and “jkl;”. Often the keys D and K have slight bumps for the middle fingers to find, without looking. On other computers F and J have bumps. You may wish to stick tape on the “home position” of two keys.

Figure 1.7 Dvorak keyboard layout

The traditional approach to teaching typing begins by emphasizing the proper posture: back slightly forward, feet on the floor, and hands curved over the home-row. The secret to learning touch typing is not to look at the keys! Each letter or key is to be hit by the specific finger assigned to it, as shown in Figure 1.8.

Figure 1.8 Finger assignments

Train yourself so that the sight of any letter automatically causes movement in the corresponding finger. Keep your eye on the paper you are copying from or on the screen that you are copying to; do not look at the keyboard. You may wish to correct errors immediately, or wait until you finish; try both ways.

A good way to start is by doing some small finger drills for exercise, to improve speed and gain confidence. Some sequences you could try are:

```
asdfg hjkl;
frdesw jukilo
frvt juym
jw9x# S(b;0
aeiou rst
```

```
staying with the home keys
straying from the key positions
further from key positions
random selections
common letters
```

<code>ea, ed, es, ing</code>	common combinations
<code>a the it to in</code>	common short words
<code>aAsSdDfF</code>	upper and lower case
<code>It is in it, is it not?</code>	sentences
<code>abcdefghijklmnopqrstuvwxyz</code>	alphabet
<code>123 456 789 0 1492 1984 2001</code>	numbers
<code>The quick brown fox jumps over the lazy dog</code>	alphabet
<code>WHILE IF THEN ELSE FOR END</code>	words in Pascal
<code>1 2 Buckle My Shoe. 3, 4 Shut The Door.</code>	
	mixed words, numbers, caps
<code>Roses are red, Violets are blue ...</code>	poems?
<code>Once upon a time there lived three bulls, a mommy bull ..</code>	stories

You could also create your own sequences that have whatever structure you like. You might choose, for example, your name, address, phone number and other things that you have to write often. However, the best practice is always to type everything properly, without looking at the keys. You will be surprised how quickly you can improve.

Evaluation of your progress is helpful because it reinforces good performance. You should keep track of your improvement in speed (words per minute) and accuracy (errors per 100 characters) but remember that there is a tradeoff here—the faster you go, the more errors you are likely to make. Your learning curve is likely to have plateaus, where there will be no progress for days, followed by large jumps of considerable improvement.

Software programs for teaching typing are available. They often include speed tests, accuracy reports, and games. We have created two simple typing applications, but you might wish to extend them later, in order to add to their functions and statistics reporting. There is an example of such an application, a typing application, in the next chapter.

1.7 Chapter 1 Review

This chapter has provided a very quick overview of many computing concepts and applications.

Hardware evolves particularly quickly and specific components become obsolete soon. Many of the numbers mentioned here will change enormously in a short time. The speeds will get faster, the sizes will get smaller, the capabilities will get larger and the prices will get smaller. Computer memory, for example, double in capacity and halve in speed every few years.

Programming is the art of instructing the computer to perform specific actions which solve a problem. One view of programming is to see it as a way of dealing with the “real” world by using a computer to build an abstract analogy of it. In any case, in order to get a computer to run a program properly, a sequence of stages must be followed (Editing, Compiling, Linking, and Executing). These stages comprise the Program Coding and Testing step of our seven step problem-solving method which was introduced in the Principles book.

To communication between people and computers at a detailed level, one must use a keyboard. To efficiently communicate using a keyboard, one should at least be familiar with the practical skill of touch typing.

Chapter 2 Computing: A Short Survey of Some Applications

Software is the stuff that makes computers go. Without software— the collection of programs that is needed to operate the computer system— computers are just a collection of lifeless hardware. This chapter introduces software by giving some examples of packaged programs, applications, which perform specific tasks.

Chapter Overview

2.1	Preview.....	30
2.2	Software and Applications.....	30
2.3	Application Software.....	33
	Editor Application.....	34
	Typing Applications.....	35
	Calculator Applications.....	36
	Retrieve Application: A Tiny Database.....	38
	Planner-Calendar Application.....	39
	Bar Plot Application.....	41
	Drill Application.....	42
	SSS: Small and Simple Spreadsheet.....	43
2.4	Chapter 2 Review.....	47

2.1 Preview

The collection of programs that is needed to operate the computer system, the *software*, consists of programs and libraries of programs. Generally, a library of programs is a group of utility programs that are related by the kind of operations they perform. Although software, like hardware, has evolved considerably in its few dozen years of existence, it generally has a longer life than the hardware on which it runs. One program, over its lifetime, may run on a range of different hardware.

In this chapter, the software that we describe mainly consists of *applications*, which are packaged programs intended for use by people who may not have programming backgrounds but wish to use computers to perform specific tasks. The uses of these packages are many and diverse, touching virtually all fields. They are used for learning, organizing, communicating, drawing, writing and playing. In this chapter we will briefly describe some common application packages through simple examples.

In later chapters, we will create and modify the Pascal source code for some of these applications. The purpose of this chapter is to introduce you to some common applications which can be programmed for a computer using Pascal, not the Pascal programming language itself.

2.2 Software and Applications

Software is the stuff that makes computers go. Computers are nothing without software! Software breathes life into computers like tape recordings bring sound to recorders. This sounds like “hype”, but it is really true, because if you were given a piece of sophisticated hardware (an advanced micro-computer for example) with no software, you could not use it. Some software is developed for a special purpose, related strictly to one area of application, but many software packages are more general and can be used in many different areas. For example, most word processing software can be used wherever writing is required, irrespective of the subject matter. The current word processing packages are more general because they also include functions to do graphics, and even some of the functions of spreadsheet software packages. In this chapter, we intend to give you an overview of general purpose software. Some typical software packages are represented in Figure 2.1.

Figure 2.1 Some typical software packages

Write (with a Word Processor)

Bull Story

Once upon a time there lived three bulls, a daddy bull, a mommy bull, and a baby bull.

And they lived happily ever after.

GetFile, Search, Cut, Paste, Format, SpellCheck, Print, Save, Help, Quit.

Organize (with a Data Base)

Who Name: Dan Druff

Where Street: 300 Main Ave
City: Northridge, CA
ZipCode: 9 1 3 3 0

How PhoneNo: (818) 885-3398

What Occupation: Professor

Why Appointment: Back Injury

When Date & Time: 91 / 2 / 29 1530

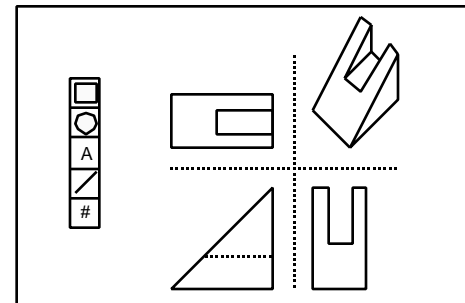
Compute (with Spreadsheet)

Get, Insert, Delete, Sort, Save, Print

Trip Expenses (rounded)

Mon	Tue	Wed	Thu	Fri	Sum	Why
70	36	28	66	0	200	Room
15	20	32	17	27	111	Food
10	15	12	13	16	66	Gas
10	11	33	0	5	59	Misc
105	82	105	96	48	436	Total

Draw (in 3 dimensions)



Combined (Concurrent) Applications

MENU

Write
Organize
Calculate
Spread
Draw
Play

WRITE (Bull Story)

Once upon a time there lived three bulls, a daddy bull, a mommy bull, and a baby bull.

DBASE (Phonos)

WHO Dan Druff

Call

TO DO

Call Jan
See Ray
Send Rent
Give Dues
Buy Milk
Meet Dean

CALCULATE:

1234.56

SPREAD (Grades)

Name	exam1	exam2	final
Able, I	88	78	80
Baker, J	95	90	75

TIME

The use of computers for writing, or *word processing*, is very common. Writing on a screen provides extreme flexibility, allowing opportunity for many revisions. Text can be checked for spelling, formatted in various ways (aligned, proportionately spaced, etc.) A feature called outlining encourages creation in a top-down manner. After using a word processor it is most painful to go back to using a typewriter.

The organization and the saving of data in a *database*, so that the data can be selected and retrieved, is very convenient and fast with a computer. There is much software available, ranging from small phone book managers to monstrous database management systems. These database application programs allow us to structure our data. This is absolutely necessary to manage the complexity of the constantly increasing amount of information we have to deal with.

Many software packages provide the ability to calculate and analyze data. *Spreadsheets* are particularly useful for analyzing data that can be put in a table, with rows and columns. They are particularly suited to “what if” queries, for values can easily be changed, and new results almost instantly obtained. *Statistical packages* take data and perform various statistical tests to find relationships within the data. Calculators are often “simulated” on computers and can be made to “pop-up” onto the screen at any point where a quick computation needs to be made. Other computing software of this kind includes packages for accounting and tax preparation.

A more recent application for computers is *drawing*. The range of applications is from business graphic charts to CAD (computer aided design), to graphic art or even to doodling. Drawing can be done from a keyboard, using keys that control the cursor position, but usually, it requires input devices such as mice, light pens, or digitizers.

The ability to exchange texts, pictures or data between computers over a network or through telephone lines, is very a very convenient, powerful and flexible tool. There are many *communication* software packages, designed for a number of systems and computers. They make on-line information services available to access databases, get news, airline flight information, exchange mail, get software, or participate in a conference. Some banks allow access from home computers to check balances, transfer funds, and pay bills. Electronic bulletin boards run by individuals or organizations to share common interests are numerous and growing.

The use of computers for *teaching* is still at an early stage. Some repetitive drill-type exercises are useful but usually at low levels and for specific subject areas. In some specific areas, simulations of natural phenomena or experiments are very meaningful to students, and an invaluable help to teaching.

A very common activity on computers is playing *games*. There exist many kinds of computer games, some involving strategy, others involving manual skills, and some others even involving social skills. Many games make much use of sound and color

Computers can be applied in virtually all areas, and obviously we cannot establish an exhaustive list of all computer applications. These will include all the applications we have already mentioned and a large number of other applications going from genealogy to cookbooks and to farm management, etc.

2.3 Application Software

Application software includes programs to perform useful functions in different areas. This chapter is devoted to eight small but useful applications:

Edit	a simple editor to help enter lists into files
Type	a typing tutor to help improve keyboard skills
Calculate	a calculator to manipulate both real and complex numbers
Retrieve	a tiny database to retrieve data from files
Plan	to create a calendar planner for organizing
Plot	to create histograms or bar plots
Drill	to provide exercises in arithmetic
Spread	to manipulate data in the form of tables

It is recommended that you become familiar with some of these applications, not necessarily all of them. The Retrieve application is a tiny database manager that could be especially convenient for organizing some of your personal data (address book, schedule, To Do lists, etc.) The data files can be created with the tiny Edit application given here, or they could be created in any larger editor like the one that came with your Pascal system. This tiny editor has the advantage of working on all versions of Pascal.

The programs for these applications are written in Pascal, and most of the details are shown later in this book. Each program is short and should fit on a page so that you can see it all at once. This small size is made possible by the use of “reusable” components from *Libraries*.

A library is a collection of utility programs that are related to a common area. For example, the Edit program uses two operations on strings (Read and Search) that are obtained from library StringLib. The Plan program uses operations on dates (WriteMonth, DaysInMonth, etc.) that are obtained from library DateLib. The use of programs from libraries reduces a programmer’s work considerably because these common operations do not have to be written anew every time a program is written. When a library has been created, it is kept in the system. Any programmer can then “import” the operations she needs from this library, and use them as if she had programmed them herself. So, if a number of general purpose libraries have been developed, the work of the programmers is reduced, as they can use the available operations directly in their programs. The creation and use of such Libraries is an important part of this book.

Modular programming is based on the concept of creating programs by using large building-blocks. To help in the modularization of a program, it is useful to create large building-blocks that can be “encapsulated” into Libraries. For example, we have presented in Chapter 8 of the Principles book, the concept of an ADT (Abstract Data Type). The types, data structures and operations of an

ADT are usually encapsulated in a library, from which necessary items may be obtained for use in other programs (or other Libraries).

In order to use an application, users need not know its internal details. Nowadays, application packages are sufficiently convenient to allow use without any need to know about programming. The applications we discuss here are quite small and not entirely complete, but all of them can be extended easily. After you have used an application, you may already have some ideas about additional features that could be useful. Later, after learning more about programming and Pascal, you may examine the actual Pascal program for an application and modify it to extend the application the way you want it. This is exactly what program “maintenance” is, already presented as the seventh step of our problem solving method.

Editor Application

TED is a Tiny Editor used to create and modify files of text. Most Pascal systems include a rather complex editor, but all are different and not transportable to other systems. This small editor runs on all Pascal systems; it is small but it is portable. It is also fast and easy to learn. Later when you have gained more experience, you will want to use the added features of a comprehensive editor, and move to the larger editor that your Pascal system provides.

This editor acts on a line at a time. It requests operations by displaying a question mark “?” and the following commands may be given (by the first letter of the command: A, B, D, etc. in upper or lower case):

```

B  to go to the Beginning of the file
D  to Delete the line at the current position
T  to Type out the entire file
P  to move the current pointer to the Previous line
N  to move the current pointer to the Next line
I  to Insert one line after current line
F  to Find a given string, starting at the current line
M  to Modify the current line
H  to provide Help, by listing all commands
R  to replace the current line by another
S  to Save a file
L  to Load a previous file which was created and saved
E  to End or exit the edit session

```

Here is an example of a typical file editing session, with comments at the right in a different font. The part typed by the user is in bold. Note that commands may be typed either in upper or in lower case.

```

?i          insert a line
Roses are red
?I          insert a line
Violets are blue
?i          insert a line

```

So are you	
?t	Type the whole thing
Roses are red	
Violets are blue	
So are you	
?p	Go to the previous line
Violets are blue	
?i	Insert following that line
Sugar is sweet	
?T	Type out the file
Roses are red	
Violets are blue	
Sugar is sweet	
So are you	
?S	Save the file
Enter name of file	
out> RosesFile	Give name of saved file
?e	Exit, end the edit session
End of edit	

TED is a program written in Pascal that uses pointers and doubly linked lists. It is rather short because it makes use of data structures and operations from a library called StringLib. You may later wish to extend it in some of the following ways:

- Add more commands, such as append, to insert more than one line
- Output with line numbers for reference
- Provide more detailed help or instructions.

Typing Applications

There are two aspects to the skill of typing, speed and accuracy. Here, we show two application programs that provide practice in these areas.

TypeTimer is an application program that presents a line of text to be typed in, and then indicates how quickly this line was typed. Accuracy is not measured because it is assumed that errors can easily be corrected. A typical run of this program is shown below; the part of the dialog that the user typed is shown in bold.

```

Typing Speed Test
You are to type the following line
Type Return when you are ready, and
type Return when you are finished
A quick brown fox jumps over the lazy dog
A quick brown fox jumps overf the laxy dog
The time taken is 50 units.
```

Time is measured by units, which are not seconds, but some arbitrary units that are consistent and serve to compare times to measure progress. Such a typing

exercise is recommended as a “warm-up” before other computing activities. Keeping track of improvement is useful.

TypeWell is another application program that presents a number of lines of various kinds of text and indicates whether the typed line is correct or has errors.

Text of two kinds can be selected. One kind is a set of “silly” sentences, each consisting of all the 26 letters of the alphabet. Beyond the common “a quick brown fox jumps over the lazy dog” it includes many more such sentences. The other kind of text includes various common words, Pascal reserved words, numbers, and other useful pieces of text. A typical run of this program is shown below, the part typed by the user is in bold type.

```
Typing Accuracy Test
What do you wish to try:
    Silly sentences or serious statements?
    Enter "silly" or "serious"
silly
Type the following
a quick brown fox jumps over the lazy dog
a quick brown fox jumps over the lazy dog
CORRECT!
Type the following
exquisite farm wench gives body jolt to prize stinker
exquisite farm wench gives body jolt to prize
stinkler
ERROR!!!
.. etc. .. etc.
Try again soon.
```

These two application programs are both written in Pascal, and they make use of strings, and files (covered in Chapters 5 and 6). Soon, after learning to use an editor, you could modify the kinds of text in the two files named “silly” and “serious”. Later you may wish to modify the program in various ways:

- to combine both speed and accuracy tests into one,
- to keep track of your progress after each exercise.
- to enter yet a third kind of file, “semi-serious”,
- to count the number of errors.

Calculator Applications

RealCalc is a program that simulates a hand held calculator. It provides the typical four arithmetic functions (addition, subtraction, multiplication, and division). Entering the letter “q” or “Q” (for “Quit”) causes the calculation to stop. An example calculation is shown below. It converts 100° Celsius to the equivalent Fahrenheit unit (212°). The part typed by the user is shown in bold type.

```
Enter a value
9.0
Enter an action
/
Enter a value
5.0
The result is  1.80
Enter an action
*
Enter a value
100
The result is  180.0
Enter an action
+
Enter a value
32
The result is  212.00
Enter an action
Q
End of calculation
```

The behavior of this calculator is similar to that of the common four-function calculator. However, it differs from most hand-held calculators in that each value and each action are on separate lines. This general form of calculator is easily extended to the manipulation of other data such as very long numbers or complex numbers.

For example one such extension, ComplexCalc, operates on complex numbers, i.e. numbers having a real part and an imaginary part. The following shows a typical calculation where, as usual, the user's responses are shown in bold.

```
Enter a value:
  Input Real part
  1.0
  Input Imag part
  2.0
Enter an action
+
Enter a value
  Input Real part
  3.0
  Input Imag part
  4.0
The result is
  Real part = 4.00
  Imag part = 6.00
Enter an action
Q
End of calculation
```

The application program RealCalc is written in Pascal, and can be extended in many ways:

- to compute squares and powers,

- to include trigonometric functions,
- to output in different forms (such as scientific notation).

ComplexCalc is also written in Pascal, and makes use of operations on complex numbers from a library, ComplexLib. The program could also be modified in many ways:

- to compute the conjugate of a complex number,
- to compute magnitudes and angles,
- to output in different forms (such as polar notation).

Other calculators based on this program could keep the same form and accept other types of data like the following.

- Large integers (much larger than the standard computer maximum sizes of 16 to 32 bits) ,
- Vectors (of coordinates in 3 dimensions),
- Matrices (or arrays of two dimensions).

Retrieve Application: A Tiny Database

Retriever, or Tiny Database, is a program that searches a file to find a given key word or phrase and retrieves the information associated with that phrase. One database file, for example, might involve a listing of various names, phone numbers, addresses, and other information such as the following.

```
Cetera, Ed      (818) 885-3398 1234 First Street, Northridge, CA 91324
DeLion, Dan    (405) 349-6400 5678 Shady Lane San Francisco, CA 90000
Druff, Dan     (818) 123-4567 2468 Shady Alley Los Angeles, CA 90123
Funt, Ella    (818) 349-1234 1500 Louis Lane, Northridge, CA 91330
Gone, Polly   (818) 548-5948 300A Shady St, Chatsworth, CA 91350
Ho, Gung      (818) 543-7654 248 Main St # 7 Northridge, CA 91324
Ono, Kim      (818) 987-6543 123 Easy St, Chatsworth, CA, 91444
Stein, F.N.   (405) 666-6666 "Frankie" 13 Lucky Lane, SF, 94114
Wood, Holly   (213) 349-6417 X1200 *** Don't call after 9pm
fire:         111-2222
paramedics    111-3333
police        111-4444
time:         222-1212
weather       333-1212
```

Each piece or line of information forms a data record. The parts within a record (name, phone number, zip code, etc.) are called data fields. Notice that here the records contain very differing fields and the fields have differing details, lengths, etc. This is a “free-form” database, as opposed to other more complex and highly structured ones.

Searching such a database called “Phonos” for the key “Funt” is done as follows (the people's part is in bold type).

```
Enter search pattern Funt
Enter the File Name Phonos
```

```
Funt, Ella (818)-349-1234 1500 Louis Lane,  
Northridge, CA 91330  
End of Search
```

Files such as this one may be searched in many different ways. We could search for a first name such as “Dan” and it would retrieve the records associated with “Dan DeLion” and “Dan Druff”. Similarly, we could search for states, e.g., “CA” or cities, say “Northridge”, or area codes, such as “818”. We could also search that database using special marks like, for example, “***” or “!!”, or even parts of words, such as “Comput” to get anything dealing with Computers, Computation, Computing, etc. Note that searching for a blank space “ ” would retrieve the entire data base! On the other hand, searching for a zip code such as “93333” would retrieve nothing.

An application program like this one could be applied to other files containing different types of information, such as part inventories, time schedules, tasks to be done, programs on a disc, collections of stamps, cars on a sales lot, people in an organization, etc.

The Retrieve program is written in Pascal. It is less than a page long mainly because it uses the operations on character strings from a library called StringLib. Using operations from a library means that we don’t have to write those parts of the program, and instead we can use someone else’s work, which we know is already tested and working. The Retrieve program stores the files as one long sequence of characters. The data file must be created separately by an editor or a word processor. You may wish to create a file of interest to you, and use the tiny database program to retrieve various details from such a file. Later you can modify this program in a number of ways, including the following.

- to enter and delete items from the tiny database instead of with an editor
- to search for fields within the records (i.e. phone numbers only)
- to sort the retrieved items in order
- to print labels for a mailing list
- to format the items
- to count the number of items of various kinds.

Planner-Calendar Application

Planner is a simple application that creates a calendar for any number of months of any year after 1900, and provides the calendar on a file so that it may be modified and “customized”. A typical run of this program follows. The trace is shown first, followed by the resulting file. As usual, the user’s part of this dialog is in bold type.

```
Enter the calendar year 1994  
Enter the first month, 1..12 1  
Enter the number of months 2
```

```

Enter the gap between months  2
Enter the gap between weeks   1
Enter the destination file name
Out>  JanFeb94
Completed the calendar file

```

Year 1994

January

Sun	Mon	Tue	Wed	Thu	Fri	Sat
---	---	---	---	---	---	---
						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					

February

Sun	Mon	Tue	Wed	Thu	Fri	Sat
---	---	---	---	---	---	---
			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					

The file “JanFeb94” produced by this program can be modified in a text editor as shown below. Most of the changes consist of comments added at the right of the calendar. Changes can also be made to the calendar part; some dates can be marked (by an asterisk, underlining, or bolding, depending on the editor).

Year 1994

January

Sun	Mon	Tue	Wed	Thu	Fri	Sat
---	---	---	---	---	---	---
						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					

03: Holiday!!
 14: Phil's Birthday
 17: Martin Luther King Day
 24: Registration, Car Repair
 31: Classes begin

February

Sun	Mon	Tue	Wed	Thu	Fri	Sat
---	---	---	---	---	---	---
			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					

11: CS assignment due
 14: Valentine's day
 26: Skiing!!!

- to produce various labels as shown on the edited version above,
- to produce an axis and border around the plot,

- to plot on both the screen and a file,
- to scale a plot to fit onto a page.

Drill Application

The application program Drill provides practice in simple multiplication. It presents a series of multiplications chosen at random, and after each multiplication waits for an answer. If the answer is correct, it displays “Correct”, otherwise, it provides the correct answer. It also keeps track of the number of correct responses, and shows this score at the end of the series. A typical run of this application follows, where, as usual, the user’s response is shown in bold. Notice that a starting value, the “seed”, is required at first; any integer value will do. The Drill program makes use of another program called a *Random Number Generator* to produce the actual values that are used to make up the questions. The random number generator produces a sequence of numbers that appear to be chosen by chance, just as if they were produced by rolling dice. The seed is used to start the random sequence. If the same seed is used on two runs of the Drill program, then the same questions will be asked.

```
Enter a seed (starting value ) 1
Type an answer to the following
and then press the Return key
8 * 3 = 24
Correct
0 * 3 = 5
The answer is 0
6 * 5 = 30
Correct
9 * 7 = 63
Correct
6 * 8 = 48
Correct
5 * 2 = 10
Correct
5 * 5 = 55
The answer is 25
Your score is 5 out of 7
```

Drill is a program written in Pascal and later, after learning more about Pascal, you may wish to modify it in some of the following ways.

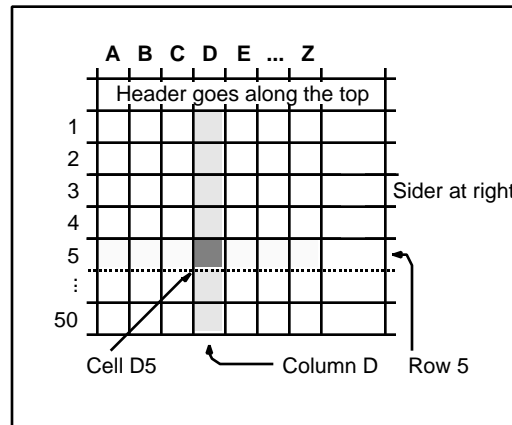
- to allow three incorrect tries before providing the solution
- to select larger numbers more often
- to allow input of a level of difficulty
- to extend the drill to additions and subtractions
- to extend to combinations of additions, multiplications, etc.
- to select the problems that you have most difficulty with

- to convert between various bases

SSS: Small and Simple Spreadsheet

Spreadsheets are very useful applications, especially in Business. Essentially, they consist of a rectangular grid of cells, as shown in Figure 2.2. Numbers can be placed in these cells and the spreadsheet program provides a repertoire of operations for manipulating these values.

Figure 2.2 Spreadsheet Grid of Cells



As an example of the typical use of a spreadsheet, Figure 2.3 shows a list of prices and quantities of chair parts. One useful action on this table would be to multiply the Price column by the Quantity column to create a third column called Total. A second action, SumAColumn, could sum this third column and put this sum at the bottom of the column. The Quantity column can be similarly summed.

Figure 2.3 Example Spreadsheet for Parts

2 Columns			
5 Rows			
Price	Quantity	Total	Part
10	20		Backs
30	40		Legs
50	60		Rungs
70	80		Seats
0	0		TOTAL

“Real” spreadsheets usually have a highly graphic and convenient user interface with a mouse allowing quick positioning of the cursor onto a cell. However every different computer requires a very different program. Each is

used in a different way. Here, we present a simple spreadsheet application program written in Pascal that can be run on all computers in the same way.

Since it is simple, it has its limitations. There can only be 26 columns (labeled alphabetically from A to Z), and 50 rows, so that a normal spreadsheet will fit onto one page. Also, all the values must be positive integers. The set of possible operations that this application can perform on a spreadsheet is simple and shown below. Again, you will be able to add to this set once you have experience in Pascal programming. Text appears only on the top line (called a Header) and on the right side (called a Sider).

Data for the program, will be kept in a file, as shown in Figure 2.4, and this file can be created using any text editor. The first values indicate the number of columns and the number of rows. Then there is a single header to briefly describe the columns. This is followed by the table of data values with brief descriptions at the right of the table. Note that Total in the header does not correspond to a column in the table (only 2 columns). This format is rather fixed and rigid to make the program simpler. Later, you will be able to modify the program to make it easier for you to use.

```

2 Columns
5 Rows
Price Quant Total Part
      10    20   Backs
      30    40   Legs
      50    60   Rungs
      70    80   Seats
      0     0   TOTAL

```

Figure 2.4 Parts Inventory File

Before: Original data				
	A	B	C	
	Price	Quantity	Total	Parts
1	10	20		Backs
2	30	40		Legs
3	50	60		Rungs
4	70	80		Seats
5	0	0	TOTAL	

After SumAColumn (B, 1, 4, B, 5)				
	A	B	C	
	Price	Quantity	Total	Parts
1	10	20	0	Backs
2	30	40	0	Legs
3	50	60	0	Rungs
4	70	80	0	Seats
5	0	200	0	TOTAL

After MulColumns (A, B, 1, 4, C)				
	A	B	C	
	Price	Quantity	Total	Parts
1	10	20	200	Backs
2	30	40	1200	Legs
3	50	60	3000	Rungs
4	70	80	5600	Seats
5	0	200	0	TOTAL

"Final" File (after Editing)				
Inventory: Chairs				
Price	Quantity	Total	Parts	
\$10	20	\$200	Backs	
\$30	40	\$1200	Legs	
\$50	60	\$3000	Rungs	
\$70	80	\$5600	Seats	
.....	
	200	\$10000	TOTAL	

The actions required for this spreadsheet program are few initially, but may grow to many. We will consider a basic set of simple operations on columns and rows. They take the form of subprograms which are imported from a Library called SpreadLib and are as follows.

- Initialize is a subprogram that sets up the spreadsheet. It requests the width of columns. All columns have the same width.
- LoadSpread is an operation to take the values from a given file and put them into the cells of the spreadsheet. The number of columns and rows must be specified first in this input file.
- SumAColumn(C1, R1, R2, C2, R3) is an operation to sum the values in column C1 (from row R1 to row R2) and put the resulting value into column C2, row R3. Usually columns C1 and C2 are the same, but they need not be. For example, in the given example of an inventory of chair parts, SumAColumn(C, 1, 4, C, 5) sums the first four values in the column C and puts this sum into the fifth row of this same column C. MaxAColumn, MinAColumn and MeanColumn are similar.
- AddColumns(C1, C2, R1, R2, C3) adds all the values in column C1 and column C2 (from row R1 to row R2) and puts the resulting value into column C3. MulColumns similarly multiplies columns. For example, in the inventory problem, MulColumns(A, B, 1, 4, C) multiplies the first column A of Prices (from rows 1 to 4) by the second column B of Quantities yielding a third column C of Total cost.
- SubColumns(C1, C2, R1, R2, C3) subtracts from C1 the values of C2, in the same manner as AddColumns above. Similarly, DivColumns divides the first column by the second column.
- ShowSpread(C1, C2, R1, R2) displays the spreadsheet on the screen from the column labeled C1 to that labeled C2, between rows R1 and R2. This makes it possible either to show the entire spreadsheet or to show only a given section of the spreadsheet. This operation can be used many times within a program to trace the effects of various actions.
- SaveSpread(C1, C2, R1, R2) saves a spreadsheet, or part of a spreadsheet, in a file where it can be edited later, or loaded in again for further manipulations.
- MultColConst(C1, R1, R2, K, C2) multiplies a column C1 (from row R1 to row R2) by a constant value K and puts the result into column C2, which could be the same as the first column C1. If this is the case, the result is that the first column was multiplied by K.
- MinOfARow(R1, C1, C2, C3, R2) finds the minimum value in row R1 between columns C1 and C2, and puts this value into column C3, row R2. Usually the two rows R1 and R2 are the same.
- AddTwoRows(R1, R2, C1, C2, R3) adds the values in the two rows R1 and R2, between the two columns C1 and C2, and puts the results into a

given row R3. SubTwoRows is similar; it subtracts the second row R2 from the first one R1.

Obviously, a number of other actions are possible, and could be added later. Now we can work with these basic operations to design a simple spreadsheet application. To do that, we write a small Pascal program shown on Figure 2.5.

Figure 2.5 An example Spreadsheet program

```
PROGRAM SpreadProg;
(* Small spreadsheet system *)

USES SpreadLib;

BEGIN
  (* Inventory spreadsheet *)
  Initialize;
  LoadSpread;
  SumAColumn(B, 1, 4, B, 5);
  ShowSpread(A, C, 1, 5);
  MulColumns(A, B, 1, 4, C);
  ShowSpread(A, C, 1, 5);
  SumAColumn(C, 1, 4, C, 5);
  ShowSpread(A, C, 1, 5);
  SaveSpread(A, C, 1, 5);
END.
```

Although this is not the first Pascal program you have seen (think back about the six program examples of Chapter 2 of the Principles book), we show it here as an example. We don't expect you to understand all the details of the program, but it should help you become familiar with the format of an actual Pascal program. Note the PROGRAM, USES, BEGIN and END statements, they are necessary to have a well defined Pascal program. The rest of this program is made of subprogram invocations that have the form we described above, and thus should not be difficult to understand. We'll introduce in the next chapters of this book all the details necessary to produce such a program.

Figure 2.6 Trace of the execution of the Spreadsheet program

```
Enter Column Width 6
Enter Source File Stock.in
  Price Quant Total Part
    10    20     0 Backs
    30    40     0 Legs
    50    60     0 Rungs
    70    80     0 Seats
     0   200     0 TOTAL

Press Return to continue

  Price Quant Total Part
    10    20   200 Backs
```

```

      30      40  1200 Legs
      50      60  3000 Rungs
      70      80  5600 Seats
      0     200      0 TOTAL

Press Return to continue

Price Quant Total Part
   10     20    200 Backs
   30     40    1200 Legs
   50     60    3000 Rungs
   70     80    5600 Seats
   0     200  10000 TOTAL

Press Return to continue

Enter Target File Stock.out

File is saved

```

Figure 2.6 shows a trace of the execution of this program: the first line is caused by the call to `Initialize`, the second line is caused by the invocation of `LoadSpread`. Then the three tables are displayed by the calls to `ShowSpread`, showing the table after each of the following operation, first `SumAColumn`, then `MulColumns`, and finally `SumAColumn` again. The last two lines of the trace correspond to the call to `SaveSpread` that saves the spreadsheet in file `Stock.out`. Note that the saved table has now three columns. Later, you may wish to extend the program or the subprogram `Library` to do other actions such as to `Move` one column to another, to `Sort` on a column showing the rank of each cell, or to `Compare` column values.

2.4 Chapter 2 Review

There are many applications of computers and they are used in a number of diverse fields. Here, we have shown some very simple applications (a tiny editor, a small data base, a simple spreadsheet, etc.) These complete applications are shown later in this book, but they can be used at this point. You need not know the details within them in order to use them. The average user certainly doesn't want to know the internal details, it would only complicate life!

The use of computer applications is significant in a number of ways. They show the "external" behavior or interface of a program, which could be "friendly" or "hostile". Use also shows limitations of the applications and suggests improvements. Then when you later achieve the appropriate facility with programming in Pascal, you may wish to return to these applications to modify and extend them in various ways. Modifying or maintaining programs is a challenging activity.

Software evolves just as quickly as hardware. Software is much easier to change than hardware and this is both a strength and a weakness. The temptation and potential for change is enormous, but if not done properly, the changes could be disastrous. Proper programs should be created in a way to facilitate changes. This is done using subprograms and Libraries and proper software engineering principles.

Libraries are the basic building blocks of applications. The Libraries consist of collections of smaller building blocks called subprograms (or procedures). In using these applications you will also learn some concepts involving the use of libraries, although only at an external black-box level. But soon you will be able to peek into these Libraries and make changes to them.

Chapter 3 Programming Language: Pascal

In this chapter we further introduce the Pascal programming language by comparing it to natural languages and by using syntax diagrams to help our understanding. Some extremely simple Pascal program examples are given as illustrations of the syntax presented. Although it will take a few more chapters to introduce all of the Pascal syntax, some other examples of complete Pascal programs are also given in this chapter.

Chapter Overview

3.1	Preview.....	50
3.2	Languages.....	50
	Syntax and Semantics.....	50
	Syntax Diagrams.....	53
3.3	Pascal Programs.....	56
	Program Format.....	56
	Program Presentation.....	59
	More Pascal Programming: Data and Actions.....	60
	Data Items.....	61
	Actions: Arithmetic Operations.....	63
3.4	More Example Programs.....	66
3.5	Chapter 3 Review.....	69
3.6	Chapter 3 Problems.....	70
3.7	Chapter 3 Programming Project.....	74
	Getting Acquainted.....	74

3.1 Preview

Natural language, whether spoken or written, is an example of a highly linear representation. When spoken, it is a sequence of words. In its written form, it is a sequence or stream of characters. Written text is organized into sentences, paragraphs, sections and chapters using mechanisms of spacing, punctuation, section headers, etc. With computer programs expressed in an artificial language, a programming language, the text is further organized by levels of indentation, and more spacing conventions that are used to show the organization into statements, forms, subprograms, modules, programs, etc.

All languages, natural or artificial, have rules that define what is meaningful and what is gibberish. These rules constitute a language's grammar or syntax. One way in which this grammar can be represented is through two-dimensional diagrams, called syntax diagrams. This representation is introduced in this chapter.

People find that two dimensional diagrams are quite easy to grasp. Many of the constructs of computer science are of this graphic type, such as trees, breakout diagrams, flowblock diagrams, dataflow diagrams, etc. However, computers at present are limited to manipulating long linear sequences of symbols, which include the characters of programming languages, text, data and, ultimately, bits.

The mismatch between the two-dimensional view of people and the one-dimensional view of computers is not as severe as it may seem. In fact, there are ways of representing two-dimensional structures as linear lists through the use of various methods, such as indentation, numbering schemes, and breakouts.

3.2 Languages

Syntax and Semantics

The term "languages" encompasses both natural languages, such as English, French or Latin, and artificial languages, such as programming languages. Both consist of linear sequences of symbols. Natural languages involve symbols, called words, that follow one another to form sequences called sentences. These sequences are constructed according to certain rules, the language's grammar or syntax. The syntax of English defines the sequence of words

You gave the ball to me

to be grammatical whereas the sequence

You gave the ball to I

is not. Because a language is intended to communicate, there is a meaning associated with syntactically correct sequences; this is its *semantics*.

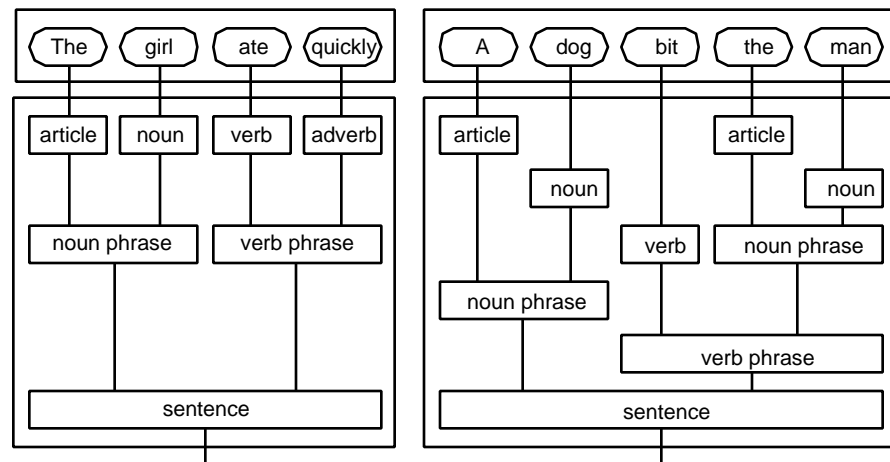
Syntax is the study of the form, representation, grammar, and structure of a language. Semantics is the study of meaning, actions, function, and behavior of a language. Briefly, syntax describes how a language looks and semantics describes what a language does. For example, consider the following two sentences:

a dog bit the man

a man bit the dog

These two statements have the same sentence structure (syntax) with only the two nouns (dog and man) interchanged. These two statements have, however, very different meanings (semantics), for the first would seldom appear in a newspaper, whereas the second is sufficiently unusual to become a headline.

Figure 3.1 Two sentence diagrams



A sentence in English can be analyzed into its *parts of speech*, this process is known as “diagramming” or “parsing” a sentence. Figure 3.1 shows sentence diagrams for two different sentences:

THE GIRL ATE QUICKLY

and

A DOG BIT THE MAN

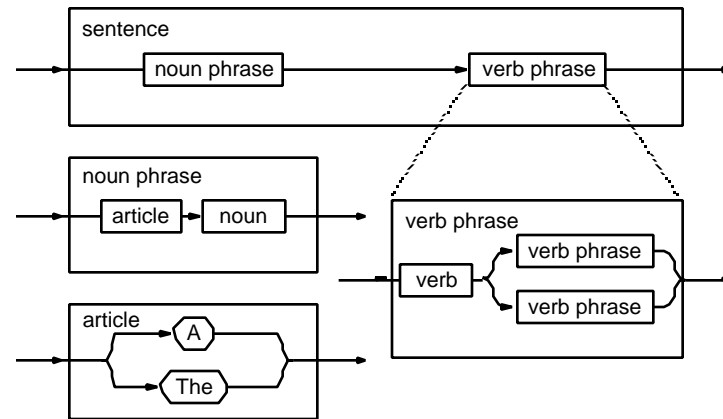
In computer science, such diagrams are known as *syntax trees*. The actual words (the symbols) of the sentences are at the top of the tree. They are called the *terminal symbols* of the syntax tree because they form the terminals of the tree’s branches.

The branches join at boxes labeled with the name of the part of speech formed by the upper branches sprouting from the box. In the diagrams in Figure 3.1, ATE is a verb and QUICKLY is an adverb. A verb followed by a noun is a verb phrase. Such parts of speech are also known as *syntactic categories*. Syntactic categories are used as terms to describe a language, and form what might be called a “language to describe a language”, which is more conveniently termed a *meta-language*.

We must distinguish between these two languages, the language being described and the descriptive language. To do so, we will use capital letters within round boxes for the ordinary language, and we will use small letters within square boxes for the meta-language.

Notice that, at right of Figure 3.1, a verb phrase can consist of a verb followed by a noun phrase whereas at left, a verb phrase is shown as a verb followed by an adverb. This means that a verb phrase may be constructed in more than one way. Such a situation is very common in natural languages and also occurs frequently in programming languages.

Figure 3.2 Examples of syntax diagrams



Syntax diagrams are one of the methods used for describing the form of languages. Figure 3.2 shows some examples of this method. The lines joining the boxes in the diagrams of the figure show the proper sequences of symbols. Valid sequences are those that can be obtained by tracing a path through the diagram. Any path that leads completely through a box, following the arrows on the lines, is syntactically correct. Any path that stops within a box is not correct. The syntax diagrams then describe all possible paths that are correct. In syntax diagrams:

- arrows indicate the possible flow of the definition
- rounded boxes indicate terminal symbols that are used as they appear in the boxes
- square boxes refer to other diagrams.

For example, if we use the syntax diagrams of Figure 3.2 to show that the sentence THE MAN ATE A BANANA is a well formed sentence, we do the following.

1. Start with diagram a, which defines a sentence. To get through this box, we must go through the noun phrase box of diagram b followed by the verb phrase box of diagram d.

2. To get through diagram b, we must pass through the article box of diagram c and then through the noun box in diagram b, which defines all the nouns of English and is thus too big to be shown here.
3. To get through diagram c, there are two possible paths, one passes through the terminal symbol A and the other through the terminal symbol THE. We take the second path.
4. The noun box, if it were shown, would show that MAN is a noun.
5. To get through diagram d, the verb phrase box, we must pass through a verb box, from which we would find that ATE is a verb. Note that the verb box is not defined here for the same reason as for the noun box: it's too big. We then have the choice of passing through the adverb box or the noun phrase box. We choose the second alternative path. This time, when we pass through the article box, we choose the first path and pick the terminal symbol A. When we pass through the noun box for the second time we pick up BANANA.

Since there is no path through the diagrams that allows us to pass through the sequence of terminal symbols DOG, ATE and QUICKLY in that order, the sentence DOG ATE QUICKLY is not a syntactically correct sentence in the language defined by these syntax diagrams.

We have defined languages as sequences of symbols. Since a symbol is not necessarily a word, this definition implies that any sequence of objects can be thought of as a language. The methods of this chapter apply to all linear sequences such as the following “Un-Natural” languages: sequences of dice throws, stops of an elevator, or floats in a parade.

Syntax Diagrams

As we have just seen, syntax diagrams describe the proper form of any sequences in a graphic way. Any path that goes all the way through the diagram from the entry point at the left to emerge at the exit point on the right, following the diagram's lines in the direction of the arrows, corresponds to a proper sequence, while all other paths are improper sequences. Syntax diagrams are sometimes called “railroad layouts”. The major use of syntax diagrams in this book will be to describe the Pascal programming language. Since you will need to be able to read syntax diagrams easily, start by studying the simple examples of syntax diagrams showing “mini-languages” involving money, names, and time that follow.

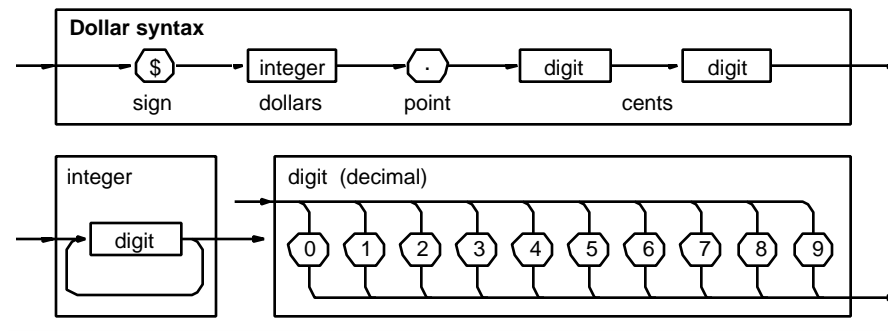
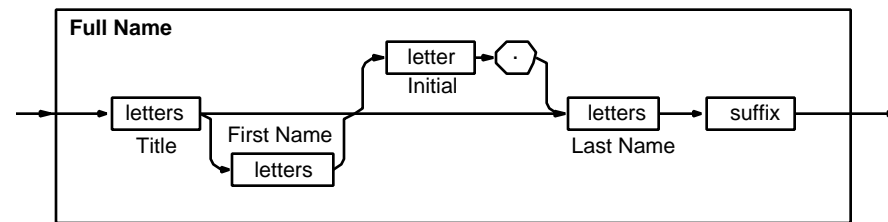
Figure 3.3 Syntax diagram for a dollar amount

Figure 3.3 shows the syntax diagram for dollar amounts. A valid sequence definition shows that the sequence must start with a dollar sign. This sign is followed by an integer, a decimal point and two digits. An integer is shown to consist of any number of digits, but a minimum of one digit. Notice how the integer syntax diagram loops back so that valid sequences can be formed by passing through the digit box any number of times. The dollar syntax diagram also contains some words such as “Dollars” and “Cents” that are not in the boxes. These describe semantics or meaning of the various parts of the sequence. Sequences having the proper dollar syntactic form include the following.

\$1.00 \$2.34 \$5678.99 \$0.12 \$003.45

Check the following sequences to convince yourself that they are all improper:

\$1 2.34 \$.56 \$7.890 \$1,234.56 \$1500
 \$1.2 \$7.89 \$1. 2 12\$ 13¢ \$1 234 567.89

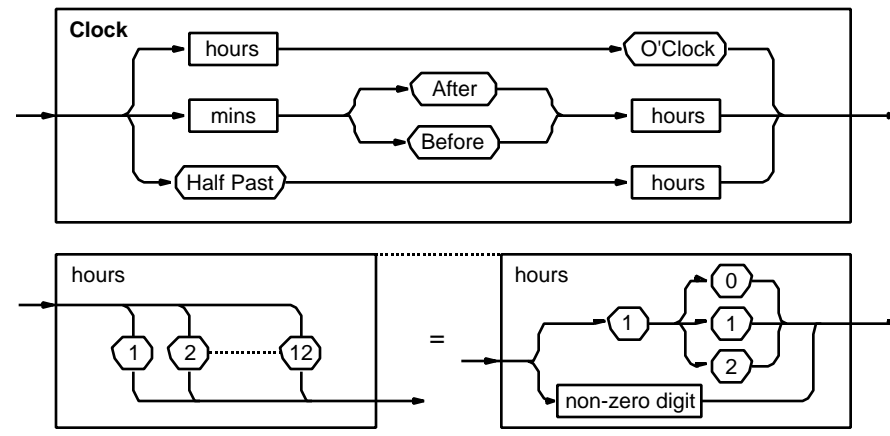
Figure 3.4 Syntax for a full name

A syntax for a full name is shown in Figure 3.4. Assuming that the letters syntax diagram is similar to the integer syntax diagram of Figure 3.3, the prefix or title could be further defined (on another syntax diagram) as being MR., MRS., MS, or some other title like DR., PROF., SIR, etc. Similarly, the suffix could be defined as a degree (BS, MS, Ph.D., MD) or other (JR., II, III, ESQ., etc.) or nothing at all. In this case, blanks are assumed as separators, so are not shown on the syntax diagrams. Notice that a name, according to this definition, need have neither a first name nor an initial. Following the straight path through the center of the diagram avoids both of these. This syntax diagram describes names of the form

MRS. J. JONES or

DR. HARRY J. JONES MD or
MR. JONES JR.

Figure 3.5 Syntax for clock time



The syntax for Clock shown in Figure 3.5, describes a common way of indicating time. With this syntax, some properly formed times are:

6 O'CLOCK 22 AFTER 2 HALF PAST 12 44 BEFORE 4

and some improperly formed times are:

49 O'CLOCK 22 BEFORE 13 HALF PAST 0 2 HALF PAST

Notice that the form of the hours (from 1 to 12) is represented in two ways. One method (at the left), is a complete listing of all 12 hours. Another method (at the right, involving non zero digits) is less exhaustive. Also note that the syntax diagram for minutes has not been shown, can you draw it?

In the rest of the chapter we'll look at the first syntax diagrams for Pascal. Syntax diagrams will help us define the Pascal programming language. However, there is one thing you should always keep in mind:

having a syntactically correct program in a programming language is no guarantee that the algorithm specified is correct.

After all, there are plenty of grammatical sentences in English that are not true. Many of them don't even have a meaning that is connected to any reality that we know, for example:

the spherical wall gargled with the purple bus.

3.3 Pascal Programs

Program Format

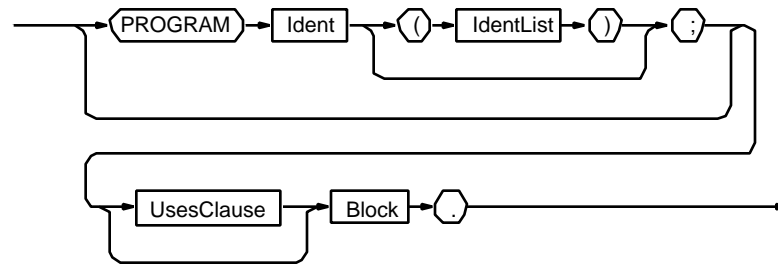
In the seven-step problem solving method that was introduced in the Principles book, step 5 was called Program Coding and Testing. Program Coding can be viewed as the process of converting an algorithm into a programming language. This conversion process from flowchart, flowblock or pseudocode into a programming language is simple compared to the process of creating the algorithm. The actual coding however, becomes very detailed and consequently error prone.

Programs in Pascal usually consist of four parts:

1. Header:
this provides a name that is indicative of the algorithm's purpose
2. Uses Clause:
this specifies the external libraries that will be used by the program
3. Declarations Part:
this describes the various items that are to be used
4. Body:
this specifies the actions to be performed

As we have seen in the preceding chapter, the syntax of the Pascal programming language can be described by syntax diagrams, made of rounded boxes, rectangular boxes and arrows. The rounded boxes contain reserved words of the language, which we shall write in upper-case letters. The rectangular boxes refer to other syntax diagrams. The lines with arrows show all possible proper paths through the diagrams.

Any sequence of symbols that is encountered while following a continuous sequence of arrows through a syntax diagram from entry to exit corresponds to a proper form, while all others are improper. Arrows between the boxes indicate points at which separators, usually one or more blank spaces, are required. Comments or new lines (carriage returns) may also serve as separators. In order to simplify our introduction of the syntax of Pascal, some of our syntax diagrams will differ to a minor degree from those in the reference section, however, their general structure will be consistent.

Figure 3.6 Syntax diagram for a Pascal program**Program**

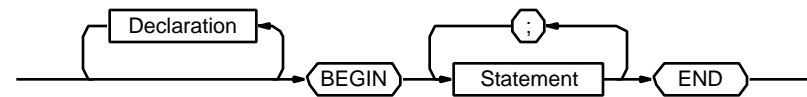
If we approach the description of the program syntax top-down, at the highest level we have programs, or program modules. These are defined by the syntax diagram of figure 3.7 which consists of a sequence of four boxes: a header followed by a Users clause, which may be bypassed, then a Block, and finally a period. Parts 3 and 4 of our previous informal description of a program are actually comprised in the Block. The Header includes the reserved word PROGRAM followed by an identifier for the program name, like:

```
PROGRAM Conversion;
```

Figure 3.7 Syntax diagram for the Uses clause**UsesClause**

The syntax diagram of the Uses clause, is shown in Figure 3.8. It consists of the reserved word **USES** followed by Id-List (a list of identifiers or names of Libraries to be used) and ending with a semi-colon. Here is a possible example of that clause:

```
USES SortLibrary, GraphLib, MyLibrary, IntLib, MoneyLib;
```

Figure 3.8 Syntax diagram for a Block**Block**

The syntax for a Block is shown in Figure 3.9. A Block consists of two main parts. The first of these two parts contains any declarations required by the program, and its syntax will be given later. There may be no declaration part, as in the *First* program in Figure 3.10, but usually there are several declarations. The second part consists of the reserved word **BEGIN**, followed by a sequence of statements separated by a semicolon and terminated by the reserved word **END**. We will define the syntax of statements in the following chapters.

A short but simple and complete Pascal program appears at the left of Figure 3.10, along with its run (or execution) to the right of it. That program is so simple that it has neither a `Uses` clause nor a `Declarations` part.

Figure 3.9 The First Pascal Program

	The results of executing the program
<pre>PROGRAM First; BEGIN Write('Hi '); WriteLn; Write('Bye'); END.</pre>	<pre>Hi Bye</pre>

The name of this simple program is `First`, and this name appears on the Header line which happens to be the first line.

The Body of this program, is that part that is sandwiched between the `BEGIN` and `END`. In this example, the body has three statements which are invocations of Pascal standard *procedures* (or subprograms):

- the first statement invokes `Write` to write out the three characters “Hi “. Note that the space after the two letters “Hi” is a character.
- the second statement calls `WriteLn` (pronounced “Write-Line”) to terminate the current line, which means that the point at which the next character output will appear is at the beginning of the next output line.
- the third statement calls `Write` again to write out the string of three characters “Bye”.

As shown at the right of Figure 3.10, this program simply writes out (displays) the two words “Hi “ and “Bye” on two separate lines. This is not a big computing accomplishment but it does show some details about Pascal.

If there were no `WriteLn` sandwiched between the two `Write` statements then the two words would be output on one line as the six characters,

```
Hi Bye
```

However this one line could also have been written by the single statement:

```
Write('Hi Bye');
```

A Pascal program consists of a sequence of characters separated by blanks to form “words” and the words form groups separated by semicolons to form statements. The program ends with a period. At this point, we encourage you to go back to Chapter 1, and to try and apply the syntax diagrams to the program Simple Pay of Figure 1.4 (the program has no `Uses` clause but has two declarations).

Program Presentation

In Pascal there are no restrictive rules about spacing, which might not be the case with other programming languages. A blank space must separate each word, but anywhere a blank occurs, it could be replaced by more blank spaces or by a carriage return (to begin a new line). The spacing is important for human readability. This first program could be written as one long line such as the following :

```
PROGRAM First;BEGIN Write('Hi ');WriteLn;Write('Bye');END.
```

The Pascal programming language is *case insensitive*, i.e. there is no distinction made between upper-case and lower-case letters. In Pascal, the word `BEGIN` has the same meaning as `begin`, and the same as `Begin`. Hence, this `First` program could also be written as:

```
Program First;Begin write('Hi ');writeln;WRITE('Bye');end.
```

or as:

```
PROGRAM FIRST;BEGIN WRITE('Hi ');WRITELN;WRITE('Bye');END.
```

However in the `Write` statements, the case of the letters that appear within the quotes is important because the letters are printed exactly as they appear.

As extremes we could write everything in lower-case, or everything in upper-case, but to make programs more readable we use both upper and lower case letters, as in natural languages. Names of variables (like names of people) will usually begin with an uppercase letter. Some special words (like `PROGRAM`, `BEGIN`, `END`, `REAL`, `ROUND`) will be entirely in upper case. Furthermore, in this book when we use words that are part of the Pascal language, we will use a typewriter-like font, as `INTEGER` and `REAL`.

A program is a form of expository writing. In the more usual forms of expository writing, technical reports, equipment manuals, etc. a great deal of the readability depends on the *style* of writing. Style is the personality and character of writing, the mode of expressing thought in language. Its chief elements are the sequence and organization of paragraphs, sentence structure and choice of words. The same is true of programming; there, style is the quality of a program in which good choices have been made in spacing, in naming, in structure, and other ways. These aspects will be considered later in this chapter and in the remaining chapters.

Reserved words are those words that have special meaning in the language (such as `BEGIN` and `END`) and cannot be used in any other way. In this book, we shall show them in upper case. Pascal does not require them to be written in upper case but that is part of the style of programming that we shall adopt. There are 35 reserved words in Pascal; they are listed on the cover of this book for quick reference. The most common ones used in the next few chapters are the following:

```
PROGRAM, BEGIN, END, CONST, VAR, IF, THEN, ELSE,  
WHILE, DO, AND, OR, NOT, DIV, MOD, FOR, DOWNT.
```

In Pascal, standard identifiers are names that have been pre-defined (such as `INTEGER`, `REAL`, `WriteLn`); here they will often be written entirely in upper-case, but that is not necessary. The standard identifiers are also listed on the cover of this book; some of the common ones follow:

```
SIN, COS, TRUE, FALSE, ROUND, TRUNC, Read, Write,  
PRED, SUCC, ABS, ODD, CHAR, BOOLEAN, INPUT, OUTPUT.
```

The reserved words and standard identifiers should not be used by programmers as names for anything else; there are plenty of other names.

More Pascal Programming: Data and Actions

A single program example, such as `First`, could be misleading because it cannot show different ways of doing things. Also, the first example must of necessity be simple. Let's now look at another example program called `Second` and shown in Figure 3.11, to review some of the previous concepts and introduce some new ones.

As we showed earlier, programs have a form consisting of a header, declarations, and a body. The `First` program had no declarations, this `Second` one declares a constant called `Year` (with value 2000) and a variable called `Age`.

Figure 3.10 The Second Pascal Program

```
PROGRAM Second;  
(* Determines the birth year *)  
CONST  
  Year = 2000;  
VAR  
  Age: INTEGER;  
BEGIN  
  Write('How old are you? ');  
  Read(Age);  
  Write('You were born around ');  
  Write((Year - Age): 4);  
END.
```

A description of the program's purpose should follow the header, or first line. Here, the description is a single line that states the function of the program. Other lines here could indicate the programmer name, the date, the inputs, the outputs and other information. Such additional information will be considered later.

Comments are textual information intended for reading by programmers. They are enclosed within special brackets `{ and }` or `(* and *)`, and are ignored by the computer. Examples are:

```
{ This is a comment }
```

and

```
(* This also is a comment *)
```

When the second type of bracket is used, there must be no spaces between the asterisk and the parenthesis. The program description on the second line of Figure 3.11 is a comment, but comments are not limited to entire lines; they can stretch over many lines or they can be only part of a line. Comments can be inserted into the program anywhere there is a space. In Pascal, comments cannot be nested within one another.

Declarations are specifications of the data items (and, as we shall see later, procedures and functions) to be used in the program. Items declared in `Second` include a constant

```
CONST
    Year = 2000;
```

whose value in this program does not change. If the value of `Year` must change, the program must be modified. Another item that is declared in `Second` is a variable

```
VAR
    Age: INTEGER;
```

which is declared to be of type `INTEGER`.

The execution of this program follows. The user's part of the dialog consists of entering the value of 20 in answer to the question, as shown in bold type.

```
How old are you? 20
You were born around 1980
```

First, the `Write` statement displays the prompt `How old are you?`. Then the `Read(Age)` statement waits for an integer value to be input, followed by the user pressing the return key. Another `Write` statement displays the message `You were born around`, and the final `Write` statement

```
Write((Year - Age): 4);
```

displays the resulting number (which is the year 2000 minus the age of 20, which is 1980). The number 4 in this last `Write` statement specifies that 4 spaces should be allotted to display the resulting year (of 1980).

This `Second` program may not be very useful but it does show important aspects of most programs; it has a prompt, an input, some (trivial) computation, and an output.

Data Items

The objects of the real world include things such as days, sheep, and money. In the abstract world as modeled by a program in a computer, the analogs of these objects are the data that are manipulated by the program. These data can be represented as boxes of a certain size and shape (called the *data type*). The boxes are given names or labels (called *identifiers*) so they can be referred to, and have contents (called *values*). The boxes are called *variables* or *constants* depending on whether or not the program can change their values.

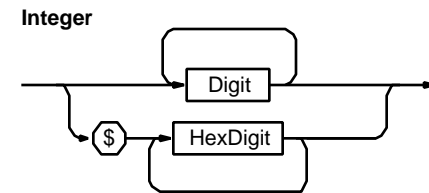
The type of an item of data is its description, specifying the range or set of values that the item may have, and also the operations that can be performed on it. Every item of data in Pascal has a type that must be specified or declared by the programmer. This requirement, known as *strong typing* is considered to be very important because it allows Pascal to check for errors made by the programmer. Type checking prevents programmers from adding “apples to oranges”.

Some of the most common items of data are numbers. They are used in different ways, usually to count or to measure. Measurements often involve values that are not a whole number of units and they are usually represented by numbers that have a decimal point. Counting involves whole integral numbers.

Numbers in Pascal are one of two types: `INTEGER` or `REAL`.

The values of data items of type `INTEGER` are whole integral numbers, either positive or negative, such as 7, +11, -40, 365, 2001, 5280, etc. Figure 3.12 gives the Pascal syntax of such integer numbers. An integer is made of an optional sign followed by a digit followed by any number of digits, i.e. there is always at least one digit. A decimal digit is one of the ten values 0 to 9. Examples of incorrectly formed integers are -3.4 and 5,280. Other improper integers are O (capital letter o), and l (lower case letter L); these can be easily confused.

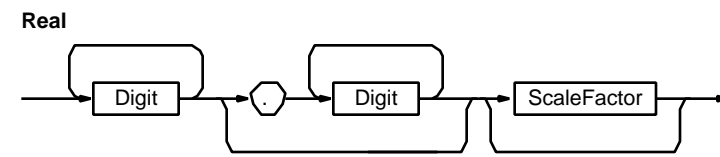
Figure 3.11 Syntax for `INTEGER` values



In the “abstract” world, integers correspond to discrete whole numbers on a “number line” that includes negative values. In the “real” world these numbers correspond to entities, such as financial worth, which can be positive or go negative. In the abstract world, counting can continue to any integer value; there is no upper limit. In the “computer” world there is always both an upper and a lower limit. The size of this limit depends on the computer. On the Macintosh for instance, this upper limit is 32,767 (or $2^{15}-1$) and the lower limit is -32,768 (or -2^{15}). On all computers the largest possible integer is named `MAXINT`.

Data items of type `REAL` have values that are numbers with decimal points. Figure 3.13 shows the syntax for real numbers in Pascal.

Figure 3.12 Syntax for `REAL` values



Real numbers always contain a decimal point. Examples of syntactically correct real numbers include 0.0, 0.5, 7.0, 3.14159, +98.6, -459.99. Notice that the sequence 5. and .5 are not valid `REAL` numbers because there is no path through this syntax diagram that can lead to these sequences. They should be written as 5.0 and 0.5; the decimal point needs digits on its left and right side.

As an alternative notation for real numbers, the exponential or scientific notation can be used for `REAL` numbers. This form is useful for very large or small values. A number such as 12300.0 can be expressed as 1.23E4. Similarly, the number 0.000123 is equivalent to 1.23E-4, which is the Pascal equivalent of the scientific notation form 1.23×10^{-4} . The number after the E (the exponent) indicates how many places the decimal point is to be moved. Its sign indicates whether it moves left (if negative) or moves right otherwise. The exponent is usually limited to a standard range.

Items such as diameters, weights and rainfall are usually measured precisely and the value given includes a decimal point. The constant `PI` (3.14159..) is a commonly used real value. However, you should know that numbers of type `REAL` cannot always be represented precisely on computers. For example, the decimal 0.20 when converted to binary is a repeating number (0.001100110011...0011...) which must ultimately be chopped to some finite length, resulting in some error.

Actions: Arithmetic Operations

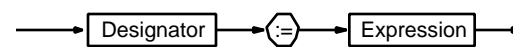
Pascal has a wide range of operations that can be performed on data. We'll consider here some of the simpler actions as an application of the formulas used to define algorithms in Chapter 3 of the Principles book. The actions are mainly arithmetic operations on numbers of the `REAL` and `INTEGER` type.

An assignment is the action of copying a value into a variable of the same type, corresponding to the pseudocode operation Set as in:

Set Pay to Hours \times Rate

In Pascal, the assignment operator is a combination of two symbols: a colon immediately followed by an equal sign—neither a blank nor a comment is permitted between the colon and the equal sign. An assignment statement has the form given by the syntax diagram of Figure 3.14.

Figure 3.13 Assignment statement



In that syntax diagram, the designator stands in particular for any variable name, and the expression may be a constant, a variable or a proper combination of these. Some simple assignment statements follow.

```

Count    := 0;
Rate     := MaxRate;
  
```

```

cost      := Price + Tax;
FORCE     := Mass * Acceleration;
Area      := Pi * Radius * Radius;
Charge    := 3 * Adults + 2 * Babies;

```

The arithmetic operations of addition, subtraction and multiplication are very similar for `INTEGER` and `REAL` values. However, the division operation differs between `REALS` and `INTEGERS`. Let's take some of the formula algorithms from Chapter 3 of the Principles book and change them into assignment statements.

```

Celsius := (5 / 9) * (Fahrenheit - 32);
Fahrenheit := (9 / 5) * Celsius + 32;

```

The division of two `REAL` values `X` and `Y` is indicated by `X/Y`, and yields a resulting `REAL` value. The division of two integers `I` and `J` is indicated by `I DIV J`, and it yields another `INTEGER` by truncating the result to the next lower `INTEGER` value. For example,

5/9 yields 0.55555, but 5 DIV 9 yields 0.

The `MOD` operator is a useful complement to the `DIV` operator on `INTEGERS`. The division of an `INTEGER` Numerator by Divisor yielding Quotient and Remainder can be viewed as a combination of `DIV` and `MOD` as follows:

```

Quotient := Numerator DIV Divisor;
Remainder := Numerator MOD Divisor;

```

For example, 5 MOD 3 is 2 and 3 MOD 5 is 3. Some interesting and useful facts about `MOD` are:

`X MOD Y` yields 0 when `X` divides `Y` evenly, for example, 8 MOD 4 is 0, and 1984 MOD 4 is 0

`X MOD 2` is 0 if `X` is even and 1 if `X` is odd, for example, 12 MOD 2 is 0, and 19683 MOD 2 is 1

`X MOD 10` is the rightmost digit of `X`, for example, 13 MOD 10 is 3, and 12345 MOD 10 is 5

Expressions involving a mixture of types, `REAL` and `INTEGER`, may appear in assignment statements. In such cases whenever one of the operands is `REAL` then the result is `REAL`. For example, in the above temperature conversion formula, the `REAL` division results in the entire expression evaluating to a `REAL` result.

In Pascal, arithmetic expressions are never ambiguous, that is to say there is only one way of evaluating an expression. For instance, to anybody the expression `A+B*C` could mean either `(A+ (B*C))` or `((A+B) * C)`. Not so in Pascal where it can only be interpreted as `(A+ (B*C))`. To achieve this, the order of evaluation of expressions is specified by an operator precedence (priority) table, like the following:

Operation	Precedence
+ -	low
* / DIV MOD	middle
unary -	high

The highest priority operations are done first, while the lowest one are done last. If operators have the same priority, then the convention is that the leftmost operator is applied first. So the expression $(10 - 1 + 2)$ will give 11 and not 7. Parentheses are evaluated first, so when in doubt use parentheses!

In a Pascal assignment statement, the identifier at the left and the expression at the right must be of the same type. There is one notable exception to this rule: if the variable is of type `REAL` and the expression is of type `INTEGER`, the value of the expression is converted to the type `REAL` before the assignment takes place. For example, if `Celsius` were declared to be `INTEGER` then the assignment statement above would not be correct. In that case, the `REAL` value of the expression on the right would have to be converted by truncation to an `INTEGER` value, by a call to the `ROUND` function, for instance.

```
IntegerCelsius := ROUND((5 / 9) * (Fahrenheit - 32));
```

Thus, the value of `Celsius` obtained from a `Fahrenheit` value of 65 would be

$$(5/9)*(65-32) = (0.5555555)*33 = 18.333333 = 18$$

Pascal does not have an exponentiation operator for taking a number to some power. One alternative is to create a subprogram to do this; another is to avoid the need for exponentiation by factoring, as in the following formula algorithm:

```
Time := Seconds + 60*Minutes + 60*60*Hours + 24*60*60*Days
Time := Seconds + 60*(Minutes + 60*(Hours + 24*Days))
```

Notice that both expressions require no exponentiation, and that the second expression also requires half as many multiplications and so is more efficient. Similarly, the following formula from Chapter 3 of the Principles book:

$$\sin(X) = X - X^3/3! + X^5/5! - X^7/7! + \dots$$

must be written in Pascal in the following manner.

```
Sin := X - X*X*X/(3*2) + X*X*X*X*X/(5*4*3*2) - ...
```

but the computation can be done in a more efficient manner, provided we know more about loops.

3.4 More Example Programs

As an advance view of what is coming, we'll give here three program example written from algorithms presented in Chapter 3 of the Principles book. Again, we have to anticipate somewhat on a number of details that will be presented in the coming chapters, but as you are familiar with the algorithms, the programs should not be too difficult to understand. They will give you more to think about, and help you apply what has been presented in this book so far.

The first program example is a temperature conversion program that was written from the algorithm drawn from Figure 3.20 of the Principles book. The complete program is shown in Figure 3.15.

Figure 3.14 A Temperature conversion Program

```
PROGRAM Convert;
(* Convert Celsius temperature to Fahrenheit *)
VAR
    Celsius, Fahrenheit: REAL;
BEGIN
    Write('Enter Degrees C: ');
    Read(Celsius);
    Fahrenheit := (9.0 / 5.0) * Celsius + 32.0;
    Write('Fahrenheit is ');
    Write(ROUND(Fahrenheit));
END.
```

The program is very simple, has no Uses clause, declares two Real variables, asks for a Celsius temperature, inputs it, applies the conversion formula, and displays the resulting Fahrenheit temperature. Note that the input of the Celsius temperature is done by calling standard procedure `Read`. Also note that the conversion formula was written with Real variables and Real constants, as we want to be sure the computation is done with Real numbers. The second line of the program is a comment giving the program's objectives. The display of the resulting temperature is done through a call to `Write`, as usual, but the value is first transformed into an integer value by a call to standard function `ROUND`.

A typical run of this program is shown below.

```
Enter Degrees C: 100.0
Fahrenheit is 212
```

The second program example is a little longer, and is the coding of the algorithm shown in Figure 3.32 of the Principles book. The complete ChangeMaker program is shown in Figure 3.16.

Figure 3.15 The ChangeMaker Pascal Program

```
PROGRAM ChangeMaker;
(* Make change for a dollar *)
VAR
    Cost:          INTEGER;
    Remainder:     INTEGER;
    Quarters:      INTEGER;
    Nickels:       INTEGER;
    Pennies:       INTEGER;
    Dimes:         INTEGER;
BEGIN
    (* Input the Cost *)
    Write('Enter the cost in cents: ');
    Read(Cost);

    (* Make the Change *)
    Remainder := 100 - Cost;
```

```

Quarters  := Remainder DIV 25;
Remainder := Remainder MOD 25;
Dimes     := Remainder DIV 10;
Remainder := Remainder MOD 10;
Nickels   := Remainder DIV 5;
Remainder := Remainder MOD 5;
Pennies   := Remainder;

(* Output the coin count *)
Writeln('The change is ');
WriteLn(Quarters: 2, ' quarters');
WriteLn(Dimes: 2, ' dimes');
WriteLn(Nickels: 2, ' nickels');
WriteLn(Pennies: 2, ' pennies');
END. (* ChangeMaker *)

```

The line following the program header is a comment stating the program's purpose. In this program there are more variables declared as the computations produce more results. All the variables are declared to be integers. The body of the program (between BEGIN and END) asks for a Cost less than a dollar, and computes the change, i.e. the number of quarters, dimes, nickels and pennies. It then displays the results. The computations are done by using the MOD and DIV arithmetic operations. We give below an example run of this program.

```

Enter the cost in cents: 32
The change is
  2 quarters
  1 dimes
  1 nickels
  3 pennies

```

The third and last program example is somewhat similar to the last example we saw in Chapter 2. We obtained it by coding the algorithm found in Figure 3.46 of the Principles book. The complete program is shown in Figure 3.17.

Figure 3.16 A Payroll Pascal Program

```

PROGRAM Payroll;
{ Repetitive pay algorithm }

CONST WeekHours = 7 * 24; (* Hours in a week *)

VAR Hours, Rate, Pay: INTEGER;

BEGIN
  Write('Give number of hours and rate ');
  Read(Hours, Rate);
  WHILE Hours >= 0 DO BEGIN
    IF (Hours < 0) OR (Hours > WeekHours) THEN
      Writeln('Error in hours')

```

```

ELSE BEGIN
  IF Hours <= 60 THEN
    IF Hours <= 40 THEN
      Pay := Hours * Rate
    ELSE
      Pay := ROUND(40*Rate +
1.5*Rate*(Hours-40))
    ELSE
      Pay := 70 * Rate + 2 * Rate * (Hours -
60);
    Writeln('The pay is ', Pay);
  END; {IF}
  Write('Give number of hours and rate ');
  Read(Hours, Rate);
END; {WHILE}
END.

```

This program is longer than the preceding examples, as the algorithm has more different cases to deal with. The line following the program header is a comment explaining what the program does. The program declares a constant and three integer variables. Note that the constant declaration is followed by a comment. The program body (between BEGIN and END) has been written with indentations to help understand the structure of the algorithm. The program statements between the line: “WHILE Hours >= 0 DO BEGIN” and the line “END; {WHILE}” will be repeatedly executed, as long as the number of hours read is positive. The indentations are such that these two lines bracket the statements to be repeated. The rest of the program involves nested selections whose details will be covered soon. The pay computations assign their result to variable Pay, which is then displayed. Note that the second assignment statement to Pay:

```
Pay := ROUND(40*Rate + 1.5*Rate*(Hours-40))
```

is slightly different from the two others. This is because of the Real constant 1.5 which causes the expression to be evaluated as a Real value. To store this Real value in an integer variable it is necessary to convert it using the ROUND standard function, which we just saw in the Convert program above.

Here is an example run of this program.

```

Give number of hours and rate 40 10
The pay is 400
Give number of hours and rate 0 15
The pay is 0
Give number of hours and rate 60 15
The pay is 1050
Give number of hours and rate 70 15
The pay is 1350
Give number of hours and rate -1 0

```

3.5 Chapter 3 Review

This chapter introduced natural and programming languages, their syntax and their semantics. The verification of the syntax of a sentence was done based on two very different forms of representation. A "stream" of text is usually linear and one-dimensional, while syntax trees used for parsing a sentence are two dimensional. Two dimensional representations are convenient for humans, while computers can deal with one dimension.

Languages, whether natural or for programming, involve sequences of symbols, and have a grammar or syntax that can be described in two dimensions. This syntax can be described by means of syntax diagrams. Syntax diagrams are used to define the Pascal programming language syntax. The semantics of the language require that explanations be given to accompany the syntax diagrams.

Some complete Pascal programs were presented even if some of the statements anticipated their presentation in the next chapters. These examples are useful to enter slowly and painlessly the world of Pascal.

3.6 Chapter 3 Problems

1. Simple Syntax

Create syntax diagrams describing the following forms:

- a. Phone Numbers (with area codes, prefix, suffix)
- b. License Plates (with three letters followed by three digits, and vice versa)
- c. Dates (such as 84/2/28 or II-28-84 or Feb. 28 1984)
- d. Military time (such as 0800 or 1547)
- e. Dewey Decimal number system (as used in public libraries)

2. Identifier

An identifier, or name, in the standard Fortran Language consists of one to six symbols, the first being a letter and the others being either letters or digits. Create a syntax diagram describing such identifiers, showing explicitly all possible paths.

3. Elevator

Create syntax diagrams describing the behavior of an elevator which travels between four floors. Typical sequences of travels (from the first to the fourth floor) are:

1,2,3,4 1,2,3,2,3,4 1,2,1,2,3,2,1,2,3,2,3,4 1,2,1,2,1,2,1,2,3,2,1,2,3,4

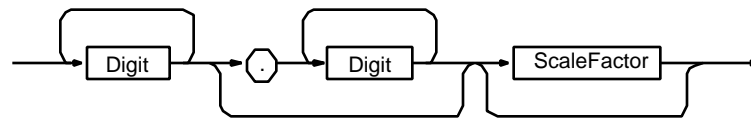
4. Roll Your Own Syntax

Find an example from everyday life which would have a structure that could be described by syntax diagrams. Examples could involve: addresses, ZIP codes, part numbers, card games, sports, trains, parades, dice, Roman numbers, combination locks, or Robert's Rules of Order.

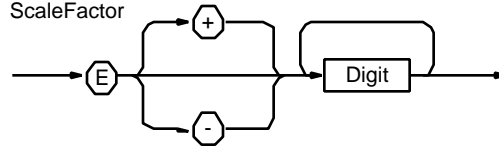
5. Unreal

Using the following syntax diagram that defines REAL number constants for Pascal:

Real



ScaleFactor



determine which of the following are in that proper form. Express those in proper form in the alternative REAL notation, with a decimal point only and without a ScaleFactor. This illustrates the great number of wrong ways that numbers may be represented, and the simplicity of syntax diagrams to define the right form.

- | | | |
|-------------|-------------|------------|
| a. 5.28E3 | b. 1.6E-05 | c. 1.35e5 |
| d. E5 | e. 1E2 | f. .5E |
| g. 2E3.4 | h. -.5E3 | i. 10E10 |
| j. 2.55EE23 | k. 3.4.1 | l. 4.E5 |
| m. 5.95E0 | n. .5 | o. 0E-3 |
| p. 2.-3E4 | q. -.1E1 | r. 3.0E4 |
| s. 3.0E4 | t. 5,280 | u. 5280 |
| v. -3.4E-6 | w. 13.4E.4 | x. 5.5E5.0 |
| y. AE3 | z. 5.5E-1E2 | |

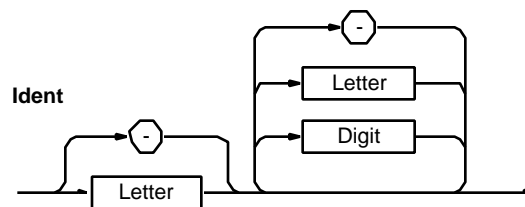
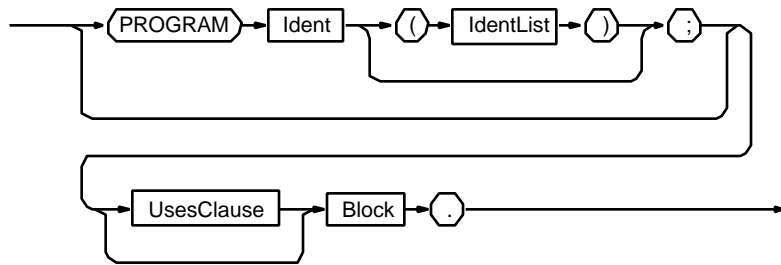
6. Integers

Create syntax diagrams to define integers in Pascal if they consist of either any number of digits or any number of hexadecimal digits preceded by the symbol "\$"

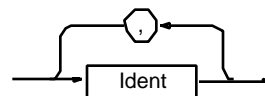
7. Top Syntax

Using the syntax diagrams at the top levels of Pascal programs:

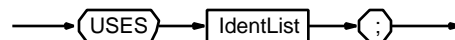
Program



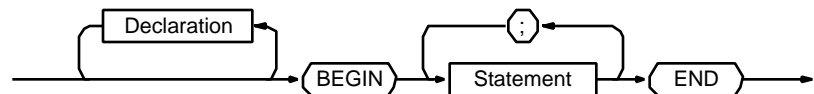
IdentList



UsesClause



Block



Answer whether the following are syntactically correct (without being concerned as to why anyone would want to do this).

- Is it possible for a program to have no declarations?
- Is it possible for a program to have no UsesClause?

- c. Is it possible for a program to have no statements in the body?
- d. Is it possible for a program to have no word `BEGIN` in it?
- e. Is it possible for a program to have no semicolons?
- f. Is it possible for a program to consist of only semicolons?

8. Exterminate

The following program is loaded with syntactic errors (bugs), as well as others. Every line has at least one error. Count the number of syntactic errors, and rewrite the program properly. Actually, computers (compilers) are good at finding such errors, but they don't rewrite the programs. (Hint: one easy way of doing this is to enter this program in your system and have the compiler do the job! But be careful! There are some errors that might confuse your compiler!)

```
PASCAL Sales;
(* Infested with bugs. Find them. * )

CONSTANT TaxRate := 06.5%
VAR Cost, total: real;

BEGEN
  WriteLN("Inputt cost");
  ReadRealcost );
  tax := TaxRate x cost;
  total = Cost + TaxRate
  (* 'Outputt the total' *)
  Write( total );
END
```

9. Least

What is the shortest possible program that can be written in Pascal? It doesn't need to do anything useful; it just needs to run without error.

10. Evaluate

Compute the following, according to the operator precedence table given in this chapter.

- | | |
|---|---|
| a. $9 - 8 - 7$ | b. $10/2/5$ |
| c. $(1 + 2)/3*4$ | d. $7 \text{ MOD } (11 \text{ MOD } 7)$ |
| e. $7 \text{ MOD } (11 \text{ DIV } 7)$ | f. $(1 \text{ MOD } 2) \text{ DIV } (3 \text{ MOD } 4)$ |

11. Represent

Write the following real numbers in exponential notation:

- a. Speed of light in a vacuum (in meters per second)

300000000.0

- b. Charge on an electron (Coulombs)

0.0000000000000000016

- c. Mass of an electron (kilograms)

0.000000000000000000000000000000911

12. Express

Write the following formulas as assignments in Pascal:

a. $E = mc^2$

b. $F = \frac{6.67 \cdot 10^{-11} m_1 m_2}{r^2}$

c. $x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$

3.7 Chapter 3 Programming Project**Getting Acquainted**

Your first programming project is to become familiar with your version of Pascal; all implementations differ somewhat. To concentrate on these details, you can start with the following simple program. It is one of the smallest that does something (displays "Hi!").

```
PROGRAM Hi ;

BEGIN
    WriteLn('Hi!');
END.
```

1. Study your editor and then enter the above program. Make some errors and correct them, both immediately after you made them, and also much later.
2. Run the above program, a process usually involving compiling and linking followed by executing.

3. Create a number of errors in the program (misspell words, delete semicolons or words, insert words) and observe the error messages.
4. Extend the program to write several more lines. Insert comments in various places. Insert gaps between lines in the output.
5. Save the program, not because it's good, but just to know how to save programs. Then retrieve the program and modify it again.

Chapter 4 Data and Actions

This chapter continues with the introduction to the Pascal programming language that was started in the last two chapters. Its main concerns are the basic components of the language with some emphasis on the way in which the language is written (i.e. the syntax or grammar) but also on the precise meaning (i.e. the semantics) of what is written.

Chapter Overview

4.1	Preview.....	76
4.2	Programming: Data and Actions.....	76
	Declarations: Syntax Diagrams from the Bottom.....	76
	Simple Input and Output in Pascal.....	80
4.3	More Programs: A Top View of Pascal.....	83
4.4	Programming Style.....	85
4.5	Layout of Programs.....	86
4.6	More Programs: Continued.....	89
	Actions: Pre-Defined Standard Functions in Pascal.....	89
	Libraries: Using Units in Pascal.....	92
4.7	A Foretaste of Procedures.....	94
4.8	Chapter 4 Review.....	96
4.9	Chapter 4 Problems.....	96
4.10	Chapter 4 Programming Projects.....	97
	Generate Conversion Tables.....	97
	Observing Errors.....	97
	Demilitarize Time.....	98
	TipTable.....	99
	STT: Sales Tax Table.....	100
	SSP: Simple Side Plot.....	101

4.1 Preview

The main topics of this chapter are the data and the actions of Pascal programs. We'll start with the declarations of the data elements, and at first will only use the simplest kind of data, numbers. The two main types for numbers, Integer and Real, were defined in Chapter 3. Declarations will be defined using syntax diagrams which were introduced in Chapter 3.

Complete programs, involving actions on data, are also presented here. The readability of programs—very important for understandability—depends on their layout. This topic, as well as some aspects of programming style, are introduced at this early stage, mainly to instill good habits from the beginning, by showing well designed program examples.

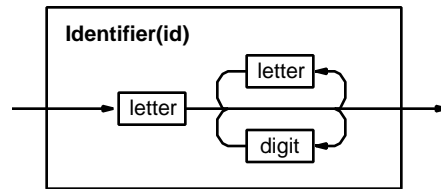
Like most chapters in this book, the goal of this chapter is to introduce some basic concepts of the Pascal programming language with many examples. You should gain familiarity with the form of programs written in the language, and a great many details concerning the language. In the course of this chapter, you will be reading many short programs, and segments of programs. After completing the chapter, you should be able to recognize the proper form (syntax) of many parts of the language. You should also be able to write some very simple programs. Most of these programs will involve only a series of actions, but some of them will also involve simple forms such as loops, decisions or sub-programs. The following chapters will present more formally what has been introduced here.

4.2 Programming: Data and Actions

Declarations: Syntax Diagrams from the Bottom

A Pascal program is written using the characters of your computer keyboard, including the 26 upper case letters, the 26 lower case letters, the ten decimal digits, and a number of other symbols (punctuation marks, brackets, etc.) These characters are combined to form numerical values, identifiers, keywords, operations, and ultimately programs. We can use syntax diagrams to describe all of these and we will now show the syntax diagrams for some of the lower level components of a Pascal program.

In Pascal, the symbolic names of variables, constants, programs, subprograms, etc., are called identifiers. The syntax diagram for identifiers is shown in Figure 4.1.

Figure 4.1 Syntax diagram for an identifier

You can see that an identifier consists of a letter followed by any number of letters or digits. However, you should note that some Pascal system keep only the first 8 (or 16 or 32) characters of the identifier. In that case `Employee1` and `Employee2` would appear to be different to you, but would be the same for the system. Reserved words, found within rounded boxes on syntax diagrams (such as `BEGIN`, `END`, `FROM`, `IF`) must not be used as identifiers. A list of reserved words is given on the Reference page at the end of this book. Examples of identifiers are shown in Figure 4.2.

Figure 4.2 Proper and improper Pascal identifiers

Examples of Identifiers

Proper	Improper
A	7
Age	2*Pi
R2D2	Pi*2
Over21	Over 21
MaxAge	Max Age
temporary	WHILE
BalanceOfPayments	

In Pascal, declarations define the meaning of an identifier. For example, a declaration defines the name of data items, indicating their type or set of possible values. Figure 4.3 shows a simplified form of the syntax diagram for declarations where a declaration consists of a choice between four different kinds of declaration. Usually in a program, the declarations are grouped in the order shown, first the constants, then the types, followed by variables and procedures. Some examples of declarations for constants and variables are shown in Figure 4.4.

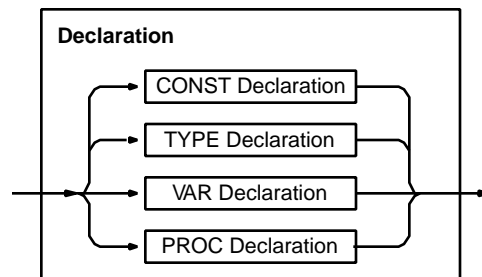
Figure 4.3 Simplified syntax diagram for a declaration

Figure 4.4 Examples of declarations of constants and variables

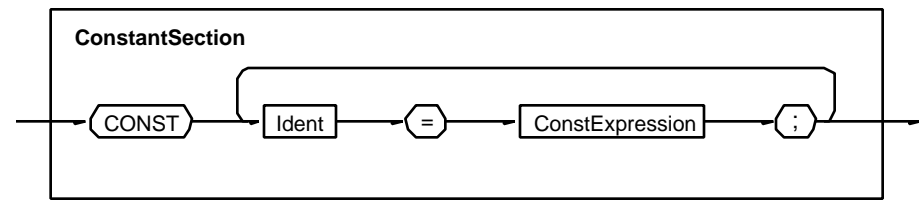
```
Examples of Declarations

CONST
    Pi      = 3.141592653589;
    Year    = 2001;
    Period  = '.';
    T       = TRUE;
    Prompt  = 'Enter a value';
    TwoPi   = 2.0*Pi;
    (* Physical constants*)
    Avogadro = 6.023E23;
    Planck   = 6.63E-34;
    Coulomb  = 8.99E9;
VAR
    Age     : INTEGER;
    Radius  : REAL;
    Grade   : CHAR;
    Male    : BOOLEAN;
    ZipCode : INTEGER;
    count   : INTEGER;
    ISBN    : INTEGER;
```

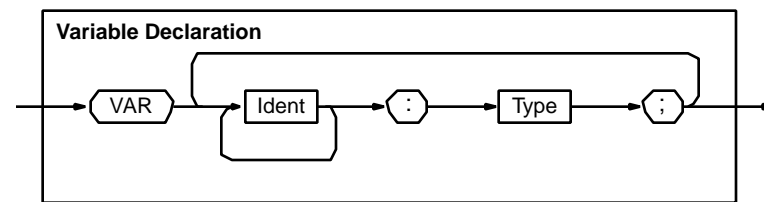
There is often a need in programs for values that remain constant during the program execution. Such constants can be used directly (as 2000) within programs, or they can be given a symbolic name through a declaration, and referred to by that name. Named constants are preferable because they make it easier to modify programs. For example, if we use a symbolic constant for the year, each time the year must be changed, only one line of the program needs to be changed. On the other hand if we use the constant directly, it is just too boring and thus error prone to change many occurrences of 2000 to 2001, and sometimes the wrong 2000 (a street address perhaps, or a salary) could get changed also. Using named constants also improves the readability of the program—it is easier to understand `Avogadro` than `6.023E23`.

The syntax diagram for the declaration of a named constant is shown in Figure 4.5. The declaration consists of the word `CONST` followed by any number of instances of the sequence: an identifier, an equal sign, a constant expression and a semicolon. We've seen the syntax diagram for the declaration of identifiers in Figure 4.1, and the diagram for constant expression is in Appendix X. The constant expression following the equal sign may involve constants and a single simple arithmetic operation such as:

```
HoursInAWeek = 7*24;
TwoPi = 2.0*Pi;
```

Figure 4.5 Syntax diagram for a declaration of a named constant

Of course we humans could compute these constants, but that is tedious, and could be error prone. Besides, the number 168 hides the two parts (7 days and 24 hours) from which it is constructed. Notice that `Pi` must be declared before `TwoPi`.

Figure 4.6 Syntax diagram for the declaration of variables

The variables used in a program must be declared as shown in the syntax diagram of figure 4.6. The declaration of variables begins with the word `VAR` followed by a list of identifiers separated by commas (which could contain only one identifier), followed by a colon followed by a type (`INTEGER`, `REAL`, etc.) There could be any number of these list of identifiers-colon-type combinations, each terminated by a semicolon. Notice the similarity between the declarations of Constants and Variables. To improve readability identifiers could be described briefly by a comment, as follows.

```
CONST
MaxAge = 150; (* max age of humans, assumed to be 150
years *)
VAR
Angle: REAL; (* angle of shaft to horizontal, given in
degrees *)
```

The choice of types, names and values for data items is an important responsibility of programmers, and that choice may not always be obvious. For example, money may be viewed as an integer (20 cents) or as a real value (0.20 dollars). If it is used in a change-making program or an accounting program it must be chosen to be an `INTEGER`, because integer operations give exact results. If it is used in a loan program (involving large sums and complicated formulas) then it could be chosen to be a `REAL` number. The counting of sheep or of a human population should be done with the `INTEGER` type, although the upper limit for `INTEGER` might be an inconvenience (there are five billion people on earth!)

Data items are referred to by names, which are identifiers, created by the programmer. The data names should be chosen so that they clearly describe the reality that is being modeled. The names should neither be too long nor too

short. For example, to count the number of months paid on a bank loan we could use the symbolic name `M` or `Mn` or `MonthsPaidOnLoan`, but some intermediate length name such as `Months` may be preferable. If this is the case, it will be useful to insert a comment with the declaration, indicating that `Months` is the number of months paid toward the loan. As we already mentioned, comments are pieces of text included in a program to help the reader understand it; they do not change the way in which the program works. The data type for the variable `Months` should also be `INTEGER`, because its value comes from counting.

Proper selection of names and types will pay off ultimately in the ease of reading and modifying of programs. More details involving Pascal declarations will be considered shortly. Other data types will be introduced later.

Simple Input and Output in Pascal

As you've already seen in the program examples of the two preceding chapters, the input and output of values using `Read` and `Write` can be done easily. The input is usually always done in a simple manner. The output can also be done as simply as you wish, but can also be formatted as elaborately as you wish. Here we will introduce and review the basic concepts of input/output in Pascal, using mostly numerical values.

The input of values is done by a `Read` statement (actually a call to the `Read` standard Pascal procedure) which has the simple form:

```
Read( InputList );
```

where `InputList` consists of a number of variable names (not constants or expressions) that are separated by commas. Some examples are:

```
Read(Size);
Read(Rate, Hours);
Read(First, Second, Third);
Read(A, B, C, D, E, F, G, H);
```

Execution of such a `Read` statement causes data values to be accepted from the keyboard and assigned to the variables of the `InputList`. The input values should be separated by blank spaces, not commas. The variables can be of various types, but the input values should appear in the same order as the variables in the `InputList`, as the values are matched with the variables in order.

The simplest kind of output involving numbers and strings of characters enclosed in quotes is done by `Write` statements having either of the following forms:

```
Write( OutputList ); or WriteLn( OutputList );
```

where `OutputList` consists of a number of expressions or quotations separated by commas. The expressions may be constants, variables, or formulas. The quotations are strings of characters enclosed between single quotes, not double quotes. Here are a few examples:

```
Write(Size);
Write('Hello ');
```



```

Write('X = ', X);
WriteLn(A, B, C, D);
WriteLn((9/5)*C + 32);
WriteLn('You are about ', Age, ' years old.');
```

The execution of a `Write` statement produces a display on the screen. Quotations are displayed exactly as shown; expressions are first evaluated and then their values are displayed. The execution of `WriteLn` has the same effect as `Write` but after displaying the information on the screen, the cursor advances to the next line. Also, `WriteLn` can be used with no `OutputList` and no parentheses to produce a blank line, as

```
WriteLn;
```

In the above example with simple `Write` statements, the format of the output is very simple and somewhat crude. For instance, the field width, that is, the space provided for writing a value, is fixed and is often much larger than required, with the result that ugly gaps appear in the displayed output. Also, `REAL` values are written in scientific notation with exponents, which is not natural for business and other applications. Such simple output is quick and convenient but it can be refined if we use *format descriptors* as follows.

It is possible to obtain some control over the layout of the output by associating format descriptors with each item in the `OutputList`. The format descriptors have two forms:

```

E:W    or
E:W:D
```

where `E` is any expression in the `OutputList`, `W` is the width of the field, and, for a `REAL` value, `D` indicates the number of digits after the decimal point. `W` and `D` can be specified by expressions having `INTEGER` values. Some formatted examples follow.

```
Write('The age is ', Age:2);
```

will display:

```

The age is  7
Write('Pay the amount of $', Pay:7:2)
```

will display:

```

Pay the amount of $1234.56
Pay the amount of $  78.90
```

depending on the value of `Pay`.

Notice the two spaces in the first example between "is" and "7", as "7" does not fill the width of its field. Also note in the second example that the decimal point is counted as one of the seven symbols in the field. In a numerical field the numbers fill the field from the right, this is called *right justification*, with spaces added to the left to make up the field width to the value of `W`. In the last example, the spaces between the dollar sign and the dollar amount is especially dangerous because digits could be put into the spaces dishonestly. This gap could be prevented by computing the exact width as will be shown shortly.

Figure 4.7 The InOutDemo Pascal program

```

PROGRAM InOutDemo;
(* Shows simple input/output *)
CONST
    YearNow = 2000;
VAR
    Age, YearBirth, Width: INTEGER;
BEGIN
    (* Prompt for Input *)
    WriteLn('Enter birth year: ');
    (* Enter & Echo an input value *)
    Read(YearBirth);
    Write('Year = ');
    WriteLn(YearBirth:4);
    WriteLn;
    (* Test for appropriate input *)
    (* Compute the approximate age *)
    Age := YearNow - YearBirth;
    (* Format & display the Output *)
    Width := 2;
    Write( 'Your age is around ' );
    Write( Age: Width );
END. (* InOutDemo *)

```

The program `InOutDemo`, shown in Figure 4.7, is a program that illustrates not only the input-output actions just discussed, but also their proper use for obtaining a user-friendly program. This program is a variation of the `Second` program we saw in Figure 3.12 in the last chapter. It simply inputs a birth year, subtracts it from the present year, and outputs the resulting `Age`. In its original version in Chapter 3 it was much shorter, but here the new version illustrates a number of concepts mainly about “user-friendly” communication between users and computers. The program first prompts the user for a value, then echoes the value just input back to the user to serve as a check. A comment `(* Test for appropriate input *)` at this point in the program indicates that code should be added at some future time to check that the value input is reasonable, for example, that it is non-negative, or not greater than the `YearNow` constant. The output shows some helpful comments and `Age` is written in a field width set to have value 2, as the following execution shows.

```

Enter birth year:
1975
Year = 1975

Your age is around 25

```

There are a number of possible improvements to this program. First, it could test the input to see whether the entered value is proper. For example, if `YearBirth` is later than `YearNow`, then the following piece of program could detect that improper input and serve as an obstacle while the input values are improper.

```

(* Test for an appropriate input *)

```

```

WHILE BirthYear > BirthNow DO BEGIN
    WriteLn('The year is improper ');
    WriteLn('Enter another value ');
    Read(BirthYear);
END;

```

Another improvement would be to compute the value of Width. The following fragment of program sets Width to 1, 2 or 3 depending on the age. This could replace the previous assignment of the value 2 to Width

```

(* Compute the Field Width of Age *)
IF Age >= 100 THEN
    Width := 3;
IF Age < 100 THEN
    Width := 2;
IF Age < 10 THEN
    Width := 1;
Write('Your age is ', Age:Width);

```

4.3 More Programs: A Top View of Pascal

In the remainder of this chapter, we'll provide some more examples of simple but complete programs to show some how algorithms are expressed in Pascal, as well as differences and similarities among programs. Our examples will also illustrate the main forms (Sequence, Selection, Repetition) and the two numeric types (INTEGER and REAL). The emphasis will be on introducing the syntax that is common to all Pascal programs. In some cases some details will be informally introduced only briefly, and the formal definition of the syntactic features will be left for the next chapter.

Our first program example is ChallengeGuess, which was written from the algorithm description found in Figure 4.40 of Chapter 4 in the Principles book. The program follows closely the pseudocode, and is reproduced in Figure 4.8.

Figure 4.8 Pascal program ChallengeGuess

```

PROGRAM ChallengeGuess;
{ Challenge a user to guess a number }

VAR Number, Guess, Count: INTEGER;

BEGIN
    Number := 1 + TRUNC(1000 * Random);
    { 0 < Number <= 1000 }
    Count := 0;
    Write('Make a guess: ');
    Readln(Guess);
    WHILE Guess <> Number DO BEGIN
        IF Guess > Number THEN
            Write('High; ');
        ELSE
            Write('Low; ');
    END;

```

```

        Count := Count + 1;
        Write('Make a guess: ');
        Readln(Guess);
    END; {WHILE}
    Writeln;
    Writeln('Congratulations! ', Guess:1, ' is
right.');
```

```

        Writeln('It took you ', Count:2, ' tries');
    END.
```

The program `ChallengeGuess` declares three integer variables, `Guess`, `Number` and `Count`, which will be used respectively to hold the value guessed by the challenger, the value originally chosen by the program, and the number of guesses necessary to find the number. The program statements are included between the `BEGIN`, `END` pair. The first statement chooses a random number between zero and 1000, by calling standard function `Random` which returns a Real value between 0 and 1. The next statement sets `Count` to zero, and the next two statements ask for and read the first `Guess`.

Then we start a `WHILE` loop whose body includes seven lines. The `Guess` is compared to the chosen `Number`, and a message is displayed indicating whether the guess was too high or too low. The `Count` is incremented and the next guess is asked for and read. The loop will terminate when the `Guess` and the `Number` will have identical values.

The last three statements display a message of congratulations with the number of guesses that were necessary to find the `Number`. Note the width of 1 that is given for `Count`: this will avoid having unnecessary spaces in the displayed line like:

```

    Congratulations! 1 is right.
```

as the width is enlarged if the value is too big for the given width. The following is an example of the execution of this program.

```

    Make a guess: 500
    Low; Make a guess: 750
    Low; Make a guess: 875
    High; Make a guess: 812
    Low; Make a guess: 844
    High; Make a guess: 828
    High; Make a guess: 820
    High; Make a guess: 816
    High; Make a guess: 814
    Low; Make a guess: 815

    Congratulations! 815 is right.
    It took you  9 tries
```

Before introducing other program examples, we'll pause and discuss programming style and program layout. These two aspects are of no importance to the computers that will run your programs, but are extremely important for the humans who will have to read, understand, modify, in other words, maintain your programs.

4.4 Programming Style

Remember that your program is first a communication: communication of the algorithm to the computer, and communication of the solution to a given problem to the programmers that will use and maintain it. To achieve good communications, it is important to develop a good style for writing programs early. The main goal of this style is to make programs readable to other programmers, and also to yourself. When you return to modify a program you did a year ago, you will be surprised at how difficult it is to understand what you did! It is much like trying to understand a shopping list that you wrote a year ago! A good programming style will make a program more readable, understandable, modifiable, and "debuggable". The program examples we have shown all have a common style.

You might be astonished to learn that poetry is closer to programming than is prose. Poetry has greater structure, words are selected very carefully, each line is spaced properly and indentation is important. On the other hand, prose is viewed as one long stream; it is broken anywhere (including the middle of a word) to begin another line.

Like poetry, programming style is an art. Your ability will improve with experience, but when you are just beginning some guidelines and rules of thumb are especially useful. The following guidelines are not "sacred" rules that should never be broken but can serve to help you develop judgment toward an individual style. Remember, it takes practice to know when to break rules.

The identifiers you choose should be as meaningful as possible. This could take some time and a little effort but it is worthwhile, for it is the most important way to improve program readability. Names in general should not be too short (I, J, K, or XX, OT have no meaning, but in a mathematical context x, y and z may have) nor should they be too long (these are error-prone and can hide the structure of the program). As a rule of thumb, a length of "8 plus or minus 3" is ideal. The names of variables are proper nouns so the first letter could be capitalized (as in Sum, Balance). If the names consist of compounded words then the first letter of each word should also be capitalized (as in OverTime, CountOfSheep, etc.)

The judicious use of comments is also very important to a good programming style. Comments should describe what is done and why it is done, but not how it is done—the code tells how it is done. Oftentimes, comments are assertions that explain how values are related at that point in the program. Comments should not occur too often (hiding the program), nor too seldom (hiding the main ideas). You'll reach a good balance with experience. In many ways, comments are like footnotes in text. Here are two examples of Pascal comments whose advice should be followed:

```
( * UNIFORM UPPER CASE IS HARDER TO READ THAN LOWER CASE
  * )

{  Comments should be meaningful, and be written in a
   readable way  }
```

Looking at our example programs, you might have noted that the spacing used in a program is very important for readability. Horizontal spacing involving indentation serves to show levels and code structure. The indentation should not be too little (one space is hard to see) nor too much (more than 5 spaces takes time); here we indent by 2 or 3 spaces. Lining up identifiers (or colons) in declarations often looks good and provides a checklist—the eye finds it easy to run down columns. Inserting blank spaces can also aid readability; for example:

```
Write('Age =', Age: 5);
```

is preferable to the squeezed

```
Write('Age=',Age:5)
```

Vertical spacing is also extremely important; blank lines provide gaps so the human eye can view clusters of lines as a unit separated from other units. A gap between the uses part of a program and the declaration part is obvious; other gaps may take some judgment. Sometimes a gap between each form (Selection or Repetition) clarifies a program, sometimes it just doesn't help. Often a comment preceding a cluster of lines ties the lines together as a unit.

Many of the above guidelines are not important by themselves, but when all taken together can make a big difference. Consistency in a single style is often sufficiently important by itself. Also, ignoring some of these guidelines may not appear serious, but when a program of many pages must be read, it can be very bothersome to deal with such “unfriendliness”.

Other aspects of style will be considered later. They include naming of procedures (as verbs), conditions (as adjectives), avoiding global variables, the indentation of forms, modularity, etc.

4.5 Layout of Programs

Let's look now at one more slightly larger Pascal program, to see its layout or arrangement. It is a program to make change for a tendered amount when purchasing something, given a *Cost*. This program corresponds to the Make Change algorithm of Figure 4.25 in Chapter 4 of the Principles book.

Figure 4.9 The MakeChange program

```
PROGRAM MakeChange;
(* Make change for a tendered amount *)

VAR
    Cost,
    Remainder,
    Tendered: INTEGER;

BEGIN
    { Input the Cost }
    Write('Enter the cost in cents: ');
    Read(Cost);
    Write('Enter the amount tendered in cents: ');
    Read(Tendered);
```

```

{ Make the Change }
Remainder := Tendered - Cost;
WHILE Remainder >= 25 DO BEGIN
    Write('Quarter ');
    Remainder := Remainder - 25;
END; {WHILE}      {Remainder < 25}
WHILE Remainder >= 10 DO BEGIN
    Write('Dime ');
    Remainder := Remainder - 10;
END; {WHILE}      {Remainder < 10}
IF Remainder >= 5 THEN BEGIN
    Write('Nickel ');
    Remainder := Remainder - 5;
END; {IF}          {Remainder < 5}
WHILE Remainder >= 1 DO BEGIN
    Write('Penny ');
    Remainder := Remainder - 1;
END; {WHILE}      {Remainder < 1}
Writeln;
END. { MakeChange }

```

The Pascal program `MakeChange`, shows one layout of the program. It begins with the minimum of documentation—a brief statement of the program’s purpose. To be complete, this documentation should also include the name of the author and the date the program was written. This date (or some kind of number, like “Version 1.5”) is important because programs get modified and there may be a number of versions available, so the date makes it convenient to find the latest one. The documentation at the head of the program should also include a brief listing and a possible description of the program’s inputs and outputs, and any limitations it might have, or other information that could be useful. For example, the following comment could be inserted at the beginning of the program.

```

(* Program to make change ( all amounts in cents )
 * by Ann Onymous 94/2/30. Version 1.1
 *
 * INPUT:
 * Cost      : cost of items, in integer, not real!
 * Tendered  : integer amount in cents
 *
 * OUTPUT:
 * a message indicating the number of quarters
 * (25 cents), the number of dimes (10 cents), the
 * number of nickels (5 cents), and the number of
 * pennies (1 cent), like:
 * "Quarter Quarter Dime Penny Penny"
 *
 * TO DO:
 * Change to output the number of coins
 *)

```

Notice that this comment extends over a number of lines. The intermediate asterisks at the left are not necessary, but do serve to group these lines. The colons are also lined up; this is not necessary, but it looks good (some other applications of vertical alignment can be seen in the complete program). Some of this documentation may seem obvious, but it should still be done as a service to others who may wish to read the program. Notice how this documentation mentions the work that remains to be done. Gaps—empty lines—between lines also serve to group parts of the program, separating one part from another for greater readability. Indentation is also very helpful for humans, and is used mostly to show what is included in the various loops. Notice also the other comments in the body of the program:

```
    { Input the cost }
    { Make the change }
```

Taken alone, these comments are a higher level algorithm. Some comments are assertions like:

```
    {Remainder < 25}
```

which indicate the state of the program variables at various points. A typical execution of the program follows.

```
Enter the cost in cents: 33
Enter the amount tendered in cents: 100
Quarter Quarter Dime Nickel Penny Penny
```

Figure 4.10 shows the same program in a different layout. Look at it and see what layout you prefer.

Figure 4.10 Other layout for program MakeChange

```
PROGRAM MakeChange2;
(* Make change for a tendered amount *)
VAR Cost, Remainder, Tendered: INTEGER;
BEGIN
  { Input the Cost }
  Write('Enter the cost in cents: '); Read(Cost);
  Write('Enter the amount tendered in cents: ');
  Read(Tendered);
  { Make the Change }
  Remainder := Tendered - Cost;
  WHILE Remainder >= 25 DO BEGIN
    Write('Quarter '); Remainder := Remainder - 25;
  END; {WHILE} {Remainder < 25}
  WHILE Remainder >= 10 DO BEGIN
    Write('Dime '); Remainder := Remainder - 10;
  END; {WHILE} {Remainder < 10}
  IF Remainder >= 5 THEN BEGIN
    Write('Nickel '); Remainder := Remainder - 5;
  END; {IF} {Remainder < 5}
  WHILE Remainder >= 1 DO BEGIN
    Write('Penny '); Remainder := Remainder - 1;
  END; {WHILE} {Remainder < 1}
```

```
Writeln;
END. { MakeChange2 }
```

The alternative layout of program `MakeChange2` shows a very different arrangement. It is a short and fat layout compared to the long and thin layout of `MakeChange` shown in Figure 4.9. However, the two versions of the program behave exactly the same and produce the same results. As you might have already guessed, the second layout of `MakeChange2` is not preferred. It appears to be less readable, because the “vertical squashing” hides the program’s structure. It is also harder to modify, because, to insert a segment of code may require much moving of the text, that may also result in errors.

Modifications and improvements to programs are always possible. For instance, this `MakeChange` program could be modified to test for proper input values, rejecting negative values for `Cost`, `Tendered` and `Remainder`. The input `Cost` and `Tendered` could also be echoed, and the total amount of the change could be output. The output could be modified as already mentioned and also, for example, to print numbers as “one penny” or with plural denominations such as “three pennies”. Many other modifications to this program are suggested in the projects at the end of this chapter.

In general, a Pascal program can be seen as having a top part (declarations) indicating WHAT (data, names, types, tools imported), and a bottom part (actions) indicating HOW the data are to be manipulated. Documentation, in the form of comments, indicates WHY something is done and supports the WHAT and HOW.

4.6 More Programs: Continued

We’ll present two more examples of complete programs before the end of this chapter. However, you should be aware that Pascal offers you a number of tools that make it possible for you to write program without having to “re-invent the wheel” every time. On the one hand, Pascal includes a number of pre-defined subprograms that are always available. On the other hand Pascal allows you to define your own subprograms, and to use subprograms from program libraries.

Actions: Pre-Defined Standard Functions in Pascal

As a convenience to the programmer, commonly used actions are often made available in a programming language. Pascal has a number of pre-defined actions, as shown in Figure 4.11. These actions are all functions, analogous to trigonometric functions like sine, cosine, etc., that accept a single value and yield a single value of a given type. Functions can be used anywhere a variable of that type may be used.

Figure 4.11 Pascal pre-defined functions

Pre-defined Functions

<code>ABS (N)</code>	yields the absolute value of any numeric type <code>N</code>
<code>ODD (I)</code>	is true if integer <code>I</code> is odd

<code>SQRT(N)</code>	yields the positive square root of any numeric type N
<code>SQR(N)</code>	yields the square of any numeric type N
<code>ROUND(R)</code>	converts a REAL value R into its nearest INTEGER value
<code>TRUNC(R)</code>	converts a REAL value R into an INTEGER by truncation
Others	including SIN, COS, LN, EXP, ORD, CHR, SUCC, PRED

`ABS(N)` is a function that operates on a numerical value of any type (REAL or INTEGER), and returns its absolute value (i.e. its positive value) of the same type. For example:

`ABS(-1)` is 1, `ABS(-2.3)` is 2.3, `ABS(4.5)` is 4.5.

`ODD(I)` is a function that operates on an INTEGER value I and indicates whether this value is an odd number. It can be used within conditions both in Repetition and Selection forms. For example:

```
IF ODD(YEAR) THEN
    Write('Cannot be a leap year');
```

`SQRT(N)` is a function that computes the positive square root of any REAL or INTEGER value N and returns a value of the same type.

`SQRT(4)` is 2, `SQRT(4.0)` is 2.0, `SQRT(8)` is 2

`SQR(N)` computes the square of any REAL or INTEGER value and returns a value of the same type. For example:

```
Hypotenuse := SQR(SQR(Base) + SQR(Height));
```

`ROUND(R)` is a type conversion function that converts a REAL number R into an INTEGER value by returning the nearest INTEGER to R.

`ROUND(3.14)` is 3, `ROUND(2.7)` is 3 and
`ROUND(-1.9)` is -2

`TRUNC(R)` is a type conversion function that converts a REAL number R into an INTEGER value by chopping off the decimal part of the number. For example,

`TRUNC(3.14)` yields 3, `TRUNC(2.7)` yields 2.

The trigonometric functions `SIN(R)` and `COS(R)` return the sine and cosine of a REAL angle R represented in radians. Angles in degrees can be converted to radians by multiplying by `Pi` and dividing by 180.

The exponential functions `LN(R)` and `EXP(R)` involve the base e (where $e = 2.71828$). These functions can be used to compute the Nth power of X by:

```
XtoPowerN := EXP( N*LN(X) );
```

The logarithm of X to any base B can be determined from:

```
LogXtoBaseB := LN(X)/LN(B);
```

There are other pre-defined actions, including `ORD` and `CHR`, which we will consider later when needed.

Let's now look at another simple Pascal program, shown in Figure 4.12, to produce a table of temperatures expressed in Celsius and Fahrenheit units.

Figure 4.12 Pascal program TempTable

```

PROGRAM TempTable;
(* Creates a table of temperature *)
(* Fahrenheit values approximate *)
VAR
  Celsius: INTEGER;
  Fahrenheit: REAL;
BEGIN
  WriteLn( 'TEMPERATURE' );
  Writeln( ' Celsius Fahrenheit ' );
  Celsius := 0;
  WHILE Celsius <= 100 DO BEGIN
    Fahrenheit := (9/5)*Celsius + 32;
    Write(Celsius: 5);
    WriteLn(ROUND(Fahrenheit): 10);
    Celsius := Celsius + 10;
  END; (* WHILE *)
END. (* TempTable *)

```

Program TempTable outputs a table of Celsius temperatures and the corresponding Fahrenheit values. The Celsius temperature starts at 0°C. Within the loop, the Fahrenheit value is computed, both the Celsius and Fahrenheit values are written as integers, and the Celsius value is increased by 10 to get the next temperature. The looping continues while the Celsius value is less than or equal to 100, and it stops when Celsius exceeds 100.

Notice that the temperature conversion formula is similar to the one in the Convert example of Figure 3.16 in the last chapter. Note however that in TempTable, the decimal points are not shown in order to make TempTable more readable. This does not change the computation which is done with REAL values because of the “/” operation. Usually it is better to write the constants as REAL and show the decimal points in order to make the program less error prone. The program produces the following output.

Temperature	
Celsius	Fahrenheit
0	32
10	50
20	68
30	86
40	104
50	122
60	140
70	158
80	176
90	194

100

212

Libraries: Using Units in Pascal

Libraries of programs are extremely important in software development as they enable software to grow in a disciplined and controlled manner. Libraries make it possible to achieve *modularization*, one of the goals of *software engineering*. Software engineering is a field of Computer Science that establishes methods for the development of good software. Libraries are basically collections of subprograms and data types that can be viewed as useful extensions to a programming language.

The Pascal libraries are called *units*. In this chapter, we will not create units, that will only come later, but we will use them. The details of the units are hidden, but the subprograms within the units are available to be shared.

We have defined library `IntLib` as a collection of various operations on `INTEGER`s that are not provided in Pascal. All these operations are subprograms, but most of them are Pascal procedures, rather than Pascal functions. Procedures calls are independent statements, analogous to the invocation of subalgorithms in pseudocode, while function calls are only used in expressions, and are therefore only parts of statements. The following operations are available in library `IntLib`.

`Incr(I, S)` is a procedure that increments an `INTEGER` `I` by a step size of `S`, where `S` is also an `INTEGER`. For example:

```
Incr(A, 1); increases variable A by 1.
Incr(B, -2); decreases variable B by 2
Incr(C, C); doubles variable C
```

`Decr(I, S)` is a procedure that decrements an `INTEGER` `I` by an `INTEGER` step size of `S`. It is very similar to the above increment operation.

`Maximize2(A, B, C)` is a procedure that finds the maximum of any two `INTEGER`s `A` and `B` and sets `C` to that value. For example:

```
Maximize2(7, 11, M);
results in M having 11 as its value.
```

`Minimize2(A, B, C)` is a procedure that finds the minimum value `C` of any two `INTEGER`s `A` and `B`. It is similar to `Maximize2`.

`Order2(A, B)` is a procedure that sorts the two input variables `A`, `B` in increasing order. For example, if the value of `X` is 11 and the value of `Y` is 7:

```
Order2(X, Y);
results in X having value 7 and Y having value 11.
```

`Order3(A, B, C)` is a procedure that sorts the three input variables `A`, `B`, and `C` in increasing order. For example, if the value of `X` is 11, the value of `Y` is 7, and the value of `Z` is 9:

```
Order3(X, Y, Z);
```

results in X having value 7, Y having value 9 and Z having value 11.

Divide(N, D, Q, R) is a procedure that divides numerator N by denominator D and produces a quotient Q and a remainder of R. For example:

```
Divide( Sum, Num, Mean, Rem);
```

divides Sum by Num yielding a quotient Mean and a remainder Rem.

IntToReal(I, R) is a procedure that converts a given INTEGER I into its corresponding REAL value R. For example:

```
IntToReal(7, X);
```

results in X having the REAL value of 7.0.

The fact that a program requires the use of one or more operations from a Library is indicated by naming the Library in the Uses clause, which follows the program header (recall the syntax diagram of Figure 3.7 in the last chapter). To use any of the above procedures from IntLib requires only to add:

```
USES IntLib;
```

at the beginning of your program. We'll now look at such a program, shown in Figure 4.13, that implements the algorithm taken from Figure 4.17 in Chapter 4 of the Principles book.

Figure 4.13 The Triangle program

```
PROGRAM Triangle;
(* Determines if a 3-sided figure is a triangle
*)
USES IntLib;

VAR SideA, SideB, SideC: INTEGER;

BEGIN
  Write('Give the three sides ');
  Read(SideA, SideB, SideC);
  Order3(SideA, SideB, SideC);
  IF SideA + SideB < SideC THEN
    Write('not a triangle ')
  ELSE
    IF SideA = SideC THEN
      Write('equilateral triangle')
    ELSE BEGIN
      IF (SideA = SideB) OR (SideB = SideC) THEN
        Write('isosceles ');
      IF SideA*SideA + SideB*SideB = SideC*SideC
      THEN
        Write('right triangle')
      ELSE
        Write('triangle ');
```

```

    END ;
END .

```

The `Triangle` program is simple but involves nested selections. It uses procedure `Order3` from library `IntLib` to put the side lengths in order. To do this, it has only to invoke it, because the “`Uses IntLib;`” statement has made all the contents of the `IntLib` library available to program `Triangle`.

The program declares three integer variables for the triangle sides. It reads the three values and orders them by invoking `Order3`. Then, it checks to see if it is not a triangle and if so displays a message. If it is a real triangle the program checks to see if it is an equilateral triangle, and if not checks the isosceles condition. It then checks to see if the triangle is a right triangle. The indentation of the program is very important as it shows the structure of the program. In particular, it shows what statements are nested in others. For instance, the first `ELSE` includes the rest of the program so that, if the figure is not a triangle, none of the following statements are executed. On the other hand, the last `ELSE` includes two `IF` statements. Nested `IF` statements must be written with care, and more will be said about them in the next chapter. In the meantime you may note that `ELSEs` cannot be preceded by semicolons.

A typical execution of the `Triangle` program follows.

```

Give the three sides 5 4 4
isosceles triangle

```

Note that the last line message was displayed by executing two `Write` statements.

4.7 A Foretaste of Procedures

In the last program example, `Triangle`, we have invoked a procedure from library `IntLib`. To be able to do that the `IntLib` library must have been defined by another programmer, and the procedure must have been completely defined. Defining procedures in Pascal is not very different from writing programs. Remember that procedures are subprograms corresponding to subalgorithms. Subprograms have a form similar to programs as they have a header, a declarations part and a body. Let’s implement the algorithm developed for the game of Fifty in Chapter 4 of the *Principles* book. That solution used a subalgorithm `Turn Of Player` (found on Figure 4.29 of that book) which must be translated into a Pascal procedure. We’ve done so in Figure 4.14, which shows the corresponding program with numbered lines to facilitate references.

Figure 4.14 The program of Fifty

```

(1)  PROGRAM Fifty;
(2)  { Play the dice game of Fifty }
(3)
(4)  VAR Score1, Score2: INTEGER;
(5)
(6)  PROCEDURE TurnOfPlayer(VAR Score: INTEGER);

```

```

(7)  { Play a turn }
(8)  VAR DiceA, DiceB: INTEGER;
(9)  BEGIN
(10)     DiceA := 1 + TRUNC(6 * Random); { 0 < Dice <= 6 }
(11)     DiceB := 1 + TRUNC(6 * Random); { 0 < Dice <= 6 }
(12)     IF DiceA = DiceB THEN
(13)         IF DiceA = 3 THEN
(14)             Score := 0
(15)         ELSE
(16)             IF DiceA = 6 THEN
(17)                 Score := Score + 25
(18)             ELSE
(19)                 Score := Score + 5;
(20)     END; {TurnOfPlayer}
(21)
(22) BEGIN
(23)     Score1 := 0;
(24)     Score2 := 0;
(25)     WHILE (Score1 < 50) AND (Score2 < 50) DO BEGIN
(26)         TurnOfPlayer(Score1);
(27)         TurnOfPlayer(Score2);
(28)     END;
(29)     IF Score1 = Score2 THEN
(30)         Write('The game is a tie ');
(31)     ELSE
(32)         IF Score1 > Score2 THEN
(33)             Write('Player1 wins ');
(34)         ELSE
(35)             Write('Player2 wins ');
(36) END. { Fifty }

```

The main program is found in lines 22 to 36. It initializes the scores (lines 23-24), repeatedly invokes `TurnOfPlayer` (lines 25-28) until one of the scores is greater than or equal to 50. Finally a message is displayed indicating the winner (lines 29-35).

The subalgorithm `TurnOfPlayer` has been packaged into a procedure (lines 6-20) which includes a header giving the name of the procedure and its parameter, the declaration of two integer variables representing the two dice, and the actions of the subalgorithm. The procedure appears before the main program as it is a declaration: in order to use a procedure we must declare it beforehand or import it from a library as we did in the triangle program. The procedure simulates two throws of each die by using standard function `Random`, as we have already done in the `ChallengeGuess` example. If the throws bring up a double, the score is modified accordingly.

The statements of a procedure are similar to the statements of a program, they are ended with an `END` statement which is followed by a semicolon and not a period. We'll present formally procedures and functions in the next chapters, so do not become anxious if things are still a little murky, this example is only

intended to show you that there are neither mysteries nor magic in programming.

4.8 Chapter 4 Review

This chapter introduced more of the basic concepts of the Pascal language, starting with identifiers and declarations of variables and constants, whose syntax diagrams were given. The chapter also introduced simple input and output operations in Pascal, mainly for numerical values.

The chapter also expanded its presentation of actions on data. These actions that included arithmetic operations built-into the language, were expanded to include both standard actions pre-defined in the language (`ABS`, `SQRT`, `FLOAT`, `ROUND`, etc.) and actions imported from various Libraries (`Incr`, `Max2`, `Order3`, etc.). One small Library, `IntLib`, was introduced to show how such Pascal units could be used.

A few Pascal programs that manipulate data were also introduced in this chapter. The programs were short and simple, but they illustrated nevertheless many aspects of Pascal. They were given as complete examples with the goal to familiarize you with the syntax of Pascal, even though the formal definition of some parts weren't covered in this chapter. Programming style and program layouts, important to produce readable and usable programs, were introduced, as well as a number of simple rules that were applied to the example programs of the chapter, and will be applied in the remainder of the book.

4.9 Chapter 4 Problems

1. Pieces of Program

Write (but do not run) a short piece of a Pascal program to do the following:

- a. **Round off** an integer to the nearest hundred, for example, 1234 rounds to 1200 and 5678 rounds to 5700.
- b. **Extend** the `MakeChange` program to return half dollars, dollars, and two dollar bills.
- c. **Tear apart** a four digit integer `D`, into its digits `D1`, `D2`, `D3`, `D4`. For example, 1984 breaks into `D1=1`, `D2=9`, `D3=8`, `D4=4`.
- d. **Perform** the `MOD` operation using other arithmetic operations such as `DIV`, multiplication and subtraction.

4.10 Chapter 4 Programming Projects

Generate Conversion Tables

Modify the temperature conversion `TempTable` program to create some other table of conversions. Some examples are:

1. **Angle Table**

Convert angles from degrees to radians for degrees ranging from 0 to 360 in steps of 10 degrees.

2. **Metric Conversion Table**

Create a table of metric amounts, say kilograms from 0 to 25, and the corresponding amounts of pounds and ounces (rounded off).

3. **Sales Tax Table**

Create a sales tax table that lists prices up to a dollar in steps of 5 cents, and for each price, gives the rounded off tax (for a given tax rate of say 6.5 percent). Then it continues in steps of a dollar up to 15 dollars.

4. **Compound Growth Table**

Create a table showing the growth (of inflation, bank balance, population, etc.) of some initial amount (say 100) when the rate of growth is a fixed percent of the present amount for each time period (month, year, etc.). Create one table for two rates (5 percent and 10 percent) for 20 time periods.

5. **Other Tables**

Create a table for any other formula that interests you (from business, science, mathematics, etc.).

Observing Errors

Become acquainted with your computing system by entering and running one of the following short programs. Then make some errors to see what happens. Finally make some modifications.

```
PROGRAM Circle;
(* Computes the circumference and area *)

CONST
    Pi = 3.14159;
VAR
    Radius, Area, Circumference: REAL;

BEGIN
    WriteLn('Enter radius of circle ');
    Write('Enter negative to quit ');
```

```

Read(Radius);
WHILE Radius >= 0.0 DO BEGIN
    Circumference := 2.0 * Pi * Radius;
    Write('Circumference is ', Circumference: 7: 3);
    WriteLn;
    Area := Pi * Radius * Radius;
    Write('The area is ', Area: 7: 3);
    WriteLn; WriteLn;
    Write('Enter next radius ');
    Read(Radius);
END (* WHILE *);
END.

```

```

PROGRAM Compound;
(* Shows compounding growth *)
(* Indicates time to double *)

VAR
    Amount, Rate: REAL;
    Year : INTEGER;

BEGIN
    Write('Enter a growth rate ');
    Read(Rate);
    Write('Year Amount ');
    WriteLn;
    Year := 1;
    Amount := 1.0;
    WHILE Amount < 2.0 DO
        BEGIN
            Amount := Amount * (1.0 + Rate);
            Year := Year + 1;
            Write(Year: 4, Amount: 7: 3);
            WriteLn;
        END (* WHILE *);
    END.

```

Demilitarize Time

Create a program to input two values representing the time in military (24 hour) units (such as 0250 and 1430) and to output the number of minutes between these times.

Hint: Use MOD and DIV to break up the input times into Hours and Minutes.

TipTable

Create a table that shows the tip corresponding to various amounts ranging from one dollar to thirty dollars, in steps of one dollar. The tip is 15 percent of the bill, rounded off.

a. Easy Tip

Create the table using `INTEGERS` for the dollar amounts, and for the number of cents. For example, the amount of 20 dollars has a tip of 300 cents.

b. Clearer Tipper

Modify the table by writing the tip in terms of dollars and cents, with the decimal point. For example, the amount 7 dollars has a tip of 1.05. In order to print the tip with the decimal point and two digits for the cents, you will have to use:

```
Write(Tip: 5: 2);
```

c. Clearest Tipper

Modify the above table by writing both the amount and the tip with a decimal point and making the amount change in smaller steps of half dollar.

d. Bigger Tipper

Modify again the above table by creating a third column in the table to show a larger tip of 20 percent.

e. Round Tipper

Modify the table so that the tip is rounded to the nearest nickel (i.e. is a multiple of 5). For example, an amount of 13.50 would correspond to a tip of \$2.025 which becomes \$2.05 in this table.

f. Fancy Tipper

Modify any of the above tables by adding a border (of asterisks, or dashes for horizontal borders and exclamation points for vertical borders) around the entire table.

g. Nicer Tipper

Modify any of the above tables by printing the dollar signs immediately to the left of each dollar amount.

h. Nicest Tipper

Modify some of the above tables so that there is not a single column, but two columns side by side.

i. More Tips

Make up your own additional modification to this table.

STT: Sales Tax Table

You are to write a program that creates a table such as the following to determine the amount of sales tax for various sales amounts. The sales tax is given as a percentage (here 6.75%) of the Sales amount.

Sale Amount	Tax 6.75%
1.00	0.067
2.00	0.135
3.00	0.202
4.00	0.270
5.00	0.337
6.00	0.405
7.00	0.472
8.00	0.540
9.00	0.607
10.00	0.675

As you “grow” the program by adding the following parts, do not make a “hard” paper copy of any tables until your very last one. This saves paper (and therefore trees).

- Begin by writing a program for a simple table as above.
- Modify this so that the taxes are rounded off to the next nearest cent.
- Enlarge the table for larger sales (up to \$20).
- Include an extra column for other countries with other tax rates (say 8.25%).
- Add extra tax ranges, before and after your range,
going from \$0.10 to \$1.00 in steps of 0.10,
going from \$20.00 to \$200 in steps of 10 dollars.
- Put two big columns together in parallel, for a fatter and shorter table.
- Beautify the table, with a header, good spacing, a box around the table, etc.
- Incorporate any other changes that you have time to do.

Be prepared to turn in the last table that you created. We may wish to compete for the best tax table. To print the dollar and cents amounts with the decimal point, you will want to use a field width as described for the printing of Clearer Tipper above.

SSP: Simple Side Plot

The given program creates a table of values of a function and also plots this function on its side. In this case the function is the square, $Y = X * X$, but other functions may be substituted for this square.

```

PROGRAM SidePlot1;
(* Plots Y vs. X sideways *)

VAR
  X, Y : REAL;
  RY   : INTEGER;

BEGIN
  X := -6.0;
  WHILE X <= 6.0 DO
    BEGIN
      (* Table part *)
      Y := X * X;
      Write(X: 5: 2);
      Write(Y: 7: 2);

      (* Plot part *)
      Write(' ');
      RY := TRUNC(Y) + 1;
      Write(0: RY);
      WriteLn;

      (* Next part *)
      X := X + 1.0;
    END (* WHILE *);
  END.

```

OUTPUT

```

-6.00  36.00                                0
-5.00  25.00                                0
-4.00  16.00                                0
-3.00   9.00                                0
-2.00   4.00                                0
-1.00   1.00                                0
 0.00   0.00                                0
 1.00   1.00                                0
 2.00   4.00                                0
 3.00   9.00                                0
 4.00  16.00                                0
 5.00  25.00                                0
 6.00  36.00                                0

```

The goal is to extend this program in many ways to be more general, more user-friendly, and more useful. Rename each extended version as SidePlot2, SidePlot3, etc.

1. Enter this program, run it, and get to understand it.
2. Re-range
Extend the range, by prompting for and entering the first and last values of x . Rename this `SidePlot2`.
3. Take step
Modify this program to enter the value of the step size.
4. Scale it

Modify it further to enter a scale factor, which multiplies y by some constant value. If the scale factor is 0.5, then it halves each value of y ; if the scale factor is 2.0, then it doubles each value.

5. Make axis
Beautify the output by making and labeling the axes, both x and y .
6. Foolproof it
Check that the step size is not negative; and does not print off a page, and any other possible unpleasant results.
7. Test and show it
Try your program with various plots of different sizes, such as:
 - a. the Square function
 - b. a trigonometric function
 - c. any other function of your own, from Physics, etc.

Chapter 5 The Four Fundamental Forms in Pascal

The goal of this chapter is to introduce the four fundamental forms: Sequence, Selection, Repetition, and Invocation, that are necessary and sufficient to create any algorithm. At the end of the chapter you will have the necessary tools to develop your own programs.

Chapter Overview

5.1	Preview.....	105
5.2	The Sequence Form in Pascal.....	105
5.3	Conditions in Pascal.....	107
5.4	Repetition Form: The WHILE statement.....	108
	Tracing Loops.....	110
5.5	WHILES, REALS, and Errors.....	114
5.6	Selection Forms in Pascal.....	117
5.7	More Selections: Combinations of Selection Forms.....	119
	Confusion in Choices.....	120
	More Nesting of Choices.....	120
	Alternative Ways to Code Selections.....	121
5.8	Select Form: Handling Many Branches.....	124
5.9	Awkward Nests: General Nesting.....	127
	Mixed Nests: Repetition and Selections.....	128
5.10	Subprograms: Using Subprograms as Black Boxes.....	130
	The ShortSort Library.....	134
	Notation for Defined Procedures.....	137
	Procedures vs. Functions.....	140
5.11	Binary Logic Library: BitLib.....	145
5.12	Chapter 5 Review.....	149
5.13	Chapter 5 Problems.....	150
5.14	Chapter 5 Programming Problems.....	152
	Sequence Problems.....	152
	Selection Problems.....	153
	Loop Problems.....	153
	Subprogram Problems.....	153
	Debugging Problems.....	153
	Selection Programs.....	156
	Procedures and Repetitions.....	156
	Josephus' Problem.....	158
5.15	Chapter 5 Programming Projects.....	159
	Project A: Change Change.....	159
	Project B: Payroll.....	160
	Project C: Quadratic Roots.....	160
	Project D: Digital Circuits.....	160
	Project E: Roll Your Own.....	161

CRN: Convert Roman Numbers.....	161
GPR: Growing Pay Roll.....	161
MWM: Many Ways to Mid.....	162
DFP: Data Flow Programming.....	164

5.1 Preview

As a continuation of the introduction of the various Pascal constructs, this chapter concentrates on the Pascal equivalents of the four fundamental forms: Sequence, Selection, Repetition, and Invocation. Most of these have already been illustrated by examples in the preceding chapters, but the forms will be presented here more formally.

The simplest form is the Sequence, a series of statements, that is considered very briefly. Logical expressions, which are based on relational operations, are considered in detail for they are used in both the Repetition and Selection forms.

The simplest expression of the Repetition form, a `While` statement, is introduced next. This introduction is accompanied by the tracing of the execution of such a statement, with some aspects of loop invariants.

The Selection form is introduced with the Pascal `IF-THEN-ELSE` statement. This statement is presented in detail, as well as nested `IF` statements.

The Pascal equivalents of the Invocation Form, `PROCEDURE` and `FUNCTION` calls, are considered from the point of view of using `PROCEDURES` and `FUNCTIONS` that are available from Libraries. This presentation is mainly based on the use of dataflow diagrams. The complete creation of subprograms will be covered in a later chapter.

5.2 The Sequence Form in Pascal

The Sequence is the simplest of the four fundamental Forms introduced in the Principles book. The Pascal version of this Form consists simply of series of statements *separated* by semicolons, and sandwiched between a `BEGIN` and an `END`. This is also known as the Pascal *compound statement*. The statements should be indented from the `BEGIN-END` pair for improved readability. Also semicolons must separate statements, but they may also *terminate* statements as shown in Figure 5.1.

Figure 5.1 A Pascal Sequence

<pre>BEGIN T := R; R := S; S := T END</pre>	<pre>BEGIN (* Swap *) Temp := First; First := Second; Second := Temp; END (* Swap *)</pre>
---	--

Shown in this figure are two versions of a program fragment to swap the values of two variables. The first version, on the left, is very minimal. It has no indentation, no meaningful names, and no semicolon after the third

assignment—no semicolon is needed there because semicolons separate statements. The second version, on the right, is more verbose. It has more meaningful names and indentation to show the program's structure. It is also more consistent since it has semicolons terminating every statement including the last (where it is not strictly necessary). This extra semicolon, just before the `END`, allows the insertion of other statements at the end of this sequence form (between the last statement and the `END`), without having to go back to the previous statement to add the extra semicolon. In addition, it is easier to remember the rule “Always put a semicolon after a statement” than a rule with an exception to cover the last statement in a sequence.

Other examples of the Sequence form follow. They can be viewed as short but complete programs, or they can be used as parts of other programs, or they can also be “encapsulated” into subprograms (`FUNCTIONS` or `PROCEDURES`, as seen in the last example of Chapter 4).

Figure 5.2 A second Pascal Sequence

```
BEGIN (* To De-Militarize Time *)
    Read(MilTime);
    Hours   := MilTime DIV 100;
    Minutes := MilTime MOD 100;
    Write(Hours, Minutes);
END (* of De-Militarizing Time *)
```

The program fragment `DeMilitarize Time`, shown in Figure 5.2, is a piece of a program showing how a military time (given as an `INTEGER` such as 2345) can be split into `Hours` and `Minutes` (such as 23 hours and 45 minutes). The `INTEGER` divide operation `DIV` yields the hours directly ($2345 \text{ DIV } 100 = 23$). The `MOD` operation produces the remainder of the integer division which is stored in `Minutes` ($2345 \text{ MOD } 100 = 45$).

We can obtain the number of minutes without using the `MOD` operation, if we use a slightly more complex expression:

```
Minutes := MilTime - 100*(MilTime DIV 100)
```

Figure 5.3 shows another program fragment that computes the value of a variable `X` raised to the `Nth` power.:

Figure 5.3 A third Pascal Sequence

```
BEGIN (* Power *)
    Read(X, N);
    P := Exp(N*Ln(X));
    WriteLn( P );
END (* of Power *)
```

It uses both the standard exponential and logarithmic functions, introduced in the Chapter 4. Since Pascal does not have an exponentiation operator, this method could be particularly useful.

The trigonometric function `Tangent` is sometimes useful for some applications, but is not directly available in Pascal. The program fragment of Figure 5.4 first converts degrees into radians in `Rads`, and then divides the sine of this angle by the cosine to yield the tangent.

Figure 5.4 A fourth Pascal Sequence

```
BEGIN (* Tangent of Degrees *)
  Write('Enter Degrees:');
  Rads := (Pi/180) * Degrees;
  Tang := SIN(Rads) / COS(Rads);
END (* of Tangent *);
```

5.3 Conditions in Pascal

Selection and Repetition forms are controlled by a logical expression, or *condition*. For this reason, we will consider conditions first, before considering these two forms.

Simple conditions, involving one relation, are shown in bold within the following (partial) forms:

```
IF X = 5 THEN ...
IF Hours > 7*24 THEN (* Error *)...
WHILE X <= Y DO ....
IF ODD(Year) THEN .....
WHILE (X - Y) < 0.1 DO ...
```

Conditions may involve arithmetic expressions (variables, operations) as well as arithmetic relations (`<`, `=`, etc.) The relations used for comparing two quantities are summarized below, in pairs which are opposites:

<code><</code> less than	<code>>=</code> greater than or equal to
<code>></code> greater than	<code><=</code> less than or equal to
<code>=</code> equal to	<code><></code> not equal to

Each of these simple relations can be used in a single logical expression (as `X <= Y`). If the variables in the logical expression have a value, then each of these relations evaluates to true or false.

We can create *compound conditions* by joining simple conditions with one of the logical operations (AND, OR, and NOT). Here are a few examples of compound conditions:

```
(A < B) AND (B < C) (* values A,B,C are increasing *)
(T = 7) OR (T = 11) (* a winning first dice throw *)
(0 <= P) AND (P <= 1) (* P in the range [0, 1] *)
NOT(ODD(Year)) (* Year is even *)
```

Although the precedence of the three logical operations is defined with OR lowest, and NOT highest, we should use parentheses to avoid any possible confusion, either when first writing the program or later when reading it.

It is often useful to negate compound conditions. To negate a logical expression (using DeMorgan's Rule, described in Chapter 5 of the Principles book) consisting of two conditions joined by an AND we simply complement each of the conditions and change the AND to an OR. For example, consider the following expression:

(A < B) AND (B < C) (* values A,B,C increasing *)

Its negation or complement is:

(A >= B) OR (B >= C) (* A,B,C are non-increasing*)

Notice that this negation differs from a similar condition:

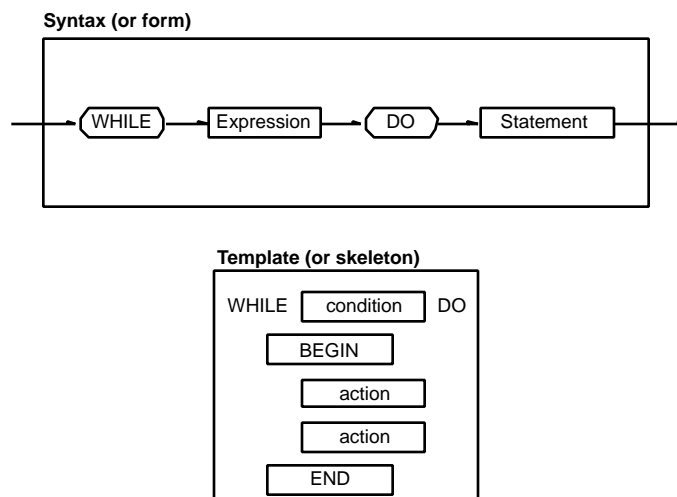
(A > B) AND (B > C) (* values A,B,C decreasing *)

We will consider such conditions in more detail when we discuss the Boolean type and Boolean expressions in Chapter 6. The informal presentation we have given here should suffice for most needs, provided that you use parentheses to avoid problems.

5.4 Repetition Form: The WHILE statement

The syntax of the Pascal WHILE loop is shown in the syntax diagram at the left of Figure 5.5. At the right of the same figure is a template that shows how the statement should be written in a Pascal program, when the body of the loop includes more than one statement.

Figure 5.5 Syntax diagram and template for Pascal WHILE statement



The Pascal WHILE statement consists of a single sequence: the word `WHILE`, followed by a logical expression, then the word `DO`, and finally a simple statement that serves as the body of the loop. Usually, the body involves more than one statement, and in that case the single statement is replaced by a compound statement. Remember that a compound statement is a group of statements enclosed within a `BEGIN-END` pair. The template shown at the right of Figure 5.5 covers this specific case. The semantics for the While statement match those of the Repetition form and are defined by the flowchart in Figure 5.6.

Figure 5.6 Flowchart showing semantics of While statement

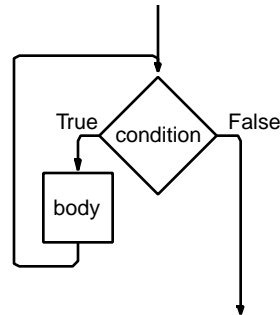


Figure 5.7 shows a typical example of a Pascal `WHILE` loop that computes the factorial of `N`.

Figure 5.7 Loop to compute the factorial

```

Fact  := 1;
Count := N;
WHILE Count > 0 DO BEGIN
    Fact := Fact * Count;
    Count := Count - 1;
END (* of while loop *) ;
  
```

Notice how we use the indentation to show the extent of the body of the loop. With this format, the `WHILE` is aligned with the corresponding `END` and the body of the loop is clearly visible.

Figure 5.8 Loop to compute the square

```

Square := 0;
OddNum := 1;
WHILE OddNum < (N + N) DO BEGIN
    Square := Square + OddNum;
    OddNum := OddNum + 2;
END (* of while loop *) ;
  
```

The segment of Pascal program shown in Figure 5.8 shows how the square of variable `N` can be computed by summing the first `N` odd integers.

Figure 5.9 Loop to enter valid data

```
Write('Enter age: ');
Read(Age);
WHILE Age < 0 DO BEGIN
    Write('Re-enter: ');
    Read(Age);
END;
Write('The age is ');
Write(Age:2 );
```

Figure 5.9 illustrates a very useful method for entering valid data. First, the user is prompted to enter the Age. Then, while the Age is negative (not a reasonable value for an age), the user is asked to re-enter the value. When a proper value has been entered, it is echoed to confirm that the intended value was actually received.

Figure 5.10 Loop to force user to order data

```
WriteLn('Enter three values: ');
WriteLn('In increasing order ');
Read(A, B, C);
WHILE (A > B) OR (B > C) DO BEGIN
    WriteLn('Order them ');
    Read(A, B, C);
END (* WHILE *);
WriteLn('Ordered values are: ');
Write(A:3, B:3, C:3);
```

The segment of Pascal program shown in Figure 5.10 is a similar example: the user is prompted to enter three values that are in increasing order. The program does not continue further unless the values are in order. This example illustrates a program that is not very user-friendly. The user should not have to enter data in order, for that could be done easily by the computer (remember we have done it in the Triangle example of Chapter 4)!

Tracing Loops

The tracing of the execution of loops by hand was considered in the Principles book. Tracing loops in this way often provides insight, suggests modifications and is a very effective way to detect errors. It is also a useful way to show relations between variables, and to find loop invariants. However, such tracing by hand is frequently tedious and error-prone. Because it is so labor intensive, it is usually limited to only a few loops. Using the computer to help in the tracing can be equally useful and is much more convenient.

We'll do it on an example algorithm seen in Chapter 5 of the Principles book. We reproduce here the algorithm pseudocode taken from Figure 5.36.

Input Amount, Duration, Payment, Rate

```

    Set Balance to Amount
    Set Time to 0
    While Time < Duration
        Set Interest to Rate × Balance chopped
        Set Balance to Balance + Interest – Payment
        Set Time to Time + 1
    Output Balance

```

This algorithm computes the Balloon Payment of a loan, and Figure 5.11 shows a Pascal version for it. However, an error was introduced during the translation into Pascal, and we have left it here!.

Figure 5.11 Pascal program for Balloon Payment of a loan

```

PROGRAM Loan;
(* Problem of Interest on a loan *)
(* Computes: the Balloon Payment total interest *)

VAR
    Amount, Duration, Payment, Interest,
    Balance, Time, IntSum: INTEGER;
    Rate: REAL;
BEGIN
    (* Input necessary information *)
    Write('Enter amount of loan: ');
    Read(Amount);
    Write('Enter payment amount: ');
    Read(Payment);
    Write('Enter the duration in months: ');
    Read(Duration);
    WriteLn('Enter annual interest rate ');
    Write('as a decimal percent: ');
    Read(Rate);
    Rate := Rate/1200; (* Convert to monthly *)

    (* Compute the Ballon Payment *)
    Balance := Amount;
    IntSum := 0;
    Time := 1;
    WHILE Time < Duration DO BEGIN
        Interest := TRUNC(Balance * Rate);
        Balance := Balance + Interest;
        Balance := Balance - Payment;
        IntSum := IntSum + Interest;
        Time := Time + 1;

        (* Begin trace *****)
        Write(Time: 2);
        Write(' Balance = ');
        WriteLn(Balance: 4);
    END

```

```
      (* End trace ***** *)

END (* WHILE *)

(* Output all required Results *)
Write('Balloon Balance is: ');
WriteLn( Balance:5);
Write('Total interest is : ');
WriteLn( IntSum:5);
END (* Loan *).
```

Notice that, in the Pascal version of Figure 5.11, the input of the data and the output of the results is now a much larger proportion of the whole algorithm. This is typical of programs that have interaction with the user. Notice also that some statements to output tracing information that appear in the Pascal program were not part of the original algorithm. These extra statements have been sandwiched between comments with many asterisks.

If we run that program with the data we used in the trace of the original algorithm, we obtain the following:

```
Enter amount of loan: 600
Enter payment amount: 100
Enter the duration in months: 6
Enter annual interest rate
as a decimal percent: 12
 2 Balance = 506
 3 Balance = 411
 4 Balance = 315
 5 Balance = 218
 6 Balance = 120
Balloon Balance is: 120
Total interest is : 20
```

This output contains information produced by the trace statements that have been added to the program. Notice that this trace “grows” downward, as opposed to the hand-trace, shown in Chapter 5 of the Principles book, which grew towards the right. The final resulting Balloon payment seems reasonable but there are problems!

The trace information shows us that the program executes the loop 5 times only, rather than the required 6 times; furthermore, the count values in the trace start at 2, whereas we expected them to start at 1. The trace output has allowed us to detect an error, an “off-by-one” error, which is very common. A remedy is to initialize the `Time` to zero before entering the loop—as was done in the pseudocode! After this correction, the program produces:

```
Enter amount of loan: 600
Enter payment amount: 100
Enter the duration in months: 6
Enter annual interest rate
as a decimal percent: 12
```



```
1 Balance = 506
2 Balance = 411
3 Balance = 315
4 Balance = 218
5 Balance = 120
6 Balance = 21
Balloon Balance is: 21
Total interest is : 21
```

This time, the loop is executed the correct number of times and the values in the Time field start at the more reasonable value of 1. Notice that the final Balance equals the total interest, which is a nice check.

This program, with its trace, can be run for other values of input like the following which checks the interest-only loan mentioned in Chapter 5 of the Principles book.

```
Enter amount of loan: 600
Enter payment amount: 6
Enter the duration in months: 6
Enter annual interest rate
as a decimal percent: 12
1 Balance = 600
2 Balance = 600
3 Balance = 600
4 Balance = 600
5 Balance = 600
6 Balance = 600
Balloon Balance is: 600
Total interest is : 36
```

A more realistic set of values would involve larger amounts, monthly payments over 5 years, and the resulting trace would have 5×12 or 60 lines. Such large amounts of output in a trace are likely to be overwhelming, so the trace could be modified to output only every fifth value. We could also output some additional information, the values of Interest and IntSum, as a table by inserting the following Selection form:

Figure 5.12 New set of trace statements

```
(* Detailed Trace of program *****)
IF Time = 1 THEN (* Header *)
  WriteLn( 'Num  Bal Int  Sum');
IF (Time MOD 5 = 0) THEN BEGIN
  Write(Time:2);
  Write( Balance:6 );
  Write( Interest:4 );
  Write( IntSum:5 );
  WriteLn;
END (* IF *);
(* End of detailed trace *****)
```

After replacing the previous trace statements of the `Loan` program by the more detailed trace statements of Figure 5.12, the following output was obtained from a run:

```
Enter amount of loan: 12000
Enter payment amount: 200
Enter the duration in months: 60
Enter annual interest rate
as a decimal percent: 12
Num  Bal Int  Sum
 5 11590 116  590
10 11159 112 1159
15 10704 107 1704
20 10229 103 2229
25  9729  98 2729
30  9204  93 3204
35  8650  87 3650
40  8069  81 4069
45  7458  75 4458
50  6816  69 4816
55  6141  62 5141
60  5432  55 5432
Balloon Balance is:  5432
Total interest is :  5432
```

The debugging of programs is often done by judiciously placing such `Write` statements within programs. The statements need not just write values, but could also write messages as in:

```
WriteLn( 'Entering the Input part');
WriteLn( 'Leaving the While Loop ');
```

Other outputs could also involve the loop invariants or the use of a tool called a debugger, which will be introduced later.

5.5 WHILES, REALS, and Errors

The `WHILE` loop is a very convenient form for repeating actions. Some actions however may be approximations, especially when dealing with `REAL` values, and, when repeated often, may result in significant errors. The following example shows one such problem involving `REAL` values.

The Pascal program `BadChanger` in Figure 5.13 is a very crude change making program whose method was considered briefly in Chapter 4 of the Principles book (Figure 4.30).

Figure 5.13 A bad change program

```
PROGRAM BadChanger;
(* A bad way to make change *)

VAR Pennies: INTEGER;
    Tendered, Cost, Remainder: REAL;
BEGIN
    (* Input necessary information *)
    Write('Enter cost of item: ');
    Read(Cost);
    Write('Enter amount tendered: ');
    Read(Tendered);

    (* Compute the change in pennies *)
    Remainder := Tendered - Cost;
    Pennies := 0;
    WHILE Remainder > 0 DO BEGIN
        Remainder := Remainder - 0.01;
        Pennies := Pennies + 1;
    END (* WHILE *);

    (* Output all required Results *)
    Write('Cost is: ');
    Write(Cost: 4:2);
    Write(' Amount tendered is: ');
    Write(Tendered: 4:2);
    Write(' Change is: ');
    WriteLn(Pennies: 3);
END (* BadChanger *).
```

In this program, a certain amount is `Tendered` for an item with a given `Cost`, and the difference is the `Remainder`. Its (unfortunate) way of making change is to repeatedly increment a `Pennies` counter and decrease the `Remainder` by 0.01 as long as the `Remainder` is greater than zero. The intent is that, when the loop is completed, the number of pennies should equal the original `Remainder`, because a penny was added to `Pennies` for each penny—\$0.01—

that is subtracted from `Remainder`. However this is not always true as can be seen from the following sample runs. Many of these runs are off by a penny!

```
Enter cost of item: 0.72
Enter amount tendered: 1.00
Cost is: 0.72 Amount tendered is: 1.00 Change is: 28
```

```
Enter cost of item: 0.61
Enter amount tendered: 0.75
Cost is: 0.61 Amount tendered is: 0.75 Change is: 14
```

```
Enter cost of item: 0.43
Enter amount tendered: 1.00
Cost is: 0.43 Amount tendered is: 1.00 Change is: 58
```

```
Enter cost of item: 0.75
Enter amount tendered: 2.00
Cost is: 0.75 Amount tendered is: 2.00 Change is: 126
```

```
Enter cost of item: 0.25
Enter amount tendered: 1.00
Cost is: 0.25 Amount tendered is: 1.00 Change is: 76
```

The reason for this error is related to a common problem of `REAL` values. `REAL` values such as 0.01 can only be approximated within a computer, so some very slight error may result. In fact, when 0.01 is converted to binary, the equivalent binary value is a repeating number (0.0001100110011...0011...), which eventually must be approximated. This leads to an error, which could be very small, but it is nevertheless an error. Then, when the subtraction of the 0.01 is repeated, this error accumulates and results in a larger error. In this case the final error is rather small, only a penny; but the error is inconsistent, occurring only some of the time. In other algorithms, the error could “grow” and become very serious.

Not all computers represent `REAL` values to the same degree of approximation so that the effects of this error are likely to differ from one Pascal system to another. Such errors and inconsistencies cannot be tolerated. There are solutions to this problem.

It is possible to avoid this error in various ways. One way is to simply avoid `REAL` values and declare `Cost`, `Tendered`, and `Remainder` to be `INTEGER`. The values would be considered as cents, rather than hundredths of a dollar. The user of the program would need to be prompted to enter values as “whole” numbers, or cents.

Another way would be to allow the input of `REAL` values, but to convert them into `INTEGER` values in the program. This could be done by multiplying the `REAL` values by 100 and then rounding the result as follows:

```
(* Enter Real money and convert it into Integer *)
Write('Enter the cost with a decimal point: ');
Read(RealCost);
IntCost := ROUND(100 * RealCost);
```

The use of REAL values also poses other problems that can be avoided if they are anticipated. For example, two REAL values should not be compared for equality because the value of $0.1 * 10$ may not exactly equal 1.0 . Instead, the difference between the two values could be compared to determine how small it is, i.e., how close is the approximation. For example, to determine if a triangle is a right triangle the condition:

$$(A*A + B*B) = C*C$$

should be replaced by the following comparison with a small value `Err`:

$$(A*A + B*B - C*C) <= Err$$

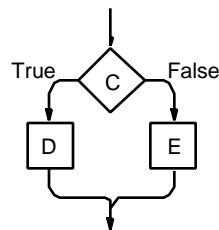
or alternatively, if the magnitude of error is more important, the comparison can be replaced by:

$$ABS(A*A + B*B - C*C) <= Err$$

5.6 Selection Forms in Pascal

The Selection Form is illustrated by the flowchart of Figure 5.14.

Figure 5.14 Flowchart for the Selection Form



This Form is interpreted in the following manner: if condition `C` is true then action `D` is performed, otherwise action `E` is performed.

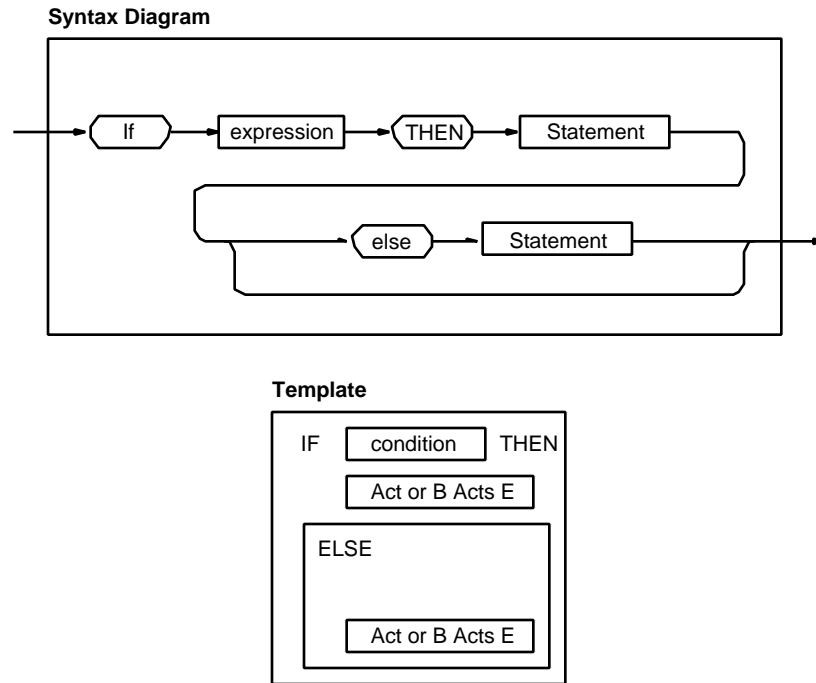
Figure 5.15 The Selection Form in Pascal

Figure 5.15 shows at left the syntax diagram, and at right the template for the Pascal version of the Selection Form. Notice that, in the syntax diagram, the ELSE-part is optional. This is also shown in the template where the optional ELSE-part is shaded.

These actually define two forms of the Selection called IF-THEN-ELSE and IF-THEN, which are:

<pre> IF expression THEN Statement ELSE Statement </pre>	<pre> IF expression THEN Statement </pre>
--	---

It is important to realize that, just as it was the case with the WHILE statement, the Statement in both the THEN-part and the ELSE-part could be any statement, that is to say:

1. A simple statement, e.g. an assignment or input-output statement.
2. A compound statement, e.g. a sequence of statements enclosed in a BEGIN-END pair,
3. Another Selection Form
4. A Repetition Form

and that each of these could, in turn, include other forms.

It is also very important to realize that **there cannot be a semicolon just before an ELSE!** If this was the case, this semicolon would mark the end of the

preceding simple IF-THEN form. Such an extra semicolon in the middle of an IF statement is the source of many errors!

The following three sample segments of Pascal code show some simple Selection forms.

```
IF Age < 18 THEN
  Minors := Minors + 1;
```

This is a simple Selection having no ELSE part; it increments Minors each time the Age is less than 18.

```
IF Age < 0 THEN
  WriteLn('Error')
ELSE BEGIN
  Count := Count + 1;
  Sum   := Sum + Age;
END;
```

This second code segment is an example of the second kind of Pascal Selection; it has an ELSE part that consists of more than one statement thus requiring the BEGIN-END pair. Note the absence of semicolon before the ELSE.

```
IF This > That THEN BEGIN
  Max := This;
  Min := That;
END
ELSE BEGIN
  Max := That;
  Min := This;
END;
```

In this third example, both the THEN and the ELSE parts have compound statements and thus, two BEGIN-ELSE pairs are required. Note the chosen indentation that shows clearly what statements belong to each part.

This last example determined the maximum and the minimum of two values. There are alternative ways of performing the same operation as shown below. Both manners can be proven equivalent by the methods seen in Chapter 5 of the Principles book.

```
Max := This;
Min := That;
IF Max < Min THEN BEGIN
  (* Swap *)
  Temp := Max;
  Max  := Min;
  Min  := Temp;
END;
```

```
Max := This;
Min := That;
IF Max < Min THEN BEGIN
  Max := That;
  Min := This;
END;
```

5.7 More Selections: Combinations of Selection Forms

It is possible to combine Selection Forms in various arbitrary manners. We will consider here some nests of Selections, both simple and complex.

Figure 5.16 Max3 algorithm and code

<i>If First < Second</i>	IF First < Second THEN
<i>If Second < Third</i>	IF Second < Third THEN
<i>Set Big to Third</i>	Big := Third
<i>Else</i>	ELSE
<i>Set Big to Second</i>	Big := Second
<i>Else</i>	ELSE
<i>If First < Third</i>	IF First < Third THEN
<i>Set Big to Third</i>	Big := Third
<i>Else</i>	ELSE
<i>Set Big to First</i>	Big := First;

Figure 5.16 shows algorithm Max3 that finds the maximum of three values. It consists of three Selections, two nested within one larger Selection. On the left of the figure is shown the pseudocode representation of the algorithm and, on the right, is the corresponding Pascal code. Notice how closely the two versions match; this is an indication of how easy it can be to convert pseudocode into Pascal.

Confusion in Choices

Notice that, in the Pascal program for Max3, there is only one semicolon, at the very end, which marks the end of the outer Selection. In Pascal, it often seems as though some programs could also be written using extra semicolons, but Pascal is not very “forgiving” of such excesses. When in doubt, don't! It is especially important not to put a semicolon just before an ELSE.

Confusion could possibly arise in the case where an IF is nested inside an IF without an intervening ELSE, as in the following:

```
IF A THEN IF B THEN C ELSE D;
```

Humans could unfortunately read and interpret this in to different ways as follows:

Proper form	Improper form
IF A THEN	IF A THEN
IF B THEN	IF B THEN
C	C
ELSE	ELSE
D;	D;

Pascal associates the lonely ELSE with the nearest preceding IF, as shown on the left, that is, D is performed if A is true and B is false. If the interpretation on the right is intended, that is, D is performed if A is false, irrespective of the value of B, then a BEGIN and END must be inserted around the inner IF and the code fragment must be written in the following manner:

```

IF A THEN BEGIN
  IF B THEN
    C
  END
ELSE
  D;

```

More Nesting of Choices

Figure 5.17 shows four algorithms for an Age Tally System that increments two counters, High and Low depending on the Age. The four algorithms are equivalent in behavior; the proofs of equivalence are similar to those found in Chapter 5 of the Principles book. The algorithms differ somewhat in structure, and lead to different programs. Notice especially the indentation. Algorithms 1 and 2 show the second Selection nested in two ways. In Algorithm 1, this Selection is nested in the “true” of the main Selection whereas in Algorithm 2 it is in the “false” leg.

Figure 5.17 Four different ways to nest choices

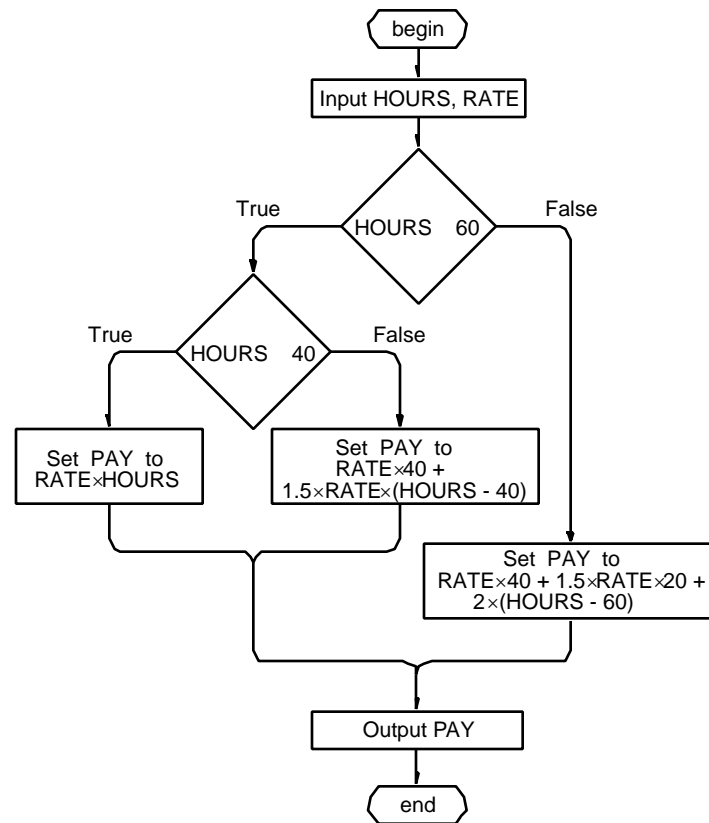
Algorithm 1	Algorithm 2
<pre> If Age < 12 If Age > 21 Increment High Else Increment Low IF Age >= 12 THEN BEGIN IF Age > 21 THEN High := High + 1; END ELSE Low := Low + 1; </pre>	<pre> If Age < 12 Increment Low Else If Age > 21 Increment High IF Age < 12 THEN Low := Low + 1 ELSE IF Age > 21 THEN High := High + 1; </pre>
Algorithm 3	Algorithm 4
<pre> If Age < 12 Increment Low Else If Age > 21 Increment High </pre>	<pre> If Age < 12 Increment Low If Age > 21 Increment High </pre>

<pre>IF Age < 12 THEN Low := Low + 1 ELSE IF Age > 21 THEN High := High + 1 ELSE ; (* do nothing *)</pre>	<pre>IF Age < 12 THEN Low := Low + 1; IF Age > 21 THEN High := High + 1;</pre>
---	--

Nesting in the false branch, as in Algorithm 2, is usually preferred because it is “top-down” and easier to read. It is also easier to extend. For example, to increment a Mid age counter the additional cases grow downwards, and this avoids the above mentioned confusion about matching the IFs and the ELSEs. Algorithm 3 shows yet another more detailed way of doing the nesting in the false branch; the Pascal code explicitly includes the case where no action is taken. Notice that the Pascal “no action” statement, generally known as the *null statement*, is just a semicolon.

Alternative Ways to Code Selections

There are many different ways to code Selections, and some ways are better than others. These possibilities will be illustrated by coding the extended payroll algorithm of Chapter 2 of the Principles book. This algorithm is shown in Figure 5.18 in both flowchart and pseudocode forms. Notice that the inner Selection is nested within the left or true “leg” of the outer Selection form.

Figure 5.18 Flowchart and pseudocode for extended pay algorithm

The Pascal code corresponding to the Selection part of this extended pay algorithm is

```

IF Hours <= 60 THEN BEGIN
  IF Hours <= 40 THEN
    Pay := Rate * Hours
  ELSE
    Pay := Rate * 40 + 1.5 * Rate * (Hours - 40);
  END
ELSE
  Pay := Rate*40 + 1.5*Rate*20 + 2*Rate*(Hours-60);

```

In this fragment of Pascal code the nested Selection is indented, and this careful use of indentation helps make the structure of the algorithm easily visible, and is of great help to readers. Since, at any point in a Pascal program where we can have one blank, we can also have an arbitrary number of blanks or new lines, such indentation does not change the program.

The way in which the Selections of this algorithm have been nested requires the Pascal equivalent to have an extra BEGIN-END pair. This solution is not extremely good, but can be improved as follows.

If the condition of the outermost Selection form is reversed, the nested Selection is moved to the false branch. The pseudocode for this modified algorithm is the following:

```
If Hours > 60
    Set Pay to Rate × 40 + 1.5 × Rate × 20 +
        2 × Rate × (Hours - 60)
Else
    If Hours ≤ 40
        Set Pay to Rate × Hours
    Else
        Set Pay to Rate × 40 +
            1.5 × Rate × (Hours - 40)
```

Here, the condition `Hours ≤ 60` has been reversed to the complement condition `Hours > 60` and the actions on the two “legs” of the Selection have also been changed. The smaller Selection is now nested in the Else leg. This structure translates into Pascal as the following:

```
IF Hours > 60 THEN
    Pay := Rate*40 + 1.5*Rate*20 + 2*Rate*(Hours-60)
ELSE
    IF Hours ≤ 40 THEN
        Pay := Rate * Hours
    ELSE
        Pay := Rate * 40 + 1.5 * Rate * (Hours - 40);
```

Notice that the omission of the BEGIN-END pair makes the logic easier to understand. The major barrier to comprehension here is all the explicit numbers, whose meaning needs to be kept in mind. This can be alleviated in a complete program through the use of named constants, declared in a CONST section at the beginning of the program. The `ExtendedPay` Pascal program, given in Figure 5.19, shows how this might be done.

Figure 5.19 The Extended Pay program

```
PROGRAM ExtendedPay;
{ An Extended Payroll program
}
{ Selection nested in Else branch of outer
Selection }

CONST RegularRate      = 10;
      ExtraRate        = 15;
      DoubleRate       = 20;
      BaseHours        = 40;
      ExtraHours        = 20;
      DoubleTimeStart  = BaseHours + ExtraHours;
      BasePay          = RegularRate * BaseHours;
      ExtraPay         = ExtraRate * ExtraHours;

VAR Hours, Pay: INTEGER;
```

```

BEGIN
    Write('Enter hours');
    Read(Hours);
    WriteLn;

    IF Hours > DoubleTimeStart THEN
        Pay := BasePay + ExtraPay +
            DoubleRate * (Hours -
DoubleTimeStart)
    ELSE
        IF Hours <= BaseHours THEN
            Pay := RegularRate * Hours
        ELSE
            Pay := BasePay +
                ExtraRate * (Hours -
BaseHours);

        Write('The gross pay is ');
        Write(Pay: 5);
        WriteLn;

    END. { ExtendedPay }

```

5.8 Select Form: Handling Many Branches

The real subject of the discussion in the last section was how to present many different flow paths as clearly as possible. Although the `ExtendedPay` algorithm had only three possible paths through it: over 60 hours, between 40 and 60 hours, and up to 40 hours, the solution was not obvious. This problem is actually complicated by some restrictions on structure imposed by the syntax of Pascal. The conclusion that we reached was that it was better to nest the inner Selection in the Else branch of the outer Selection. To see that this applies generally, we'll expand the `ExtendedPay` algorithm by verifying that the number of hours input is reasonable—not more than the number of hours in a week! The pseudocode for this new version is:

```

    If Hours > Hours in Week
        Error
    Else
        If Hours > 60
            Double pay
        Else
            If Hours > 40
                Time and Half
            Else
                Regular

```

Notice that this structure has a uniform pattern that can easily be extended to cover more branches. In this structure, the conditions are tested in the order given and whenever one is true, the corresponding action is done and any following conditions and actions are ignored; so the next action begins after the very end of this form.

If we take the Selection structure of the pseudocode and translate it to Pascal, we obtain the following:

```
IF Hours > HoursInWeek THEN
  (* Error message *)
ELSE
  IF Hours > DoubleTimeStart THEN
    (* Compute pay including double time hours *)
  ELSE
    IF Hours > ExtraTimeStart THEN
      (* Compute pay including extra time hours *)
    ELSE
      (* Compute pay for regular time hours only *)
```

One thing is clear, although this structure is consistent and can be readily be extended, it has its limitations. Because of our indentation rules, the more branches we have, the more we indent and the space for the statements gets shorter and shorter. If we had 20 possible branches, we would be indenting 60 spaces and have almost no room left on a line. We therefore recognize the regularity of the structure and modify the indentation rules to obtain the following:

```
IF Hours > HoursInWeek THEN
  (* Error message *)
ELSE IF Hours > DoubleTimeStart THEN
  (* Compute pay including double time hours *)
ELSE IF Hours > ExtraTimeStart THEN
  (* Compute pay including extra time hours *)
ELSE
  (* Compute pay for regular time hours only *)
```

This structure can be extended to arbitrarily many branches. It is common to refer to the conditions in the Selections as *selectors*, there is one selector for each branch. When we use this select structure, it is crucial to remember that the selectors are tested in the order shown and that the branch is selected by the first selector to be true. The action that follows the last ELSE is the one that is performed when none of the selectors is true. Figure 5.20 shows two Pascal versions of the same algorithm. The purpose of that figure is to show that there is more than one possible ordering for the branches, as long as the selectors are specified properly.

Figure 5.20 Two versions of a Grade program

Write('Enter percentage');	Write('Enter percentage');
Read(Percent);	Read(Percent);
WriteLn;	WriteLn;
Write('The grade is ');	Write('The grade is ');

IF Percent<DLimit THEN	IF Percent >= ALimit THEN
Write('F');	Write('A');
ELSE IF Percent<CLimit THEN	ELSE IF Percent>=BLimit THEN
Write('D');	Write('B');
ELSE IF Percent<BLimit THEN	ELSE IF Percent>=CLimit THEN
Write('C');	Write('C');
ELSE IF Percent<ALimit THEN	ELSE IF Percent>=DLimit THEN
Write('B');	Write('D');
ELSE	ELSE
Write('A');	Write('F');

The values for the named constants, ALimit, BLimit, etc. would be defined elsewhere in the CONST declarations.

Finally, yet another version of the ExtendedPay program shows a very different structure and formulas compared with the previous examples. This new version is shown in Figure 5.21.

Figure 5.21 A new version of Extended Pay

```

PROGRAM NewExtendedPay;
{ An Extended Payroll program                               }
{ Decision nested in Else branch of outer Decision        }
{ Normal indentation                                       }
}

CONST RegularRate      = 10;
      ExtraIncrease     = 5;
      DoubleIncrease    = 5;
      BaseHours         = 40;
      ExtraHours        = 20;
      HoursInWeek       = 7 * 24;
      DoubleTimeStart   = BaseHours + ExtraHours;

VAR Hours, Pay: INTEGER;

BEGIN
    Write('Enter hours: ');
    Read(Hours);
    WriteLn;

    IF Hours >= 0 THEN { regular pay}
        Pay := RegularRate * Hours;

    IF Hours > BaseHours THEN { add extra pay }
        Pay := Pay + ExtraIncrease * (Hours - BaseHours);

    IF Hours > DoubleTimeStart THEN { add double pay }
        Pay := Pay + DoubleIncrease *
                (Hours - DoubleTimeStart);

```

```
IF (Hours < 0) OR (Hours > HoursInWeek) THEN
    Pay := 0;

WriteLn('The gross pay is ', Pay: 5);

END. { NewExtendedPay }
```

In this new version, the pay computation is done in an entirely different manner. First the regular rate is applied, then the overtime rate is added if necessary, then the double time rate is added if necessary, and finally the pay is reset to zero if the number of hours is invalid. In this version the Selections are not nested, which means that all the conditions are checked all the time, which is not the case with nested selections.

5.9 Awkward Nests: General Nesting

Not all Selections nest as neatly as the examples we have shown in the previous section. Sometimes it is very awkward to nest the Selections so that the algorithm is easy to understand. As an example of this situation consider the triangle classification algorithm that was discussed in Chapter 4 of the Principles book, and shown in the last chapter. Figure 5.22 shows the original algorithm and a new Pascal version.

Figure 5.22 The Triangle algorithm and program

Input Sides A, B, C	IF (A+B)<=C THEN
Sort sides so A <= B <= C	Write('Not a triangle')
	ELSE
If A + B < C then	IF ABS(A-C)<Err
Output "Not a triangle "	THEN
else	
if A = C then	Write('Equilateral')
Output "Equilateral"	ELSE BEGIN
else	IF
if (A=B) or (B=C) then	(ABS(A-B)<Err) OR
Output "Isosceles "	(ABS(A-C)<Err) THEN
if A + B = C + C	
then	Write('Isosceles');
Output "Right triangle"	IF
else	
Output "Triangle"	ABS(A*A+B*B-C*C)<Err
	THEN
	Write('Right triangle')
	ELSE
	Write('triangle')
	END

The structure of the heart of this algorithm consists of a sequence of two Selections nested within a Selection form, that is itself nested within yet another Selection. The corresponding piece of Pascal program is given to the right of the pseudocode in the figure. Notice especially the indentation: there are three levels. Notice also that only one semicolon is needed, that is to separate the two Selections in sequence. Two other semicolons could be added, can you guess where?.

As we saw earlier, `REAL` numbers should never be compared for equality. This is why in that version of the `Triangle` program, we test for approximate equality by comparing the absolute value of the difference with an acceptable error, `Err`. When the difference is less than `Err`, we say that the two values are sufficiently close to be taken as equal. The admissible error must be specified by the programmer as some small quantity such as:

```
CONSTANT Err = 0.0001;
```

Mixed Nests: Repetition and Selections

Programs that involve combinations of Repetitions and Selections nested in one another are very common. The actual creation of algorithms that make use of such nested combinations is considered in detail in Chapter 6 of the Principles book, but the coding of such nests is straightforward, as is shown in the example of Figure 5.23.

Figure 5.23 A Change Maker program

<i>Input Cost</i>	Write('Enter cost: ');
<i>Input Tendered</i>	Read(Cost);
	Write('Enter amount tendered: ');
<i>Set Change to Tendered - Cost</i>	Read(Tendered);
<i>Set Quarters to 0</i>	Change := Tendered - Cost;
<i>Set Nickels to 0</i>	Quarters := 0;
<i>Set Pennies to 0</i>	Nickels := 0;
<i>While Change > 0</i>	Pennies := 0;
<i>{Loop invariant:</i>	
<i>Tendered = Pennies +</i>	WHILE Change > 0 DO
<i>5 x Nickels +</i>	
<i>25 x Quarters +</i>	(** Loop invariant: **)
<i>Change}</i>	(** Tendered = Pennies + **)
<i>If Change 25 then</i>	(** 5*Nickels + 25*Quarters + **)
<i>Set Quarters to Quarters + 1</i>	(** Change **)
<i>Set Change to Change - 25</i>	
<i>Else</i>	IF Change >= 25 THEN BEGIN
<i>If Change 5 then</i>	Quarters := Quarters + 1;
<i>Set Nickels to Nickels + 1</i>	Change := Change - 25;
<i>Set Change to Change - 5</i>	END
<i>Else</i>	ELSE
<i>Set Pennies to Pennies + 1</i>	IF Change >= 5 THEN BEGIN
<i>Set Change to Change - 1</i>	Nickels := Nickels + 1;
<i>Output Quarters</i>	Change := Change - 5;
<i>Output Nickels</i>	END
<i>Output Pennies</i>	ELSE BEGIN
	Pennies := Pennies + 1;
	Change := Change - 1;
	END; { nested IF and WHILE}
	WriteLn('Quarters = ', Quarters);
	WriteLn('Nickels = ', Nickels);
	WriteLn('Pennies = ', Pennies);

The example of Figure 5.23 is a change making algorithm corresponding to the fourth version of Change Maker developed in Chapter 5 of the Principles book. On the left of the figure is the algorithm in pseudocode. This “Change by Selections” consists of a Loop with double-nested Selections within it. Notice that the loop invariant is shown in the pseudocode. The body of the equivalent Pascal program is shown at the right of the pseudocode. Notice the indentation of the IF statements within the WHILE statement.

In such mixed, nested program constructs, it is often convenient and helpful to comment the ENDS of each of the compound statements as, for example

```
END (* of IF *)
```

or

```
END (* of WHILE *).
```

Notice also the spacing of the statements: we've left a gap around the entire IF form.

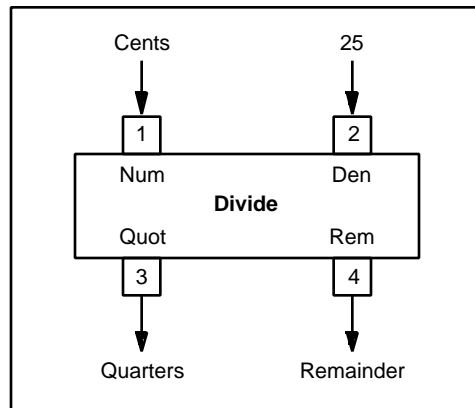
Assertions, such as the loop invariant, are shown as comments with two asterisks beginning with (** and ending with **) to mark a difference from other comments.

Style matters, such as the above comments, space gaps, and indentation are not required in Pascal, but are most useful to readers of the program. Remember, programs are read more often than they are written!

5.10 Subprograms: Using Subprograms as Black Boxes

Up to now we have considered subprograms to be black boxes, that is, single actions, that transform inputs into outputs. This view has allowed us to use such large actions very early in learning the practice of programming. In particular the input/output actions of Pascal are enclosed in subprograms `Read`, `ReadLn`, `Write`, and `WriteLn`. In Chapter 4, we've also used a subprogram from a Library (`Order3`), and we also have had a peek at what a Pascal subprogram looked like (`TurnOfPlayer`). Later, we will look inside the boxes in detail to "see how they work", and, eventually, we will create our own. Until then, we will mainly use subprograms that are available from Libraries. The black boxes or subprograms were constructed as procedures and functions in Pascal that were grouped to form libraries. A library in Pascal is called a *unit*. Here, we will use the unit `IntLib` that comprises operations on Integers.

To make use of black boxes requires knowledge of only a few rules. The main concept is the distinction between inputs and outputs, which is obvious once a dataflow diagram has been drawn. Inputs have arrows going into a dataflow box, and outputs have arrows coming out of the box. For example, in the `Divide` box of Figure 5.24, there are two inputs labeled `Num` and `Den` (short for Numerator and Denominator) and two outputs labeled `Quot` and `Rem` (for Quotient and Remainder). A subprogram can have any number, including none, of inputs and outputs or *parameters*, depending upon what it needs to perform its function. In addition to `Divide`, which has two inputs and two outputs, we shall see subprograms with inputs but not outputs, others with outputs but no inputs and even some with neither inputs nor outputs. Usually, to avoid complexity, the numbers of inputs and outputs is kept small.

Figure 5.24 Dataflow diagram for the Divide subprogram

When we use a subprogram as an action, we say that we are “invoking” or “calling” the subprogram. When this happens, certain data in the program that is using the subprogram, usually referred to as the “caller” are connected with the inputs and outputs of the subprogram. Inputs may be connected to a variable or a constant, but outputs cannot be connected to a constant (because the output value needs to be put somewhere and the value of a constant can’t be changed). Inputs of various boxes may be connected together; the same variable may connect to different inputs. Outputs may not be connected together because one output may have a different value than another, but no variable can have two different values at the same time.

Figure 5.25 Definition of Divide subprogram

Divide (integer divide)

imported from IntLib Library has 4 parameters all of type
INTEGER. Called by:

Divide(Num, Den, Quot, Rem);

where the Input parameters are:

1. Num, for Numerator
2. Den, for Denominator

and the Output parameters are:

3. Quot, for Quotient
4. Rem, for Remainder.

Figure 5.25 defines the Divide subprogram, a very commonly used operation. It is defined by the way in which it is invoked:

Divide(Num, Den, Quot, Rem);

The four parameters which define such an action are called formal parameters. They could have been defined in some other order but, once defined, the order of definition is important. We’ve also shown this parameter numbering on each of the arrows on the dataflow diagram of Figure 5.24 to avoid confusion. In the case of Divide the first two parameters are inputs and the last two are outputs.

Calling or invoking such an action simply involves replacing the formal parameters by some actual parameters. For example, to convert a given number of cents into the equivalent number of quarters and pennies we need simply write:

```
Divide(Cents, 25, Quarters, Pennies);
```

Similarly, a given number of ounces can be converted into the equivalent number of pounds and ounces (remember: 16 ounces equal one pound) by:

```
Divide(Ounces, 16, Pounds, Ounces);
```

Notice especially that `Ounces` here is both an input and an output. This is natural in Pascal, but may be a problem in other programming languages.

Figure 5.26 Dataflow diagram for Subtract subprogram

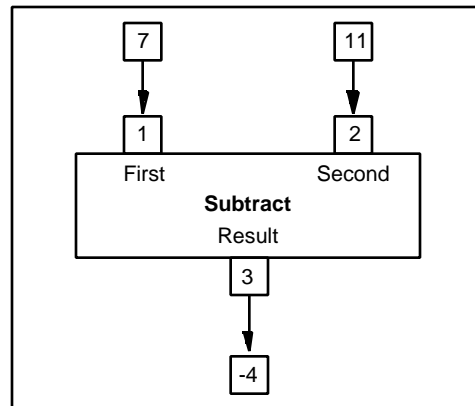


Figure 5.26 shows the dataflow diagram for subprogram `Subtract`. It is defined with the following form:

```
Subtract(First, Second, Result);
```

and subtracts the value of the `Second` parameter—the subtrahend—from the `First` parameter—the minuend—to yield the difference as the `Result`.

Subprograms `Add` and `Multiply` have the same form as `Subtract`, that is, they have two inputs and one output. They are all available from Library `IntLib`.

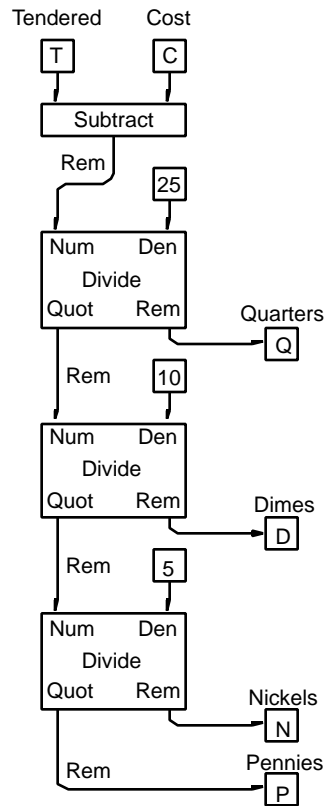
The square of any integer value `Val` can be created by connecting the two inputs to `Multiply` together as:

```
Multiply(Val, Val, Square);
```

Figure 5.27 shows a complete program in Pascal demonstrating the use of the subprograms `Divide` and `Subtract`, which are imported from the Library called `IntLib`. We will actually create this Library, and its contents later, but now we'll merely use it. The dataflow diagram on the left of the figure shows the dataflow through the computation part of the program, which corresponds to the boxed part of the Pascal program on the right. The dataflow diagram

corresponds to the diagram that was presented in Figure 3.32 of Chapter 3 of the Principles book.

Figure 5.27 Dataflow diagram and Pascal program for the SubChange algorithm



```

PROGRAM SubChange;
{ ChangeMaker using procedures      }
{ imported from the library IntLib }

USES
    IntLib; { For Divide and Subtract }

VAR
    Cost, Tendered, Remainder,
    Pennies, Nickels, Dimes,
    Quarters:      INTEGER;

BEGIN
    { Input Cost and Tendered amounts }
    WriteLn('Enter both the cost ');
    WriteLn('and the amount tendered ');
    WriteLn('as cents: ');
  
```

```

Read(Cost, Tendered);

{ Make the change using subprograms }
Subtract(Tendered, Cost, Remainder);
Divide(Remainder, 25, Quarters, Remainder);
Divide(Remainder, 10, Dimes, Remainder);
Divide(Remainder, 5, Nickels, Pennies);

{ Output the resulting coin counts }
WriteLn('The change is ');
WriteLn(Quarters: 2, ' quarters');
WriteLn(Dimes: 2, ' dimes');
WriteLn(Nickels: 2, ' nickels');
WriteLn(Pennies: 2, ' pennies');

END.

```

The program `SubChange` finds the change from a given amount for an item of a certain cost. Notice that the main part of the program consists of the four calls to the imported actions `Subtract` and `Divide`. This program can be written in other ways using `Repetitions`, etc. If you recall we already defined a `Change Maker` program in Chapter 4 (Figure 4.9), and also in this chapter (Figure 5.24). If you compare `SubChange` to these, you'll note that it is shorter and simpler.

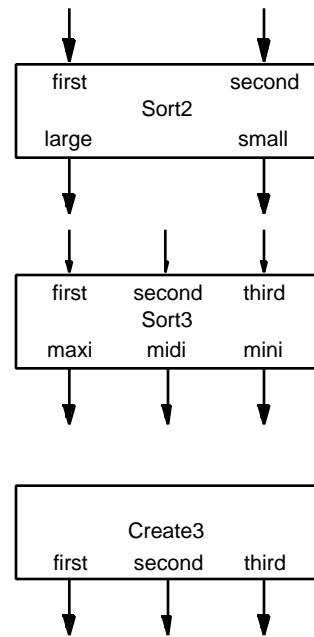
The ShortSort Library

It would be useful to have subprograms in a Library that sorted two and three variables. For example, the procedure `Sort2(P, Q, L, S)` could act on any two integers `P, Q` and produce outputs that are ordered: `L` the larger and `S` the smaller. Similarly procedure `Sort3(A, B, C, L, M, S)` would output the three integer values of `A, B, and C` ordered in `L` (large), `M` (middle) and `S` (small). Subprograms that sorted four or five variables could be created using these smaller sort subprograms as building blocks. These sort subprograms are known as "short sorts" for they involve only a few variables. Later, subprograms that can sort any number of values will be done using arrays.

In Pascal, libraries are built as units. A Pascal unit is made of two parts: the *Interface* and the *Implementation*. Interfaces are specifications of groups of data items and actions that are available from units. These interface specifications describe *what* is available, not *how* it is constructed. They show the names of subprograms, the parameters that are passed in and passed out, and a brief statement of the behavior of the subprograms. Along with each Interface part there is an Implementation part that actually defines in detail what the interface part describes generally, i.e. how the data structures and the operations are implemented.

Definition of `ShortSortLib` (informal)

Defines sort subprograms of 2 and 3 variables and a subprogram for generating test data for these subprograms.

Figure 5.28 Informal interface specifications for Sort2, Sort3 and Create3 subprograms

```
Sort2(First, Second, Large, Small)
```

Sorts two integer values.

Input: First, Second

Output: Large, Small

```
Sort3(First, Second, Third,
      Maxi, Midi, Mini)
```

Sorts three integer values

Input: First, Second, Third

Output: Maxi, Midi, Mini

```
Create3(First, Second, Third)
```

Generates three values (from 0, 1, 2) each time it is called. It repeats after 27 calls.

Input: none

Output: First, Second, Third

Figure 5.28 shows an informal interface for ShortSortLib in the form of text and diagrams. In the corresponding Pascal code, the first procedure, Sort2 would be described in the interface by:

```
PROCEDURE Sort2(    P, Q: INTEGER;  { pass-in }
                  VAR L, S: INTEGER); { pass-out }
```

In the procedure heading, the word VAR precedes the parameters that are passed out. In the data flow diagrams this means that whenever the diagrams

have arrows leaving them, a corresponding VAR appears in the procedure heading of the interface definition. This graphic view makes it clearer; parameters that are passed in have no VAR preceding them, parameters that are passed out are preceded by VAR. We need not be further concerned about these VARs now, for we are not creating units, we are using already created ones. Later we will create our own.

Figure 5.29 Program to test ShortSortLib and its output

PROGRAM ShortSortTest;	Data	Sort3	Check
{ Test ShortSortLib }	A B C	L M S	L M S
	0 0 0	0 0 0	0 0 0
USES ShortSortLib;	0 0 1	1 0 0	1 0 0
	0 0 2	2 0 0	2 0 0
VAR A, B, C, D, E, F,	0 1 0	1 0 0	1 0 0
I, J, K, L, M, S: INTEGER;	0 1 1	1 1 0	1 1 0
	0 1 2	2 1 0	2 1 0
BEGIN	0 2 0	2 0 0	2 0 0
WriteLn(' Data Sort3 Check');	0 2 1	2 1 0	2 1 0
WriteLn(' A B C L M S L M S');	0 2 2	2 2 0	2 2 0
I := 27;	1 0 0	1 0 0	1 0 0
	1 0 1	1 1 0	1 1 0
WHILE I <> 0 DO BEGIN	1 0 2	2 1 0	2 1 0
Create3(A, B, C);	1 1 0	1 1 0	1 1 0
Write(A: 2, B: 2, C: 2);	1 1 1	1 1 1	1 1 1
Sort3(A, B, C, L, M, S);	1 1 2	2 1 1	2 1 1
Write(' ');	1 2 0	2 1 0	2 1 0
Write(L: 2, M: 2, S: 2);	1 2 1	2 1 1	2 1 1
	1 2 2	2 2 1	2 2 1
Sort2(A, B, D, E);	2 0 0	2 0 0	2 0 0
Sort2(E, C, F, S);	2 0 1	2 1 0	2 1 0
Sort2(D, F, L, M);	2 0 2	2 2 0	2 2 0
Write(' ');	2 1 0	2 1 0	2 1 0
Write(L: 2, M: 2, S: 2);	2 1 1	2 1 1	2 1 1
	2 1 2	2 2 1	2 2 1
WriteLn;	2 2 0	2 2 0	2 2 0
I := I - 1;	2 2 1	2 2 1	2 2 1
END;	2 2 2	2 2 2	2 2 2
END.			

Figure 5.29 shows a Pascal program to test our ShortSortLib library, and the results obtained by executing it. Note the `USES ShortSortLib;` at the beginning of the program, that makes it possible for it to use procedures from ShortSortLib. This program performs the test by using `Create3` to generate all possible combinations of the data values 0, 1 and 2. Each time, the program prints the combination of values. For each of these combinations it calls `Sort3` to put the values into order into three variables L, M and S, whose values are then printed under the column labeled Sort3. As a comparison, the test program then uses `Sort2` and the original values to simulate the action of `Sort3`, again putting the results into L, M and S, and printing their values under

the column labeled Check. The correctness of Sort2 and Sort3 are then verified by comparing the values of L, M and S in the Sort3 and Check columns. Another Pascal program, Grader, shown in Figure 5.30, is a “friendly” forgiving average program. It simply uses the ShortSortLib subprograms to average the best and mid grades and forget the worst.

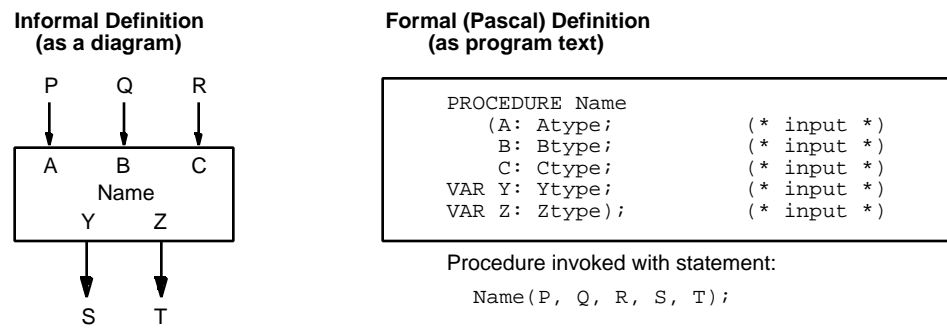
Figure 5.30 Program Grader

```
PROGRAM Grader;
(* Forgiving Grading program *)
USES
    ShortSortLib; { for Sort3 }
VAR
    Midterm, Final, Projects,
    Best, Mid, Worst, Grade: INTEGER;
BEGIN
    WriteLn( 'Enter three percentages: ');
    Read( Midterm, Projects, Final ); WriteLn;
    Sort3( Midterm, Projects, Final, Best, Mid,
    Worst );
    Grade := (Best + Mid) DIV 2;
    Write( 'The "forgiven" mean is ', Grade:3 );
END. { Grader }
```

Notice particularly that we were able to create a program, such as Grader, that uses these subprograms, even though we only knew the Interface part, and had no knowledge of the inner details from the Implementation part. Of course we must first specify in a USES clause that we wish to use the operations from this Library before we can use them. Using these short sorts, we could similarly create program segments to sort 4 variables (in two different ways) and 5 variables (more than 4 ways). Try it.

Notation for Defined Procedures

Procedures may be defined very conveniently by using dataflow diagrams. Arrows into the diagrams show input parameters, and arrows coming out of the diagram show output parameters. The names of the procedures and their parameters are given next to the arrows as shown at the left of Figure 5.31. Such diagrams however cannot yet be understood by computers, so they must be transformed into a “linear” notation that defines procedures in terms of text, shown at the right of the figure.

Figure 5.31 General form of Pascal procedure definition

In Pascal, procedures are defined strictly as a sequence of symbols with considerable structure. Before we show how a procedure such as `Divide` is completely defined in a programming language, it is useful to list what must be communicated. The simple invocation of the procedure in a program statement such as :

```
Divide( A + B + C, 3, Mean3, Rem);
```

assumes many things that must be specified in the procedure definition

```
Divide(Num, Den, Quot, Rem).
```

Parts of this definition are as follows:

1. The name of the procedure is descriptive.
`Divide` is an obvious name; it is not `DIVIDE` nor `divide`.
2. The number of parameters is fixed.
`Divide` has exactly 4 parameters, no more no less.
3. Order of the parameters is important, once it is established it is fixed.
The Numerator is first followed by Denominator, then the Quotient and Remainder.
4. The names of the parameters should be meaningful.
The formal names in the definition need not be the same as the actual names in the program given in the `IMPLEMENTATION` part.
5. The types of the parameters (`INTEGER`, `REAL`, etc.) must be respected.
`Divide` has all parameters of the same type, `INTEGER`.
6. The direction of passage of the parameters, input or output, must be indicated.
Here the first two parameters are input and the last two are output.
7. Assertions may be indicated; when they are, must be respected.
For example, the denominator should never be zero.
8. The parameters in a procedure call are separated by commas, not spaces or colons.

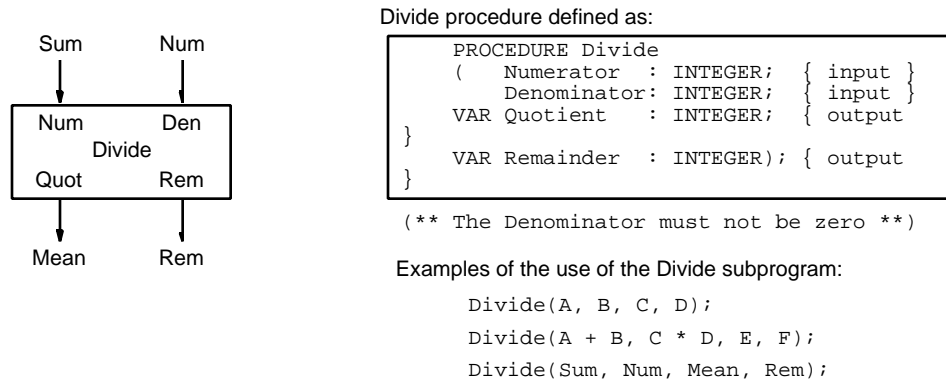
Figure 5.32 Specification of Divide subprogram

Figure 5.32 specifies `Divide` both as a dataflow diagram and as a linear form in Pascal. The linear form begins with the keyword `PROCEDURE`, followed by the name `Divide`. This is followed by parentheses that enclose all the parameter specifications. The parameters are given in a fixed order. Each parameter has three parts:

1. passage is indicated by preceding the name by `VAR` in the case of output parameters; there is no `VAR` preceding the input parameters.
2. the name of the parameter is given, followed by a colon.
3. the type of the parameter is given after the colon.

Following the parameter list some explanations, conditions, limitations, or other comments may appear.

It is also possible to describe `Divide` in a slightly different way by grouping some of the similar parameters. For example, the first two parameters are input parameters and so could be grouped together. Similarly the last two parameters are output parameters and can be combined following a `VAR`. We can also shorten the parameter names, just to illustrate an alternative definition that fits on a couple of lines.

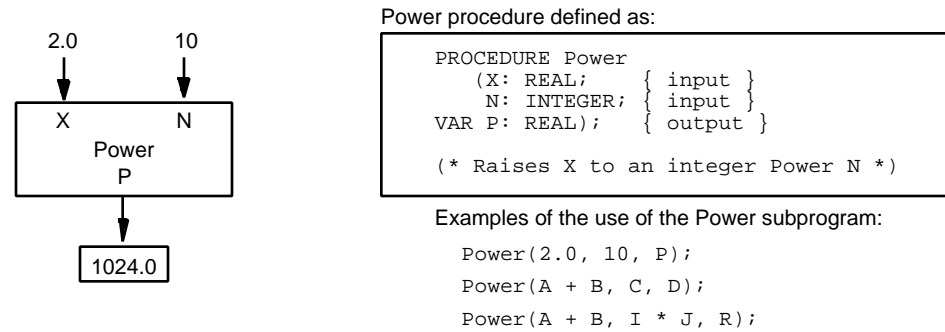
```

PROCEDURE Divide(Num, Den: INTEGER; VAR Quot, Rem:
INTEGER)
(** Divides Num by Den to yield Quot and Rem; Den
cannot be 0 **)

```

Libraries, or units in Pascal, are collections of procedures. The Interface part consists of definitions as shown above, which are sufficient for using the procedures. The Implementation part of a unit will provide the details, but these can be hidden from the users of Libraries.

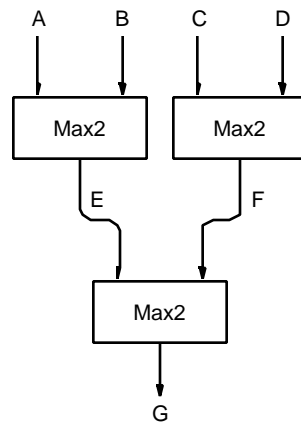
The Power procedure, defined in Figure 5.33, shows parameters of different types.

Figure 5.33 Specification of the Power subprogram

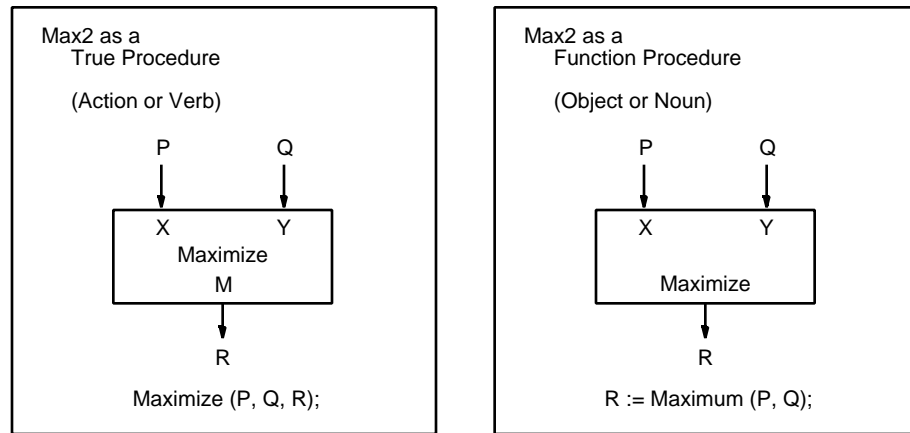
Procedures vs. Functions

We have already seen the use of dataflow diagrams to describe subprograms as black boxes. Subprograms defined in this way can always be implemented as procedures. Sometimes they can also be implemented as functions.

Functions can be thought of as subprograms that return a value, and this value is associated with the name of the subprogram. This name can thus represent a value, and therefore must have a type associated with it. In mathematics, particularly in trigonometry, we are familiar with the idea of a function. The function $\sin(x)$ has a single value, the trigonometric sine of the angle x .

Figure 5.34 Dataflow diagram for Max4

Procedure Max4, illustrated in Figure 5.34, is created out of three applications of Max2. Max2 may be implemented as either a function or a procedure. If we implement it as a procedure, we think of it as an action and therefore label it with a verb, e.g., Maximize. If we implement it as a function, we think of it as a value and therefore label it with a noun, e.g. Maximum.

Figure 5.35 Two forms for the subprogram *Max2*

As another example, Figure 5.35 shows the subprogram *Max2* represented in two ways. On the left side of the figure, it is represented as a procedure `Maximize(A, B, C)`, whose action is to put the maximum value of *A* and *B* into *C*. On the right side of the figure, it is represented as a function `Maximum(A, B)`, whose value is the maximum of the two values *A* and *B*. The procedure call is a statement, whereas the function call is an expression that could be part of a statement. These two must obviously be used in two different manners. Let's compare the Pascal statements needed to compute the maximum of four variables *A*, *B*, *C* and *D*. If we are using procedure *Maximize*, we would write:

```
Maximize(A, B, E);
Maximize(C, D, F);
Maximize(E, F, G);
```

With function *Maximum*, we would write:

```
E := Maximum(A, B);
F := Maximum(C, D);
G := Maximum(F, G);
```

In fact, with function *Maximum*, we could rewrite the last three statements as only one statement:

```
G := Maximum(Maximum(A, B), Maximum(C, D));
```

This one line of nested function calls is equivalent to the sequence of three invocations of *Maximize*. The nested function calls also avoid the use of temporary objects *E* and *F*, which are necessary for the procedure version.

In this particular case, either procedure or function forms can be used. In general, however, the procedure form is more powerful because it allows for the possibility of more than one output. Thus, although the function forms may be shorter and more convenient, they are not as general as procedures. For example, our previous *Sort2* procedure cannot be implemented as a function since it produces two results.

In the interface section of a unit, the two forms `Maximize` and `Maximum` would be defined differently as:

Definition of a procedure:

Definition of a function:

```
PROCEDURE Maximize(X, Y: INTEGER;
                   VAR M: INTEGER);
FUNCTION Maximum(X, Y: INTEGER )
               : INTEGER;
```

Earlier in this chapter and also in Chapter 4, we referred to `IntLib`, a library of useful operations that can be applied to `INTEGER`s. Figure 5.36 shows the Interface part of the `IntLib` unit.

Figure 5.36 IntLib interface part

```
UNIT IntLib;

INTERFACE

    PROCEDURE Add(First, Second: INTEGER; VAR Result:
                                     INTEGER);
    { Result := First + Second }

    PROCEDURE Subtract(First, Second: INTEGER; VAR Result:
                                     INTEGER);
    { Result := First - Second }

    PROCEDURE Multiply(First, Second: INTEGER; VAR Result:
                                     INTEGER);
    { Result := First * Second }

    PROCEDURE Divide(Num, Den: INTEGER; VAR Quot, Rem:
                                     INTEGER);
    { Quot := First DIV Second
      Rem  := First MOD Second }

    PROCEDURE Incr(VAR I: INTEGER; J: INTEGER);
    { I := I + J }

    PROCEDURE Decr(VAR I: INTEGER; J: INTEGER);
    { I := I - J }

    PROCEDURE Order2(VAR X, Y: INTEGER);
    { Sorts A and B in increasing order }

    PROCEDURE Order3(VAR A, B, C: INTEGER);
    { Sorts A, B and C in increasing order }

    PROCEDURE Maximize2(A, B: INTEGER; VAR C: INTEGER);
    { Returns the maximum of A and B in C }

    PROCEDURE Minimize2(A, B: INTEGER; VAR C: INTEGER);
    { Return the minimum of A and B in C }

    PROCEDURE IntToReal(I: INTEGER; VAR R: REAL);
    { Convert integer value of I into real R }
```

The action of the first four procedures specified in this figure is obvious from their names, and the associated comment. Also included are other useful actions already introduced in Chapter 4.

The arithmetic actions just defined as procedures have, as might be expected, verbs as names: `Add`, `Subtract` and `Multiply`. These operations could have

been written as functions, and in that case the names should have been `Sum`, `Difference` and `Product`. These names are nouns representing the values of the functions. `Sum` represents the value of the sum of its two arguments. This duality of values and actions, nouns and verbs, procedures and functions, provides a beautiful complementarity of views.

Procedure `Divide` cannot be represented as a function because it has two results—`Quotient` and `Remainder`. On the other hand, since the division of `REAL` values returns only one value, it can be represented as a function, say `RealQuotient`. This enables us to express our previous temperature conversion formula as the following nest of `REAL` functions (`Product` and `Difference` have been redefined to work on `REAL` values):

```
C := Product(RealQuotient(5, 9), Difference(F, 32));
```

In the Pascal definition of a function, the type of value returned is given at the end of the header following a colon, for example, function `Sum` we just mentioned would have the following definition:

```
FUNCTION Sum(First, Second: INTEGER): INTEGER;
```

You can compare this with the equivalent procedure definition where output parameters are specified in the parameter list with a preceding `VAR`, as we have seen above in the following definition:

```
PROCEDURE Add(First, Second: INTEGER; VAR Result:
INTEGER);
```

The complete definition of such library procedures and functions is done in the implementation part of the library, and will be considered in detail in a later chapter.

To emphasize the difference between procedures and functions, Figure 5.37 shows the interface specification for a collection of procedures that make up the `MaxMinLib` library. This unit gives an example of both functions and procedures that act on `INTEGER` values.

Figure 5.37 MaxMinLib unit

```
UNIT MaxMinLib;
```

```
INTERFACE
```

```
PROCEDURE Maximize2(A, B: INTEGER; VAR C: INTEGER);
{ Sets C to be the maximum value of A and B }
```

```
PROCEDURE Minimize2(A, B: INTEGER; VAR C: INTEGER);
{ Sets C to be the minimum value of A and B }
```

```
FUNCTION Maximum2(A, B: INTEGER): INTEGER;
{ Returns the maximum value of A and B }
```

```
FUNCTION Minimum2(A, B: INTEGER): INTEGER;
{ Returns the minimum value of A and B }
```

Notice that `Maximize` is a verb and is implemented as a procedure, while `Maximum` is a noun and is implemented as a function.

5.11 Binary Logic Library: `BitLib`

`BitLib` is a Pascal unit that deal with binary digits, or bits. In fact, `BitLib` implements the abstract data type `BIT` (see Chapter 8 of the Principles book). This means that the unit defines the type `BIT` as well as a number of operations for values of this type. These operations are similar to the logical operations that were discussed in Chapter 5 of the Principles book. They bring us to the very low logic level of digital computer components. The actions will be denoted as `And2`, `Or2`, `Not1`, `And3`, etc. with the integer indicating the number of inputs to the data flow diagram.

The type `BIT` defined by the unit is a type having only two values: 0 and 1. Input and output of a single bit value is done using procedures `ReadBit(X)` and `WriteBit(Y)`.

Type `BIT` has values 0 and 1 only.

Input and Output:

`ReadBit(X)`: reads in a value of type `BIT` into `X`. Only the values 0 and 1 are accepted.

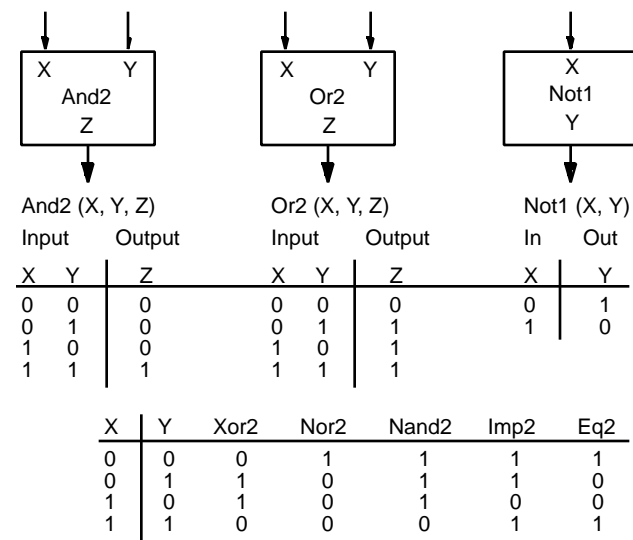
`WriteBit(X)`: writes out the value of `Y`

In addition the `BitLib` unit comprises eight actions illustrated in Figure 5.38. The following three actions correspond to the diagrams of that figure.

`And2(X, Y, Z)`: is a binary action having two inputs and one output as shown at the left of Figure 5.38. The output `Z` has a value of 1 only when both inputs have value 1; otherwise the output value is 0.

`Or2(X, Y, Z)`, shown in the middle of Figure 5.38, has an output value `Z` of 1 when either one or the other (or both) of the inputs `X` and `Y` have the value 1; otherwise the value is 0.

`Not1(X, Y)`, shown at the right of Figure 5.38, is a unary action with one input `X` and one output `Y`; the output is the opposite value (or complement) of the input value. When `X` is 1 then `Y` is 0 and vice versa.

Figure 5.38 The actions of *BitLib*

The five other binary actions (with two inputs and one output) are also shown at the bottom of Figure 5.38. Notice that the exclusive-or, denoted `Xor2`, has an output of 1 only when exactly one of the inputs has value 1. This differs from the basic `Or2` action (that is also called the inclusive-or), which has an output of 1 when both inputs have value 1.

`Nor2` and `Nand2` are two other binary actions that are often used as components in electronic logic circuits such as are used in computers. The action `Eq2` indicates when its input values are equal. The implication `Imp2` is not used much in computing, but is more important in the logic of Philosophy.

Finally, `Create2(P, Q)` is an action (not shown) that produces a different combination of the two parameters each time it is called; it repeats combinations after 4 calls. `Create2` is very similar to procedure `Create3` of `ShortSortLib`, but has two parameters instead of three.

We'll build a binary "half-adder" from the bit actions of `BitLib` that has two inputs `X` and `Y` and two outputs, `Sum` and `Carry`. The relations between its input and outputs is shown in Figure 5.39.

Figure 5.39 Definition of Half-adder

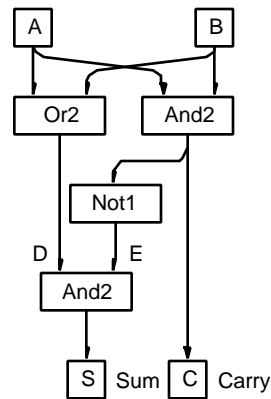
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The figure describes binary addition in the four cases: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 2$ (i.e. 10 in binary, which is 0 with a carry of 1). The carry in the first three cases is 0. It is called a half-adder because it performs

half the operation required to perform binary addition; the other half is to add in the carry from the previous stage.

We'll use the operations of `BitLib` to build a half adder whose dataflow diagram is shown on Figure 5.40. This dataflow diagram shows the actual interconnection of the digital logic components. The method of creating and optimizing such systems is the topic of other courses. Here we simply use this diagram to trace through it.

Figure 5.40 Dataflow diagram for binary half adder



The Pascal program corresponding to this dataflow diagram is fairly easy to construct, using the actions of `BitLib`. The `TestHalfAdder` Pascal program of Figure 5.41 tests the actions of `BitLib`, using `Create2` to produce all four possible inputs to the half adder.

Figure 5.41 The TestHalfAdder program

<pre> PROGRAM TestHalfAdder; USES BitLib; VAR A, B, Carry, D, E, Sum: BIT; Count: INTEGER; BEGIN Count := 4; WriteLn('A B Sum Carry'); WHILE Count > 0 DO BEGIN Create2(A, B); (* Binary half-adder *) And2(A, B, Carry); Or2(A, B, D); Not1(Carry, E); And2(D, E, Sum); WriteBit(A); Write(' '); WriteBit(B); Write(' '); WriteBit(Sum); Write(' '); WriteBit(Carry); WriteLn; Count := Count - 1; END; END.</pre>	<pre> Output from execution TestHalfAdder: A B Sum Carry 0 0 0 0 0 1 1 0 1 0 1 0 1 1 0 1</pre>
---	---

Notice that the three actions `And2`, `Or2` and `Not1` of `BitLib` are not independent. For example, `Or2(P, Q, R)` can be created from the other two, using DeMorgan's first law, as:

```

Not1(P, S);
Not1(Q, T);
And2(S, T, U);
Not1(U, R);
```

We can also write a short Pascal program, `TestOr2` of Figure 5.42, to test this, using `Create2` again to create all possible inputs.

Figure 5.42 The TestOr2 program

PROGRAM TestOr2;	Output from TestOr2
USES BitLib;	
VAR P, Q, R, S, T, U: BIT;	P Q R Or2
Count: INTEGER;	0 0 0 0
BEGIN	0 1 1 1
Count := 4;	1 0 1 1
WriteLn('P Q R Or2');	1 1 1 1
WHILE Count > 0 DO BEGIN	
Create2(P, Q);	
(* Or2 from And2 and Not1 *)	
Not1(P, S);	
Not1(Q, T);	
And2(S, T, U);	
Not1(U, R);	
WriteBit(P);	
Write(' ');	
WriteBit(Q);	
Write(' ');	
WriteBit(R);	
Write(' ');	
(* Create comparison result *)	
Or2(P, Q, R);	
WriteBit(R);	
WriteLn;	
Count := Count - 1;	
END;	
END.	

The equivalence of operation Or2 and the sequence of four actions above, is shown by comparing the two columns R and Or2 in the output at the right of Figure 5.42.

5.12 Chapter 5 Review

This chapter introduced more formally the four fundamental forms in Pascal: Sequence, Selection, Repetition and Invocation.

The simplest form is the Sequence form which consists of a series of statements. The statements are separated by semicolons. The only unusual problem is whether to put a semicolon following the last statement of the series. Strictly speaking, it is not necessary, but for consistency, we will often put one there.

The Repetition form was covered before the Selection form, because the Pascal syntax for the Repetition is simpler. However we did not consider the nesting of repetitions here; that will be considered later.

The Selection form was covered in detail, along with arbitrary nests of Selections, and the many ways to do things. Some simpler nests coded with the `IF-THEN` forms and the `IF-THEN-ELSE-IF` form were also considered in this chapter.

The Invocation form was described and used, but the creation of subprograms was not discussed at this time. Libraries of subprograms were considered, mainly from the view of data-flow diagrams. Procedures and functions were compared and contrasted.

This chapter did not cover the creation of very complex programs consisting of these four fundamental forms. It emphasized the creation of rather small programs. In the next chapters, we can begin to consider creating more complex programs.

5.13 Chapter 5 Problems

1. Is It Possible

- a. Can there be a semicolon before an `ELSE`?
- b. Can there be an `END` before an `ELSE`?
- c. Can there be a semicolon before the `END` that terminates `WHILE`?
- d. Can there be a `DO` immediately before an `END`?
- e. Can there be a semicolon just after a `DO`?
- f. Can there be an `ELSE` just before an `END`?
- g. Can there be two `END`s in sequence separated by a semicolon?

2. Check Syntax

The following statements contain errors of syntax; you are to find them. The ability to find such errors is not important because compilers detect them easily. These problems are given here to help you learn the proper syntax. Do not be concerned with the meaning of the program fragments.

- a. `IF X < Y OR Z THEN X := MIN;`
- b. `WHILE I < 5 DO`
 `WriteLn(I);`
 `Inc(I);`
 `END;`
- c. `IF First < Second THEN`

```
        Write(First);  
    ELSE  
        Write(Second);  
d. WHILE A <= B DO  
    BEGAN  
        Inc(A);  
        Dec(B);  
    END;  
e. IF this = that THEN  
    that = this;  
f. WHILE 5 = 5 DO  
    Write('help');
```

3. Coding Problems

Translate the following fragments of pseudocode involving conditions C_i and statements S_j into pieces of programs in Pascal.

1. *While* C_1
 S_1
 While C_2
 S_2
 S_3
 S_4
2. *If* C_1
 If C_2
 S_1
 Else
 S_2
 S_3
 Else
 S_4
 If C_3
 S_5
3. *While* C_1
 S_1
 If C_2
 S_2
 While C_3
 S_3


```

4.  If C1
    While C2
      While C3
        S4
        S5
        S6
      Else
        If C4
          S7
        Else
          S8
          While C5
            S9

5.  While C1
    While C2
      While C3
        If C4
          S1
        Else
          If C5
            S2
          Else
            If C6
              S3
            Else
              If C7
                S4
              Else
                While C8
                  S5
                  S6

```

5.14 Chapter 5 Programming Problems

Write Pascal programs for some of the following algorithms which are given in the Principles book. After a few examples, this coding process may seem very trivial, but concentrate on choosing proper types, reasonable identifier names (for constants as well as variables), “friendly” prompts, meaningful output, and good indentation, layout and comments. You need not run these programs yet; later they may be used as parts of larger programs. The last 2 or 3 problems in each group are more challenging.

Sequence Problems

1. CHARGE algorithm of Figure 3.1.
2. IDEAL algorithm of Figure 3.4.
3. ISBN Remainder of Figure 3.5
4. TIME algorithm of Figure 3.7.
5. TEMPERATURE algorithms of Figure 3.8
6. MEAN and VARIANCE algorithms of Figure 3.10 for the number of hours worked on five days.
7. SINE algorithm (for first 3 terms) of Figure 3.12.
8. BASE algorithm of Figure 3.13.

Selection Problems

1. PAY algorithms of Figures 2.15 and 2.16.
2. MORE-CHARGE algorithm of Figure 3.39
3. DAYS and LEAP algorithm of Figure 3.40
4. COMPARE algorithm of Figure 4.5.
5. PEOPLE CLASSIFICATION algorithms of Figure 5.17.
6. MAJORITY algorithm of Figure 3.16
7. GRADES algorithms of Chapter 5 Section Nested Selections, Methods 3 and 4.
8. MAJORITY algorithms of Chapter 5 Section Larger Selection Forms, Methods 1, 2, 3, 4 and 5.

Loop Problems

1. DIVIDE algorithm of Figure 5.34.
2. FACTORIAL algorithms of Figures 5.14 and 5.15.
3. BETTER PRODUCT algorithm of Figure 5.35.
4. DICE GAME of Figure 3.3.

Subprogram Problems

(assume that the given sub-programs are available)

1. SECONDS algorithms of Figure 5.42.
2. MINIMUM TIME DIFFERENCE algorithm Figure 5.45
3. TIME-DIFFERENCE algorithms of Figure 5.43.

Debugging Problems

1.

```
PROGRAM BadSwap;
  (* Swaps two values without a third *)
  (* contains one bug for you to find *)
  (* Find the error by either looking *)
  (* at the code, or by running tests *)

VAR first, second: REAL;

BEGIN
  WriteLn('Enter two real values ');
  Read(first);
  Read(second);

  first := first * second;
  second := first / second;
  first := first / second;

  Write('Swapped values are ');
  Write(first: 9 );
  WriteLn(second: 9);
END.
```

2.

```
PROGRAM BadFactorial;
  (* This program almost computes *)
  (* the factorial of any input    *)
  (* You find the problem here     *)

VAR index, fact, num: INTEGER;

BEGIN
  Write('enter a value ');
  Read(num);

  fact := 1;
  index := num;
  WHILE index >= 0 DO BEGIN
    fact := fact * index;
    Dec(index);
  END;

  Write('Factorial is ');
  WriteLn(fact: 5);
END.
```

3.

```
PROGRAM BadMin3;
```

```
(* Finds the minimum of 3 values      *)
(* but it contains one error          *)
(* Find it by looking at the code     *)
(* which is called "opaque box"      *)
```

```
VAR A, B, C, mini:  INTEGER;

BEGIN
  Write('Enter three values ');
  Write('use integers only ');
  Read(A);
  Read(B);
  Read(C);

  IF (A < B) AND (A < C) THEN
    mini := A
  ELSE IF (B < A) AND (B < C) THEN
    mini := B
  ELSE
    mini := C;

  Write('The minimum value is ');
  WriteLn(mini: 4);
END.
```

4.

```
PROGRAM BadMid3;
(* Finds the mid value of 3 values    *)
(* in most cases, but not all cases   *)
(* You are to find the problem here *)

VAR A, B, C, mid: INTEGER;

BEGIN
  WriteLn('Enter three values ');
  Read(A);
  Read(B);
  Read(C);

  IF (B < A) AND (A < C) OR
     (C < B) AND (A < B) THEN
    mid := A
  ELSE
    IF (A < B) AND (B < C) OR
       (C < B) AND (B < A) THEN
      mid := B
    ELSE
      mid := C;

  Write('The mid value is ');
```

```
WriteLn(mid: 2);  
END.
```

Selection Programs

Create Pascal programs to solve the following problems. Pay particular concern to checking the input values, and displaying the output. Provide sufficient runs to test the program. Plan first.

1. Classify Quadrilaterals

Four-sided figures may be classified according to their inner angles as: quadrilaterals, trapezoids, parallelograms, or rectangles. Write a program that takes the four angles of a quadrilateral, in degrees, starting with the smallest and continuing clockwise with adjacent angles in order and identifies these figures. Modify this program to include kites (where at least one pair of the non consecutive angles is equal).

2. Dice Poker

The game of dice poker (or Indian Dice) involves the throwing of five dice, their spot values constitute the “hand”, and the evaluation of the hand is as described below. Write a program to input the five spot values, evaluate the hand, and display the kind of hand (“five of a kind”, etc.). The algorithm can be simplified if the values are sorted.

- a. *Five of a kind*, means that all dice have the same value,
- b. *Four of a kind*, means that 4 dice have the same value,
- c. *Full house*, means that 3 are of one value and 2 of another value,
- d. *A Straight*, means that the 5 values are in consecutive order,
- e. *Three of a kind*, means that three dice are of one value,
- f. *Two pairs*, means 2 dice of one value and 2 of another,
- g. *One pair*, means that only two dice have the same value,
- h. *No pairs*, means none of the above.

3. Human Time Out

Write a program that inputs time in 24-hour military form and outputs one of the following four forms, whichever is most appropriate:

```
"H O'clock"  
"M minutes after H"  
"Half past H"  
"M minutes before H"
```

where M and H are the required minutes and hours.

Procedures and Repetitions

Libraries of various kinds were described in this chapter, for instance `ShortSortLib` that has a number of procedures like `Sort2` and `Sort3`. Use these, and any others you are familiar with, to write the following programs.

1. Many Sorts of Sorts

Create two programs to sort 4 `INTEGER`s, using the `Sort2` procedures, but interconnected in two different ways.

Create a program to sort 7 `INTEGER`s using `Sort3`.

2. Sorts and Sports

Create a segment of a program to sort 5 `INTEGER` values, then use it in the following program.

Five judges of a sporting event each provide an `INTEGER` input score from 1 to 10 that describes their evaluation of the sports performance. The overall score of the event is determined by dropping the highest and lowest scores, and averaging the remaining three scores. This scoring continues for any number of evaluations until a negative value is input, signifying the end of the competition.

Redo this program (or parts of it) by using different subprograms.

3. Forgiving Grader

A student's percentage score is determined by averaging the highest four percentage scores out of five (exam1, midterm, final, projects, quizzes), thus "forgiving" the lowest score. This grading is to continue for any number of students until a negative value is input, signifying the last of the students.

Do this in two different ways, using different sub-programs.

4. Demilitarize Time

Time can be expressed in "military" or 24-hour form where 20:30 means 8:30pm. Create an algorithm to convert such a military time into "non-military" time. A sequence of times is to be converted until a negative time is input.

Create another program to determine the time elapsed between any two given military times on the same day.

Create another program which determines the elapsed time between any two given times on two successive days.

Create another program to determine the shorter elapsed time between three given times on the same day.

5. Date of Easter

The algorithm to determine the date of Easter for any given year could involve a number of `Div` and `Mod` actions. Create a program to continuously determine the date of Easter according to the following algorithm for each input year `Y` until a negative year is input:

1. The “golden number”, G is $(Y \bmod 19) + 1$.
2. The century number C is $(Y \text{ Div } 100) + 1$.
3. The number of years X in which a leap year was dropped, e. g., 1900, so as to keep in step with the sun is $(3C \text{ Div } 4) - 12$.
4. A correction Z to synchronize Easter with the moon’s orbit is $(8C + 5) \text{ Div } 25$
5. If $D = (5Y \text{ Div } 4) - X - 10$ then March $((-D) \bmod 7)$ is a Sunday—if $(-D) \bmod 7 = 0$ then March 7 is a Sunday.
6. The “Epact” E specifies when a full moon occurs. $E = (11G + 20 + Z - X) \bmod 30$. If $E = 25$ and G is greater than 11, or if $E = 24$ then E is increased by 1.
7. Easter is on the “first Sunday following the first full moon that occurs on or after March 21”. The “calendar moon” used for finding Easter is defined as the N th of March where $N = 44 - E$. If $N < 21$ then set N to $N + 30$.
8. To advance N to a Sunday, set $N = N + 7 - ((D + N) \bmod 7)$.
9. If $N > 31$ then the date of Easter is the $(N - 31)$ April; otherwise the date is N March.

Josephus’ Problem

It is said that, after the Romans had captured Jotapat, Josephus and forty other Jews took refuge in a cave. Josephus, much to his disgust, found that all except himself and one other man were resolved to kill themselves rather than fall into the hands of their conquerors. Fearing to show his opposition too openly, he consented, but declared that the operation must be carried out in an orderly way, and suggested that they should arrange themselves round a circle and that every third person should be killed until only one man was left, who must then commit suicide. It is alleged that he placed himself and the other man in the 31st and 16th places. You are to write a Pascal program that prints out the execution order in the generalized case where, instead of every third person every n th person is killed. The program should input two `INTEGER`s, M , the number of people in the circle, and N , the execution ordinal. With $M = 8$ and $N = 4$, the deaths occur in the order shown:

Position in circle	Execution order
1	5
2	4
3	6
4	1
5	3
6	8

7	7
8	2

5.15 Chapter 5 Programming Projects

Project A: Change Change

You are to modify the program `SubChange` of this chapter in any five of the following ways. Do this in stages ending with one larger program (not five smaller ones).

1. Make the program more **general**, by allowing the output of half-dollars and dollar bills.
2. Make the program more **robust**, by having it reject any improper values (such as negative costs).
3. Make the program more informative, by writing out the amount of change as:
“The change is 28 cents” or
“The change is 2 dollars and 13 cents”.
4. Make the program more **convenient**, by putting it into a loop and having it continue making change until a negative value is input.
5. Make the program **grammatically correct**, by writing singular and plural counts such as
“1 quarter 0 dimes 0 nickels 3 pennies”.
6. Make the program output **shorter**, by not writing out the zero count as:
“1 quarter 3 pennies”.
7. Make the program output more **verbose** by writing out the numbers in English as:
“one quarter three pennies”.
8. Make the program output more **complete** by writing out the connectives also as:
“The change is one quarter and three pennies”.
9. Make the program more **user friendly**, by allowing input as real numbers with decimal points such as 0.75, 1.25
10. Make the program more **productive**, by having it keep track of the numbers of coins of each type (not attempting to put out any quarters if it has none).
11. Make some improvements of your own.

Project B: Payroll

Create a program describing the payroll system of the Principles book Chapter 2, Section 2.4. Do this in stages: first, create the generalized version, then modify it to include the extended version, which pays double time for hours worked over 60. Next modify these two with the changes in the foolproof version, which checks that the input number of hours worked is both greater than or equal to zero and less than the number of hours in a week and, finally, embed all of this in a loop that repeats the pay process for many people.

Project C: Quadratic Roots

The solutions to (roots of) the quadratic equation

$$Ax^2 + Bx + C = 0$$

are given by the quadratic formula:

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Create a program to determine these solutions and test it for the following values:

A	B	C	
0	0	0	(Many roots)
0	0	1	(No roots)
0	1	2	(One root)
0	5	0	(Special root)
1	2	2	(Complex roots)
4	0	-16	(Similar roots)
1	1	-6	(Real roots)

Study also the cases where B^2 is much larger than $4 \times A \times C$.

Project D: Digital Circuits

If you have some knowledge of switching theory (or would like to gain some), use the `BitLib` procedures to create the following (and show some input-output traces).

1. `Maj3(A, B, C, M)`, a majority of three binary inputs.
2. `FullAdd(A, B, C1, S, C2)`, a full adder of 3 inputs and 2 outputs (input A, B and carry C1 from previous stage; output sum S and carry C2 to the next stage).

3. `HalfSub (X, Y, D, B)`, a half subtractor, with 2 inputs and 2 outputs.
4. `FullSub (X, Y, Z, D, B)`, a full subtractor.
5. Anything else of interest.

Project E: Roll Your Own

It is much harder to create a project than it is to investigate or solve an already created one. Create a project.

CRN: Convert Roman Numbers

You are to create a program to convert a given integer from the normal or Arabic form into the Roman form. The Arabic number will be, at first, in the range 1 to 300, but will grow later. The Roman number will first be a simple kind that allows 4 repetitions of symbols, but will also grow later.

You may need to review the concept of Roman numbers; the following may refresh your memory.

I is 1, V is 5, X is 10, L is 50

C is 100, D is 500, M is 1000

The Arabic 1984 in the simple Roman form is:

MDCCCLXXXIII

and in the standard Roman form is:

MCMLXXXIV

First do the simpler Roman numbers. Limit the Arabic input value to a maximum of 300. Call this program `Roman1`. Use the `ChangeMaker` program as a guide. Do sufficient tests to convince yourself that it works.

Secondly, extend the above program to larger Roman numbers. Call this program `Roman2`.

Finally, modify `Roman2` to accept and convert to the standard Roman form. Call this program `Roman3`. Test it thoroughly.

GPR: Growing Pay Roll

This assignment is to gain experience in coding and “growing” programs, using the `PayRoll` algorithms of the *Principles* book Chapter 2, Section 2.4. Assume all the variables to have `REAL` values. Write these programs in Pascal **exactly** in the structure shown; do not change the structure of the algorithms—yet! Use reasonable names (not `R`, `H`, `P`) and readable style with indentation.

1. **Write** a program consisting of the **Generalized** version of Figure 2.14 embedded in a loop that continues until a negative number of hours

worked has been entered. Name this program `Pay1`, run it and test it. Print the program and make a record of some test runs.

2. **FoolProof** the above version as shown in Figure 2.16, and save it as `Pay2`. Make the resulting program readable with good indentation. Test it for all possible cases. Print this version and record your test runs.
3. **Extend** the above program as a **Cascaded** equivalent as on the right of Figure 2.18 and save it as `Pay3`. Print out your program, and show some test runs.

Time permitting:

4. Modify the structure of any of the above programs to use the nested `ELSE IF` Selections whenever possible. Test it.
5. Extend the `PayRoll` program further by computing the `NetPay`, which is the `GrossPay` less the `Deductions`. The deductions include some `Misc` amounts that are input, and also include `Taxes` at a fixed rate, say 20%, of the `GrossPay`.
6. Print out the pay in the form of a pay check, with a date and the amount. Leave space for the name and the written amount to be filled in by hand.
7. Further modify the program to sum all the money paid out to the employees and to the government in the form of taxes, and to output these sums at the end.
8. Make any other changes, modifications, improvements that you have time for.

MWM: Many Ways to Mid

This lab project involves many very different ways to do the same thing. Study the problem of finding the middle value `Mid` of any three variables, say `A`, `B`, `C`, having different `INTEGER` values as described in the Principles book Chapter 5, Problems. You will also need the definition of `MaxMinLib` from this chapter.

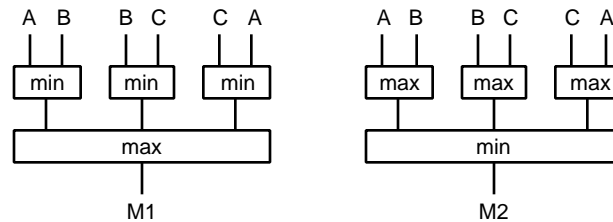
Method 1: Using bigger building blocks, Procedures

Use the procedures `Maximize2`, `Minimize2` from `MaxMinLib`, and connect these together to get the `Mid` value of three variables as given in one of the following diagrams. Test it for all possible cases; but do not print out these tests, just print out the program.

Method 2: Using other bigger building blocks, Functions

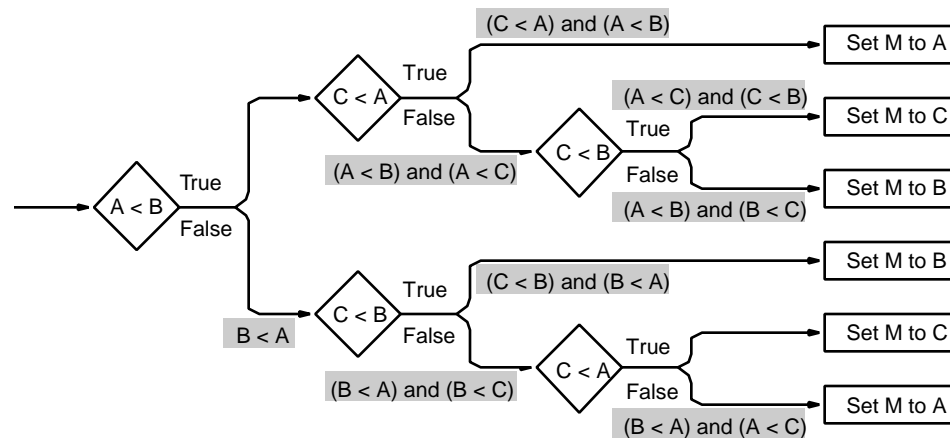
Copy the above program and modify it to use the functions `Maximum2`, `Minimum2` from `MaxMinLib`, and connect these together to get the `Mid` value by

using the other diagram that follows. Write the expression for the value of `Mid` as one line. Embed the computation of `Mid` within a loop to make it easy to test this program for all possible cases (save a tree). Print out the test results.



Method 3: Using Complex Nests

Create a program corresponding to the following general nest of Selection forms. Use only the simple conditions shown (do not use ANDs, ORs etc. in the conditions). Test this program. You may wish to document this program with assertions at various points.



Method 4: Another way (Time Permitting)

Create yet another different program to find the `Mid` value of three variables having different values.

DFP: Data Flow Programming

This lab project will involve programming at a high level, using subprograms from Libraries. The programs involve data flows rather than control flows! You will be creating programs using procedures which will be tested within the Pascal test program shown at the end of this problem. You do not need to be concerned with the details of the Pascal test program, `TestProg`; just use it as given. Your main task is to make various modifications to the data flow part in the middle of the given program `TestProg`.

1. Enter the program TestProg below.

```

PROGRAM TestProg;
(* Tests of programs in Integer Library *)

PROCEDURE Sort2(  First, Second: INTEGER;
                  VAR Large, Small : INTEGER);
VAR Temp: INTEGER;

BEGIN
    Large := First;
    Small := Second;
    IF Large < Small THEN BEGIN (* Swap *)
        Temp := Large;
        Large := Small;
        Small := Temp ;
    END (* SWAP *)
END;

PROCEDURE Sort3(  First, Second, Third: INTEGER;
                  VAR Maxi, Midi, Mini : INTEGER);
VAR Temp1, Temp2, Temp3: INTEGER;

BEGIN
    Sort2(First, Second, Temp1, Temp2);
    Sort2(Temp2, Third, Temp3, Mini);
    Sort2(Temp1, Temp3, Maxi, Midi);
END;

VAR 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: INTEGER;
    Count: INTEGER;
BEGIN
    WriteLn('Enter number of tests ');
    Read(Count);
    WriteLn('Enter the test values ');
    WHILE Count > 0 DO BEGIN
        Read(A);
        Read(B);
        Read(C);

        (* Enter Subs here *)
        Sort2(A, B, D, E );
        Sort2(E, C, F, S );
        Sort2(D, F,  L, M );
        (* End of the Subs *)

        Write(' ');
        Write(L: 4, M: 4, S: 4);
        WriteLn;
    END;
END;

```

```
        DEC ( Count ) ;  
    END ;  
END .
```

2. Run TestProg to check that it does sort any 3 different values. Output the result of this test. Draw the data flow diagram next to the program.
3. Run TestProg to check that it does sort any 3 values (some which may be the same). Can you do this in less than 27 test values? Output the test results.
4. Modify TestProg to create a Sort4 using only Sort3. Output the test results when all values are different. Draw the data flow diagram on this output page.
5. Replace the lines that input the test values with a call to a procedure Test4(A, B, C, D), which assigns a different set of test values each time it is called, which you must write and insert in TestProg. Output the test results again.
6. Modify TestProg to create a Sort4 using only Sort2, and output the test results when all values are different; use the Test4 procedure. Draw the data flow diagram again.
7. Find yet another different way to create a Sort4 using Sort2 and use the above Test4 procedure to test this program. Draw the data flow diagram again.
8. Modify the program to create a Sort5 using Sort3 in different ways.

Chapter 6 Pascal with Bigger Blocks

This chapter continues the presentation of the programming language Pascal by introducing other forms, deeper nests, different data types and more details. These complements are not as fundamental or important as the topics of the previous chapter, but are useful and convenient in the development of clear and correct programs. Remember, bigger is not always better.

Chapter Overview

6.1	Preview.....	169
6.2	Conglomerations.....	169
	Mixed and Nested Forms.....	169
6.3	More Data.....	171
	External.....	171
	CASE Form.....	173
6.4	More Repetition Forms.....	175
6.5	The For Loop.....	177
6.6	Character Type.....	182
	in Pascal.....	182
	Representation of Characters.....	185
6.7	Boolean Type in Pascal.....	190
6.8	More Types.....	193
	Big Types in Pascal.....	193
	Even Bigger Types.....	193
	Smaller Types.....	194
6.9	Programmer Defined Types.....	195
	Enumerated Types.....	195
	Subrange Types In Pascal.....	198
6.10	Strings in Pascal.....	198
6.11	Simple Files in Pascal.....	201
	More Files: Input and Output.....	205
6.12	Modification of Programs.....	207
6.13	Programming Style.....	209
	Documentation.....	209
	Further Guidelines for Identifiers.....	212
	A Bad Style Horror Story.....	213
	Criticism of the MeanMean Program.....	215
6.14	Errors in Programming.....	216
	Syntactic Errors.....	216
	Execution Errors.....	217
	Logical Errors.....	217
	Other Errors.....	217
6.15	Debugging, Testing, Proving.....	217
6.16	Chapter 6 Review.....	219
6.17	Chapter 6 Problems.....	219

6.18	Chapter 6 Programming Problems.....	221
6.19	Chapter 6 Programming Projects.....	224
	Plotting Programs.....	224
	Modify Calculator.....	226
	Statistics (Rainfall).....	227
	Project Plotup.....	227
	BSD: Big Statistics Data.....	228
	BTD: Big Text Data.....	229
	SMC: Small Monthly Calendar.....	230

6.1 Preview

The more important part of this Chapter is the material on programming style, documentation, errors, testing and debugging, since a knowledge of these areas is essential to serious programming. Without it, you will waste much time floundering around making blunders, then searching for and correcting them.

First, we consider larger programs that are created from complex nests of the four fundamental forms. The coding of these algorithms into readable programs is straightforward. Indentation is important for clarity. Next, we consider briefly the processing of larger amounts of data, in particular, arbitrary amounts of external data. We also introduce some other Pascal statements, especially the extensions of the Selection and Repetition forms (`CASE`, `FOR` and `REPEAT`). Many examples are shown of deeper nests, especially of the Repetition form.

Some additional data types, especially the Character and Boolean types, are introduced along with some example programs illustrating these types.

A particular goal of this chapter is to expose you to additional tools, concepts and skills. Many of these are alternatives to tools you have previously seen, but this will allow you to better choose the appropriate tools, best suited for your purposes. For example, the `FOR` loop is concise and convenient, but it is also limited in some ways over the more general `WHILE` loop. You should be capable of knowing when to use either of these loops.

Another important goal of this chapter is to provide an appreciation for programming style, documentation, modification, testing, and debugging. This is done through examples. There are really three ways of learning to become a good programmer: practice, practice and practice with, in each case, a great deal of self criticism. This chapter will put you on the right track.

6.2 Conglomerations

Mixed and Nested Forms

In almost all non trivial programs, you will find mixed combinations of forms. In Pascal, such combinations, whether they be sequences or nests, should be considered as a structure of forms rather than as a conglomeration of individual statements. We will give a number of examples that illustrate mixed forms and their nesting.

As we have already mentioned, a program is essentially a one-dimensional sequence of words and symbols that represents a two-dimensional structure. In order to understand the program, you must appreciate this two-dimensional structure. Thus, the careful use of statement indentation in the program is an important way of helping the reader see the two-dimensional nature of the program. Strictly speaking, indentation is not necessary, but it is very

convenient and certainly helps in explaining, understanding and extending programs.

The precise details of the style of indentation are not as important as the fact that the indentation exists and follows rules that are consistently applied. In this book, we use an indentation of 3 or 4 spaces per nesting level; in other books it may vary from 2 spaces to 6 spaces. You will eventually decide on the rules that you will adopt for yourself, but in the meantime you could imitate the style shown here. You shouldn't be in a rush to develop your own style yet; it will come naturally in time.

Figure 6.1 Pseudocode and Pascal program for Signed Product algorithm

<i>Input X</i>	PROGRAM SignedProduct;
<i>Input Y</i>	{ Shows Loops nested in a Choice }
<i>Set Count to X</i>	
<i>Set Prod to 0</i>	VAR Count, Product, X, Y: INTEGER;
<i>If Count > 0</i>	BEGIN
<i>While Count > 0</i>	Write('Enter two values: ');
<i>Set Prod to</i>	Read(X, Y);
<i>Prod + Y</i>	Count := X;
<i>Set Count to</i>	Product := 0;
<i>Count - 1</i>	
<i>Else</i>	IF Count > 0 THEN
<i>While Count < 0</i>	WHILE Count > 0 DO BEGIN
<i>Set Prod to</i>	Product := Product + Y;
<i>Prod - Y</i>	Count := Count - 1;
<i>Set Count to</i>	END { WHILE }
<i>Count + 1</i>	ELSE
<i>Output Prod</i>	WHILE Count < 0 DO BEGIN
	Product := Product - Y;
	Count := Count + 1;
	END { WHILE };
	Write('The product is ',
	Product: 7);
	END. { Signed Product }

The program Signed Product from the Principles book Chapter 5 (Fig. 5.48) is shown in Figure 6.1, both as pseudocode and as a Pascal program. In this program there are two While loops nested within a Selection form. Notice the three levels of indentation corresponding to the three levels of detail: the program level, the branches of the IF statement, and the bodies of the loops. Also notice that the identifier names in the program have been chosen to be meaningful.

As you can see in Figure 6.1, spacing between lines is also very useful to separate the various parts of a program. Here, the initial part of the program body is separated from the nested Selection, which is separated from the output part. Some may also wish to insert space between the `WHILE` forms within the `IF` form. But spacing, like indentation, can easily be overdone. Experiment.

The alignment of symbols, such as the colon-equals symbols within sequences of assignments, is sometimes viewed as helping readability; however, if followed slavishly, it can also mask a program's structure. The most beneficial aspect of the writing style in programs is consistency. For instance, if the indentation step varied from 1 to 6 spaces in our program in Figure 6.1, it would be a hindrance to the reader. Having a fixed indentation step lets the reader see the various "blocks" of our program at first glance. Remember, the goal in using a writing style is the understandability of programs, not the mindless adherence to rules.

6.3 More Data

External

One of the abilities that makes computers such a powerful tool is their ability to process large amounts of data, one item at a time. For example, the daily quantity of rainfall over many months may need to be analyzed to compute various averages, accumulations, extremes, trends, and other statistics. Other such large quantities of data could include prices, sales figures, efficiencies and grades. One common method of processing such external data is considered in Chapter 6 of the Principles book. With this method, the end of the data is marked by a special value, the terminator or end of file marker, that is different from all possible data values. This value is usually specified by including it as the first value in the external data.

Figure 6.2 shows the pseudocode for the BigMax algorithm from the Principles book Chapter 6 (Fig. 6.6), and the corresponding Pascal program. In this program, the terminating value for the data is entered first. This value (for example, `-999`) differs from the other typical values by being negative, large and unusual. Thus, it is not only unique, it is easily remembered by the user. Following this value, other values are entered and processed, that is, they are compared to the current value of `Max`, which is updated if necessary, until the terminating value is reached; this marks the end of the data. Finally the maximum value is output. A sample output produced by the execution of this program is shown in Figure 6.3.

Possible modifications to this program include the computation of other statistics, means, minimum value, second-largest value, etc. For some types of data it may be necessary to use `REAL` values instead of `INTEGERS`. Reading the input from a file would also be natural for this type of problem.

Figure 6.2 Pseudocode and Pascal program for BigMax

<i>Input Terminator</i>	PROGRAM BigMax;
<i>Input Val</i>	(* Shows a Choice nested in a Loop*)
<i>Set Max to Val</i>	(* The data is sandwiched between *)
<i>Input Val</i>	(* occurrences of a unique marker *)
<i>While Val Terminator</i>	(* value. This program assumes *)
<i> If Max < Val</i>	(* at least one value is input *)
<i> Set Max to Val</i>	VAR Terminator, Value, Max :
<i> Input Val</i>	INTEGER;
 <i>Output Max</i>	 BEGIN
	Write('Enter terminal value: ');
	Read(Terminator);
	Write('Enter first value: ');
	Read(Value);
	Max := Value;
	 Write('Next value: ');
	Read (Value);
	WHILE Value <> Terminator DO
	BEGIN
	IF Max < Value THEN
	Max := Value;
	Write('Next value: ');
	Read(Value);
	END { WHILE };
	 Write('The maximum value is ',
	Max:7);
	 END { BigMax }.

Figure 6.3 Execution of the BigMax program

```
Enter terminal value: -999
Enter first value: 34
Next value: 56
Next value: 67
Next value: 76
Next value: 65
Next value: 54
Next value: 43
Next value: 35
Next value: -12
Next value: 89
Next value: 98
Next value: 22
Next value: 12
Next value: -999
```

The maximum value is 98

CASE Form

Pascal's CASE statement is a convenient way of expressing a selection from a number of choices, that would otherwise be expressed as a series of deeply nested IF-THEN-ELSE-IF... constructs. This alternative form can be used where each of the conditions involves the comparison of a specific value with several different constant values. This is illustrated in Figure 6.4 where the pseudocode and corresponding Pascal program are shown for the Price algorithm from the Principles book Chapter 6 (Fig. 6.8). This program selects a price depending on the quantity of items sold.

Figure 6.4 Pseudocode and Pascal program for Price algorithm

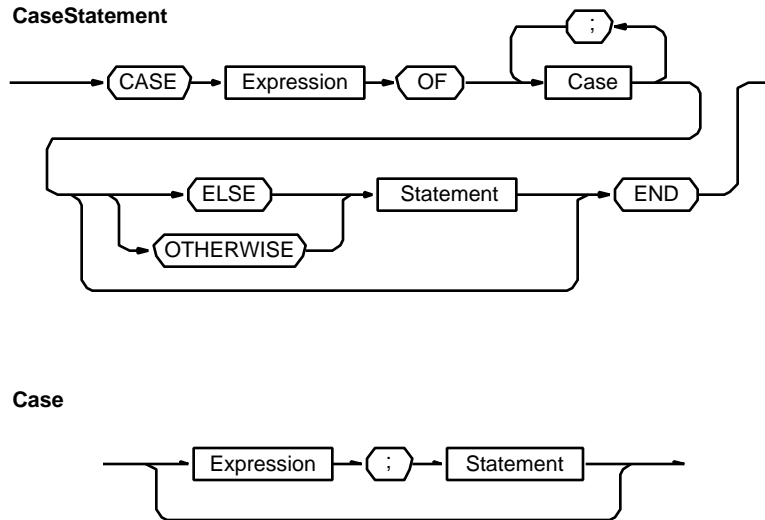
<i>Input Quantity</i>	PROGRAM Price;
<i>Select Quantity</i>	(* Select price depending *)
1:	(* on quantity sold *)
Price ← 99	VAR Quantity, Price: INTEGER;
2, 3:	BEGIN
Price ← 98	Write('Specify quantity: ');
4, 5, 6:	Read(Quantity);
Price ← 95	CASE Quantity OF
7, 8, 9:	1:
Price ← 90	Price := 99;
Otherwise:	2, 3:
Price ← 85	Price := 98;
<i>Output Price</i>	4, 5, 6:
	Price := 95;
	7, 8, 9:
	Price := 90;
	OTHERWISE
	Price := 85;
	END;
	WriteLn('Price is ', Price: 3);
	END. { Price }

Figure 6.5 shows the syntax diagram for the Pascal CASE statement. A CaseLabelList consists of any number of constants separated by commas. For example, one list representing the non working days of some month might be:

1, 2, 8, 9, 15, 16, 21..24, 29, 30

In this example, the sequence 21..24 is a Pascal notation called a “subrange” and denotes the values 21, 22, 23, 24. The OTHERWISE part is optional; the word ELSE may be used instead of OTHERWISE.

Figure 6.5 Syntax diagram for Pascal *CASE* statement



The number of days in a month depends on the month and could be written as a CASE statement as follows:

```

(* Days in a month of a leap year *)
CASE month OF
  1:
    Days := 31;
  2:
    Days := 29;
  3:
    Days := 31;
  4:
    Days := 30;
  5:
    Days := 31;
  6:
    Days := 30;
  7:
    Days := 31;
  8:
    Days := 31;
  9:
    Days := 30;
  10:
    Days := 31;
  11:
    Days := 30;

```

```

12:
    Days := 31;
OTHERWISE
    Write('error');
END;

```

Alternatively, the above example could be shortened by combining the days into longer label lists as:

```

(* Days in a Month of a Leap year *)
CASE Month OF
    9, 4, 6, 11 :
        Days := 30;
    1, 3, 5, 7, 8, 10, 12:
        Days := 31;
    2:
        Days := 29;
OTHERWISE
    Write('error');
END;

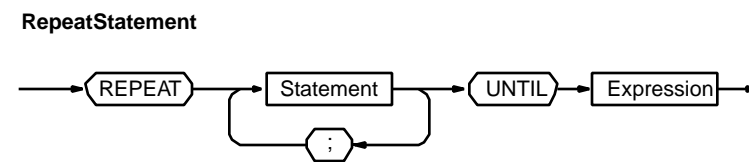
```

Notice that not all the deeply nested Selection forms that are expressed in pseudocode with a Select form can be written with the Pascal CASE statement. The reason is simple: the CASE statement requires that the conditions always involve constants that are not REAL numbers.

6.4 More Repetition Forms

The REPEAT-UNTIL form was first introduced in Chapter 5 of the Principles book. It is a Repetition form where the associated condition is evaluated *after* the body of the loop has been executed, rather than *before* as in the WHILE loop. Consequently, the body of the loop is executed at least once. The syntax of the Pascal REPEAT-UNTIL statement is defined in the syntax diagram in Figure 6.6.

Figure 6.6 Syntax diagram for Pascal *REPEAT* statement



When written, the REPEAT and UNTIL reserved words sandwich the indented statements forming the body of the loop, as in the following:

```

REPEAT
    Statements
UNTIL condition

```

Figure 6.7 shows two almost equivalent Pascal versions of algorithm OddSquare discussed in Chapter 6 of the Principles book. Both calculate the

square of an INTEGER, Num, by summing the first Num odd integers. In the version on the left of the figure, OddSquareWhile, a WHILE statement is used, and in the version on the right of the figure, OddSquareUntil, a REPEAT-UNTIL statement is used. Notice that the loop conditions in the two versions are the negations of each other—if you think about the meanings or “while” and “until”, you will see why this is so. Notice also that, in OddSquareWhile, a BEGIN-END must be used to delimitate the body of the loop, whereas in OddSquareUntil the REPEAT-UNTIL pair encloses the body of the loop. These two versions are only *almost* equivalent because of another important difference between the semantics of the two statements. In the REPEAT-UNTIL loop, the body is *always* executed at least once, whereas, in the WHILE loop, if the loop condition is initially false, the body is not executed at all. Consequently, if the value entered is zero, OddSquareUntil gives the wrong result 1, whereas OddSquareWhile gives the correct result of zero.

Figure 6.7 Two versions of the program OddSquare

<pre> PROGRAM OddSquareWhile; (* Calculate the square of*) (* N by summing the first *) (* N odd numbers *) VAR Square, OddNum, Num: INTEGER; BEGIN Write('Specify number to be squared '); Read(Num); Square := 0; OddNum := 1; WHILE OddNum <= ABS(Num + Num) DO BEGIN Square := Square + OddNum; OddNum := OddNum + 2; END; WriteLn('Square is ', Square: 4); END. { OddSquareWhile } </pre>	<pre> PROGRAM OddSquareUntil; (* Calculate the square of*) (* N by summing the first *) (* N odd numbers *) VAR Square, OddNum, Num: INTEGER; BEGIN Write('Specify number to be squared '); Read(Num); Square := 0; OddNum := 1; REPEAT Square := Square + OddNum; OddNum := OddNum + 2; UNTIL OddNum > ABS(Num + Num); WriteLn('Square is ', Square: 4); END. { OddSquareUntil } </pre>
---	--

Here is an example of the execution of both programs.

```

Specify number to be squared: 15
Square is  225

```

The program Mean, shown in Figure 6.8, is another example of the use of the REPEAT-UNTIL statement. This program calculates the mean of a sequence of input values terminated by the value -999. Notice that it requires that at least

one value other than -999 be input. If the first value is the end of data marker, -999, then, as a consequence of the way the REPEAT-UNTIL loop operates, the program assumes that the -999 is a genuine data value and waits for another -999 to terminate the program. Study the program carefully so that you understand why this is so.

Figure 6.8 A Mean program

```
PROGRAM Mean;
(* Calculate mean of list of values *)
(* terminated by EndOfData value    *)

CONST EndOfData = -999;

VAR Sum, Num, Value: INTEGER;
    Mean: REAL;

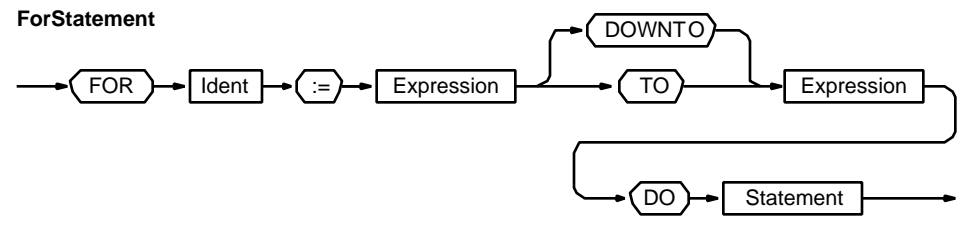
BEGIN
    Sum := 0;
    Num := 0;
    Write('Enter list of values terminated by ',
          EndOfData, ' : ');

    Read( Value );
    REPEAT
        Sum := Sum + Value;
        Num := Num + 1;
        Read( Value );
    UNTIL Value = EndOfData;
    Mean := Sum / Num;

    WriteLn('Mean of ', Num, ' values is ', Mean:6:2);
END. { Mean }
```

6.5 The For Loop

In Chapter 6 of the Principles book, we introduced the For loop for use in our pseudocode to provide a Repetition Form that made it convenient to specify the number of iterations to be performed. The Pascal equivalent of this loop is the FOR statement, whose syntax is defined in the syntax diagram of Figure 6.9

Figure 6.9 Syntax diagram for Pascal FOR statement

This syntax diagram shows that the statement has two forms:

```
FOR var := expr1 TO expr2 DO
    statement
```

and

```
FOR var := expr1 DOWNT0 expr2 DO
    statement
```

In both forms, *var* is the counter variable for the loop, called the *loop control variable*, and *statement* is the body of the loop. In execution of the first of these forms, *var* is initialized to the value of *expr₁*, and the body of the loop is executed repeatedly with the value of *var* being increased by 1 between each iteration until its value is greater than the value of *expr₂*. The execution of the second form is similar, the difference being that the value of *var* is *decreased* by 1 between each iteration, and looping continues until the value of *var* is less than the value of *expr₂*. The body of the loop is not executed at all if, in the case of the *TO*, the value of *expr₁* exceeds the value of *expr₂* or, in the case of the *DOWNT0*, the value of *expr₁* is less than the value of *expr₂*.

The following are three sample fragments of Pascal code illustrating the use of the *FOR* statement:

1.

```
Fact := 1;
   FOR Count := 1 TO Num DO
       Fact := Fact * Count;
```

Here, the factorial of *Num* is calculated with the *Count* increasing by 1 at each iteration.

2.

```
Fact := 1;
   FOR Count := Num DOWNT0 1 DO
       Fact := Fact * Count;
```

This code is essentially the same as in example 1 above, except that *Count* is decreased by 1 at each iteration.

3.

```
Square := 0;
   OddNum := 1;
   FOR Count := 1 TO Num DO BEGIN
       Square := Square + OddNum;
       OddNum := OddNum + 2;
   END;
```

This is yet another example of finding the square of Num by summing the first Num odd integers.

There are a number of restrictions on the use of the `FOR` statement, and it is necessary to know what they are.

1. The expressions describing the initial and final values of the loop control variable are evaluated only once, on entry to the loop before the body of the loop is executed for the first time.
2. The statements in the loop body must not change the value of the loop control variable.
3. The loop control variable is usually, but not always, of type `INTEGER`, but cannot be of type `REAL`.
4. The step, the amount by which the value of the loop control variable is increased or decreased at each iteration, is fixed at 1, and cannot be changed.
5. The loop control variable may not retain its value after the loop is done.

In addition, there are some other restrictions on the loop control variable of the `FOR` statement. At this time, you might not understand them, but you may come back to this page later, so you should be aware of these additional restrictions. The loop control variable cannot be:

- a formal parameter of a procedure
- imported from another unit
- a component of an array, of a record, or a variable designated by a pointer

In short, there are many things it cannot be!

If you find that these restrictions are too severe for your needs, then you can use a `WHILE` statement instead. The `WHILE` statement is always more general, but the `FOR` statement is often more convenient.

Figure 6.10 shows side-by-side a `FOR` statement and its equivalent `WHILE` statement.

Figure 6.10 Equivalent `FOR` and `WHILE` statements

<pre>FOR C := M TO N DO BEGIN statements END;</pre>	<pre>C := M; WHILE C <= N DO BEGIN statements C := C + 1; END;</pre>
---	---

The `FOR` statement is shorter, but the equivalent `WHILE` statement is more general, since it may also be used for loop control variables of the `REAL` type. Actually, if there were only one statement in the loop body, then the `FOR` statement would not need the `BEGIN-END` pair, and so would be much shorter than the equivalent `WHILE` statement.

Figure 6.11 shows a Pascal program that is a refinement of the graph plotting algorithm discussed in Chapter 6 of the Principles book. It has been refined to plot two functions on the same graph, both sideways with the X axis vertical and the Y axis horizontal. This program can serve as a summary of the previous discussion of nesting and of the different forms of loops.

Figure 6.11 The Pascal program *SidePlotXY*

```
PROGRAM SidePlotXY;
(* Plot of two functions vs. X *)
(* Both sideways, Y horizontal *)

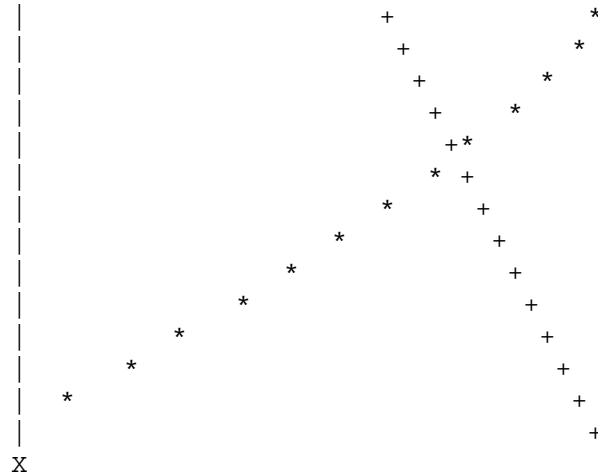
CONST MaxY = 40;

VAR X, Y1, Y2, First, Last, Incr, Factor: REAL;
    Q1, Q2, Step: INTEGER;

BEGIN
    { Input plot parameters }
    Write('Enter first value: ');
    Read(First);
    Write('Enter last value: ');
    Read>Last);
    Write('Enter scale factor: ');
    Read(Factor);
    Write('Enter an increment: ');
    Read(Incr);
    WriteLn;

    { Draw horizontal Y axis }
    FOR Step := 0 TO MaxY DO
        IF (Step MOD 5 = 0) THEN
            Write('+')
        ELSE
            Write('-');
    Write(' Y ');
    WriteLn;

    { Do the Plot on its side }
    X := First;
    WHILE X <= Last DO BEGIN
        Y1 := SIN(3.14159 * X / 180.0);
        Y1 := Factor * Y1;
        Q1 := ROUND(Y1);
        Y2 := 0.005 * X;
        Y2 := Factor * Y2;
        Q2 := ROUND(Y2);
        FOR Step := 0 TO MaxY DO
            IF Step = 0 THEN
```

6.6 Character Type

in Pascal

A character is a single keyboard symbol. It is not a sequence of symbols like a string, but an individual “lonely” symbol. Characters can be placed in various categories:

the ten decimal digits: 0, 1, 2, 3,..., 9

the 26 lower-case letters: a, b, c,..., z

the 26 upper-case letters: A, B, C,..., Z

punctuation and miscellaneous characters, such as !, @, #, [,], :, ", >, =, etc.

There also exist non printing characters that do not correspond to a keyboard symbol. These characters are also called *control characters* and are useful for text formatting, telecommunications, or user interface. They include for instance Ring Bell, Carriage Return, Form Feed, Line Feed, Escape.

Variables whose values are characters, are of type CHAR. They can be declared, assigned, compared, input and output much like INTEGER or REAL variables, but there are important differences.

A Character value must be distinguished from the name of a variable, so single quotes surround characters within programs (but not characters input at the keyboard).

The following declarations show the distinction between character constants and variables.

```
CONST
  PERIOD = '.';
```

```

        FAIL    = 'f';
        NO      = 'N';
        BLANK   = ' ';
        MALE    = 'M';
        F       = 'F';
VAR
    C, Ch, Grade, Reply, Sex,
    Initial, Return, Numeral: CHAR;

```

Assignment of a character value to any variable of type CHAR is easily done:

```

Reply := 'Y';
Sex   := 'F';
Grade := FAIL;

```

Character values can be compared for equality. When they are compared using the greater-than or less-than relation, the ordering is given by the standard sequence of their character code (called ASCII, as seen in the next section). The 26 letters can be compared by their alphabetic order, with 'A' < 'B' < 'C' etc. Some examples involving comparisons are:

```

IF Grade <= 'C' THEN
    Write('Good ');
WHILE ('C' < Grade) AND (Grade <= 'F') DO
    (* Something!! *)
IF (Sex = 'M') OR (Sex = 'm') THEN
    Write('Male ');

```

Numeric character values are also in the usual order: '0' < '1' < '2', etc. These character numerals should not be confused with INTEGER values. CHAR and INTEGER are two entirely different types. The ordering of characters other than the alphabetic and numeric follows the standard character codes, which will be considered in the next section. Because of this, the numeric characters are all considered less than the alphabetic letters, and the upper-case letters are considered less than the lower case letters!

There are only a few operations that involve character values. Arithmetic operations on characters are meaningless. However, the function call Succ(Ch) returns the CHAR value that succeeds the current value of the CHAR variable Ch in the standard sequence. Similarly, the call Pred(Ch) returns the preceding character value.

If the value of CHAR variable L is a lower case letter, the value returned by UpCase(L) is the corresponding upper case character—the value of other characters is unchanged. It can be used to simplify statements such as:

```

IF (Reply = 'Y') OR (Reply = 'y') THEN
    Write('Yes ');

```

which becomes:

```

IF (UpCase(Reply) = 'Y') THEN
    Write('Yes ');

```

The input and output of characters is done as usual with the `Read` and `Write` statements. When `Read(C)` is executed, whatever character is entered (without quotes) gets stored into `CHAR` variable `C`. The following is a code fragment that increments a counter on receiving the required `$` character:

```
Read(Reply);
IF Reply = '$' THEN
    Count := Count + 1;
```

However, since characters are read one by one, all the characters in the input stream are considered. When we read integers, we did not have to worry about extra blanks or end of line characters, as the system would skip them. When we read characters we have to be aware of all characters, as the system does not skip any! For instance, the following segment of program is intended to input the sex, denoted by 'M' or 'F' "safely". It requests the input of one character (either `M` or `m` for male, or `F` or `f` for female), and reads in one character. While the input character is not one of these four acceptable values it keeps looping.

```
(* Input Sex *)
Write('Enter Sex: M or F ');
Read(Sex);
WHILE (UpCase(Sex) <> 'M') AND (UpCase(Sex) <> 'F') DO
BEGIN
    Write('Enter M or F only ');
    Read(Sex);
END;
```

Unfortunately, this piece of program does not quite work in the case where the user does not enter an acceptable response. The output obtained if, for example, the user entered the improper response `x` is:

```
Enter Sex: M or F x
Enter M or F only Enter M or F only
```

which seems somewhat mystifying. The problem is that the computer requires a Return after an input. This pressing of the Return key generates a second input character, a Carriage Return. The sequence of events is thus:

1. The initial "Enter Sex: M or F " is displayed.
2. The user enters two characters, an `x` and a Carriage Return.
3. The program reads the `x`, rejects it as improper and displays the first "Enter M or F ".
4. The program reads the Carriage Return, rejects it as improper and displays the second "Enter M or F ".

This problem is easily solved by inserting another `Read` after each `Read(Sex)`, to "eat-up" this Return character. Nothing is done with this Return character which was read in; it simply prevents this wrong character from being read in by the next `Read` statement. A comment in the program should document this situation as shown in the modified version:

```
(* Input Sex *)
Write('Enter Sex: M or F ');
```



```

Read(Sex);
Read(Return); { to "eat" Carriage Return character }
WHILE (UpCase(Sex) <> 'M') AND (UpCase(Sex) <> 'F') DO
BEGIN
    Write('Enter M or F only ');
    Read(Sex);
    Read(Return); { "eat" Carriage Return }
END;

```

This kind of problem is easy to see when the program is tested after it is written. The modifications are minor and easy to make.

When dealing with character variables, each Read takes a single character from the input line of characters. Thus, successive characters can be read by a sequence of Read statements. For example, the following piece of program reads all the characters of a sentence and counts the blanks in it. The number of blanks could correspond to the number of words if multiple blanks between words are not used.

```

(* Counts Words (blanks) in a Sentence *)
WriteLn('Enter sentence to be analyzed');
BlankCount := 0;
Read(Ch);
WHILE CH <> Period DO BEGIN
    IF CH = Blank THEN
        BlankCount := BlankCount + 1;
    Read(Ch);
END (* WHILE *);
Write('Blank count = ', BlankCount: 2);

```

Note that identifiers Blank and Period refer to character constants previously declared.

Representation of Characters

In the computer, each character is represented by a numerical code in the range 0–255, which, in binary, uses eight bits. The most common character code is known as ASCII (American Standard Code for Information Interchange). Figure 6.13 shows the portion of the ASCII table for the decimal values ranging from 32 to 126. The code values below 32 correspond to control characters that do not print. The values above 126 correspond to special characters.

Figure 6.13 Portion of ASCII codes table for values 32–127

Dec Value	Char	Dec Value	Char	Dec Value	Char	Dec Value	Char	Dec Value	Char
32	SP	48	0	64	@	80	P	96	`
112	p	33	!	49	1	65	A	81	Q
97	a	113	q	34	"	50	2	66	B

82	R	98	b	114	r	35	#	51	3
67	C	83	S	99	c	115	s	36	\$
52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e
117	u	38	&	54	6	70	F	86	v
102	f	118	v	39	'	55	7	71	G
87	W	103	g	119	w	40	(56	8
72	H	88	X	104	h	120	x	41)
57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j
122	z	43	+	59	;	75	K	91	[
107	k	123	{	44	`	60	<	76	L
92	\	108	l	124		45	-	61	=
77	M	93]	109	m	125	}	46	.
62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o
127	DEL								

We see in Figure 6.13 that the ten digits 0 to 9 have ASCII decimal values ranging from 48 to 57. Notice also that the capital letters begin at an ASCII decimal value of 65, and the lower-case letters begin at 97. Thus, there is a difference of 32 between any upper-case letter and its lower case equivalent. The space character, denoted by SP in the figure, has an ASCII value of 32. The character with value 127, DEL, is the keyboard delete character. The table does not show the first 32 ASCII characters because they do not print. However, you might want to use some of them in a program, for instance Bell, Backspace, Horizontal Tab, Line Feed, Vertical Tab, Form Feed, Carriage, Return. It is possible to do that by using their ASCII code preceded by '#', as in the following example.

```
CONST BELL = #7;
      CR   = #13;
```

The function `ORD(C)` is a function that gives the ordinal number of the character `C` in the standard ASCII sequence. For example, `ORD('A')` is 65, `ORD('B')` is 66 and `ORD('Z')` is 90. The values of the `ORD` function for the lower case letters range consecutively from 97 through 122.

It is important to realize that a character numeral such as `'2'` is not equal to the `INTEGER` value of 2. The two items are of very different types and should not even be compared. However, there are a number of ways to convert between the two types.

The conversion of a decimal digit, `Numeral`, of `CHAR` type into its corresponding `INTEGER` type, `Digit`, is useful in a number of applications. It could be done as shown in Figure 6.14.

Figure 6.14 Conversion of a character digit to an integer

```
(* Convert CHAR Numeral to INTEGER Digit *)
IF Numeral = '0' THEN
    Digit := 0
ELSE IF Numeral = '1' THEN
    Digit := 1
ELSE IF Numeral = '2' THEN
    Digit := 2
ELSE IF Numeral = '3' THEN
    Digit := 3
ELSE IF Numeral = '4' THEN
    Digit := 4
ELSE IF Numeral = '5' THEN
    Digit := 5
ELSE IF Numeral = '6' THEN
    Digit := 6
ELSE IF Numeral = '7' THEN
    Digit := 7
ELSE IF Numeral = '8' THEN
    Digit := 8
ELSE IF Numeral = '9' THEN
    Digit := 9
ELSE
    WriteLn('Error ');
```

Alternatively, this conversion can be done in a one line statement by using the fact that the `ORD` of character 0 is 48, and the other digits follow. So the character value '2' with an `ORD` value of 50, is simply 2 away from the value of `ORD('0')`, which is 48. Assuming that character variable `Numeral` contains a numeric character, we find the corresponding `INTEGER` value by simply subtracting 48 from `ORD(Numeral)`:

```
Digit := ORD(Numeral) - ORD('0');
```

Notice that, rather than subtracting explicit value 48 (which by itself has no meaning), we subtracted `ORD('0')`, which relates to character '0'. This is self-documenting code.

For conversion from an `INTEGER` code to the corresponding ASCII character, standard function `CHR` is used. For example, `CHR(65)` is 'A' and `CHR(48)` is the character zero. Conversion of an `INTEGER` digit to its corresponding `CHAR` digit can be done by:

```
Numeral := CHR(ORD('0') + Digit);
```

To test whether a character actually corresponds to a decimal digit we only need to use the simple condition:

```
IF ('0' <= Ch) AND (Ch <= '9') THEN
  ...
```

Our next example program, `AsciiTable`, shown in Figure 6.15 is a program that generates the same ASCII table as in Figure 6.13, but arranged in three columns. This program makes considerable use of the `CHR` function. Notice that in the table the first character (with decimal value 32) corresponds to a blank space, and the last character (with value 127) corresponds to Delete.

Figure 6.15 Pascal program to generate a partial ASCII table and its output

PROGRAM AsciiTable; (* Print a table of ASCII values *)		ASCII Table		
CONST Gap = ' ';		Dec Ch	Dec Ch	Dec Ch
VAR I, J: INTEGER;		32	64 @	96 `
BEGIN		33 !	65 A	97 a
{ Write a Table Header }		34 "	66 B	98 b
Write('ASCII Table');		35 #	67 C	99 c
WriteLn;		36 \$	68 D	100 d
WriteLn;		37 %	69 E	101 e
Write(' Dec Ch Dec');		38 &	70 F	102 f
Write(' Ch Dec Ch ');		39 '	71 G	103 g
WriteLn;		40 (72 H	104 h
{ Fill in table of 3 columns }		41)	73 I	105 i
FOR I := 32 TO 63 DO BEGIN		42 *	74 J	106 j
J := I;		43 +	75 K	107 k
Write(Gap, J: 3, Gap, Gap);		44 ,	76 L	108 l
Write(CHR(J));		45 -	77 M	109 m
Write(Gap, Gap);		46 .	78 N	110 n
J := I + 32;		47 /	79 O	111 o
Write(Gap, J: 3, Gap, Gap);		48 0	80 P	112 p
Write(CHR(J));		49 1	81 Q	113 q
Write(Gap, Gap);		50 2	82 R	114 r
J := I + 64;		51 3	83 S	115 s
Write(Gap, J: 3, Gap, Gap);		52 4	84 T	116 t
Write(CHR(J));		53 5	85 U	117 u
WriteLn;		54 6	86 V	118 v
END { FOR };		55 7	87 W	119 w
END { AsciiTable }.		56 8	88 X	120 x
		57 9	89 Y	121 y
		58 :	90 Z	122 z
		59 ;	91 [123 {
		60 <	92 \	124
		61 =	93]	125 }
		62 >	94 ^	126 ~
		63 ?	95 _	127

The Four Function Calculator example described in Chapter 6 of the Principles book (Fig. 6.25) used character variables. The corresponding Pascal program, Calculate, is shown in Figure 6.16.

Figure 6.16 The Pascal program for a Four Function Calculator with *REAL* values

```
PROGRAM Calculate;
(* Four Function Calculator of Real Values *)

VAR Value, Result: REAL;
    Ch, Action, Return: CHAR;

BEGIN
    Write('Calculate Real Numbers ');
    WriteLn;
    Write('End with ''Q'' for quit ');
    WriteLn;
    Write(' Enter a value: ');
    WriteLn;
    Read(Result);
    Read(Return); { to "eat" Carriage Return }
    Write(' Enter an action: ');
    WriteLn;
    Read(Action);
    Read(Return); { to "eat" Carriage Return }

    Action := UpCase(Action);
    WHILE (Action <> 'Q') DO BEGIN
        Write(' Enter a value ');
        WriteLn;
        Read (Value);
        Read(Return); { to "eat" CarriageReturn }
        IF (Action = '+') OR (Action = 'A') THEN
            Result := Result + Value
        ELSE IF (Action = '-') OR (Action = 'S') THEN
            Result := Result - Value
        ELSE IF (Action = '*') OR (Action = 'M') THEN
            Result := Result * Value
        ELSE IF (Action = '/') OR (Action = 'D') THEN
            Result := Result / Value
        ELSE
            Write('Error ')
        { END IF };
        Write(' The Result is ');
        Write(Result: 10:3); WriteLn;
        Write(' Enter an action ');
        WriteLn;
        Read(Action);
```

```
        Read(Return); { to "eat" Carriage Return }
        Action := UpCase(Action);
    END { WHILE };

    WriteLn('End of Calculation ');

END { Calculate }.
```

The following is an example output from a typical run of program Calculate of Figure 6.12.

```
Calculate Real Numbers
End with 'Q' for quit
Enter a value
9.0
Enter an action
/
Enter a value
5.0
The Result is      1.800
Enter an action
*
Enter a value
100
The Result is  180.000
Enter an action
+
Enter a value
32
The Result is  212.000
Enter an action
q
End of Calculation
```

6.7 Boolean Type in Pascal

As we saw in the Principles book, the Logical type is useful in a number of applications. In Pascal, that Logical type is actually called `BOOLEAN` type. Data items or variables of type `BOOLEAN` can only have one of two values: `TRUE` or `FALSE`. The following fragment of Pascal code illustrates declarations involving the `BOOLEAN` type.

```
CONST T    = TRUE;
      F    = FALSE;
      YES  = TRUE;
      NO   = FALSE;

VAR      Male, Done, Over18, Increasing, Win: BOOLEAN;
          Tall, Aged, Female, Triangle, Error: BOOLEAN;
```

Based on these declarations, the following code fragment shows examples of assignments involving BOOLEAN variables and values.

```
Male      := TRUE;
Done      := F;
Over18    := (Age > 18);
Aged      := Over18;
Tall      := (Height > 72);
Triangle  := (Small + Mid > Large);
```

Notice that in three of the samples above, the value being assigned to a BOOLEAN variable is the result of an arithmetic comparison. Operators on BOOLEAN variables are the three logical operators AND, OR and NOT, as illustrated below:

```
Equilateral := (A = B) AND (A = C );
Win          := (S = 7) OR (S = 11);
Female       := NOT Male;
```

The direct input and output of Boolean values is not possible in Pascal. Input can be done indirectly by entering the characters 'T' and 'F' as follows:

```
(* ReadBool *)
Write('Enter truth value. ');
Write('Give T or F only ');
Read(Reply);
Read(Return);
Write(Reply);
Reply := UpCase(Reply);
WHILE (Reply <> 'T') AND (Reply <> 'F') DO BEGIN
    WriteLn('T or F only ');
    Read(Reply); Write(Reply);
END;
IF Reply = 'T' THEN
    Truth := TRUE
ELSE
    Truth := FALSE;
```

This ReadBool algorithm can easily be put in the form of a useful Pascal procedure. Similarly, the output of the BOOLEAN value of variable Val can also be done indirectly by the following statements.

```
(* WriteBool *)
IF Val THEN
    Write('TRUE ')
ELSE
    Write('FALSE ');
```

These statements will be used to define a Pascal procedure WriteBool(Val).

Boolean expressions, consisting of complex combinations of variables and operations, are commonly used in both WHILE and IF statements. For clarity, parentheses should be used, especially when BOOLEAN and arithmetic expressions are mixed. For example:

```
((Inning <= 9) OR (Score1 = Score2)) AND NOT Rain
```

Truth tables for any given Boolean expressions can be created by nesting loops as in the program `TruthTable`, shown in Figure 6.17, which proves DeMorgan's Theorem:

```
(NOT P) AND (NOT Q) = NOT(P OR Q)
```

Notice in that `TruthTable` program the `BOOLEAN` Loop control variables, `First` and `Second`. The nested loops provide all four possible combinations of these two `BOOLEAN` values. An expression having three such variables would require a further nest and would produce eight combinations. Four variables would produce sixteen rows, and in general n variables produce 2^n rows.

Figure 6.17 The Pascal program *TruthTable*

```
PROGRAM TruthTable;
(* Shows Boolean data and operations *)

VAR First, Second, Left, Right: BOOLEAN;

PROCEDURE WriteBool(Val: BOOLEAN);

BEGIN
  IF Val THEN
    Write('TRUE  ')
  ELSE
    Write('FALSE ');
END; { WriteBool }

BEGIN
  { Write Header }
  WriteLn('Proof of DeMorgan theorem ');
  WriteLn;
  WriteLn('First Second Left Right ');
  WriteLn('----- ----- ----- ');

  { Loop through all truth value combinations }
  FOR First := FALSE TO TRUE DO
    FOR Second := FALSE TO TRUE DO BEGIN
      { Write out Input values of First, Second }
      WriteBool(First);
      WriteBool(Second);

      { Separate Input values from the output }
      Write('| ');

      { DeMorgan's Result }
      Left := (NOT First) AND (NOT Second);
      Right := NOT(First OR Second);
```



```

        { Write out the new values of Left, Right }
        WriteBool(Left);
        WriteBool(Right);
        WriteLn;
    END { Inner FOR };
END { TruthTable }.

```

Notice the use of procedure `WriteBool` whose statements were defined earlier. These statements were used to declare a procedure by enclosing them in a `BEGIN-END` pair, and preceding this with a procedure header. The following is the output obtained after executing the `TruthTable` program.

Proof of DeMorgan theorem

First	Second	Left	Right
-----	-----	-----	-----
FALSE	FALSE		TRUE TRUE
FALSE	TRUE		FALSE FALSE
TRUE	FALSE		FALSE FALSE
TRUE	TRUE		FALSE FALSE

6.8 More Types

Big Types in Pascal

So far, the only numeric types that we have discussed have been `INTEGER`, and `REAL`. These types describe numbers of a very common range. For example, values of type `INTEGER` include all whole numbers between $-32,768$ and $32,767$ (represented in 16 bits). Variables of this `INTEGER` type are however insufficient for many purposes, such as counting the population in even a small town!

In order to be able to represent larger integer values in Pascal, another type, `LONGINT`, is available. This type uses 32 bits and covers the range going from $-2,147,483,648$ to $2,147,483,647$. The usual Integer operations (`Read`, `Write`, `+`, `-`, `*`, `DIV`, `MOD`, etc.) apply to these long integers as well. Similarly, for numbers beyond the range of `REAL` numbers, -3.38×10^{38} to 3.38×10^{38} , there is a `DOUBLE` data type that covers the range -1.79×10^{308} to 1.79×10^{308} .

Even Bigger Types

If the extended type `LONGINT` is still insufficient for our needs, then it is possible to create even larger types. As an example, we will create a new type called `ELINTEGER` (Extra Long Integer), which will express integers of one hundred or even more digits. Later, we will create an entire Library, called `ELintegerLib`, with various arithmetic operations on these very long

integers. The operations will not be in their usual form (+, -, *, /), but rather in the form of procedures such as:

```
ELIntegerAdd(I, J, K);
```

which adds the ELINTEGER I to ELINTEGER J, and produces another ELINTEGER K. The symbols for comparison (<, =, <=, etc.) of ELINTEGERs would also be in the form of procedures such as:

```
Equals(I, J), or IsLessThan(K, 2)
```

Smaller Types

Often, some variables need only take only a few values. For example, there are only two sexes (male, female), four directions (North, East, South and West), seven days of the week (Sunday, Monday, etc.) and twelve months of the year (January, February, etc.) In the older programming languages, these were often coded as small INTEGER values. For example, Sunday was assigned a value of 0, Monday was 1, Tuesday was 2, etc. and Saturday was assigned a 6. This led to more readable statements. For example, instead of writing

```
IF Date = 5 THEN ...,
```

it is clearer to write

```
IF Date = Friday THEN ...".
```

Pascal programs could still use such a mechanism, but it has problems. For example, we could define the following constants:

```
CONST
  Male   = 0;  Female = 1;
  North  = 0;  East   = 1;
  South  = 2;  West   = 3;
  Sun    = 0;  Mon    = 1;
  Tue    = 2;  Wed    = 3;
  Thur   = 4;  Fri    = 5;
  Sat    = 6;
```

All of these are of the same numeric type, and so if an error is made by comparing two different types, for example:

```
IF Date = Male...
```

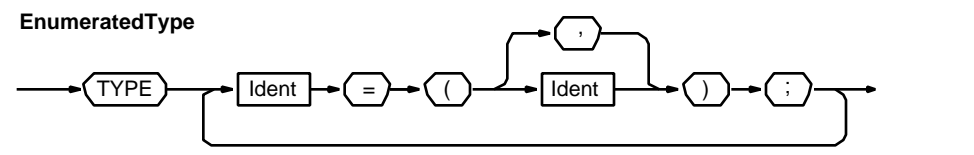
it cannot be detected by the system. This particular error of comparing Date with Male is also very difficult for humans to detect. The English language and its two meanings for "Date" could cause considerable confusion. Pascal, with its emphasis on type-checking, has an alternative mechanism that allows a programmer to create different types for each different kind of data being worked with. The next section shows how to create types for working with the above kinds of data, including SexType, DirectionType, and WeekDayType.

6.9 Programmer Defined Types

Enumerated Types

It is often convenient for a programmer to define a new data type. It allows the “real-world” to be reflected in its model represented by a program, through data types that match more closely the real world data than the `INTEGER`, `REAL`, `CHAR` data types that are part of Pascal. Oftentimes, the programmer defined data types have only a few values and can thus be defined by listing, or enumerating their possible values. An *enumerated type* is defined by giving names, using distinct identifiers, to each possible value of the data type. The syntax of such a definition is shown in the syntax diagram shown in Figure 6.18.

Figure 6.18 Syntax diagram for declaration of an Enumerated Type



We could, for example, define the seven weekdays by the following type:

```
TYPE WeekDay = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

The values of this `WeekDay` type (`Sun`, `Mon`, etc.) need not all consist of three characters. Here, we have used a uniform size for consistency only, and in fact we could have used the complete names.

We give below some examples of declarations of some other common enumerated types. Notice that the word `TYPE` must precede the declarations. For documentation purposes, the name of the type often includes the word `Type`. Sometimes the name of the type is written entirely in capitals.

```

TYPE
  SexType      = (MALE, FEMALE);
  GradeType    = (F, D, C, B, A);
  DirectionType = (North, East, South, West);
  DeviceKind   = (keyboard, printer, console);
  SEASON       = (Spring, Summer, Fall, Winter);
  StateType    = (increasing, decreasing, still);
  ProfType     = (Lecturer, Assistant,
                  Associate, Full);
  CoinType     = (penny, nickel, dime,
                  quarter, halfDollar);
  StudentType  = (Freshman, Sophomore, Junior,
                  Senior, Graduate);
  MonthType    = (January, February, March, April,

```

```
May, June, July, August,  
September,  
October, November, December);
```

Values of the type are defined by unique identifiers. The order in which the identifiers are given is by definition an increasing order. This ordering makes it possible to compare the values of an enumerated type. For example, with the above declarations, `January` is less than `February`, and a grade of `F` is less than a grade of `D`. Order may not always be important, for example, although `MALE` was listed before `FEMALE` and therefore would compare as less than, this is really an accidental effect of having to list something first. Also the same identifier cannot appear in two types. For example, if `Sun` is a value of type `WeekDay`, then it cannot also be enumerated in some other type, for example, in a `SolarSystemType`.

Once the enumerated types have been defined, they can be used to declare variables, in the same manner that built-in types, like `INTEGER`, etc. can be used.

```
VAR Day, Date, today, tomorrow: WeekDay;  
    Grade, G, MaxGrade, Final, average: GradeType;  
    Month: MonthType;  
    State: StateType;
```

There are only a few possible actions on values of enumerated types, including assignment and comparison as shown in the following fragments of code. Notice how readable they are.

```
Day := Mon;  
IF Day = Fri THEN  
    ...  
IF Grade <= D THEN  
    ...  
IF (May < Month) AND (Month < July) THEN  
    ...  
WHILE State = increasing DO  
    ...
```

There are also a few standard functions that operate on enumerated types:

`SUCC(X)`

This function returns the successor value, i.e. the next value in the enumeration list. For example, if `Day` has the value `Tue` then `SUCC(Day)` has the value `Wed`. If `Day` has the value `Sat` then `SUCC(Day)` is an error since `Sat` has no successor in the enumeration list.

`Pred(X)`

This function is the converse of `SUCC` and returns the predecessor value in the enumeration list. For example, the value of `PRED(Wed)` is `Tue`, while `PRED(Sun)` is an error.

Using this for example, the following program fragment loops through all the working days of the week.

```
Day := Mon;
WHILE Day <= Fri DO BEGIN
  (* body *)
  Day := SUCC(Day);
END;
```

In the real world, some data types are really cyclic and contrary to the way in which the `SUCC` function works, the successor of `Sat` is `Sun`. The next value of such a cyclic type can easily be determined by adding a test, as in the following piece of program.

```
(* Determine The Next Day *)
IF Today = Sat THEN
  Tomorrow := Sun
ELSE
  Tomorrow := SUCC(Today);
```

Standard function `ORD` that we have seen applied to character values, also operates on enumerated types.

`ORD(X)`

is a function that indicates the order or position of `X` (with the first value being 0). Thus, the value of `ORD(January)` is 0 and `ORD(December)` yields 11.

There are no standard input and output operations for enumerated types, but they can easily be created. For example, the following fragment of program can output the weekday value of the variable `Day`.

```
(* Output of weekday for given Day of WeekDay type *)
IF Day = Sun THEN
  Write('Sunday ')
ELSE IF Day = Mon THEN
  Write('Monday ')
ELSE IF Day = Tue THEN
  Write('Tuesday ')
ELSE IF Day = Wed THEN
  Write('Wednesday ')
ELSE IF Day = Thu THEN
  Write('Thursday ')
ELSE IF Day = Fri THEN
  Write('Friday ')
ELSE IF Day = Sat THEN
  Write('Saturday ')
ELSE (* none of the above *)
  Write('Error ');
```

Subrange Types In Pascal

Another programmer defined type, the *subrange type* is a type whose values are restricted to a contiguous subrange of values of another type—the base type. The values must be consecutive, one following the other in order. For example, the digits 0 to 9 form a subrange of the `INTEGER` type. Similarly, the days `Mon` to `Fri` are a subrange of the `WeekDay` type. Subrange types are declared in the following manner:

```
TYPE typename = Minvalue .. Maxvalue;
```

where `Minvalue` is less than `Maxvalue`, and both are constants of the base type.

Examples of declarations of subrange types are:

```
TYPE DayOfMonth      = 1..31;
   MidWeekType       = Mon..Fri;
   YearType          = 1900..2000;
   BitType           = 0..1;
   DiceThrow         = 1..6;
   DecimalDigit      = 0..9;
   VHFChannels       = 2..13;
   WorkHours         = 0..168;
   CountdownTyp      = -10..0 ;
   GradeType         = 'A' .. 'F';
   PercentType       = 0..100; (* INTEGER not REAL *)
```

```
VAR Year, BirthYear: YearType;
    MidTerm, Project: PercentType;
```

The set of operations that are applicable to a subrange type is “inherited” from its base type, the type of the original range. Subranges cannot involve `REAL` values.

It is also possible to use characters as the base type of a subrange type. The `GradeType` shown above includes the five characters from `A` to `F` (including `E`).

Error checking is another important feature of the Pascal subrange types. If a value outside of the range is assigned to a variable of a subrange type, this causes an error message, warning the programmer of an unintended situation. For example, if the variable `HoursWorked` were of type `WorkHours`, then it would be limited to the number of hours in a week ($7 \times 24 = 168$). If a value higher than this were assigned to it, an error would occur.

6.10 Strings in Pascal

Strings are sequences of characters. In Pascal, they are described by the built-in type `STRING`, which is limited to a maximum of 255 characters. Shorter strings than this may be declared by specifying an explicit maximum length. For example, if 25 characters were sufficient to represent a person’s name, a variable `Name` could be declared of type `STRING[25]`. Variables to hold

sentences could be declared of type `STRING[80]`, etc. As we have already seen, constant string values are sandwiched between single quotes. String constants and variables are declared in a manner similar to the following examples.

```
CONST SPACE      = ' ';
      PERIOD      = '.';
      GREETING    = 'Hello there ';
      PROMPT      = 'Enter first name, then last ';
```

```
VAR LastName, FirstName, FullName,
    Initial, Street: STRING[25];
    Address, Sentence, Saying, Poem: STRING;
```

Simple actions on strings include input and output through `Read` and `Write`, as well as assignment. Concatenation is the operation of joining together two strings, and is denoted by a plus (+) sign between the two strings. For example, the body of a very short program that uses the above declarations, and reads two input strings (separated by a Return) is:

```
BEGIN
  Write(PROMPT);
  Read(FirstName, LastName);
  FullName := FirstName + SPACE + LastName;
  Write(GREETING, FullName, PERIOD);
END.
```

Besides `Read` and `Write` there are a number of standard functions and procedures that operate on strings. Among these, the following three functions are used often.

`LENGTH(Str)`

is a function that counts the number of characters in string `Str`. For example, `Length(Greeting)` has the value 12.

`POS(Pat, Str)`

is a function that searches for the occurrence of a sequence of characters, the pattern `Pat`, in the string `Str`, and returns the position of the first occurrence of that pattern if there is one, otherwise it returns the value of zero (0). For example:

```
POS( SPACE, 'John Motil' ) returns the value of 5.
POS( 'Mo' , 'John Motil' ) returns the value of 6.
POS( 'mo' , 'John Motil' ) returns the value of 0.
```

`COPY(Str, Position, Num)`

is a function that returns a substring of `Num` characters from string `Str` beginning at `Position`; this copy action does not modify string `Str`. For example:

```
Copy('Mississippi', 4, 3) returns the value 'sis'.
```

As an example of a standard procedure operating on Strings, we'll look at `Delete`.

`Delete(Str, Position, Num)`

is a procedure (not a function) that deletes from string `Str` a number of characters `Num` beginning at `Position`. For example, after the two actions:

```
Str := 'Pascal';
Delete(Str, 3, 3);

the value of Str will be 'Pal'.
```

Comparisons of Strings use the alphabetic order (called *lexicographic ordering*). One string is less than another if it comes before it alphabetically (according to the ASCII order). Equality requires exactly the same characters, including blanks. For example:

```
'Age' < 'Beauty'   because Age comes before Beauty
'able' > 'Able'     because capital letters come
earlier

                               in the  ASCII order
'Rose' <> 'Rose '   because the second Rose has a space
'love' < 'lovely'   because love is shorter
```

Figure 6.19 shows a Pascal program `NameParse` that demonstrates some of the operations that can be performed on strings.

Figure 6.19 Pascal program demonstrating string actions

```
PROGRAM NameParse;
(* Demonstrates some String actions *)
(* that involve names of people *)

TYPE ShortString = STRING[25];

CONST Space      = ' ';
      Hyphen     = '-';
      Greeting   = 'Hello there ';

VAR FirstName, LastName, FullName: ShortString;
    Count, NameCount, Gap: INTEGER;

BEGIN
  Write('Enter the number of names: ');
  ReadLn(NameCount);
  WriteLn;
  WHILE NameCount > 0 DO BEGIN
    Write('Enter a name, last name first: ');
    Read(FullName);
    Gap := POS(Space, FullName); { NOTE }
    IF Gap > 0 THEN BEGIN
      LastName := Copy(FullName, 1, Gap);
      Delete(FullName, 1, Gap); { NOTE }
      FirstName := FullName;
      IF Length(LastName) <= 4 THEN
```



```
        WriteLn('That is a short last name');
    IF Pos(Hyphen, LastName) <> 0 THEN
        WriteLn('That is a hyphenated name');
    IF FirstName = 'Bill' THEN { etc., etc. }
        WriteLn('Bill is a good name ');
    FullName := FirstName + Space + LastName;
    WriteLn(Greeting, FullName);
    WriteLn;
END { IF };
NameCount := NameCount - 1;
END { WHILE };
END { NameParse }.
```

Program `NameParse` asks for the number of input names, and repeatedly applies the same processing to all the input names. It reads a full name made of two parts separated by a space. It locates the space, and copies the first part of the string into the last name, and the last part of the string into the first name. It then checks the last name and the first name and outputs a number of messages, before displaying a greeting with the recomposed full name.

The output from a typical run of program `NameParse` follows.

```
Enter the number of names 3

Enter a name, last name first Motil John
Hello there John Motil

Enter a name, last name first Gates Bill
Bill is a good name
Hello there Bill Gates

Enter a name, last name first Do-Dah Bill
That is a hyphenated name
Bill is a good name
Hello there Bill Do-Dah
```

6.11 Simple Files in Pascal

A file is an ordered collection of data, such as the characters of a text, the item counts in an inventory, or the records of a patient. Files are often used to store large amounts of data on secondary storage devices (such as tapes or disks). The data in files may be accessed and manipulated by programs.

The simplest kind of files are sequential files where the data may be read exactly in the order in which they were written. Files that are organized so that their data can be accessed in any order are known as *random access files* and are much more complex. Random access files are not treated in this book.

Sequential files can be viewed as long linear tapes similar to audio or video tapes. At present, all files will be of type `TEXT`, which means that they are a

sequence of ASCII characters, where `INTEGER` and `REAL` values are represented in the form that they appear in Pascal programs (as a sequence of characters).

Actions on files involve reading and writing, opening and closing, and assigning. Apart from the standard input and output files, which correspond to the keyboard and the screen `Text Window`, files are accessed through *file variables* of type `TEXT`. Thus, an actual file on a disk might be referred to through a variable, like `DataFile`. To do this, we need a declaration like:

```
VAR DataFile: TEXT;
```

This file variable must be associated to the actual data file through a call to the procedure `Assign`. For example:

```
Assign(DataFile, 'Rain.Data');
```

associates the file whose name on disk is `Rain.Data` to the file variable `DataFile`.

Before data values can be read from a file, the file must be prepared by executing a `Reset` statement referencing the file variable, in this case:

```
Reset(DataFile);
```

Following this, each `Read` statement aimed at the data in the file, must include the name of the file variable as its first argument. Thus, the statement

```
Read(DataFile, Value);
```

will read the next data item from the file associated with `DataFile`, the disk file `Rain.Data`, into the variable `Value`. Finally when all the data has been read (often towards the end of a program), the file associated with `DataFile` is closed with a statement of the form:

```
Close(DataFile);
```

When reading a file, it is possible to detect the end of the file, by using a call to the Boolean function `EOF`:

```
EOF(DataFile)
```

and this condition should be tested before reading a value.

Figure 6.20 compares two versions of the program `Mean` that we saw earlier in this chapter. `Mean1`, on the left, takes data from the keyboard, whereas `Mean2` reads the data from the file `Rain.Data`, and echoes the values read to the screen. Note the differences between the two programs carefully, extra blank lines have been inserted into `Mean1` so corresponding statements are always side-by-side.

Figure 6.20 Comparison of two versions of the Mean program

<pre> PROGRAM Mean1; (* Finds the Mean of many values *) (* Uses a terminating sandwich *) VAR Terminator, Sum, Value, Count: INTEGER; Mean: REAL; BEGIN { Setup for entering many values } Write('Enter a terminal value: '); Read(Terminator); WriteLn('Enter all the values: '); { Do the Mean computation } Sum := 0; Count := 0; Read(Value); WHILE Value <> Terminator DO BEGIN Sum := Sum + Value; Count := Count + 1; Read(Value); END { WHILE }; Mean := Sum / Count; { Output the Resulting Mean } Write('The mean value is '); Write(Mean: 7:2); END { Mean1 }.</pre>	<pre> PROGRAM Mean2; (* Finds the Mean of many values *) (* Uses simple text files *) VAR Terminator, Sum, Value, Count: INTEGER; Mean: REAL; DataFile: TEXT; BEGIN { Setup for entering many values } Assign(DataFile, 'Rain.Data'); Reset(DataFile); Read(DataFile, Terminator); WriteLn('The values are: '); { Do the Mean computation } Sum := 0; Count := 0; Read(DataFile, Value); WHILE Value <> Terminator DO BEGIN WriteLn(Value:4); { ECHO } Sum := Sum + Value; Count := Count + 1; Read(DataFile, Value); END { WHILE }; Mean := Sum / Count; { Output the resulting Mean } Write('The mean value is '); Write(Mean: 7:2); Close(DataFile); END { Mean2 }.</pre>
--	--

Figure 6.21 shows a comparison of the outputs of Mean1 and Mean2.

Figure 6.21 Comparison of results from *Mean1* and *Mean2*

Contents of file Rain.Data	
-1	
11	
22	
33	
44	
55	
66	
77	
88	
99	
-1	
Output obtained from Mean1	Output obtained from Mean2
Enter a terminal value: -1	
Enter all the values	The values are:
11	11
22	22
33	33
44	44
55	55
66	66
77	77
88	88
99	99
-1	
The mean value is 55.00	The mean value is 55.00

Writing values into a file is similar to the reading process just shown. As with reading, we need a file variable, which is declared in the same manner:

```
VAR OutputFile: TEXT;
```

The actual file name is then assigned to the file variable, through a call to the procedure `Assign`:

```
Assign(OutputFile, 'Results.Data');
```

In this case, the output from the program will be written to a file called `Results.Data`. The file is then opened or prepared for use by the statement:

```
Rewrite(OutputFile);
```

which sets or rewinds the file to its beginning, so that the data written by the program will overwrite any values that already existed in the file. Each `Write` statement to this file must include the file variable as its first argument, as in:

```
Write(OutputFile, Value);
```

Each execution of such a `Write` statement appends a value to the end of the file.

Finally, when all the writing is completed, i.e. the end of the program is reached, the file is closed with a statement identical to what we saw above for input files.

```
Close(OutFile);
```

More Files: Input and Output

A very common situation in computing involves reading from one file, doing some actions, and then writing to another file. As an example, suppose we wish to take a text file of mixed upper and lower case letters and convert it to entirely upper case letters. This might be useful for counting occurrences of words, or checking spelling. We can then search for the one capitalized word (such as "THE") rather than the many variations of the word (such as "The" or "the" or "THE").

Figure 6.22 Program Capper

```
PROGRAM Capper;
(* Capitalizes all lowercase letters *)
VAR Ch: CHAR;
    InFile, OutFile: TEXT;
BEGIN
    Assign(InFile, 'Source.Data');
    Reset(InFile);
    Assign(OutFile, 'Target.Data');
    Rewrite(OutFile);
    WriteLn(OutFile, 'Header');
    WHILE NOT EOF(InFile) DO BEGIN
        Read(InFile, Ch);
        Ch := UpCase(Ch);
        Write(Ch); { to the Screen }
        Write(OutFile, Ch);
    END { WHILE };
    Close(InFile);
    Close(OutFile);
END { Capper }.
```

The program `Capper`, shown in Figure 6.22, converts all the characters of some input file, `Source.Data`, into upper case characters and puts these into an output file called `Target.Data`. Again, file variables `InFile` and `OutFile` are used to refer to the actual files `'Source.Data'` and `'Target.Data'`, external to the program. The first two calls to the `Assign` procedure establish this correspondence.

The main body of the program is a simple loop which reads a character from the input file, capitalizes it, echoes it to the screen, and writes it to the output file. This process is repeated as long as the end of the input file is not read

```
WHILE NOT EOF(InFile) DO
```

Finally, once this loop is finished, the two files are closed.

Figure 6.23 Program Flasher

```
PROGRAM Flasher;
(* Flash-card Questions & Answers *)
(* Shows the input of a file name *)

VAR Question, Answer, Reply, FileName: STRING;
    Qafile: TEXT;
BEGIN
    { Enter file name from keyboard }
    Write('Enter name of QA file: ');
    ReadLn(FileName);
    Assign(Qafile, FileName);
    Reset(Qafile);
    WriteLn;

    { Do question and answer }
    ReadLn(Qafile, Question);
    WHILE Question <> 'END' DO BEGIN
        WriteLn(Question);
        ReadLn(Reply);
        ReadLn(Qafile, Answer);
        IF Reply = Answer THEN
            WriteLn('Correct ')
        ELSE
            WriteLn('It is ', Answer);
        ReadLn(Qafile, Question);
        WriteLn;
    END { WHILE };
END { Flasher }.
```

Program Flasher, shown in Figure 6.23, is a program that asks people questions and analyzes their answers. The questions are read from a file, while the answers are read from the keyboard. The program compares this response to the answer stored in the file, and indicates whether it is correct (and if not, it provides the correct answer). The Questions and Answers file alternates questions and answers lines, as in the following example.

```
What is 1 + 1?
2
Who is buried in Grant's tomb?
Grant and his wife
What is the chemical symbol for gold?
Au
END
```

Questions and answers here are simply strings that are read one at a time by `ReadLn(QAfile, Question)`, which stops reading at the End-of-line or

Return character. A typical run of this program follows with the user's response in bold.

```

Enter name of QA file: MiscQA
1 + 1 =
2
Correct
Who is buried in Grant's tomb?
Grant
It is Grant and his wife
What is the chemical symbol for gold?
AG
It is Au

```

A important feature of this program is that it can access different files which could correspond to different categories or levels of questions. The name of the actual file to be accessed is input by the user in the first few lines of the program body. It is read into string variable `FileName`, and file variable `QAFile` is assigned this name. This mechanism is important because the program need not be recompiled for each different input file.

6.12 Modification of Programs

As we have mentioned when we introduced the seven-step problem solving method in the Principles book, it is very common to modify programs. For example, the Bisection algorithm, discussed in Chapter 6 of the Principles book, could be modified in two very different ways. One version, `SquareRoot`, finds the square root of any real number. Another version, `Guesser`, plays a guessing game involving only integers. Both versions shown in Figure 6.24, are based on the same general Bisection algorithm. These two algorithms are so similar, and yet so different.

Figure 6.24 Two versions of the Bisection algorithm

<i>Input X</i>	<i>Input X</i>
<i>Set High to X</i>	<i>Set High limit</i>
<i>Set Low to 0</i>	<i>Set Low limit</i>
<i>Set SqRoot to</i> $(High + Low) / 2$	<i>Set Guess to</i> $(High + Low) / 2$
<i>While (SqRoot × SqRoot) < X</i>	<i>While Guess is not correct</i>
<i>If (SqRoot × SqRoot) > X</i>	<i>If Guess is too high</i>
<i>Set High to SqRoot</i>	<i>Lower High limit</i>
<i>Else</i>	<i>Else</i>
<i>Set Low to SqRoot</i>	<i>Raise Low Limit</i>
<i>Set SqRoot to</i> $(High + Low) / 2$	<i>Set Guess to</i> $(High + Low) / 2$
<i>Output SqRoot</i>	<i>Output Guess</i>

The bisection algorithm basically operates by taking two limiting values (Low and High) and adjusting them successively to bracket the required result with ever closer bounds. The adjustment is made by finding the mid-point between the Low and High, and assigning that to one or the other of these extreme points. This effectively halves the “search space” at each adjustment, and the process converges very quickly to a solution. The two corresponding Pascal programs and their output from typical runs are shown in Figure 6.25.

Figure 6.25 The Pascal programs *SquareRoot* and *Guesser*

<pre> PROGRAM SquareRoot; (* Find Square Root by Bisection *) CONST Err = 0.001; VAR High, Low, Mid, X: REAL; BEGIN Write('Enter a value: '); Read(X); WriteLn; Low := 0.0; High := X; Mid := (Low + High) / 2.; WHILE ABS(Mid*Mid - X) > Err DO BEGIN IF (Mid * Mid) > X THEN High := Mid ELSE Low := Mid; WriteLn(Mid: 7: 3); Mid := (Low + High) / 2.; END (* WHILE *); Write('The square root is '); Write(Mid: 7: 3); END { Square Root }. </pre>	<pre> PROGRAM Guesser; (* Guessing using Bisection *) CONST Max = 1024; VAR High, Low, Mid, Trial, NumTrials: INTEGER; Reply, Return: CHAR; BEGIN WriteLn('Pick an integer '); WriteLn('from 0 to 1023. '); WriteLn; Low := 0; High := Max; Mid := (Low + High) DIV 2; Trial := 1; WHILE Trial <= 10 DO BEGIN Write('Is it less than '); Write(Mid:4, ' '); Read(Reply); Read(Return); Reply := UpCase(Reply); IF Reply = 'Y' THEN High := Mid ELSE Low := Mid; Mid := (Low + High) DIV 2; Trial := Trial + 1; END { WHILE }; WriteLn; Write('Your number was ', Mid: 4); END { Guesser }. </pre>
---	---

Enter a value 2.0	Pick an integer from 0 to 1023
1.000	
1.500	Is it less than 512 y
1.250	Is it less than 256 n
1.375	Is it less than 384 n
1.437	Is it less than 448 y
1.406	Is it less than 416 n
1.422	Is it less than 432 y
	Is it less than 424 y
The square root is 1.414	Is it less than 420 n
	Is it less than 422 n
	Is it less than 423 y
	Your number was 422

The `SquareRoot` program involves `REAL` numbers, and finds that value of the midpoint whose square is near the input value within a given error amount. In this case the error amount was fixed as a constant at 0.001. For the given input value of 2 it required 7 iterations before finally arriving at the square root. For larger values and smaller errors it would also require more iterations before halting. The intermediate values shown in the given trace were produced by the `Write` statement inside the loop.

The `Guesser` program is very similar, but it involves `INTEGER` values and differs in details. The easiest detail to miss is the divide operation, which must be changed from `/` to `DIV`. Two new variables, `Reply` (a `CHAR`) and `Trial` (another `INTEGER`) also need to be introduced. This program does a fixed number of trials ($\log_2 \text{Max}$ or 10 in this case), so the `WHILE` loop could have been replaced by a `FOR` loop.

We have used extra blank lines to keep the programs main structure in parallel. A comparison of these two programs shows that `Guesser` is very interactive, or input-output intensive, whereas `SquareRoot` is much less interactive, with a single input and a single output (the trace given with all the intermediate values is not necessary).

As usual, it is possible to extend these programs. For example, `Guesser` could be generalized to guess within any number of trials (not just the 10 trials of the range 0 to 1023). The number of trials could be defined as a constant, or could be entered by the user. The program could also be made robust by providing more detailed instructions (if requested), and by testing the input responses. This will be done in the next section.

6.13 Programming Style

Documentation

Style in programming may seem unnecessary, but it is actually very important. Documentation and comments may seem obvious at the time they are written,

but at a later time, when we have forgotten the details, any information is very valuable. As we progressed in this book, we have already incorporated many of the concepts of style into our programs. For example, we have often used comments, and we have carefully selected suitable variable names. We have also used indentation constantly and consistently. Remember, consistency of form is often better than the details of the particular way in which the statements are arranged.

We'll illustrate the principles of *internal documentation* with a program example. Internal documentation is the element of documentation that is part of the program text. Our example, program `Guesser2`, is shown in Figure 6.26 together with the output obtained from a typical run. At the beginning of the program (in a starred box) is a brief statement of the goal of the program, the author and date of creation. Other relevant information could also be put there, including a list of inputs and outputs, special limitations, assumptions and warnings. The starred boxes should not be overused, but there should be one at the beginning of each program, and also at the beginning of each procedure and function.

Figure 6.26 Documented version of Guesser

<pre> PROGRAM Guesser2; (***** (* Simple Guessing Game using Bisection *) (* Shows fancy layout and documentation *) (* by A. Nonymous on February 28, 1994 *) *****) CONST YES = 'Y'; NO = 'N'; NUM = 10; { Number of tries } MAX = 1024; { Highest value } { MAX is 2 to the power NUM } VAR High, Low, Mid, Trial, NumTrials: INTEGER; Reply, Return: CHAR; BEGIN { Offer instructions to user } WriteLn('Do you want instructions? '); Write('If so enter Y (for yes): '); ReadLn(Reply); WriteLn; </pre>	<pre> Do you want instructions? If so enter Y (for yes): y This is a guessing game You may pick any integer From 0 to 1023 It will be guessed in 10 tries Pick a number Trial 1 Is it less than 512? y Trial 2 Is it less than 256? n Trial 3 Is it less than 384? m Enter Y or N only: b Enter Y or N only: n </pre>
--	--

<pre> { Provide instructions if requested } Reply := UpCase(Reply); IF Reply = YES THEN BEGIN WriteLn('This is a guessing game '); WriteLn('You may pick any integer'); WriteLn('From 0 to ', MAX-1: 4); WriteLn('It will be guessed in '); WriteLn(NUM: 2, ' tries '); WriteLn; END { instructions }; { Initialize } WriteLn('Pick a number '); Low := 0; High := MAX; Mid := (Low + High) DIV 2; Trial := 1; { Loop a number of times } WHILE Trial <= NUM DO BEGIN { Prompt for an input } WriteLn('Trial ', Trial: 2); Write('Is it less than '); Write(Mid: 4, '? '); ReadLn(Reply); Reply := UpCase(Reply); { Test for correct input } WHILE (Reply <> YES) AND (Reply <> NO) DO BEGIN Write('Enter Y or N only: '); ReadLn(Reply); Reply := UpCase(Reply); END { WHILE wrong input }; WriteLn; { Adjust the middle value } IF Reply = YES THEN High := Mid ELSE Low := Mid; Mid := (Low + High) DIV 2; Trial := Trial + 1; END { WHILE }; { Output the final guess } Write('Your number was '); Write(Mid: 4); END { Guesser2 }. </pre>	<pre> Trial 4 Is it less than 448? y Trial 5 Is it less than 416? n Trial 6 Is it less than 432? y Trial 7 Is it less than 424? y Trial 8 Is it less than 420? n Trial 9 Is it less than 422? n Trial 10 Is it less than 423? y Your number was 422 </pre>
--	---

The two constants YES and NO (better than Y and N) are used for the convenience of reading. The number of trials, NUM, and the highest value, MAX, are also declared as constants for the flexibility they offer: with them, it is easy to change the game to work with other ranges. This method of allowing

easy change of “constants” is called parameterization, and is very important in large production programs, where there may be many occurrences of a constant. The relationship between the parameters `NUM` and `MAX` is also given as a comment for convenience of modification. Note that we have used capital letters to write the constants names. This is part of our style, and makes it possible to differentiate between variables and constants rather easily.

The main program has gaps which break it up into “chunks”, which are easier to comprehend when reading. Each chunk is preceded by a comment describing what is done in the chunk (not how it is done). These comments alone form a high-level algorithm:

```
{ Offer instructions to user }
{ Provide instructions if requested }
{ Initialize }
{ Loop a number of times }
{ Prompt for an input }
{ Test for correct input }
{ Adjust the middle value }
{ Output the final guess }
```

Notice also the “friendly” nature of the dialogue as shown to the right of the program:

- it offers to help with instructions,
- it reports the trial number,
- it prompts the user,
- it separates the trials,
- it forgives bad inputs.

Moderation is as important with documentation as it is with everything in life. Too much documentation may be overwhelming, and too little may be useless.

Further Guidelines for Identifiers

We already know that identifiers chosen for naming variables, constants, procedures, functions and programs cannot be Pascal keywords. The compiler will prevent us from making that mistake. However, to avoid confusion it is also wise not to choose identifiers that are names of standard procedures or functions. If you do, the system will accept it but you won’t be able to use the standard item anymore as it is hidden by your new declaration.

We have seen that capitalizing the names of constants is useful because it allows us not to confuse variables and constants when reading the program. We encourage you to capitalize the first letter of variables, and to keep your identifiers reasonably short. Long identifiers like

```
Textwithnogapsisalsoveryhardtoread,
```

are a pain. When an identifier is made of several words, the first letter of each word should be capitalized as

```
InterestRate, OverTime, CountOfSheep
```

Identifiers should only comprise letters and digits. Apart from rare exceptions, very short names should not be used at all

```
A, B, C, I, J, X1, X2
```

Instead, meaningful short names should be used:

```
Sum, Index, Top, Difference, Root1, Root2
```

Procedure names should have the first letter capitalized and be commands (verbs):

```
Sort, ComputeCost, Search
```

Function names should be nouns, as they represent a value:

```
Tangent, Volume, CompoundInterest
```

Type names should contain the word `Type` or its root, or end with `T`

```
BitType, WeekdayTyp, AccountT, EmployeeT
```

In all the programs, spaces should be used inside lines to make the text more readable. In particular, assignment signs and arithmetic operators should be preceded and followed by a single space:

```
Y1 := SIN(3.14159 * X / 180.0);
```

In procedure and function calls, parameters should be spaced as in the example:

```
Divide(First, Second, Quotient, Remainder);
```

A Bad Style Horror Story

The program shown in Figure 6.27 actually runs as it stands, and was designed in earnest by a person with very little style. We have made only very minor changes to it. It is severely unfriendly. We're sure that by looking at it, you can indicate a number of things that are not done well, and that you can suggest improvements as well. Do not try to fix it up—it is not worth repairing. It's the kind of program that must be redone completely.

Figure 6.27 The mean Mean program

```
PROGRAM MeanMean;
(* Mean Unfriendly program *)
(* Try it; You'll hate it *)
CONST ZERO=0; ONE=1;
VAR I, C, N, X, S, true: INTEGER;
LABEL OUT;
BEGIN
Write ('HI. I'M YOUR FRIEND');
Write ('TO HELP YOU AVERAGE'); WriteLN;
Write ('HOW MANY INPUTS DO YOU HAVE');
Read ( N );
S := 0;
```

```
REPEAT WriteLn;
Write ('IMPUT COUNT '); WriteLn;
Read (C);
Write ('INPUT X '); Read(X);
Write ('ECHO VALUE'); Write (X: 5);
S := S + X DIV N;
IF (C = N)
THEN BEGIN
Write ( S, 10 );
goto out;
END;
IF (X = -99) THEN BEGIN
S := S - X DIV N;
Write ( S: 10 );
goto out; END;
Write ('DO YOU WANT ANOTHER VALUE?'); WriteLn;
Write ('TYPE 1 IF TRUE 0 IF FALSE. '); WriteLn;
Read ( true );
UNTIL true = 0;
out: ;
END.
```

A typical output produced by an execution of this program follows.

```
HI. I'M YOUR FRIENDTO HELP YOU AVERAGE
HOW MANY INPUTS DO YOU HAVE3

IMPUT COUNT
1
INPUT X 1
ECHO VALUE    1DO YOU WANT ANOTHER VALUE?
TYPE 1 IF TRUE 0 IF FALSE.
1

IMPUT COUNT
2
INPUT X 3
ECHO VALUE    3DO YOU WANT ANOTHER VALUE?
TYPE 1 IF TRUE 0 IF FALSE.
1

IMPUT COUNT
3
INPUT X 5
ECHO VALUE    5          2
```

Criticism of the MeanMean Program

This is a severely unfriendly program both for the computer user, and also for any programmer who must maintain it. Unfortunately, it's bad programs like this one that need the most maintenance. It supposedly computes the mean value (but doesn't say so). It is actually a mean program, hostile, treacherous and awful!

The writer of this computerized trash probably has an attitude problem that leads to the program being even unfriendly when intending to be friendly. The chatty, condescending prompts "I'm your friend to help you average" are not necessary. The user could not care less. The text prompts in complete uppercase letters are less readable than if they were done in lower case, or even better, in mixed upper and lowercase letters. Even with all these prompts there is no indication to the user that the program expects integers only and provides an approximate answer.

The user is first prompted to input the number of input values, this is awkward. As you know, the program could have used a special end of data marker value to help the user. In fact there is a similar provision for terminating the program with a preset terminating value of -99. Unfortunately, that value cannot be changed without modifying the program, and the user is never told about it! Furthermore, after entering each value the user is asked if another value is to be entered. This repetition—taking two lines yet—is bothersome after a very short while. The response to this question would more naturally be Yes or No, or even Y or N, instead of typing 1 or 0.

The prompt "INPUT COUNT" is not at all clear. The prompt to enter a value is "INPUT X", but the user has no knowledge or interest in X. Also the spelling error becomes bothersome after it has been repeated several times.

You'll note that every value is echoed, which is good practice, but each echo is preceded by the message ECHO VALUE which is unnecessary. The final average however has no message indicating that it is the output.

So much from the user's point of view. Sometimes a program is user unfriendly because the programmer wishes to make the program simpler or shorter. However, here, the inside program view is equally horrible. The program is very difficult to read, as it has no indentation and no gaps to separate its various parts. The variable names are very primitive and not meaningful. The one attempt at a longer name, `true`, is confusing because this name describes one of the standard constants of type `BOOLEAN` (but here `true` is of type `INTEGER`!) There is very little useful documentation, and the little documentation given is useless and confusing.

Finally, even though this program runs, it suffers from some errors and will give bad results at times. The method of finding the average proceeds by `INTEGER` division (which truncates the result) of each value before accumulating that value. Every value that is input after the "INPUT X" prompt is divided by the stated number of inputs and added to the sum, which gives a very poor approximation of the mean. To save time and improve accuracy, the values should all be summed first and then divided once. The resulting output should

have been either a `REAL` value, or an `INTEGER` value with an indication of whether it is exact or approximate.

As a final bad point, take the case when the program is terminated by the input of `-99`. By the time this is detected, the value has already been added to the average, so it must now be subtracted. Definitely not a nice feature.

The only good thing that we may say is that there are many prompts. However, these are often very cryptic, badly phrased, or misleading...

6.14 Errors in Programming

In every complex creative activity, errors are possible, and programming is no exception. In programming, the errors are often called “bugs” and the process of finding and removing them is called “debugging.” Unfortunately, even the smallest program bug can cause a serious failure.

There are a number of different sources of errors, often classified as:

- Syntactic (compile-time errors)
- Semantic (execution or run-time errors)
- Logical (performance errors)

Syntactic Errors

arise from improper forms in a language involving:

Spelling	<code>INTERGER</code> vs. <code>INTEGER</code>
Punctuation	Misplaced semicolons
Typography	letter <code>O</code> instead of digit <code>0</code>
Sequence	declaration part before uses part
Spacing	space between the <code>:</code> and the <code>=</code> in the assignment <code>:=</code>
Type mismatch	assigning a <code>REAL</code> to a <code>CHAR</code>
Omission	of declarations or <code>ENDS</code>
Misuse	using <code>IF</code> , <code>END</code> etc. as names for variables
Incompleteness	not closing quotes or parentheses

Most of the syntactic errors are detected by the Pascal compiler, and so are fairly easy to find.

Execution Errors

are not found by the compiler, but do appear during a run.

Some examples of these “run-time” errors involve:

Undefined variables	(not initialized)
Unexpected value	(value out of range)
Invalid operations	(divide by zero)
Infinite loop	(non-halting program)

Logical Errors

are those which may not prevent a program from running, but do produce wrong results and may go undetected. Some such errors include:

Off by one	(looping one too many or too few times)
Wrong comparison	(less than instead of greater than)
Wrong operation	(increasing instead of decreasing)
Reversed operation	(equal vs. not equal)
Side effect	(inadvertent change of variable)

Other Errors

may also be encountered, these include:

- Misunderstanding of functions,
- Misuse of operating system, and
- Misbehavior of the computer.

6.15 Debugging, Testing, Proving

Debugging is the process of finding and “exterminating” program bugs. Keep in mind that this process can be very time-consuming. One method of debugging is to trace the program, by hand, using some simple values. However, for anything other than the simplest of programs, this can be very tedious and error-prone. Another useful technique is to add output instructions to the running program at strategic points. In this manner, a `Write` statement can be inserted at various points to output some information. When looking for a bug, the Bisection method may be used to place the `Write` statements (first, in the middle of the program, then in the middle of one of the program halves, etc.), in order to successively narrow down the possibilities.

As a first step to debugging, it is essential to find out *where* the action flows. This can be indicated by writing tracing comments to form a trail, such as:

```
INITIALIZING
ENTERING FIRST LOOP
LEAVING SWAP SUB
```

Once the flow of actions has been determined, we can output *what* values variables have at intermediate points. This is also done through the use of `Write` statements, yielding a computer-generated trace like:

```
A = 2
```

or

```
X IS 5 AND Y IS 7
```

It is also possible to use `Write` statements to generate a trace of *how* variables are related, for example, in a loop invariant. This can be done with code sequences such as:

```
IF (S + B) = (I * I) THEN  
  WriteLn('S + B = I * I');
```

Similar statement sequences can also be used to trace *when* a particular condition occurs:

```
IF (X MOD 2) = 0 THEN  
  WriteLn('X IS NOW EVEN');
```

```
IF Y >= 0 THEN  
  WriteLn('Y IS POSITIVE');
```

Such conditional output statements can also be very useful in reducing the amount of output generated inside a loop to an amount that can easily be examined, as:

```
IF I = 3 THEN  
  WriteLn('I = ', I: 1, ' J = ', J: 9);
```

After the debugging `Write` statements have served their purpose, they can be removed (or commented out, or embedded in a Selection form for further use).

TESTING is the process of checking, or verifying to determine proper performance. It is a means for gaining confidence in a program's results. The fact that a program works for one set of input data, does not necessarily mean that it will work for another set. One path through a program is not necessarily typical of all other paths.

One method of testing a program is to try all possible sets of data (checking all paths), but this is often impractical because too many combinations are necessary. For example, if there are 20 logical conditions in a program (either in Selections or Loops), then there may be over a million different paths!

Another method is to test using random values, but such values may be too typical, and may not include critical values. The best and most reliable way to test is to use carefully selected values based both on the particular problem being solved by the program, and the way in which the algorithm is structured. The selection of such test data is independent of the programming language and is discussed in the Principles book. Some data should be simple and easy to verify by hand computation. Some data should test for extreme or boundary values (such as zero, very large and very small values). Some data should be faulty or wrong, to check whether the program is robust and fails "softly." It is

also helpful if the test values are unique, each value testing and isolating one program part at a time.

Unfortunately, as the old saying goes, testing can reveal the presence of bugs, but only in the simplest cases can it reveal their absence.

In theory, the only really sure way is to *prove* the correctness of programs much as one proves mathematically the correctness of a geometrical theorem.

Unfortunately, such techniques are very complex and are beyond the scope of this book. However, informal use of assertions (including loop invariants) is a step in the proper direction.

6.16 Chapter 6 Review

This Chapter mainly presented alternative ways of constructing more complex programs. At the same time, new features of the Pascal programming language were introduced.

These newly introduced forms included the `CASE` statement as an alternative to the `IF-THEN` statement in special cases. However, the Selection form, represented by the `IF` statement, remains more general. The `WHILE` statement was shown to have two alternatives: the `REPEAT` statement and the `FOR` statement. Again these extra repetition statements are not necessary, but may be convenient.

Two new data types, `CHAR` and `BOOLEAN`, were also introduced through examples. Deeper nests, especially of the Repetition form, were considered with examples related to plotting functions.

Programming style, program layout and program documentation were considered. Examples were given to show how to produce good readable programs. A badly styled program was also discussed to show how difficult things can be when programming style goes lacking. Program modification, program testing, and program debugging were also discussed.

6.17 Chapter 6 Problems

1. For to While

Write the following `FOR` statement in an equivalent `WHILE` form:

```
FOR C := FIRST TO LAST DO S1;
```

2. Repeat to While

Write the following `REPEAT` statement in an equivalent `WHILE` form:

```
REPEAT S1; UNTIL C1.
```

3. While to Repeat

Write the following WHILE statement in an equivalent REPEAT form:

```
WHILE C1 DO S1;
```

4. Case to Ifs

Write the following CASE in terms of the IF statement:

```
CASE E1 OF K1: S1; K2, K3: S2; K4: S3; S4; END
```

5. Char

Four successive character A, B, C and D are input to a program. Write short pieces of program to compare these characters and to output whether or not the following are true.

- a. All the characters are digits.
- b. None of the characters is a digit.
- c. Some of the characters are vowels.
- d. The characters are in increasing order.
- e. There is no repetition within the sequence of characters.

6. Numeric Characters

Four successive characters E, F, G, H are input to a program. One of the characters F, G or H could be a period; all the others are digits. Thus, the sequence of characters can be interpreted as representing an amount of money. Write a piece of program to compare the characters and to output the value of the amount in cents. For example 1.25 is output as 125.

7. Logical evaluation

Given 5 logical variables P, Q, R, S and T, where P and Q both have the value TRUE, and R and S both have the value FALSE, and the value of T is undefined, evaluate the truth values of each of the following.

- a. P OR (R AND T)
- b. Q OR (R OR T)
- c. R AND (S OR T)
- d. P AND (T OR R)
- e. Q AND NOT (S OR T)

8. More

Create Pascal programs for the following problem statements:

6.18 Chapter 6 Programming Problems**1 Gas**

Create an algorithm that inputs sequences of two values Miles and Gals representing the mileage and gallons of gasoline at a succession of refills. The algorithm is to compute and output the immediate average miles-per-gallon (labeled Short for short range), and also the overall average mpg (since the beginning of the data), which is labeled Long for long range. A typical input-output sequence follows (and should end with negative mileage).

INPUTS		OUTPUTS	
Miles	Gals	Short	Long
1000	20		
1200	10	20	20
1500	20	15	16.67
...

2 GPA

The grade point average of a student is computed from all the course grades G and units U . Corresponding to each grade is a numeric point P (where A has 4 points, B has 3 points, etc.). The products of each grade point and its number of units are then summed. This sum is divided by the total number of units, to yield the grade point average. Create an algorithm to compute the grade point average for a sequence of pairs of values G, U (ending with negative values).

3 Speed

Create an algorithm to analyze the speed during a trip of N stops. At each stop, the distance D and time T from the previous stop are recorded. These pairs of values are then input to a program which computes each velocity ($V = D/T$) and outputs it. It also ultimately indicates the maximum speed on the trip, and the overall average (total distance divided by total time).

SAMPLE RUN ($N = 5$)

D	T	V	
45	1	45	
100	2	50	
55	1	55	
120	2	60	$\text{Avg} = 380/8 = 47.5$
60	2	30	$\text{Max} = 60$

4 Unbiased Mean

In some sports, a number of judges each ranks performance on a scale from 1 to 10. To adjust for biases, both the highest and lowest values are eliminated before computing the average. Create an algorithm to compute such an average for M judges on N performances.

5 More Plots

Modify the program `SidePlotXY` of this Chapter as follows:

- Mark points on the axes.
- Plot a third function.
- Input the size of the grid.

6 Upright Plots

Create an “upright” plot program, similar to the `SidePlotXY` program in this Chapter. Modify it as shown above.

7 Calendar

Create a program to print out a calendar for a month, given the number of days in the month and the starting day of the week. Create a grid of horizontal lines and vertical lines.

8 Encrypt-Decrypt

Create a program to input a file of characters, to encrypt it for security purposes in such a way that it is not easily understandable and to output it to another file. Then create another program that reconverts the encrypted file and returns the original file. Your algorithm could a simple arithmetic function on the ASCII values or use a random number generator.

9 Flasher2

Modify the Flasher program of this chapter to keep track of the number of correct answers.

10 Binconvert

Create an algorithm to convert a sequence of binary input characters (not integers) into their corresponding decimal values. For example, 1101 is the decimal 13

- a) Write the algorithm, if the input is read from left to right (ending with a period).
- b) Write the algorithm, if the input is read from right to left (ending with a blank).

11 When in Rome...

One method for converting an Arabic number into a Roman number is to separately convert each digit (the units, tens, hundreds, and thousands positions) as shown.

1	9	8	4
M	CM	LXXX	IV

Write an algorithm that accepts as inputs any values up to 3999, and outputs the corresponding Roman numbers.

- a) Do this, if the number is entered digit by digit (least significant digits first, like 4 8 9 1).
- b) Do this, if the number is entered digit by digit (most significant digits first, like 1 9 8 4).
- c) Do this, if the number is entered as an integer, like 1984.

6.19 Chapter 6 Programming Projects

Plotting Programs

Many interesting programming problems involving nests of loops can be encountered in creating plots using a printer. The following plots should be created in a general manner with the sizes entered at the beginning of execution. Arrays are not necessary for any of these; later, some may be done more easily using arrays.

1. Starbox

```
* * * * *
*           *
*           *
*           *
*           *
*           *
* * * * *
```

2. Tic-Tac Grid

```
  #  #
  #  #
  #  #
# # # # # # # #
  #  #
  #  #
# # # # # # # #
  #  #
  #  #
  #  #
```

3. Diamond

```
      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *
```

4. Pine Tree

```
      [
    [ [ [
  [ [ [ [ [
[ [ [ [ [ [ [
[ [ [ [ [ [ [ [
```



```

[ [ [
[ [ [

```

5. Triangle1

```

1
12
123
1234
12345
123456
1234567
12345678

```

6. Triangle2

```

1
22
333
4444
55555
666666
7777777
88888888

```

7. Calendar

(given beginning weekday & month)

```

S M T W T F S
      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

```

8. Greeting (input 9)

(gap every 5 years)

```

Happy Anniversary
Happy Anniversary
Happy Anniversary
Happy Anniversary
Happy Anniversary

```

```

Happy Anniversary
Happy Anniversary
Happy Anniversary
Happy Anniversary

```

9. More Greetings

(given input age of 21)

```

Happy Birthday      Happy Birthday

```

```

Happy Birthday      Happy Birthday
Happy Birthday      Happy Birthday
Happy Birthday      Happy Birthday
Happy Birthday      Happy Birthday

Happy Birthday      Happy Birthday
Happy Birthday      Happy Birthday
Happy Birthday      Happy Birthday
Happy Birthday      Happy Birthday
Happy Birthday      Happy Birthday

Happy Birthday

```

10. Checkered Grid

```

* * *      * * *      * * *      * * *
* * *      * * *      * * *      * * *
* * *      * * *      * * *      * * *

      * * *      * * *      * * *
      * * *      * * *      * * *
      * * *      * * *      * * *

* * *      * * *      * * *      * * *
* * *      * * *      * * *      * * *
* * *      * * *      * * *      * * *

      * * *      * * *      * * *
      * * *      * * *      * * *
      * * *      * * *      * * *

* * *      * * *      * * *      * * *
* * *      * * *      * * *      * * *
* * *      * * *      * * *      * * *

```

Modify Calculator

Modify the program `Calculate` given in this chapter in the following ways:

- To include a call for Help (by entering action H) that just lists the possible actions.
- To accept the action = or t (for “type”) to output the result instead of showing it after each action.
- To allow actions to be also given by upper case characters.
- To format the output in a better way.
- To extend to other actions to include square root, trigonometric functions, etc.
- To have a “memory” similar to other hand-held calculators.
- To have an “undo” command that “forgets” the previous action.
- To do anything else you could find convenient in a calculator.

Statistics (Rainfall)

Create an algorithm and the corresponding program to analyze the rainfall over a period of any number of days. The rainfall figures (real numbers, giving the daily amount of rain) are input in order, and end with a negative number. The following group of “statistics” is computed as the numbers are read in (they are not stored in arrays!).

1. MEAN is the average rainfall per day, computed by summing all the amounts and dividing by the number of days.
2. MAX is the maximum amount of rain that fell in any day.
3. MAXDAY is a day on which the maximum rain fell.
4. MIN is the minimum amount of rain that fell in any day, not including zero.
5. RANGE is the amount of variation in rainfall.
6. DRYDAYS is the number of days of no rainfall.
7. MAXCHANGE is the largest change in amount of rainfall between two consecutive days.
8. MAXDROP is the largest drop in amount of rainfall between two consecutive days.
9. MEANCHANGE is the average of the absolute amount of change between two consecutive days.
10. VARIANCE is the difference between the mean of the square of the values and the square of the mean of the values.
11. DEVIATION is the square root of the variance.
12. SECONDMAX is the second largest value of rainfall.
13. DRYRUN is the length of the largest period of days without rain.
14. WETRUN is the length of the largest period of days with rain.
15. MAXWEEK is the week which had the most accumulated rainfall.
16. WETRIPLE is the largest accumulated rainfall over three consecutive days.

Project Plotup

Create a program to plot a function in the proper orientation, with the Y axis vertical and the X axis horizontal (which is just the opposite of the SidePlotXY of this chapter). Then modify it in some of the following ways.

- a. Draw the X and Y axes, using the plus "+" symbol.
- b. Label the above axes.
- c. Extend the plot to two coordinates (allowing negative X values).
- d. Extend the plot to four coordinates.

- e. Allow for a second function to be plotted.
- f. Enclose the entire plot within a box.
- g. Add your own modifications.

A possible plot format follows:

```

                                Y
                                +
*****
*                               +                               * 10
*                               +                               *
*                               +                               * 8
*                               +                               *
*                               +                               * 6
*                               +                               *
*                               +                               * 4
*                               +                               *
*                               +                               * 2
*                               +                               *
* ++++++0+++++*+ X
*                               +                               *
*                               +                               *
*                               +                               *
*                               +                               *
*                               +                               *
*                               +                               *
*                               +                               *
*                               +                               *
*                               +                               *
*****
                                0  2  4  6  8 10

```

BSD: Big Statistics Data

This project will involve files of data values and statistical actions on these values. You will be given a file of rainfall amounts (in millimeters) for any number of days, and you will compute the following, in order, with the program growing in complexity as the project continues!

Reference:

Chapter 6 of the Principles book and this chapter.

1. ECHO.

First create a simple Pascal program that reads `INTEGER` values from a file called `Rain.DATA` until it reaches a negative value, which is the terminating value. This program simply reads the values and then writes them out to the screen.

2. COUNT.
Modify (or grow) the above program so that while it reads and writes the input values, it counts them.
3. MEAN.
Modify the above program to find the mean value of all the data values, and to output this value.
4. MAX.
Modify again the above program to find the maximum value of all the data values, and indicate also on what day this fell.
5. MIN.
Modify the above program to find the minimum non-zero amount of rain that fell in one day, and also show on what day this fell.
6. WETDAYS.
Modify the program again to count the number of days of rainfall.
7. WETRUN.
Modify it again to find the length of the largest period of days with rain.
8. Time Permitting: See other possible modifications in the STATISTICS programming project above.

BTD: Big Text Data

This project will involve files of data values and statistical actions on these values. You will be given a file of text (such as the Gettysburg Address) and you will compute the following, in order, with the program growing!

Reference:

Chapter 6 of the Principles book and this chapter.

1. ECHO.
First create a simple Pascal program which reads Character values from a file called `Gettysburg.Text` until it reaches a dollar sign, which is the terminating value. This program simply reads the values and then writes them out to the screen.
2. COUNT.
Modify (or grow) the above program so that while it reads and writes the characters, it counts them. Modify it also to count the number of words, lines and sentences.
3. MEAN.
Modify the above program to find the mean or average word length and to output this value. Find also the average number of words per line.

4. MAX.
Modify again the above program to find the length of the longest word.
5. MIN.
Modify the above program to find the length of the shortest sentence.
6. VOWELS.
Modify the program again to count the number of vowels in a file.
7. MORE.
Indicate other actions of the above kinds that could be computed for a text file. Do one or two of these.

SMC: Small Monthly Calendar

You are to plan a program and code it to output a calendar of a single month (similar to the Calendar Planner Application). You will input the number of days N in the month and the day of the week F (where $F = 1$ on Sunday, $F = 2$ on Monday, etc.) on which the first of the month falls. For example, the following calendar is for a month with $N = 30$ days, with the first day occurring on a Saturday, so $F = 7$. You may use the Pascal FOR statement now; arrays (whatever they are) are not necessary yet, but may be used at a later time.

Sun	Mon	Tue	Wed	Thu	Fri	Sat
---	---	---	---	---	---	---
						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						

Modify this program in the following ways,:

- a. Putting a border around the outside,
- b. Putting each day into a box (of plusses)
- c. Making the size of each box or cell variable
- d. Putting more months

Chapter 7 Better Blocks: Procedures and Libraries

This chapter presents the creation and use of Pascal subprograms, i.e. the Pascal procedures and functions. Subprograms are often part of libraries, and the chapter introduces the Pascal units that are used to implement libraries.

Chapter Overview

7.1	Preview.....	233
7.2	Procedures in Pascal.....	233
	Use and Definition.....	233
7.3	Syntax of Subprogram Forms.....	235
	Procedures in Pascal.....	235
	More Examples of Pascal Procedures.....	239
	Power Procedure.....	242
7.4	Passing Parameters.....	242
	In, Out, In and Out, and Neither.....	242
	BigChange.....	242
	BigPay.....	245
	A Miscellany of procedures.....	248
	A Second Miscellany of procedures.....	250
7.5	Procedures with Char, Boolean and Other Types.....	252
	Generalized Item Types.....	255
7.6	Procedures with User-Defined types.....	256
7.7	More on Passing Parameters.....	259
7.8	Nested Procedures.....	261
7.9	Functions in Pascal.....	264
	Many Functions.....	266
7.10	SubPrograms: Variations on a theme.....	269
7.11	Recursion in Pascal.....	272
7.12	Libraries in Pascal.....	275
	Units.....	275
	UtilityLib: a custom-made utilities Library.....	278
	Other Libraries: DateLib, BitLib, CharLib.....	281
	The Interaction of Many Libraries.....	288
7.13	Function and Procedure Types.....	291
7.14	Top-Down Development.....	296
	Pay Again.....	296
7.15	Chapter 7 Review.....	299
7.16	Chapter 7 Problems.....	300
7.17	Chapter 7 Programming Problems.....	302
	Random Projects.....	302
	DateLib.....	305
	Create Libraries.....	306
	FinanceLib.....	307

	Change Again: Done Properly with Procedures.....	308
	MeanLib.....	308
7.18	Chapter 7 Programming Projects.....	310
	DMT: DeMilitarizeTime Lab with Procedures.....	310
	SLL: Small Library Project.....	311

7.1 Preview

The encapsulation of both actions and data types into subprograms is a very important part of computing, and is treated in great detail in this chapter, which makes it one of the most important chapters.

First, procedures and their syntax are considered, and numerous examples are given. Nested procedures are also treated, with more examples. The concept of functions as limited procedures is considered next, along with many examples.

In Pascal, libraries, which are collections of related subprograms, are created through the units mechanism. Units consist of an Interface part, which is a specification of *what* the procedures in the unit do, and an Implementation part, which indicates *how* the procedures do what they do. The creation of libraries through Pascal units is described, again with many complete examples.

In this chapter, you will learn how to create the kinds of procedures, functions and libraries that you have been using and re-using in previous chapters.

7.2 Procedures in Pascal

Use and Definition

A *procedure* is a very useful “black box” that *encapsulates* (comprises all the elements of) a subalgorithm. Once a procedure is declared, it may be used whenever the subalgorithm is required. First, consider the procedure `Max`, which operates on two data values, `X` and `Y` and produces a value `M`, which is the maximum of the two values `X` and `Y`. `Max` can then be used, for example, to find the maximum of three values `A`, `B` and `C` as in the body of the program `Max3` from Chapter 7 of the Principles book (Fig. 7.7):

```
Read(A, B, C);  
Max(A, B, E);  
Max(E, C, L);  
WriteLn(L);
```

Here, `Max` is first used to set `E` to the maximum of the values `A` and `B`. Then `Max` is used again to set `L` to the maximum of the values `E` and `C`. In other words, we have used the actions encapsulated in `Max` twice to build the larger operation of finding the maximum of three values.

We will use the procedure `Divide`, which is a little more complex than `Max`, to illustrate how procedures are defined in Pascal. `Divide` operates on a numerator `Num` and denominator `Den`, and produces a quotient `Quot` and remainder `Rem`. A call to `Divide` is denoted as `Divide(Num, Den, Quot, Rem)`. We can use `Divide` in many different ways as shown in the following examples.

Conversion of many units is done conveniently by `Divide` as:

```
Divide(Footage, 5280, Miles, Feet);
Divide(TotalOz, 16, Pounds, Ounces);
```

Two INTEGER values can be averaged simply by:

```
(* Mean of Two INTEGER Values *)
Divide(A + B, 2, Mean2, Rem);
Write(Mean2: 3);
IF Rem = 0 THEN
    Write(' exactly')
ELSE
    Write(' approximately');
```

It is simple to determine whether Year is a leap year with:

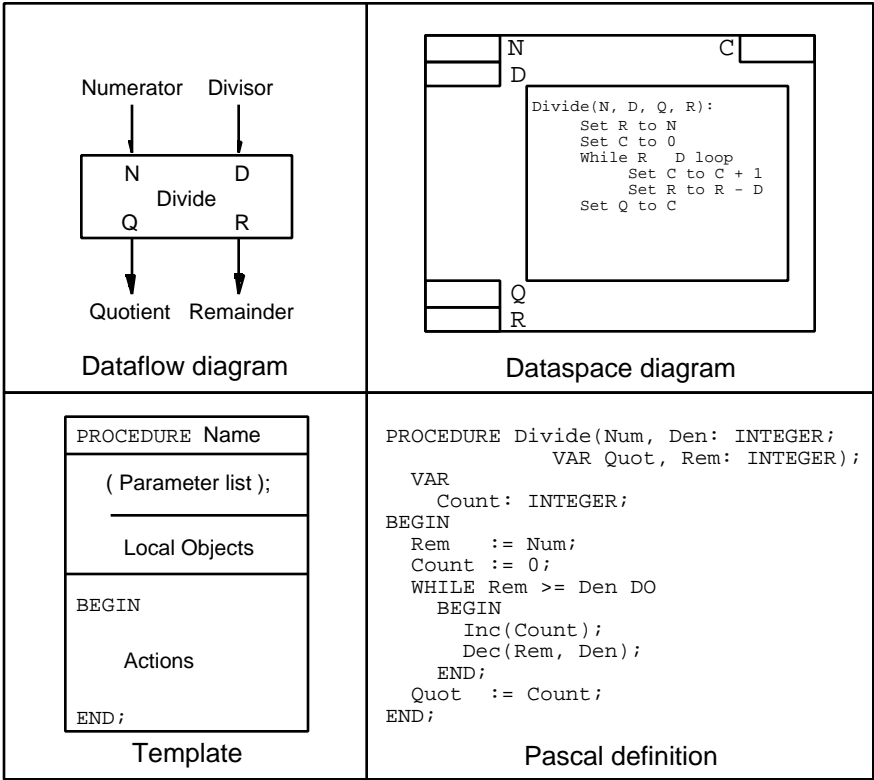
```
(* Simple Leap: Fails for 1900 *)
Divide(Year, 4, Q, Remainder);
IF Remainder = 0 THEN
    WriteLn(Year: 4, ' is a leap year')
ELSE
    WriteLn(Year: 4, ' is not a leap year');
```

The algorithm for computing the check digit in the ISBN, the International Book Number, is described in Chapter 3 of the Principles book. The check digit is computed from the weighted sum WtSum of the first nine digits of the number, D1, D2, D3, D4, D5, D6, D7, D8, D9, as follows.

```
(* ISBN Check Digit Computation *)
WtSum := D1 + 2*D2 + 3*D3 + 4*D4 + 5*D5 +
        6*D6 + 7*D7 + 8*D8 + 9*D9;
Divide(WtSum, 11, Q, Check);
IF Check = 10 THEN
    Write('Check digit is X')
ELSE
    Write('Check digit is ', Check:1);
```

In Figure 7.1, Divide is described in four different ways. The top two representations are taken from Chapter 7 of the Principles book. The dataflow diagram shows the four parameters of which two are passed in and two are passed out. The data space diagram shows the actual space occupied by the variables associated with the procedure Divide. Also shown is the pseudocode of the procedure, from which is seen that it makes use of a temporary variable C, and that this variable is private to the procedure and cannot be accessed by any of the programs that use Divide.

Figure 7.1 Procedure descriptions



The template for the Pascal definition of a procedure is shown at the bottom left of Figure 7.1. From the template, we see that the first part of the procedure consists of its name, which should be descriptive of the procedure’s purpose. This is followed by a parameter list, which must specify all the parameters, their names, data type, and whether they are being passed in or passed out. Then comes a declaration part for the local variables, which are hidden within the procedure. Finally, there is a series of statements that describe the actions performed in the body of the procedure. Finally, in the bottom right quadrant of Figure 7.1, the complete Pascal definition of the `Divide` procedure, with more descriptive names for all the variables, is shown so that its correspondence with the template can be seen.

7.3 Syntax of Subprogram Forms

Procedures in Pascal

As we saw in the preceding section, procedures are a very convenient and powerful form of abstraction. We can use them without burdening our minds with the details of how they carry out their purpose. Viewed from the user’s standpoint, a procedure has a name and a list of parameters. There are three kinds of parameters:

- Input parameters—data values are passed into the procedure; in `Divide`, `Num` and `Den` were input parameters.
- Output parameters—data values are passed out of the procedure; in `Divide`, `Quot` and `Rem` are output parameters.
- Input-output parameters—data values are passed in and out of the procedure.

Pascal implements only two parameter passing mechanisms: *pass by value*, used for input parameters and *pass by reference*, used for both output and input-output parameters. Consequently, there is no distinction made in Pascal between output and input-output parameters. Comments could be used to indicate the difference. It is often convenient to list the input parameters first but this is not necessary.

Figure 7.2 Syntax diagrams for a ProcedureCall statement

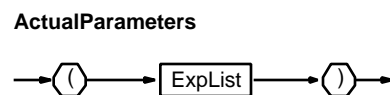
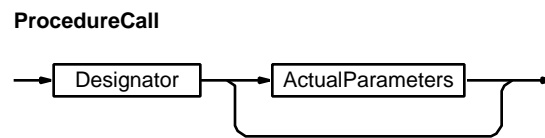
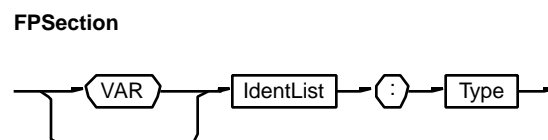
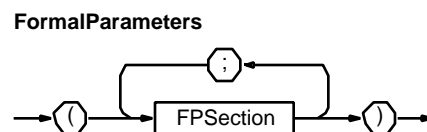
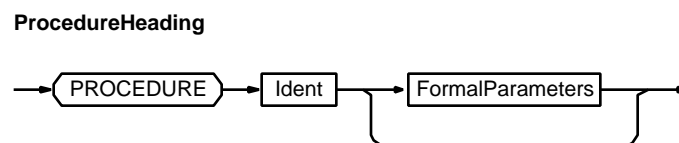
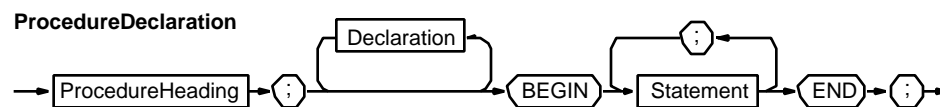


Figure 7.3 Syntax diagrams for a ProcedureDeclaration

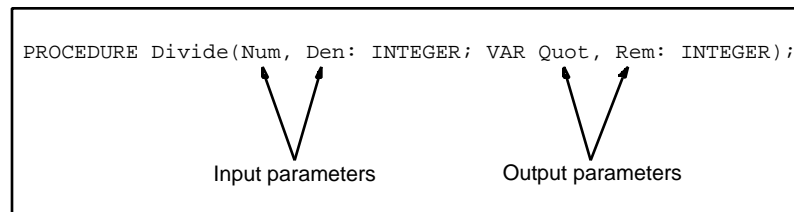


A procedure is invoked by a `ProcedureCall` statement, whose syntax is defined by the diagrams in Figure 7.2. From these diagrams we see that a `ProcedureCall` consists of the name of the procedure followed possibly by the `ActualParameters`, which is a list of expressions separated by commas and enclosed in parentheses. The procedure itself is specified by a `ProcedureDeclaration` whose syntax is defined by the diagrams in Figure 7.3.

These diagrams show that a `ProcedureDeclaration` consists of a `ProcedureHeading`, a semicolon, a `Block` and another semicolon. In turn, the `ProcedureHeading` consists of the name of the procedure followed possibly by the `FormalParameters`, which is essentially a list of identifiers and types separated by semicolons and enclosed in parentheses. The close correspondence between a `ProcedureCall` and a `ProcedureHeading` emphasizes the fact the `ProcedureCall` is the point at which data communication between the user of the procedure and the procedure itself takes place.

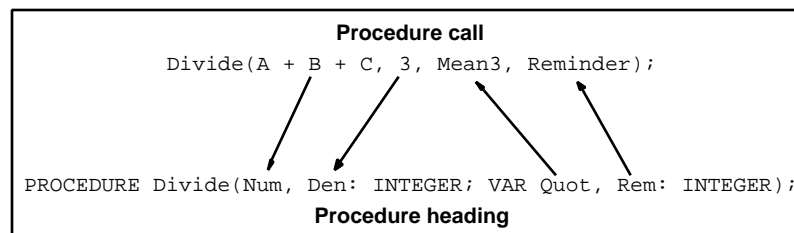
The distinction between input parameters, and output or input-output parameters is made by prefixing the output or input-output parameters with the word `VAR`. Thus the `ProcedureHeading` for the `Divide` procedure is as shown in Figure 7.4.

Figure 7.4 The procedure heading for Divide



The correspondence between the `ActualParameters` of the `ProcedureCall`, and the `FormalParameters` of the `ProcedureHeading` is on a strict left-to-right one-by-one match. It is required that each actual parameter that corresponds to an output or input-output parameter be a variable—it must not be an expression. On the other hand, an actual parameter that corresponds to an input parameter can be an expression or a constant. An example of such a correspondence is given in Figure 7.5.

Figure 7.5 Correspondence of actual and formal parameters



When a procedure call is executed, the following Steps take place:

1. The caller's point of return is recorded. This is the point at which execution will continue after the procedure being called has completed its actions, i.e. the next statement after the procedure call.
2. Space is allocated for all input parameters and local variables. Using `Divide` as an example, space is allocated for the input parameters `Num` and `Den` as well as for the internal variable `Count`.
3. The output and input-output formal parameters are linked to their corresponding actual parameters in the procedure call. `Quot` is linked to `Mean3` and `Rem` is linked to `Remainder`.
4. The actions specified in the body of the procedure are performed.
5. The space allocated in Step 2 is deallocated.
6. Execution is resumed from the point of return recorded in Step 1.

This sequence of steps is discussed in greater detail in Chapter 7 of the Principles book.

To summarize the material covered in this section, let's look at a complete program example. The complete Pascal program in Figure 7.6 is based on the `AverageExample` algorithm found in Chapter 7 of the Principles book.

Figure 7.6 The `AverageExample` program

```
PROGRAM AverageExample;
(* Mean of two INTEGER Values *)

VAR First, Second, Average, Remainder: INTEGER;

PROCEDURE Divide( Num, Den: INTEGER;
                  VAR Quot, Rem: INTEGER);
(* Integer division of Num by Den giving a
quotient
Quot and a remainder Rem *)
VAR Count: INTEGER;
BEGIN
  Rem := Num;
  Count := 0;
  WHILE Rem >= Den DO BEGIN
    Inc(Count);
    Dec(Rem, Den);
  END;
  Quot := Count;
END;

BEGIN
  Write('Give two integers: ');
  Read(First, Second);
  Divide(First + Second, 2, Average, Remainder);
  Write(Average: 3);
  IF Remainder = 0 THEN
    Write(' exactly ')
```

```

ELSE
  Write(' approximately');
END. { AverageExample }

```

This program starts by asking for and reading in two values, `First` and `Second`. It then calls procedure `Divide`, which is declared earlier in the program—a procedure must be declared in the program or in an external unit before it can be used. The expression `First + Second` is evaluated, and the resulting value is passed in to become the value of `Num` in `Divide`. Similarly, the value 2 is passed into `Den`. In the procedure, the value in `Num` (the sum of `First` and `Second`) is divided by `Den` (2) and the resulting quotient and remainder are assigned to `Quot` and `Rem`. Since these are output parameters that were linked to `Average` and `Remainder` in the procedure call, these two variables are set to the quotient and remainder obtained in the division. Execution then continues with the `Write` statement and the value of `Average` is output. Then, either the word “exactly” is output, if the value of `Remainder` is zero, otherwise the word “approximately” is output.

Notice that, in the declaration of procedure `Divide`, the word `VAR` has two different meanings according to its context—depending on whether it occurs in a procedure header or a declaration. Also note the use of standard procedures `Inc` and `Dec` to increment and decrement a variable:

`"Dec(Rem, Den)"` is equivalent to `"Rem := Rem - Den."`

More Examples of Pascal Procedures

Figure 7.7 shows the program `Change`, which is based on the `Change` algorithm of Chapter 7 of the *Principles* book, and contains a `Divide` subprogram. The main program makes change, using a series of calls to the procedure `Divide`. This program illustrates a number of concepts.

Procedure `Divide` is essentially the same as the one defined in the `Average` example of Figure 7.6. Here, it has longer and more meaningful parameter and variable names (`Numerator` instead of `Num`, and `Remainder` instead of `Rem`). In the procedure header, each parameter is set on a separate line, with its type and indication of method of passing. This header could also have been written as one long line or as the following:

```

PROCEDURE Divide
  ( Numerator, Denominator: INTEGER; { pass-in }
  VAR Quotient, Remainder: INTEGER);    { pass-out}

```

The form shown in Figure 7.7 is more redundant than this one because it repeats the `INTEGER` type and the `VAR` indicator. The form of Figure 7.7 is more verbose but also less prone to error. In particular, one common programming error occurs when a `VAR` precedes lists of output parameters of different types. The `VAR` actually applies only to the parameters of the first list; it does not apply to the other lists. For example, consider the header:

```

PROCEDURE SubProg(    A, B, C: INTEGER;
                     VAR X, Y: INTEGER; Z: REAL)

```

Here the VAR indicator applies to parameters X and Y only; they are passed by reference, but Z is passed by value. If it is intended for Z to be also passed by reference, then it needs to be prefaced by another VAR, as in:

```
PROCEDURE SubProg(    A, B, C: INTEGER;
                    VAR X, Y: INTEGER;
                    VAR Z:      REAL)
```

Figure 7.7 The Pascal program Change

```
PROGRAM Change;
(* Calculate change to be given *)

VAR Tendered, Cost, Remainder,
    Quarters, Dimes, Nickels, Pennies: INTEGER;

PROCEDURE Divide(Numerator: INTEGER; { input
parameter }
    Denominator: INTEGER; { input parameter }
    VAR Quotient:  INTEGER; { output parameter }
}
    VAR Remainder: INTEGER);{ output parameter }
}
VAR Count: INTEGER; { Local variable }
BEGIN
    Remainder := Numerator;
    Count    := 0;
    WHILE Remainder >= Denominator DO BEGIN
        Inc(Count);
        Dec(Remainder, Denominator);
    END;
    Quotient := Count;
END; { Divide }

BEGIN
    { Input the transaction data }
    Write('Enter the amount tendered in cents ');
    Read(Tendered);
    Write('Enter the cost in cents ');
    Read(Cost);

    { Carry out the computation of change }
    Remainder := Tendered - Cost;
    Divide(Remainder, 25, Quarters, Remainder);
    Divide(Remainder, 10, Dimes, Remainder);
    Divide(Remainder, 5, Nickels, Pennies);

    { Output the results }
    WriteLn('The change is:');
    WriteLn(' ', Quarters: 2, ' quarters');
    WriteLn(' ', Dimes: 2, ' dimes');
```

```

WriteLn(' ', Nickels: 2, ' nickels');
WriteLn(' ', Pennies: 2, ' pennies');
END. { Change }

```

As a good programming practice, it is very useful to include assertions of various kinds in procedures. To make them more visible, it is common to write assertions as comments with two asterisks. Examples of such useful assertions in procedures are preconditions, postconditions and loop invariants.

Preconditions are assertions that describe the values of variables before the procedure executes. For example, in the `Divide` procedure the `Denominator` should not be zero, and also the `Numerator` should exist, as:

```

(** PRE-COND: Numerator exists,          **)
(**          Denominator is not zero **)

```

Postconditions are assertions that describe the values of variables after a procedure ends. For example, in `Divide`, after execution, the `Remainder` will be non-negative and less than `Denominator`:

```

(** POST-COND:                               **)
(** 0 <= Remainder < Denominator **)

```

Loop Invariants are assertions that describe the behavior of a loop. The loop in `Divide` has the invariant:

```

(** INVAR:
    **)
(** Numerator = Denominator*Quotient + Remainder **)

```

Sometimes a postcondition is put near the end of the procedure. But a better practice is to put it at the beginning of the procedure with the precondition, as part of the procedure documentation. Also, when the assertions are obvious they are often not shown.

Power Procedure

As another procedure example consider the useful exponentiation operation that is not part of Pascal. This operation raises a number to some power. If the power is a positive `INTEGER` then procedure `PPower` in Figure 7.8 multiplies the number by itself the required number of times.

Figure 7.8 The PPower procedure

```

PROCEDURE PPower( Base: REAL; { input }
                  Exponent: INTEGER; { input }
                  VAR Result: REAL); { output }
(* Compute Base raised to the power Exponent *)
(** PRE-COND: Exponent >= 0          **)
(** POST-COND: Result = Base to the power
Exponent **)

```

```
VAR Count: INTEGER;  
BEGIN  
  Result := 1.0;  
  FOR Count := 1 TO Exponent DO  
    Result := Result * Base;  
  END (* PPower *);
```

7.4 Passing Parameters

In, Out, In and Out, and Neither

Procedures are representations of independent algorithms and, as such, they present a great many different aspects that cannot be conveyed by the study of only a few examples. In this section, we will show many examples of small procedures, to illustrate the great diversity of details.

In Pascal, data are passed from the actual parameters of a call statement to the formal parameters of the called procedure, either by value or by reference. Input parameters are passed by value, while output and input-output parameters are passed by reference. It is very important that this distinction be absolutely clear in your mind for two reasons:

- Expressions may only be passed to input parameters, while the actual parameters corresponding to output and input-output parameters must be variables.
- Variables passed to input parameters cannot have their values changed by the procedure, whereas variables passed to output and input-output parameters can, and generally do, have their values changed by the procedure.

To make this distinction clear, appropriate comments can be put in the procedure headers. The comments { input }, { output } and { through } can be put at the right of the corresponding parameters, as we have done in our previous examples and will do in this section. You are likely to find that some of the procedures shown in the following programs can be used in your programming. You can copy their definitions directly into your program, or you can use them as models on which to base procedures tailored to your particular needs. They could also be put into a Library and be available to all program.

BigChange

The first example, program `BigChange` in Figure 7.9, is an enlarged version of the program `Change` shown earlier in Figure 7.7.

Figure 7.9 Program BigChange

```
PROGRAM BigChange;  
(* Change maker with procedures *)
```

```

PROCEDURE Spellout(Number: INTEGER); { input }
(* Write digits in English *)
BEGIN
  CASE Number OF
    0: Write('zero');    1: Write('one');
    2: Write('two');     3: Write('three');
    4: Write('four');    5: Write('five');
    6: Write('six');     7: Write('seven');
    8: Write('eight');   9: Write('nine')
  ELSE
    Write(Number: 3);
  END;
END; { Spellout }

PROCEDURE EnterPos(VAR PositiveNumber:
INTEGER);{ output }
(* Ask for and return a positive integer *)
VAR InputValue: INTEGER;
BEGIN
  Write(' Enter a positive number ');
  Read(InputValue);
  WHILE InputValue < 0 DO BEGIN
    Write('Error in value; enter it again ');
    Read(InputValue);
  END;
  PositiveNumber := InputValue;
END; { EnterPos }

PROCEDURE Divide( Numerator: INTEGER; { input }
                  Denominator: INTEGER; { input }
                  VAR Quotient: INTEGER; { output }
                  VAR Remainder: INTEGER);{ output }
(* Divide Numerator by Denominator and return
Quotient
and Remainder *)
VAR Count: INTEGER; { Local variable }
BEGIN
  Remainder := Numerator;
  Count := 0;
  WHILE Remainder >= Denominator DO BEGIN
    Inc(Count);
    Dec(Remainder, Denominator);
  END;
  Quotient := Count;
END; { Divide }
VAR Tendered, Cost, Remainder,
    Quarters, Dimes, Nickels, Pennies: INTEGER;
BEGIN
{ Input the transaction data }

```

```
WriteLn('Enter the cost in cents:');
EnterPos(Cost);
WriteLn('Enter the amount tendered in cents:');
EnterPos(Tendered);
{ Carry out the computation of change }
Remainder := Tendered - Cost;
Divide(Remainder, 25, Quarters, Remainder);
Divide(Remainder, 10, Dimes, Remainder);
Divide(Remainder, 5, Nickels, Pennies);
{ Output the results }
WriteLn('The change is:');
Write(' the number of quarters is ');
SpellOut(Quarters);
WriteLn;
Write(' the number of dimes is ');
SpellOut(Dimes);
WriteLn;
Write(' the number of nickels is ');
SpellOut(Nickels);
WriteLn;
Write(' the number of pennies is ');
SpellOut(Pennies);
WriteLn;
END. { BigChange }
```

The BigChange program makes use of a number of different procedures:

spellout (Number)—Number is an input parameter.

This procedure outputs small integer values, passed in Number, in words instead of digits. Since Number is an input parameter, it is passed by value. Only the numbers zero through nine are output as words; outside that range, numerical characters are used.

Enterpos (PositiveNumber)—PositiveNumber is an output parameter.

This procedure reads in an INTEGER called InputValue, tests whether it is positive, and if not keeps prompting and reading in numbers. When the input value is positive then it is assigned to whatever variable PositiveNumber corresponds to in the calling program. The procedure can be used to enter positive values for variables such as Age, Height, IdNumber, etc. The VAR preceding the variable PositiveNumber indicates passing by reference, as does the comment { output } at its right. A second VAR precedes the declaration of the local variable InputValue.

Divide (Numerator, Denominator, Quotient, Remainder)—Numerator and Denominator are input parameters; Quotient and Remainder are output parameters.

This procedure receives two values (Numerator and Denominator) and passes out two values (Quotient and Remainder). It has been discussed in detail and needs no further explanation.

Following the declarations of these three procedures are the declarations of the variables that are used in the program's body. They are listed close to the BEGIN because that is where they are used. Things that go together should be together. There are seven variables all of type INTEGER.

Running this program for some typical data yields the following output:

```
Enter the cost:
    Enter a positive number 13
Enter the amount tendered:
    Enter a positive number 100
The change is:
    the number of quarters is three
    the number of dimes is one
    the number of nickels is zero
    the number of pennies is two
```

BigPay

Figure 7.10 Program BigPay

```
PROGRAM BigPay;
(* Compute the net pay and the deductions for a
number
of employees, whose data is kept in input file
Employee.Data. The results are given in a table
and the total net pay is displayed *)
VAR Total: REAL;
    EmployeeNum, EmployeeCount: INTEGER;
    InputFile: TEXT;

PROCEDURE GrossPay(VAR Pay: REAL);
(* Compute the gross pay corresponding to the
data
read from an open InputFile *)
CONST Break = 40;

VAR Hours, Rate: REAL;
BEGIN
    ReadLn(InputFile, Hours);
    ReadLn(InputFile, Rate);
    Write(Hours:7:2, Rate:6:2);
    IF Hours < Break THEN
        Pay := Hours * Rate
    ELSE
        Pay := Break * Rate + 1.5 * (Hours - Break);
    END; { GrossPay }

PROCEDURE GetMiscDeductions(VAR MiscDeductions:
REAL);
(* Compute miscellaneous pay deductions *)
```

```
CONST ConstMiscDeductions = 10.75;
BEGIN
  MiscDeductions := ConstMiscDeductions;
END; { GetMiscDeductions }

PROCEDURE Deductions(Gross: REAL; VAR Total:
REAL);
  (* Compute the total pay deductions from a gross
pay *)
  CONST Rate = 0.27;

  VAR Tax, Misc: REAL;
  BEGIN
    Tax := Rate * Gross;
    GetMiscDeductions(Misc);
    Total := Tax + Misc;
    Write(Tax: 7: 2, Misc: 6: 2);
  END; { Deductions }

PROCEDURE NetPay(VAR Amount: REAL);
  (* *)
  VAR Gross, Deduct, ActualPay: REAL;

  BEGIN
    GrossPay(Gross);
    Write(Gross: 7: 2);
    Deductions(Gross, Deduct);
    ActualPay := Gross - Deduct;
    WriteLn(ActualPay: 7: 2);
    Amount := Amount + ActualPay;
  END; { NetPay }

BEGIN
  Assign(InputFile, 'Employee.Data');
  Reset(InputFile);
  ReadLn(InputFile, EmployeeCount);

  Total := 0;
  WriteLn('Emp  Hours  Rate  Gross Tax  Misc
Actual');
  WriteLn('Num      Pay      Pay');
  FOR EmployeeNum := 1 TO EmployeeCount DO BEGIN
    Write(EmployeeNum: 3);
    NetPay(Total);
  END;

  Write('The total net pay is      ');
  WriteLn(Total:7:2);
```

```
Close(InputFile);  
END. { BigPay }
```

Our second example is a Pascal version of the BigPay algorithm we have seen in Chapter 7 of the Principles book. This version is shown in Figure 7.10, and has been written so that it obtains its data from a file whose layout is:

```
Number of employees  
One data record showing hours worked and rate of pay  
for each employee.
```

The actual contents of the file used with this example are the following.

```
10  
45.5 11.25  
40.0 6.50  
37.5 10.25  
43.0 11.50  
40.0 4.50  
42.4 14.50  
40.0 6.75  
40.0 8.50  
52.0 4.50  
28.5 4.50
```

These data correspond to ten employees with hours and rates of pay as shown. The body of the program, between the last BEGIN-END pair, first sets up the `InputFile` for reading, and reads the number of employees to be processed and sets that in `EmployeeCount`. It then initializes the variable to accumulate the total pay to zero, and outputs the header for the pay ledger that will be produced. It then executes a loop once for each employee. In the body of the loop, it prints the employee number, calls the procedure `NetPay` to calculate the employee's net pay and add it to the total. The program includes the following procedures:

GrossPay(Pay)

This procedure has a single output parameter, `Pay`, in which the calculated gross pay is set. Hours worked and Rate of pay for the employee are read from the `InputFile` and the `Pay` computed. Note that since `InputFile` is a global name—it is not passed as a parameter—it must be declared *before* the procedure is defined.

GetMiscDeductions(MiscDeductions)

This procedure has a single output parameter, `MiscDeductions`. In fact, this procedure is a stub—a temporary version of the procedure used during the development of the program. The details of the computation of the miscellaneous deductions have not been elaborated; it just returns a constant value of 10.75 for each employee, which is specified as a constant.

Deductions(Gross, Total)

This procedure has one input parameter, `Gross`, and one output parameter, `Total`. Note that the parameter `Total` is quite separate

from the variable `Total` declared at the top of the program. There are two local variables, `Tax` and `Misc`.

NetPay (Amount)

This procedure has a single input-output parameter `Amount` to which the `ActualPay` is added.

The result of running `BigPay` with the sample data in the input file is given in Figure 7.11. You may note that the net pay total is not exact (it is one cent too much); this is proof of the disadvantage of using `REAL` variables.

Figure 7.11 Results produced by BigPay

Emp Num	Hours	Rate	Gross Pay	Tax	Misc	Actual Pay
1	45.50	11.25	458.25	123.73	10.75	323.77
2	40.00	6.50	260.00	70.20	10.75	179.05
3	37.50	10.25	384.37	103.78	10.75	269.84
4	43.00	11.50	464.50	125.42	10.75	328.33
5	40.00	4.50	180.00	48.60	10.75	120.65
6	42.40	14.50	583.60	157.57	10.75	415.28
7	40.00	6.75	270.00	72.90	10.75	186.35
8	40.00	8.50	340.00	91.80	10.75	237.45
9	52.00	4.50	198.00	53.46	10.75	133.79
10	28.50	4.50	128.25	34.63	10.75	82.87
The total net pay is						2277.39

A Miscellany of procedures

Figure 7.12 Program MiscProcs1

```

PROGRAM MiscProcs1;
(* Illustrates many procedures *)

PROCEDURE Instructions;
(* Prints brief instructions *)
BEGIN
  WriteLn(' Enter percentages ');
  WriteLn(' as whole numbers. ');
  WriteLn(' End on a negative ');
END { Instructions };

PROCEDURE WriteStar15;
(* Prints a line of 15 stars *)
VAR Count: INTEGER;
BEGIN
  FOR Count := 1 TO 15 DO
    Write('*');
  WriteLn;
END { WriteStar15 };

```



```
PROCEDURE Decr(VAR Value: INTEGER); (*
input-output *)
BEGIN
    Value := Value - 1;
END; { Decr }

PROCEDURE WriteStar(Number: INTEGER); (* input
*)
(* Prints a line of Number stars*)
BEGIN
    WHILE Number > 0 DO BEGIN
        Write('*');
        Decr(Number);
    END;
    WriteLn;
END { WriteStar };

PROCEDURE WriteChar(Symbol: CHAR; Number:
INTEGER);
(* Prints line of N Chars *)
VAR Count: INTEGER; (* local *)
BEGIN
    FOR Count := 1 TO Number DO
        Write(Symbol);
    WriteLn;
END { WriteChar };

BEGIN { Main Program }
    WriteStar(24);
    Instructions;
    WriteChar('-', 24);
END. { MiscProcs1 }
```

Our next example is program `MiscProcs1`, shown in Figure 7.12, that embodies a mixed collection of procedures that serves to further illustrate some more forms of procedures.

Instructions()

This first procedure, simply displays some instructions to the user of the program. It involves no passing of parameters and has no local variables.

WriteStar15()

This procedure outputs a line of 15 asterisks and terminates the current output line. It also has no parameters, but has a local variable `Count` used to count the asterisks.

WriteStar(Number)

This is a more general version of `WriteStar15` that has one input parameter `Number`, which specifies the number of asterisks to be output. It does not need to use a local variable since the value of the

input parameter `Number` serves as a counter. Since input parameters are passed by copying their value, when the value of `Number` is changed in the procedure, it does not change the value of the corresponding actual parameter in the main program.

Decr(`Value`)

This procedure has a single input-output parameter `Value`, which it decrements by 1, i.e. it is similar to standard procedure `Dec`. It serves to illustrate the use of an input-output parameter, one whose value is passed in, modified and then passed out to the same variable in the calling program.

WriteChar(`Symbol`, `Number`)

outputs a line consisting of the `Symbol` character repeated `Number` times. It is a further generalization of `WriteStar`.

The body of `MiscProcs1` illustrates the calling of the above procedures in trivial ways. Notice especially that each call is a statement. The output obtained from running the program is:

```
*****
Enter percentages
as whole numbers.
End on a negative
-----
```

A Second Miscellany of procedures

Figure 7.13 Program MiscProcs2

```
PROGRAM MiscProcs2;
(* Illustrates many procedures *)
VAR A, B, C: CHAR;
    R, S, T: REAL;
    I, J, K: INTEGER;

PROCEDURE SnapTrace;
(* Show snapshot of 3 values *)
BEGIN
    WriteLn('First  = ', C);
    WriteLn('Second = ', I: 4);
    WriteLn('Third  = ', R: 7: 2);
END { SnapTrace };

PROCEDURE TemperatureFtoC( Fahrenheit: REAL; { input }
                           VAR Celsius: REAL); { output }
(* Converts the temperature from Fahrenheit to Celsius *)
BEGIN
    Celsius := (5.0 / 9.0) * (Fahrenheit - 32.0);
END { TemperatureFtoC };

PROCEDURE AreaCircle(Radius: REAL; VAR Area: REAL);
```

```

(* Computes area of circle *)
BEGIN
  Area := Pi * Radius * Radius;
END { AreaCircle };

PROCEDURE Maxi2( X, Y: INTEGER; { input }
  VAR Max: INTEGER); { output }
(* Finds Maximum of two integers *)
BEGIN
  IF X > Y THEN
    Max := X
  ELSE
    Max := Y;
  END { Maxi2 };

PROCEDURE Maxi3( A, B, C: INTEGER; { input }
  VAR Largest: INTEGER); { output }
BEGIN
  Maxi2(A, B, Largest);
  Maxi2(Largest, C, Largest);
END { Maxi3 };

BEGIN { Main Program }
  TemperatureFtoC(212.0, R);
  WriteLn('212F is ', R: 7: 2, 'C');
  Write('Max of 3, 6, 5 = ');
  Maxi3(3, 6, 5, I);
  WriteLn(I: 2);
  FOR J := 1 TO 5 DO BEGIN
    Write('Area of Circle of radius ', J: 1, ' is');
    AreaCircle(J, S);
    WriteLn(S: 6: 2);
  END;
  C := '#';
  SnapTrace;
END. { MiscProcs2 }

```

A second miscellany of procedures, program MiscProcs2 shown in Figure 7.13, contains some procedures that are generally useful, either as such or as the basis from which particular procedures can be crafted.

SnapTrace()

Like Instructions seen above, this procedure also has no parameters and no local variables. It outputs the values of three global variables C, I and R. Note that these variables must be declared before this procedure is defined. Calls of this procedure could be inserted at various points in a program to test or debug it.

TemperatureFtoC(Fahrenheit, Celsius)

is a simple temperature conversion procedure that has one input, Fahrenheit, and one output, Celsius, and no local variables.

AreaCircle(Radius, Area)

is another simple algebraic formula packaged as a procedure. Notice that Pi does not need to be declared since it is a standard function.

Maxi2(X, Y, Max)

has two inputs X and Y and an output Max. It applies only to integers.

Maxi3(A, B, C, Largest)

has three inputs A, B, C and one output Largest. It calls the previous Maxi2 procedure twice.

Again, the body of this program illustrates the calling of these procedures in simple ways. The output obtained from running MiscProcs2 is given in Figure 7.14.

Figure 7.14 Results of program MiscProcs2 execution

```
212F is 100.00C
Max of 3, 6, 5 = 6
Area of Circle of radius 1 is 3.14
Area of Circle of radius 2 is 12.57
Area of Circle of radius 3 is 28.27
Area of Circle of radius 4 is 50.27
Area of Circle of radius 5 is 78.54
First = #
Second = 6
Third = 100.00
```

7.5 Procedures with Char, Boolean and Other Types

Figure 7.15 Program MoreProcs

```
PROGRAM MoreProcs;
(* Illustrates use of CHAR and BOOLEAN in
procedures *)

PROCEDURE CharToDigit(Ch: CHAR; VAR Digit:
INTEGER);
(* Converts numeric character to INTEGER *)
(** Pre-cond: ORD(Ch) must be in the range 48 to
57 **)
BEGIN
    Digit := ORD(Ch) - ORD('0');
END { CharToDigit };

PROCEDURE ReadDigit(VAR Digit: INTEGER);
(* Reads character digits *)
VAR Ch: CHAR;
BEGIN
    Read(Ch);
    WHILE (Ch < '0') OR (Ch > '9') DO BEGIN
```

```
    Write('?');
    Read(Ch);
END;
CharToDigit(Ch, Digit);
END { ReadDigit };

PROCEDURE ReadBool(VAR Truth: BOOLEAN);
(* Reads char T or F as truth value *)
VAR Ch: CHAR;
BEGIN
    Read(Ch);
    Ch := UpCase(Ch);
    WHILE (Ch <> 'T') AND (Ch <> 'F') DO BEGIN
        Write('Try again ');
        Read(Ch);
        Ch := UpCase(Ch);
    END;
    IF Ch = 'T' THEN
        Truth := TRUE
    ELSE
        Truth := FALSE;
    END { ReadBool };

PROCEDURE WriteBool(Truth: BOOLEAN);
(* Writes truth value *)
BEGIN
    IF Truth THEN
        Write('TRUE ')
    ELSE
        Write('FALSE');
    END { WriteBool };

VAR A, B, C, D, E, F, G, H, I, Sum: INTEGER;
    BoolValue: BOOLEAN;
    Return: CHAR;
BEGIN
    WriteLn('Enter ISBN ');
    ReadDigit(A);    ReadDigit(B);
    ReadDigit(C);    ReadDigit(D);
    ReadDigit(E);    ReadDigit(F);
    ReadDigit(G);    ReadDigit(H);
    ReadDigit(I);
    Sum := A + 2*B + 3*C + 4*D + 5*E + 6*F +
           7*G + 8*H + 9*I ;
    Write('Check is ');
    IF (Sum MOD 11) = 10 THEN
        WriteLn('X')
    ELSE
        WriteLn(Sum MOD 11: 1);
    Read(Return);
```

```
Write('Enter T or F ');
ReadBool(BoolValue);
Write('The Boolean value entered was ');
WriteBool(BoolValue);
WriteLn;
END. { MoreProcs }
```

Procedures that involve CHAR and BOOLEAN types are similar to those involving INTEGER types. However, these non-numeric types offer some differences, as illustrated in program `MoreProcs` of Figure 7.15.

CharToDigit, is a small procedure to convert the ten characters '0' to '9' into the corresponding ten INTEGER values. The procedure simply subtracts the ORD of '0', which is 48, from the ORD of the input character. The pre-condition written into the procedure states that the ORD of the input character must be within the range 48 to 57 ('0' to '9').

Problems may arise when an input character to this procedure is not in the proper range, '0' to '9'. In fact, the procedure makes no test and converts any character into an INTEGER—it must be remembered that a pre-condition is only a comment, it takes no part in the execution of the procedure. `CharToDigit` should be modified to test whether the character read is within the proper range, and only then convert the input. It could also be modified to display an error, but that still leaves the problem of what value to assign to the output parameter. Another possibility is to return a BOOLEAN value, called `Done`, which is usually TRUE but which can be set to FALSE in the case of any error condition. Then after each procedure call this `Done` condition should be tested to see if the program can continue on. This stresses the significance of indicating pre-conditions for procedures. It is important to make sure that the pre-conditions are respected, for the alternatives are messy.

ReadDigit is a procedure that reads a character which is to represent a digit. If the character is a digit it is converted to the corresponding INTEGER. If the input is not a digit then a question mark is output, another character is read in, and this process repeated until the input is a digit. Notice that `ReadDigit` calls the first `CharToDigit` procedure.

ReadBool, is similar in structure to `ReadDigit` above. It can be used to input Boolean values. It accepts only the two Character values T and F (either upper or lower case), and assigns to its output parameter, `Truth`, one of the corresponding logical values TRUE or FALSE.

WriteBool, is a procedure that writes out the string 'TRUE' or 'FALSE' depending on the value of its input BOOLEAN parameter, `Truth`. We have already seen it in Chapter 6 (Fig. 6.17).

The `MoreProcs` main program uses `ReadDigit` (and indirectly `CharToDigit`) to compute the check digit of the International Standard Book Number. It also calls `ReadBool` and `WriteBool` to illustrate the use of these procedures. The output obtained from a typical run is:

```

Enter ISBN
354091248
Check is 7
Enter T or F t
The Boolean value entered was TRUE

```

Generalized Item Types

Figure 7.16 Program GeneralProcs

```

PROGRAM GeneralProc;
(* Shows more general item type *)

TYPE ItemType = CHAR;

PROCEDURE Maj3( First, Second, Third: ItemType;
  VAR Majority: ItemType);
(* Majority of three values *)
BEGIN
  IF First = Second THEN
    Majority := First
  ELSE
    Majority := Third;
  END { Maj3 };

VAR A, B, C, M: ItemType;

BEGIN
  Write('Enter 3 values ');
  Read(A, B, C);
  Maj3(A, B, C, M);
  WriteLn('The majority is ', M);
END { GeneralProc }.

```

GeneralProc, in Figure 7.16, demonstrates one way of generalizing a procedure so that it can be readily adapted to a variety of needs. Throughout this program, the data being manipulated is declared as being of type `ItemType`. Notice that `ItemType` appears within the majority procedure `Maj3` and also in the declaration part. In the instance of `GeneralProc` shown, `ItemType` is defined as being the `CHAR` type.

```
TYPE ItemType = CHAR;
```

By changing this one line, other types could be used including `BOOLEAN`, user-defined, and subrange types. For example, adapting to the two `INTEGER` values 0 and 1 can be done simply by modifying this declaration to:

```
TYPE ItemType = 0..1;
```

It is important to realize that the `Maj3` procedure applies only when there are only two different values; 0 and 1, or `TRUE` and `FALSE` or 'M' and 'F', etc. If three different values are used the resulting majority would be meaningless.

For example if the three input values are 2, 5, 1 then the output would be 1, and if the input values were the same but in another order, 2, 1, 5, the output would be 5. Majority has meaning only for binary values.

7.6 Procedures with User-Defined types

As we have seen in the preceding example, procedures are not restricted to Pascal's standard built-in types. They can be used equally well with user-defined types, enumeration types and subrange types. In fact, since such types have no ready-made input nor output, procedures are the most convenient means for creating input and output actions for these types.

The program `WeekPay`, in Figure 7.17, does a simple payroll calculation for a week and illustrates the use of user-defined data types in procedures.

Figure 7.17 Program WeekPay

```
PROGRAM WeekPay;
(* Compute the weekly pay *)
TYPE WeekDayTyp = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
    MonthType = 1..12;
VAR Day: WeekDayTyp;
    Month: MonthType;
    PayRate, Hours, SumHours, D: INTEGER;

PROCEDURE ReadMonth(VAR Month: MonthType);
(* Read and validate month value *)
VAR Temp: INTEGER;
BEGIN
    Write('Enter Month as number (1 to 12): ');
    Read(Temp);
    WHILE (Temp < 1) OR (Temp > 12) DO BEGIN
        Write('Try again ');
        Read(Temp);
    END { WHILE };
    Month := Temp;
END { ReadMonth };

PROCEDURE ReadDay(VAR Day: WeekDayTyp);
(* Read and validate Day value *)
VAR DayStr: STRING;
BEGIN
    Write('Enter the week day and spell it all out: ');
    ReadLn(DayStr);
    IF DayStr = 'Sunday' THEN Day := Sun
    ELSE IF DayStr = 'Monday' THEN Day := Mon
    ELSE IF DayStr = 'Tuesday' THEN Day := Tue
    ELSE IF DayStr = 'Wednesday' THEN Day := Wed
    ELSE IF DayStr = 'Thursday' THEN Day := Thu
    ELSE IF DayStr = 'Friday' THEN Day := Fri
```



```

ELSE IF DayStr = 'Saturday' THEN Day := Sat
ELSE WriteLn('Error in day ');
END { ReadDay };

PROCEDURE WriteDay(Day: WeekDayTyp);
(* Output name of Day *)
BEGIN
CASE Day OF
Sun: Write('Sunday '); Mon: Write('Monday ');
Tue: Write('Tuesday '); Wed: Write('Wednesday ');
Thu: Write('Thursday '); Fri: Write('Friday ');
Sat: Write('Saturday ');
ELSE
Write('Error');
END { CASE };
END { WriteDay };

PROCEDURE NextDay(VAR Day: WeekDayTyp);
(* Find next day *)
BEGIN
IF Day <> Sat THEN
Inc(Day)
ELSE
Day := Sun;
END { NextDay };

BEGIN
ReadMonth(Month);
Write('Enter the pay rate: ');
ReadLn(PayRate);
WriteLn('Input the first day ');
ReadDay(Day);
WriteLn;
SumHours := 0;
FOR D := 1 TO 7 DO BEGIN
Write('Enter hours for ');
WriteDay(Day); Write(': ');
Read(Hours);
SumHours := SumHours + Hours;
NextDay(Day);
END { FOR };
WriteLn('The total pay is ', PayRate * SumHours:5);
END { WeekPay }.

```

In this program, `WeekDayTyp` is an enumerated type defined for the seven days of the week:

```
TYPE WeekDayTyp = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

All the days are represented by three characters, simply for consistency. A second type, `MonthType`, is defined as a subrange:

```
TYPE MonthType = 1..12;
```

It could equally well have been defined as an enumerated type with months Jan, Feb, etc.

The program comprises the following procedures.

ReadMonth, a procedure that asks for a value, and when the value received is in the appropriate range (1 to 12), that value is assigned to the output parameter `Month`.

ReadDay, a procedure that requests a string depicting a day of the week. Depending on the string input, it assigns to parameter `Day` the appropriate value from the type `WeekDayTyp`.

WriteDay, a procedure which displays the day of the week corresponding to the input value.

NextDay makes the week cyclic, so that the next day of 'Sat' is 'Sun'.

The main program computes the “simple” pay (without overtime) of one individual for a week beginning at any day of the week and continuing for seven days. Figure 7.18 shows a typical execution of the program.

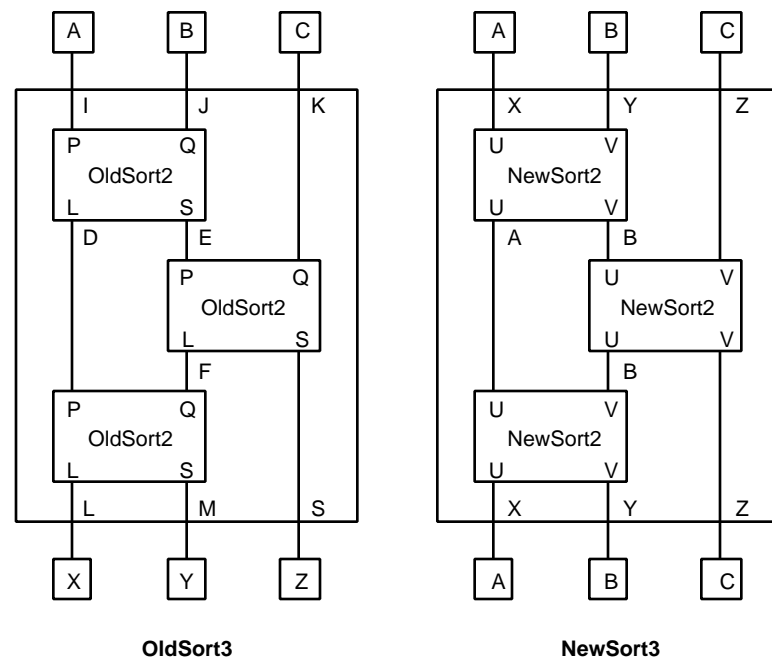
Figure 7.18 Execution of the WeekPay program

```
Enter Month as number (1 to 12): 3
Enter the pay rate: 10
Input the first day
Enter the week day and spell it all out: Thursday

Enter hours for Thursday : 15
Enter hours for Friday : 12
Enter hours for Saturday : 3
Enter hours for Sunday : 0
Enter hours for Monday : 10
Enter hours for Tuesday : 12
Enter hours for Wednesday : 5
The total pay is 570
```

7.7 More on Passing Parameters

As there are always several ways of solving a problem, there are also several ways of implementing a subprogram as a Pascal procedure. For example, let us consider subprogram `Sort3`, discussed in Chapter 7 of the Principles book. Figure 7.19 shows two dataflow diagrams for `Sort3` that differ only in labeling.

Figure 7.19 Dataflow diagrams for two versions of Sort3

On the left of the figure is OldSort3, which separates the input parameters I, J, K, from the output parameters, L, M, S (for Large, Medium, and Small). Similarly in that diagram, OldSort2 has two input parameters P and Q, which are separate from its two output parameters L and S (for Larger and Smaller).

Instead of OldSort3's six parameters, NewSort3 has just three input-output parameters, X, Y and Z. Upon completion of NewSort3's actions, X contains the largest of the three values and Z has the smallest. Similarly, NewSort2 has two input-output parameters, U and V and after execution, U has the larger of the two values and V the smaller.

A comparison of the old and new methods shows that the old forms have more variables, but the caller's original input values remain unchanged. The new methods use fewer variables but probably change the caller's original values. Sometimes the destruction of the old values is not important, so the new method would be preferred. At some other times the original values are important and must be protected, so the old method would then be preferred. New is not always better than old!

Figure 7.20 Comparison of two Pascal versions of Sort3

<pre> PROGRAM OldSortProg; (* Sorts three integer values *) PROCEDURE OldSort2 (First, Second: INTEGER; VAR Large, Small : INTEGER); VAR Temp: INTEGER; BEGIN Large := First; Small := Second; IF Large < Small THEN BEGIN { Swap } Temp := Large; Large := Small; Small := Temp ; END { Swap } END; { OldSort2 } PROCEDURE OldSort3 (First, Second, Third: INTEGER; VAR Maxi, Midi, Mini : INTEGER); VAR Temp1, Temp2, Temp3: INTEGER; BEGIN OldSort2(First, Second, Temp1, Temp2); OldSort2(Temp2, Third, Temp3, Mini); OldSort2(Temp1, Temp3, Maxi, Midi); END; { OldSort3 } VAR A, B, C, X, Y, Z: INTEGER; BEGIN Write('Enter three integers '); Read(A, B, C); OldSort3(A, B, C, X, Y, Z); Write('The sorted values are '); Write(X:4, Y:4, Z:4); END. { OldSortProg } </pre>	<pre> PROGRAM NewSortProg; (* Sorts three integer values *) PROCEDURE NewSort2 (VAR ToBeLarge: INTEGER; VAR ToBeSmall: INTEGER); VAR Temp: INTEGER; BEGIN IF ToBeLarge < ToBeSmall THEN BEGIN { Swap } Temp := ToBeLarge; ToBeLarge := ToBeSmall; ToBeSmall := Temp; END { Swap } END; { NewSort2 } PROCEDURE NewSort3 (VAR ToBeMaxi: INTEGER; VAR ToBeMidi: INTEGER; VAR ToBeMini: INTEGER); VAR Temp1, Temp2, Temp3: INTEGER; BEGIN NewSort2(ToBeMaxi, ToBeMidi); NewSort2(ToBeMidi, ToBeMini); NewSort2(ToBeMaxi, ToBeMidi); END; { NewSort3 } VAR A, B, C: INTEGER; BEGIN Write('Enter three integers '); Read(A, B, C); NewSort3(A, B, C); Write('The sorted values are '); Write(A:4, B:4, C:4); END. { NewSortProg } </pre>
---	--

Figure 7.20 shows the Pascal versions of OldSort3 and NewSort3 placed next to each other for comparison. The two algorithms are very similar. Notice that OldSort3 involves many more parameters and local variables. In fact,

NewSort3 has no local variables. OldSortProg also uses more variables than NewSortProg.

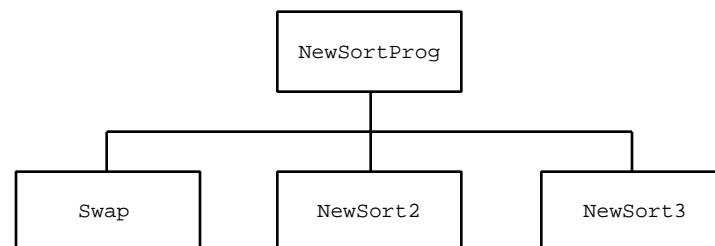
7.8 Nested Procedures

In Chapter 7 of the Principles book, you have seen through structure charts or contour diagrams that there were many ways of arranging or nesting procedures in a program. As an example, consider a modification to NewSortProg of Figure 7.20, where the swapping of two values, that is done in NewSort2, is extracted and made into a separate procedure. This new procedure, Swap, has two input-output parameters.

```
PROCEDURE Swap
  (VAR this: INTEGER;
   VAR that: INTEGER);
VAR temp: INTEGER;
BEGIN
  temp := this;
  this := that;
  that := temp;
END; { Swap }
```

Figure 7.21 is a structure chart corresponding to this new version of the NewSortProg.

Figure 7.21 Structure chart for NewSortProg



If we look at the program NewSortProg in Figure 7.20, we can see that it actually calls only NewSort3, and does not call Swap or NewSort2, as the structure chart suggests. Is our structure chart wrong? In fact, it is not. The way things are, the program could call all three procedures, which are available and on the same level. If we want to show the actual functional relationships between program components that a structure chart is supposed to show, we must draw another chart. Figure 7.22 represents the actual functional dependence of the NewSortProg program. It shows that the program calls NewSort3 which in turn calls NewSort2 which calls Swap.

Figure 7.22 Functional structure chart for NewSortProg

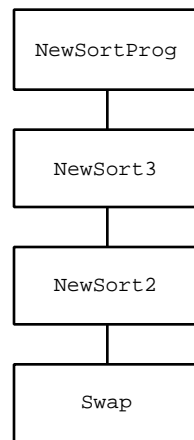


Figure 7.23 shows, side by side, two different versions of `NewSortProg`, based on these structure charts. On the left is `Sequence`, corresponding to the structure chart of Figure 7.21, where all the procedures are arranged in sequence. First comes `Swap`, which is used in the following procedure `NewSort2`, which is itself used in `NewSort3`, which is called in the main program.

Figure 7.23 A comparison of sequential and nested procedures

<pre> PROGRAM Sequence; (* Sorts three integer values *) PROCEDURE Swap (VAR this: INTEGER; VAR that: INTEGER); VAR temp: INTEGER; BEGIN temp := this; this := that; that := temp; END; { Swap } PROCEDURE NewSort2 (VAR ToBeLarge: INTEGER; VAR ToBeSmall: INTEGER); BEGIN IF ToBeLarge < ToBeSmall THEN Swap(ToBeLarge, ToBeSmall); END; { NewSort2 } PROCEDURE NewSort3 (VAR ToBeMaxi: INTEGER; VAR ToBeMidi: INTEGER; VAR ToBeMini: INTEGER); BEGIN NewSort2(ToBeMaxi, ToBeMidi); NewSort2(ToBeMidi, ToBeMini); NewSort2(ToBeMaxi, ToBeMidi); END; { NewSort3 } VAR A, B, C: INTEGER; BEGIN Write('Enter three integers '); Read(A, B, C); NewSort3(A, B, C); Write('The sorted values are '); Write(A:4, B:4, C:4); END. { Sequence } </pre>	<pre> PROGRAM Nest; (* Sorts three integer values *) PROCEDURE NewSort3 (VAR ToBeMaxi: INTEGER; VAR ToBeMidi: INTEGER; VAR ToBeMini: INTEGER); PROCEDURE NewSort2 (VAR ToBeLarge: INTEGER; VAR ToBeSmall: INTEGER); PROCEDURE Swap (VAR this: INTEGER; VAR that: INTEGER); VAR temp: INTEGER; BEGIN temp := this; this := that; that := temp; END; { Swap } BEGIN IF ToBeLarge < ToBeSmall THEN Swap(ToBeLarge, ToBeSmall); END; { NewSort2 } BEGIN NewSort2(ToBeMaxi, ToBeMidi); NewSort2(ToBeMidi, ToBeMini); NewSort2(ToBeMaxi, ToBeMidi); END; { NewSort3 } VAR A, B, C: INTEGER; BEGIN Write('Enter three integers '); Read(A, B, C); NewSort3(A, B, C); Write('The sorted values are '); Write(A:4, B:4, C:4); END. { Nest } </pre>
---	---

The right half of Figure 7.23 shows another version of the same program, *Nest*, corresponding to the structure chart of Figure 7.22, and where the procedures are

nested one inside the other. `NewSort3` calls `NewSort2` nested within it, which, in turn, calls `Swap`, which is nested within it.

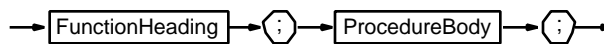
The behavior of these two programs is identical. However, the difference lies in accessibility. The procedures of `Sequence` are all accessible by the main program, whereas the procedures of `Nest` are accessible and may be called only by the procedures within which they are nested. These nested procedures are hidden, like local variables can be hidden. More of this hiding is considered later and in the Principles book.

7.9 Functions in Pascal

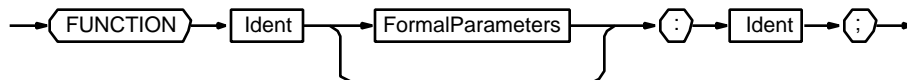
Although functions and procedures are very much alike, they differ from each other in several important ways. The major difference is that functions always return a single value. In contrast, the results of executing a procedure are either contained in one or more output parameters, or consist of some other effect such as displaying something on the screen, or writing some data to a file. This leads to another difference which is that function calls are used as parts of expressions, whereas procedure calls are complete statements. Consequently, it is usually better to name functions by nouns like `Sum`, `Size`, `Maximum`, etc. On the other hand, procedures are best described by verbs, e.g., `Accumulate`, `FindSize`, `Sort`, etc.

Figure 7.24 Syntax diagrams for Pascal functions

FunctionDeclaration



FunctionHeading



Functions in Pascal are defined by the syntax diagrams shown in Figure 7.24. As can be seen from these, they differ from `ProcedureDeclarations` only in the heading. Figure 7.25 shows a template for the declaration of a function, together with an example defining the trigonometric sine of an angle given in degrees. This function `SinD`, is useful because most versions of Pascal expect the argument to be in radians.

Figure 7.25 Template and example of function declaration

<pre> FUNCTION FuncName (list-of-parameters) : Return-type; Declarations; BEGIN body with FuncName := result; END; { FuncName } </pre>	<pre> FUNCTION SinD (Degrees: REAL) : REAL; VAR Radians: REAL; BEGIN Radians := Degrees / 57.3; SinD := SIN(Radians); END; { SinD } </pre>
--	--

Besides the obvious use of FUNCTION instead of PROCEDURE, note that the list of parameters is now followed by a colon and the data type of the function result. The returned value can have any of the types studied so far, including enumeration and sub-range types. The value of a function in Pascal may only be a single value, a *scalar*; that is to say, it may not be an array or a record. As a function is supposed to return a single result, it is strongly recommended that all the function parameters be input parameters.

Figure 7.26 Procedure vs. Function forms of finding maximum value

<pre> PROGRAM MaxProcedure; VAR A, B, C, D, E, F, G: INTEGER; PROCEDURE Maximize(X, Y: INTEGER; VAR M: INTEGER); BEGIN IF X > Y THEN M := X ELSE M := Y; END; { Maximize } BEGIN WriteLn('Enter four values'); ReadLn(A, B, C, D); Maximize(A, B, E); Maximize(C, D, F); Maximize(E, F, G); WriteLn('The maximum is ', G:3); END. { MaxProcedure } </pre>	<pre> PROGRAM MaxFunction; VAR A, B, C, D, G: INTEGER; FUNCTION Maximum(X, Y: INTEGER) : INTEGER; BEGIN IF X > Y THEN Maximum := X ELSE Maximum := Y; END; { Maximum } BEGIN WriteLn('Enter four values'); ReadLn(A, B, C, D); G := Maximum(Maximum(A, B), Maximum(C, D)); WriteLn('The maximum is ', G:3); END. { MaxFunction } </pre>
--	--

Figure 7.26 shows a comparison of two programs that find the maximum of four values, using a subprogram that can find the maximum of two values. On the left of the figure, the operation is implemented as a procedure, `Maximize`, while, on the right, it is implemented as a function, `Maximum`. Notice the great similarities, but also the differences between the two forms. The boxed areas in the figure highlight the significant differences. Procedure `Maximize` has a third parameter for the result, whereas function `Maximum` has only two parameters.

Use, or invocation, of the subprogram is also very different between procedure form and function form, as can be seen in the main parts of these two programs.

Program `MaxProcedure` requires two temporary variables `E` and `F` and three separate statements:

```
Maximize(A, B, E);
Maximize(C, D, F);
Maximize(E, F, G);
```

Program `MaxFunction` requires no temporary variables, but the result is a rather long statement:

```
G := Maximum(Maximum(A, B), Maximum(C, D));
```

In this case, the function form seems to be the most convenient, but in general, procedures are more powerful. Procedures can always be used, whereas functions only apply when one single value is to be returned.

Many Functions

Program `ManyFunctions`, in Figure 7.27, shows a number of examples of functions. These illustrate the structure of function declarations, and also demonstrate the use of these functions. Most of these functions are quite general, and you can use them directly by copying them into your programs. They could also be made into a Library, say `MiscFuncs`, and used in any program. We will show how this is done later in this chapter.

Figure 7.27 Program Many Functions

```
PROGRAM ManyFunctions;
TYPE DIGIT = 0 .. 9; { Sub range }
    ItemType = CHAR;

VAR Year : INTEGER;

FUNCTION Fahrenheit(C: REAL): REAL;
BEGIN
    Fahrenheit := (9.0 / 5.0) * C + 32.0;
END; { Fahrenheit }

FUNCTION TanD(Degrees: REAL): REAL;
CONST RadiansPerDegree = 2.0 * Pi / 360.0;
VAR Radians: REAL;
```

```
BEGIN
  Radians := RadiansPerDegree * Degrees;
  TanD := SIN(Radians) / COS(Radians);
END; { TanD }

FUNCTION Maxi2(First, Second: INTEGER): INTEGER;
BEGIN
  IF First < Second THEN
    Maxi2 := Second
  ELSE
    Maxi2 := First;
END; { Maxi2 }

FUNCTION Maj3(First, Second, Third: ItemType):
ItemType;
BEGIN
  IF First = Second THEN
    Maj3 := First
  ELSE
    Maj3 := Third;
END; { Maj3 }

FUNCTION IsLeap(Year: INTEGER): BOOLEAN;
BEGIN
  IF Year MOD 4 = 0 THEN
    IsLeap := TRUE
  ELSE
    IsLeap := FALSE;
END; { IsLeap }

FUNCTION CharDec(Ch: CHAR): DIGIT;
BEGIN
  CharDec := ORD(Ch) - ORD('0');
END; { CharDec }

FUNCTION Size(It: INTEGER): INTEGER;
VAR Count: INTEGER;
BEGIN
  Count := 0;
  WHILE It > 0 DO BEGIN
    It := It DIV 10;
    Inc(Count);
  END { WHILE };
  Size := Count;
END; { Size }

FUNCTION PowerP(X: REAL; N: INTEGER): REAL;
VAR I: INTEGER;
    P: REAL;
BEGIN
```

```
P:= 1.0;
FOR I := 1 TO N DO BEGIN
  P := P * X;
  N := N - 1;
END { FOR };
PowerP := P;
END; { PowerP }

BEGIN
  WriteLn('-40 degrees C is ',
Fahrenheit(-40.0):10:2);
  WriteLn('Tangent of 45 degrees is ',
TanD(45.0):10:2);
  WriteLn('The max of 3, 6, 5 is ', Maxi2(3,
Maxi2(6, 5)):3);
  WriteLn('The majority of Y N Y is ', Maj3('Y',
'N', 'Y'));
  Write('Enter the year ');
  Read(Year);
  IF IsLeap(Year) THEN
    WriteLn(Year: 4, ' is a leap year')
  ELSE
    WriteLn(Year: 4, ' is not a leap year');
    WriteLn('The year is ', year:Size(Year));
    WriteLn('1 plus 1 is ', CharDec('1') +
CharDec('1'):1);
    WriteLn('2 to the power 20 is ', PowerP(2.0,
20):10:0);
  END. { ManyFunctions }
```

The simple functions shown in `ManyFunctions` involve various data types, including `CHAR`, `BOOLEAN`, and enumeration types. You might have noticed that the majority of the function calls shown in the body of the program involve constant arguments. This is done for brevity, because actual parameters of functions can also be variables or expressions. Another thing worth of notice is that the functions do not test the values passed in. For example, the tangent function, `TanD`, does not test before dividing by the cosine of the angle. Thus, if the angle is 0° or 180° , there will be a division by zero error. Although the users of such functions should ensure that bad parameters are not passed, the **functions should always check their pre-conditions**.

Figure 7.28 shows the output corresponding to the execution of the given main program.

Figure 7.28 Execution results for `ManyFunctions`

```
-40 degrees C is   -40.00
Tangent of 45 degrees is   1.00
The max of 3, 6, 5 is 6
The majority of Y N Y is Y
Enter the year 1992
```

```

1992 is a leap year
The year is 1992
1 plus 1 is 2
2 to the power 20 is 1048576

```

Fahrenheit and TanD are functions that return a REAL value. Notice that TanD uses a constant declaration involving an expression.

Function Maxi2 operates on integers only. The main program shows the nesting of applications of Maxi2 in an expression to compute the maximum of three integers.

Function Maj3 returns the majority of any two values that are declared as ItemType. In this case ItemType has been declared as CHAR.

BOOLEAN function IsLeap makes a simplistic determination of whether or not a given year is a leap year. The algorithm does not provide the correct result for the year 1900 or 2100 (and similar non leap centuries).

Function CharDec converts a given character digit into its corresponding numeric digit. It does not check that the given value is in the proper range to be a digit.

Finally, function Size determines the number of digits in an integer. Such a function can be very useful within Write or WriteLn statements to determine the output width of numbers. This is how it is used in the body of the program, to output the year.

7.10 SubPrograms: Variations on a theme

You know that there is always more than one way of solving a problem; you have seen it repeatedly in the Principles book. Similarly, a subprogram may be packaged, or made available to the user, in several different manners. We will illustrate this by the UnCapitalize subprogram that converts an upper case letter (between A and Z) into its corresponding lower case form, and does not change any lower case letters. The method used for this is simple when it is realized that the ORD of an upper case letter differs from the corresponding lower case letter by 32 (look back to the ASCII table seen in Chapter 6).

The main part of the UnCapitalize action is as follows:

```

(* Convert Char Ch to lowercase LowerCh *)
IF ('A' <= Ch) AND (Ch <= 'Z') THEN BEGIN
    Code := ORD(Ch);
    NewCode := Code + 32;
    LowerCh := CHR(NewCode);
END;
(* LowerCh is the corresponding lower case char *)

```

This can also be written in various equivalent shorter forms without using the temporary integer variables Code and NewCode, for example:

```

IF ('A' <= Ch) OR (Ch <= 'Z') THEN

```

```
LowerCh := CHR(ORD(Ch) + 32);
```

We can package this simple action in four different manners, by using either procedures or functions, and by using different kinds of parameters. Program UnCapping of Figure 7.29 shows the four versions, together with code that demonstrates their use.

Figure 7.29 Program UnCapping

```
PROGRAM UnCapping;
CONST UPPERLowerDiff = 32;

PROCEDURE UnCapOf(Ch: CHAR; VAR LowerCh: CHAR);
VAR Code, NewCode: INTEGER;
BEGIN
  IF ('A' <= Ch) AND (Ch <= 'Z') THEN BEGIN
    Code := ORD(Ch);
    NewCode := Code + UpperLowerDiff;
    LowerCh := CHR(NewCode);
  END
  ELSE
    LowerCh := Ch;
END; { UnCapOf }

PROCEDURE UnCap(VAR Ch: CHAR);
BEGIN
  IF ('A' <= Ch) AND (Ch <= 'Z') THEN
    Ch := CHR(ORD(Ch) + UpperLowerDiff);
END; { UnCap }

FUNCTION UnCapped(Ch: CHAR): CHAR;
BEGIN
  IF ('A' <= Ch) AND (Ch <= 'Z') THEN
    UnCapped := CHR(ORD(Ch) + UpperLowerDiff)
  ELSE
    UnCapped := Ch;
END; { UnCapped }

FUNCTION IsUnCapped(Ch: CHAR): BOOLEAN;
BEGIN
  IF ('A' <= Ch) AND (Ch <= 'Z') THEN
    IsUnCapped := FALSE
  ELSE
    IsUnCapped := TRUE;
END; { IsUnCapped }

VAR Ch, NewCh, Extra: CHAR;
BEGIN { Main }
  { Testing UnCapOf }
  Write('Enter a character followed by Return: ');
  Read(Ch);
```

```

Read(Extra); { eat Return }
UnCapOf(Ch, NewCh);
WriteLn('The lower case is ', NewCh);

{ Testing UnCap }
Write('Enter a character followed by Return: ');
Read(Ch);
Read(Extra); { eat Return }
UnCap(Ch);
WriteLn('The lower case is ', Ch);

{ Testing UnCapped }
Write('Enter a character followed by Return: ');
Read(Ch);
Read(Extra); { eat Return }
WriteLn('The lower case is ', UnCapped(Ch));

{ Testing IsUnCapped }
Write('Enter a character followed by Return: ');
Read(Ch);
Read(Extra); { eat Return }
IF IsUnCapped(Ch) THEN
    WriteLn('It is lower case ')
ELSE
    WriteLn('It is capitalized');
END. { Uncapping }

```

UnCapOf(Ch, LowerCh) is a procedure where Ch is an input parameter, and LowerCh is an output parameter. Its name was chosen for easy reading as the statement “Uncap value of Ch is LowerCh”.

UnCap(Ch) is another procedure where Ch is an input-output parameter. Its name is a simple verb, “Uncap it”.

UnCapped(Ch) is a function that returns the uncapped value of the character Ch. It is used as a part of an expression as in the example:

```
IF UnCapped(Ch) = 'y' THEN ...
```

IsUnCapped(Ch) is yet another way of viewing this Uncapping process, but it differs from the others. It returns a BOOLEAN value (TRUE or FALSE) depending on whether the argument Ch is capped or not.

Which one of these four versions should be chosen? It all depends on how you intend to use this subprogram. All these forms are correct, but none is more useful, convenient or natural for all purposes. The proper one to select depends on how it will be used. If this action is part of another action then it should be a function. On the other hand, if the action should stand alone then it should be a procedure.

7.11 Recursion in Pascal

The process of recursion, introduced in Chapter 7 of the Principles book, involves a subprogram that calls itself. In Pascal, both procedures and functions can be recursive without having to declare it in any special manner. Thus, it is very easy to implement a recursive solution to a problem.

In the introduction to recursion, a comparison was made between the pseudocode for an iterative and a recursive algorithm for calculating the square of a number Num by summing the first Num odd numbers. Figure 7.30 repeats this comparison and also shows the Pascal implementation of these two algorithms both as procedures and functions. Study the four implementations carefully, comparing both horizontally and vertically.

Figure 7.30 Iterative vs. recursive implementations of the OddSquare algorithm

<pre> IterativeSquare(Num, <u>Square</u>): Set Square to 0 For Count = 1 to Num by 1 Set Square to Square + 2 × Count - 1 </pre>	<pre> RecursiveSquare(Num, <u>Square</u>): If Num = 1 Set Square to 1 Else RecursiveSquare(Num-1, Square) Set Square to Square + 2 × Num - 1 </pre>
<pre> PROCEDURE IterSquareProc(Num: INTEGER; VAR Square: INTEGER); VAR Count: INTEGER; BEGIN Square := 0; FOR Count := 1 TO Num DO Square := 2 * Count - 1 + Square; END; { IterSquareProc } </pre>	<pre> PROCEDURE RecurSquareProc(Num: INTEGER; VAR Square: INTEGER); BEGIN IF Num = 1 THEN Square := 1 ELSE BEGIN RecurSquareProc(Num - 1, Square); Square := 2 * Num - 1 + Square; END; END; { RecurSquareProc } </pre>

<pre> FUNCTION IterSquareFunc(Num: INTEGER) : INTEGER; VAR Square, Count: INTEGER; BEGIN Square := 0; FOR Count := 1 TO Num DO Square := 2 * Count - 1 + Square; IterSquareFunc := Square; END; { IterSquareFunc } </pre>	<pre> FUNCTION RecurSquareFunc(Num: INTEGER) : INTEGER; VAR Square: INTEGER; BEGIN IF Num = 1 THEN RecurSquareFunc := 1 ELSE RecurSquareFunc := 2 * Num - 1 + RecurSquareFunc(Num - 1); END; { RecurSquareFunc } </pre>
---	---

These examples of recursion have all consisted of procedures or functions that called themselves directly; this is known as *direct recursion*. However, not all recursion is direct, as illustrated in the program `ZigZagSquare`, shown in Figure 7.31, which consists of two procedures `Zig` and `Zag` that call each other—this is known as *indirect recursion*:

Figure 7.31 Indirect recursion

```

PROGRAM ZigZagSquare;
(* Square by recursive procedure uses two, mutually
   recursive procedures. *)

PROCEDURE Zag(Num: INTEGER; VAR Square: INTEGER);
FORWARD;

PROCEDURE Zig(Num: INTEGER; VAR Square: INTEGER);
BEGIN
    WriteLn('In Zig: Num = ', Num: 2); { Trace }
    Zag(Num - 1, Square);
    Square := Num + Square;
END; { Zig }

PROCEDURE Zag(Num: INTEGER; VAR Square: INTEGER);
BEGIN
    WriteLn('In Zag: Num = ', Num: 2); { Trace }
    IF Num = 0 THEN
        Square := 0
    ELSE
        Zig(Num - 1, Square);
    END; { Zag }

VAR TrialNum, TopOddNum, Sq: INTEGER;

BEGIN

```

```
Write('Enter a value ');
Read(TrialNum);
TopOddNum := 2 * TrialNum - 1;
Zig(TopOddNum, Sq);
WriteLn('The square is ', Sq:2);
END. { ZigZagSquare }
```

Since recursion in general, and indirect recursion in particular, are often difficult to understand, a `WriteLn` statement that prints out a debugging trace has been added to each of the mutually recursive procedures. The output obtained from a typical run is:

```
Enter a value 4
In Zig: Num = 7
In Zag: Num = 6
In Zig: Num = 5
In Zag: Num = 4
In Zig: Num = 3
In Zag: Num = 2
In Zig: Num = 1
In Zag: Num = 0
The square is 16
```

In this example, the algorithm computes the square of 4 by summing the first four odd numbers, 1, 3, 5 and 7. The main body of the program, after having obtained the number, 4, computes the value of the fourth odd number, $2 \times 4 - 1$. It then calls `Zig`. `Zig` does not call itself directly, but does call `Zag`, which then calls `Zig`; so indirectly `Zig` calls itself. In the above trace, no computation is done before the base case (`Num = 0`) has been found since the computation (`Square := Num + Square;`) is only done in `Zig` after returning from the recursive calls.

This mutual referencing of the two procedures poses a slight problem in their declaration. The syntax rules of Pascal require that a procedure be defined before it can be referenced. Since `Zig` and `Zag` reference each other, it is impossible to meet this requirement directly. To handle this, the header of `Zag` is specified first with the directive `FORWARD` indicating that its full definition will appear later. This allows a procedure like `Zag` to be referenced within `Zig` before it is fully declared.

The program `GlobalLocal`, given in Figure 7.32, is a program that emphasizes the difference between local and global declarations.

Figure 7.32 Global-Local example

```
PROGRAM GlobalLocal;
(* Shows Global vs. Local Variables *)
VAR Ch: CHAR; (* Global declaration *)

PROCEDURE GloLo;
(* Uses local declaration of Ch *)
VAR Ch: CHAR; (* Local declaration *)
BEGIN
```

```

    Read(Ch);
    IF Ch <> '.' THEN BEGIN
        GloLo;
        Write(Ch);
    END;
END; { GloLo }

PROCEDURE GloGlo;
(* Uses global declaration of Ch *)
BEGIN
    Read(Ch);
    IF Ch <> '.' THEN BEGIN
        GloGlo;
        Write(Ch);
    END;
END; { GloGlo }

BEGIN
    WriteLn('Enter a sentence ending with a period');
    GloLo;
    WriteLn;
    WriteLn('Enter a sentence ending with a period');
    GloGlo;
    WriteLn;
END. { GlobalLocal }

```

When the variable `Ch` is declared locally, as shown in `GloLo`, a given input string is reversed. Thus, the sequence "EVIL DID I LIVE." is output as "EVIL I DID LIVE" (with no period) by `GloLo`. But when the globally declared `Ch` is used as in `GloGlo`, then the same input sequence yields an output of 15 periods!

You are challenged to trace this program to understand this behavior. As a hint, note that each invocation of a procedure (recursive or not) has its own private space for parameters and local variables. So, the 16 recursive calls to `GloLo` generate 16 different `Ch` variables, each loaded with a different character from the input string. However, there is only one global variable ...

7.12 Libraries in Pascal

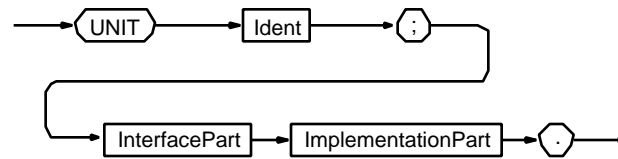
Units

Procedures and functions that are related in the actions they perform, or the type of data that they manipulate, are frequently gathered together to form a *library*. In Pascal, libraries can be created using *units*. These units are collections of procedures, functions and other resources (constants, types, etc.) that are made available for use by other programs. The units can be compiled separately, and then referenced by the program that wants to use them through the `USES` clause. This keeps the details of the units hidden, while sharing. It

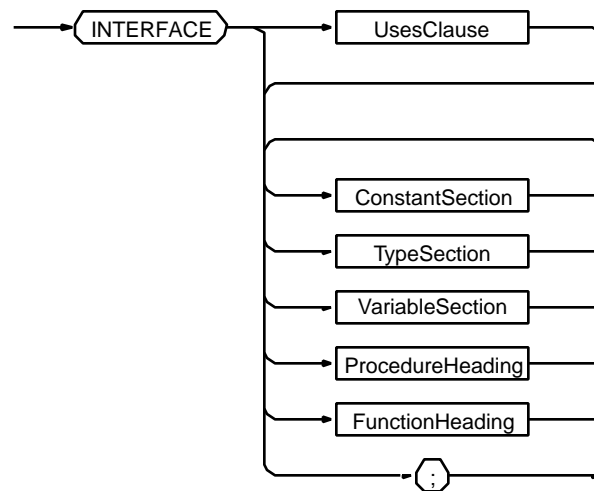
results in programs that are smaller and better structured, because of their use of units.

Figure 7.33 Syntax diagrams for Units

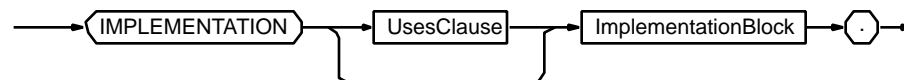
Unit



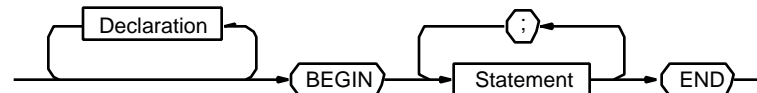
InterfacePart



ImplementationPart



ImplementationBlock



The syntax of a unit is defined by the syntax diagrams in Figure 7.33. Units have two main parts: a public part, the *Interface Part*, that is available to users, and a private part, the *Implementation Part*, which can be hidden from users. The public part describes *what* is available to users; the private part hides *how* this is implemented.

The Interface Part is the public part, which indicates what is available to any users; it specifies what can be used by other programs or units. This part can

contain declarations of named constants, of variables, of types, and only the headers of procedures and functions. These do not need to be in any definite order. This part can also contain a `USES` clause that refers to other units that it uses, but this clause should be at the beginning of the Interface Part.

The Implementation Part is the private part of a unit. It contains the entire declarations of the procedures and functions that were defined by headers only in the Interface Part. The Implementation Part could also contain declarations and definitions for resources that are available only within the unit. The order of listing these is not important. It could also contain a `USES` clause that refers to other units, and that should be at the beginning of the Implementation Part. Variables declared in this part are called *private* or *owned* variables, and are available to all the subprograms within the unit but not outside of it. The values in these variables are retained after the procedures and functions have been executed.

In the Implementation Part, there is also an optional section that starts with `BEGIN` and contains a set of statements that can initialize variables, open files, etc. This section is always executed before any of the programs which use this unit.

A template for a unit is shown in Figure 7.34.

Figure 7.34 Unit template

```
UNIT Name-of-Unit;
INTERFACE (***** Public *****)
  USES UseList;
  CONSTs
  TYPEs
  VARs
  PROCEDURE Headers
  FUNCTION Headers
IMPLEMENTATION (***Private***)
  CONSTs
  TYPEs
  VARs
  PROCEDURES
  FUNCTIONS
BEGIN (** Initialization **)
  Statements;
END. (* Name-of-Unit *)
```

Figure 7.35 shows two versions of the program `ShortSort`, which we discussed in Chapter 5 (Fig. 5.29).

Figure 7.35 ShortSort program and Unit

<pre>PROGRAM ShortSort; (* Sorts three integer values *) PROCEDURE Sort2(First, Second: INTEGER; VAR Large, Small : INTEGER); VAR Temp: INTEGER; BEGIN Large := First; Small := Second; IF Large < Small THEN BEGIN Temp := Large; (* Swap *) Large := Small; Small := Temp;</pre>	<pre>UNIT ShortSortLib; INTERFACE PROCEDURE Sort2(First, Second: INTEGER; VAR Large, Small : INTEGER); PROCEDURE Sort3(First, Second, Third: INTEGER; VAR Maxi, Midi, Mini : INTEGER); IMPLEMENTATION PROCEDURE Sort2(First, Second: INTEGER;</pre>
--	--

On the left of the figure is a complete program, `ShortSort`, that asks for, inputs, sorts and displays three integer values. That program is a single monolithic entity. On the right of Figure 7.35, the `ShortSort` procedures have been built into a library, `ShortSortLib`, that is used by a small program, `ShortSortProg`. Both programs `ShortSort` and `ShortSortProg` have the same behavior, and produce the same results from the same inputs. However, they have a different structure, and `ShortSortProg` is much simpler than `ShortSort`. The `ShortSortLib` unit is an example of a very minimal and simple unit (you might note we haven't included all the procedures that were planned in Chapter 5).

UtilityLib: a custom-made utilities Library

Normally, libraries are collections of declarations that are related in some way. We'll introduce here a library example, `UtilityLib` in Figure 7.36, that shows some of the main characteristics of libraries. The major reasons for using libraries involve in particular convenience, consistency, speed, readability, and reliability.

Figure 7.36 The `UtilityLib` library

```
UNIT UtilityLib;

INTERFACE (***** public *****)

CONST GRAMSInPOUND = 454;    (* INTEGER *)
      TWOPI      = 2.0 * Pi;  (* REAL *)
      BELL       = #7;       (* CHAR *)
      USER      = 'John Motil';(* STRING *)

TYPE DigitTyp = 0..9;
      ItemType = CHAR;

FUNCTION Int(R: REAL): INTEGER;
(* Chops R to nearest whole *)
FUNCTION Float(I: INTEGER): REAL;
(* Converts Integer to Real *)
PROCEDURE Incr(VAR R: REAL; S: REAL);
(* Increments Real R by amount S *)
PROCEDURE WrLn;
(* Writes a new line *)

IMPLEMENTATION (***** private *****)
FUNCTION Int(R: REAL): INTEGER;
(* Chops R to nearest whole *)
BEGIN
  Int := TRUNC(R);
END;

FUNCTION Float(I: INTEGER): REAL;
```

```
(* Converts Integer to Real *)
BEGIN
  Float := I;
END;

PROCEDURE Incr(VAR R: REAL; S: REAL);
(* Increments Real R by amount S *)
BEGIN
  R := R + S;
END;

PROCEDURE WrLn;
(*Rename of WriteLn *)
BEGIN
  WriteLn
END;
END. { UtilityLib }
```

Convenience is a principal reason for using libraries. For example, `UtilityLib` contains a set of useful constants such as `GRAMSInPOUND`, `TWOPI` or `BELL` (the character that causes the terminal to make a sound—remember that a character can be represented by its ASCII code preceded by a #). It is easier to use these than to memorize the numeric constants.

Generality is another reason for using libraries. Some of the constants in a library may have values that change only occasionally, such as `USER` in `UtilityLib`. These constants can serve to “parameterize” programs; when changes are necessary they are made in this one place and imported elsewhere; this is better than trying to find many places to make the change.

Structure is a reason for using libraries that is closely related to the previous one. Types such as the subrange `DigitTyp`, for example, can be defined once in a single library and used whenever necessary. Also, types, such as `ItemType`, can be defined as required to be `CHAR`, `INTEGER`, `DigitTyp`, etc. and then used in a more general procedure such as `Max2` as shown in the program `UtilityProg`. This is better than having many versions of `Maxes`, (`MaxInt`, `MaxChar`, `MaxReal`, etc.) one for each different type. This is another form of parameterization.

Language extension is a very important reason for creating libraries. For example, an increment procedure `Incr` can be created to apply to `REAL` values. A `Decr` procedure can be similarly created.

Consistency is yet another reason for using libraries. For example Pascal has a function `TRUNC` to convert from `REALS` to `INTEGERS`, but there is no function to convert the other way. A function `Float` could be created as shown, to convert `INTEGER` values to `REAL` values.

The function `TRUNC` is available in other languages under the name `Int`. An `Int` function could be created in Pascal simply by re-naming `TRUNC` as shown.

Brevity, is a further reason for using a library. For example, if you do not wish to use `WriteLn` you could abbreviate it to `WrLn`. Similarly, `Write` could be abbreviated to `Wr`, or `Print` or anything else. This is done simply by renaming `Write` as shown.

Readability could also be enhanced by using Libraries. Naming a constant `GRAMSInPOUND` indicates its purpose; renaming the conversion function `TRUNC` to `RealToInt` may also be more easily remembered, if not more readable.

Program `UtilityProg`, shown in Figure 7.37, is a program which simply tests the above library, `UtilityLib`.

Figure 7.37 Program UtilityProg

```
PROGRAM UtilityProg;
(* Tests parts of UtilityLib *)

USES UtilityLib;

PROCEDURE Max2( A, B: ItemType;
               VAR C: ItemType);
BEGIN
  IF A > B THEN
    C := A
  ELSE
    C := B;
END; { Max2 }

VAR Ch1, Ch2, Ch: CHAR;
    X: REAL;
BEGIN
  Write('Done by ', USER);
  Writeln;
  Write(BELL);
  Write('Enter 2 characters ');
  Read(Ch1, Ch2);
  Max2(Ch1, Ch2, Ch);
  Writeln('The maximum value is ', Ch);
  X := Pi;
  Incr(X, TWOPI);
  Writeln('Three Pi = ', X:8:4);
  Write('Two Pi chops to ');
  Writeln(Int(TWOPI):2);
  Write('Int of Pi is ');
  Write(Float(Int(Pi)):4:2);
END. { UtilityProg }
```

In that program, note that procedure `Max2` is based on `ItemType` from `UtilityLib`. Also note the use of various constants, procedures and functions from that library.

Other Libraries: DateLib, BitLib, CharLib

Our next library example is a library that is concerned with dates, `DateLib`. A UNIT defining `DateLib` is shown in Figure 7.38. To simplify the example, only three functions have been declared: `IsLeap`, `DaysInMonth` and `DayOfYear`. These are sufficient, however, to allow a programmer to use this library to create various programs. For example, let us create a program to find the number of days to Christmas, from any given date in the year.

Figure 7.38 The DateLib library

```
UNIT DateLib;

INTERFACE
  FUNCTION IsLeap(Year: INTEGER): BOOLEAN;
  FUNCTION DaysInMonth(Year, Month: INTEGER)
    : INTEGER;
  FUNCTION DayOfYear(Year: INTEGER;
    Month: INTEGER; Day: INTEGER)
    : INTEGER;

IMPLEMENTATION

  FUNCTION IsLeap(Year: INTEGER): BOOLEAN;
  BEGIN
    IF Year MOD 400 = 0 THEN
      IsLeap := TRUE
    ELSE IF Year MOD 100 = 0 THEN
      IsLeap := FALSE
    ELSE IF Year MOD 4 = 0 THEN
      IsLeap := TRUE
    ELSE (* not divisible *)
      IsLeap := FALSE;
    END; { IsLeap }

  FUNCTION DaysInMonth(Year, Month: INTEGER)
    : INTEGER;
  VAR Days: INTEGER;
  BEGIN
    CASE Month OF
      9, 4, 6, 11:
        Days := 30;
      1, 3, 5, 7, 8, 10, 12:
        Days := 31;
      2:
        IF IsLeap(Year) THEN
          Days := 29
        ELSE
          Days := 28;
        END;
    DaysInMonth := Days;
  END; { DaysInMonth }

  FUNCTION DayOfYear(Year: INTEGER;
    Month: INTEGER; Day: INTEGER)
    : INTEGER;
  VAR Julian, Mon: INTEGER;
  BEGIN
    Julian := 0;
```

```

FOR Mon := 1 TO Month DO
  Julian := Julian + DaysInMonth(Year, Mon);
  Inc(Julian, Day);
  DayOfYear := Julian;
END; { DayOfYear }
END. { DateLib }

```

At the level of the functions supplied by DateLib, the program DaysToXmas is very easy to write. It consists simply of reading a date, converting it to the DayOfYear, which gives the ordinal number of any date in the year, and subtracting that from the DayOfYear of Christmas (which is a little less than 365 or 366). It is assumed that both dates are in the same year and that the present date is before Christmas. The DayOfYear function calls the procedure DaysInMonth, which in turn calls IsLeap. In a more complete version of DateLib, there would be more procedures and functions, such as:

```

ReadDate(Year, Month, Day);    { date input }
WriteDate(Year, Month, Day);   { date output }
GoodDate(Year, Month, Day):BOOLEAN; { date validation }
UpDate(Year, Month, Day);      { next day's date }
WeekDate(Year, Month, Day);    { day of week }
Calendar(Year, Month);         { output month calendar }

```

Our reduced set of functions is sufficient to write DaysToXmas, shown in Figure 7.39.

Figure 7.39 Program DaysToChristmas

```

PROGRAM DaysToXmas;
(* Finds Days to Christmas with DateLib *)

USES DateLib;

VAR Year, Month, Day: INTEGER;
    DayOfYearNow, DayOfYearXmas, Elapsed: INTEGER;
BEGIN
  WriteLn('Enter Year, Month, Day ');
  Read(Year, Month, Day);
  DayOfYearNow := DayOfYear(Year, Month, Day);
  DayOfYearXmas := DayOfYear(Year, 12, 25);
  Write('Days to Christmas = ');
  Elapsed := DayOfYearXmas - DayOfYearNow;
  WriteLn(Elapsed:3);
END. { DaysToXmas }

```

Three examples of executions of this program follow.

```

Enter Year, Month, Day
1996 12 24
Days to Christmas = 1

Enter Year, Month, Day

```

```
1996 1 1
Days to Christmas = 359

Enter Year, Month, Day
1996 4 1
Days to Christmas = 269
```

The library `BitLib`, in Figure 7.40, defines binary digits of a type `BitType`, and three operations, `And`, `Or` and `Not`, that manipulate these binary digits.

Figure 7.40 The `BitLib` library

```
UNIT BitLib;
(* A Library of Bits *)

INTERFACE
  TYPE BIT = 0..1;

  PROCEDURE ReadBit(VAR B: BIT);
  PROCEDURE WriteBit(B: BIT);
  PROCEDURE And2(X, Y: BIT; VAR Z: BIT);
  PROCEDURE Or2(X, Y: BIT; VAR Z: BIT);
  PROCEDURE Not1(X: BIT; VAR Z: BIT);
  PROCEDURE And3(A, B, C: BIT; VAR D: BIT);
  PROCEDURE Or3(A, B, C: BIT; VAR D: BIT);

IMPLEMENTATION
  PROCEDURE ReadBit(VAR B: BIT);
  BEGIN
    Write('Give a bit '); Read(B);
    WHILE (B<>0) AND (B<>1) DO BEGIN
      Write('Enter 0 or 1 ');
      Read(B);
    END;
  END; { ReadBit }

  PROCEDURE WriteBit(B: BIT);
  BEGIN
    Write(B:2);
  END; { WriteBit }

  PROCEDURE And2(X, Y: BIT; VAR Z: BIT);
  BEGIN
    Z := X * Y;
  END; { And2 }

  PROCEDURE Or2(X, Y: BIT; VAR Z: BIT);
  BEGIN
    Z := X + Y - X * Y;
  END; { Or2 }
```

```
PROCEDURE Not1(X: BIT; VAR Z: BIT);
BEGIN
  Z := 1 - X;
END; { Not1 }

PROCEDURE And3(A, B, C: BIT; VAR D: BIT);
BEGIN
  D := A * B * C;
END; { And3 }

PROCEDURE Or3(A, B, C: BIT; VAR D: BIT);
BEGIN
  Or2(A, B, D);
  Or2(C, D, D);
END; { Or3 }
END. { BitLib }
```

Program BitProg, in Figure 7.41, uses the procedures of this BitLib library, and shows how all four combinations of values can be generated to verify DeMorgan's theorem, in a manner similar to what you saw in Chapter 6, extended to three variables.


```

PERIOD = #46; LINEFEED = #10;
RETURN = #13; ESCAPE   = #27;

FUNCTION CharToInt(Ch: CHAR): INTEGER;
(* Convert a numeric character to decimal digit *)
FUNCTION IntToChar(Digit: INTEGER): CHAR;
(* Convert a decimal digit to a character *)
FUNCTION IsDigit(Ch: CHAR): BOOLEAN;
(* Indicate if character is a digit*)

IMPLEMENTATION
CONST UPPERLowerDIFF = 32;

FUNCTION CharToInt(Ch: CHAR): INTEGER;
(* Convert a Character to Decimal Digit *)
BEGIN
  IF ('0' <= Ch) AND (Ch <= '9') THEN
    CharToInt := ORD(Ch) - ORD('0')
  ELSE
    CharToInt := -1;
END; { CharToInt }

FUNCTION IntToChar(Digit: INTEGER): CHAR;
(* Convert a Decimal Digit to a Character *)
BEGIN
  IF (0 <= Digit) AND (Digit <= 9) THEN
    IntToChar := CHR(ORD('0') + Digit)
  ELSE
    IntToChar := SPACE;
END; { IntToChar }

FUNCTION IsDigit(Ch: CHAR): BOOLEAN;
(* Test if Ch is a Digit *)
BEGIN
  IF ('0' <= Ch) AND (Ch <= '9') THEN
    IsDigit := TRUE
  ELSE
    IsDigit := FALSE;
END; { IsDigit }
END. { CharLib }

```

Notice that, because the constant `UPPERLowerDIFF` is defined in the `IMPLEMENTATION` part of the `UNIT`, it is private to the library and is not available to its users. Some other functions and procedures should be added to the library, like:

```

FUNCTION IsVowel(Ch: CHAR):BOOLEAN;
FUNCTION IsLetter(Ch: CHAR): BOOLEAN;
FUNCTION IsCapital(Ch: CHAR):BOOLEAN;
FUNCTION Uncapped(Ch: CHAR):CHAR;{ check lower case }
PROCEDURE EnterDigit(VAR D: INTEGER); { input digit }

```

The Interaction of Many Libraries

A large program will probably make use of many libraries, and these could interact. A good organization of libraries makes programs easy to build and to maintain, whereas a unstructured organization is likely to cause problems. To illustrate this, we will consider here a “large” system comprising three libraries and two separately compiled programs. Each library will be rather small, and the procedures will be familiar, but the whole system will help us make many significant points.

Lib2, in Figure 7.43, is a library that consists of two procedures, **Max2** and **Sort2** acting on items of **ItemType**. The **ItemType** is declared to be a **CHAR**, but it could be readily changed to another type such as **INTEGER** or **REAL**. However, once such a change is made then **Lib2** must be recompiled. Also, all modules that use **Lib2** must be recompiled. The order of compilation is important.

Figure 7.43 Library Lib2

```
UNIT Lib2;

INTERFACE
  TYPE ItemType = CHAR;

  PROCEDURE Sort2(  A, B : ItemType; VAR L, S: ItemType);
  PROCEDURE Max2(  A, B : ItemType; VAR C: ItemType);

IMPLEMENTATION

  PROCEDURE Sort2(  A, B : ItemType; VAR L, S: ItemType);
  BEGIN
    L := A;
    S := B;
    IF L < S THEN BEGIN
      L := B;
      S := A;
    END;
  END; { Sort2 }

  PROCEDURE Max2(A, B : ItemType; VAR C: ItemType);
  VAR D: ItemType ;
  BEGIN
    Sort2(A, B, C, D);
  END; { Max2 }
END. { Lib2 }
```

Lib3, in Figure 7.44, is another library that contains a single procedure to sort three items of **ItemType**. Notice that it uses the **ItemType** and **Sort2** both from **Lib2**.

Figure 7.44 Library Lib3

```

UNIT Lib3;

INTERFACE
USES Lib2;

    PROCEDURE Sort3(  X, Y, Z : ItemType;
                      VAR L, M, S: ItemType);

IMPLEMENTATION
    PROCEDURE Sort3(  X, Y, Z : ItemType;
                      VAR L, M, S: ItemType);
    VAR E, F, G: ItemType;
    BEGIN
        Sort2(X, Y, E, F);
        Sort2(F, Z, G, S);
        Sort2(E, G, L, M);
    END; { Sort3 }
END. { Lib3 }

```

Lib4, in Figure 7.45, is yet another library that comprises two procedures: **Max4**, which computes the maximum of four values and **Sort4**, which sorts four values. **Lib4** uses **ItemType** and **Max2** from **Lib2**, and **Sort3** from **Lib3**.

Figure 7.45 Library Lib4

```

UNIT Lib4;

INTERFACE
USES Lib2, Lib3;

    PROCEDURE Max4(  A, B, C, D: ItemType; VAR E : ItemType);
    PROCEDURE Sort4(  A, B, C, D: Itemtype; VAR W, X, Y, Z:
                      ItemType);

IMPLEMENTATION

    PROCEDURE Max4(  A, B, C, D: ItemType; VAR E : ItemType);
    BEGIN
        Max2(A, B, E);
        Max2(E, C, E);
        Max2(E, D, E);
    END; { Max4 }

    PROCEDURE Sort4(  A, B, C, D: Itemtype; VAR W, X, Y, Z:
                      ItemType);
    VAR P, Q, R, S, T: ItemType;
    BEGIN
        Sort3(A, B, C, P, Q, R);

```

```
Sort3(Q, R, D, S, T, Z);
Sort3(P, S, T, W, X, Y);
END; { Sort4 }
END. { Lib4 }
```

Prog5, whose skeleton is shown in Figure 7.46, is a program that sorts any 5 values of `ItemType`. It uses `ItemType` from `Lib2` and `Sort3` from `Lib3`. The details of creating a `Sort5` using `Sort3`s are not shown, and left as an exercise.

Figure 7.46 Program Prog5

```
PROGRAM Prog5;
(* Uses Sort3s for a Sort5 *)

USES Lib2, Lib3;

VAR A, B, C, D, E, F, G, H: ItemType;

BEGIN
  WriteLn('Enter 5 values ');
  { ..... }
END. { Prog5 }
```

Prog7, whose skeleton is shown in Figure 7.47, is a program which computes the maximum of 7 values of `ItemType` using 2 `Max4`s. It uses `ItemType` from `Lib2` and `Max4` from `Lib4`. The details are also not shown here. You do it.

Figure 7.47 Program Prog7

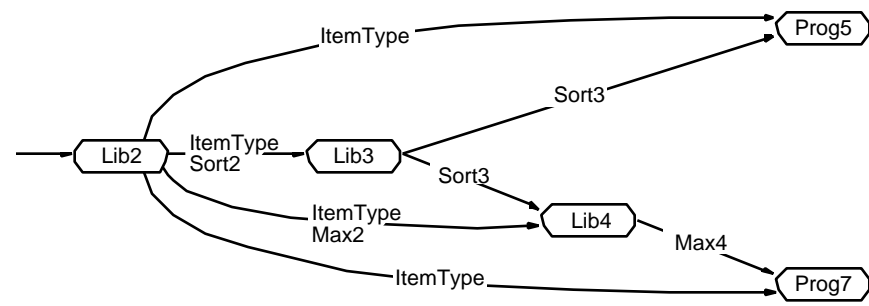
```
PROGRAM Prog7;
(* Uses 2 Max4s for a Max7 *)

USES Lib2, Lib4;

VAR A, B, C, D, E, F, G, H: ItemType;

BEGIN
  WriteLn('Enter 7 values ');
  { ..... }
END. { Prog7 }
```

The interaction amongst these 5 parts (three Libraries and two programs) can be best seen from Figure 7.48, where an arrow from a library to another program or unit indicates that the library is being used. For instance, we can see in the figure that `Prog7` uses `Lib2` and `Lib4`, and also that `Lib4` uses `Lib2` and `Lib3`.

Figure 7.48 The interaction between libraries

The structure of these interactions determines the order in which the units and programs should be compiled. When a change is made to some unit, it must be recompiled as well as every other unit or program that uses it. For example, if we change the declaration of `ItemType` in `Lib2`, then all the units and programs must be recompiled, starting with `Lib2` in the order indicated by the arrows. After `Lib2` is recompiled, `Lib3` must be recompiled. Then either `Lib4` or `Prog5` could be recompiled, but `Prog7` must necessarily be recompiled after `Lib4` has been recompiled.

As another example, if `Lib4` were modified, for example to add a `Min4` procedure, then only `Prog7` would need recompilation (once `Lib4` had been recompiled). Similarly, if `Lib3` were modified to add a `Majority3` procedure, then recompilation would be necessary for `Lib3`, `Lib4`, `Prog5` and `Prog7`.

The Pascal programming system attaches a *time stamp* to any unit at the moment of compilation. During the compilation process, whenever use is made of a unit, a check is made on the time stamps to see if that unit must be recompiled. This would be the case if some other unit that it uses had been recompiled since the unit itself was compiled. If so, an incompatibility exists and is indicated. In this way, the Pascal programming system helps you to maintain consistency. It makes sure that you create your programs based on only the latest versions of units that you use! This error prevention is very important especially when creating very large systems!

7.13 Function and Procedure Types

A *FUNCTION type* (or a *PROCEDURE type*) is a template for a function (or a procedure). Such a type is used as any other Pascal type, in particular to declare variables or parameters. For example, the `SIN` function and the `SQR` function both have the form of a function with one `REAL` input parameter and a `REAL` result. The corresponding template or *FUNCTION type* could be declared as:

```
TYPE RealFunc = FUNCTION(X: REAL): REAL;
```

Similarly, the `Maximum` function we have seen earlier in this chapter, is a binary operator that has the form of a function, with two `INTEGER` input parameters and a single `INTEGER` result. A similar `Minimum` function would have the same form. Their *FUNCTION type* could be declared as:

```
TYPE BinOpType = FUNCTION(X, Y: INTEGER): INTEGER;
```

Suppose that `Maximum` and `Minimum` have been defined as functions with the headers:

```
FUNCTION Maximum(I, J: INTEGER): INTEGER;  
FUNCTION Minimum(I, J: INTEGER): INTEGER;
```

Suppose also that a binary operator `Bop` is declared as:

```
VAR Bop: BinOpType;
```

then we can treat `Bop` just like any other variable and assign values to it; for example:

```
Bop := Maximum;
```

Following this assignment, the execution of a statement like

```
M3 := Bop(Bop(R, S), T);
```

is the equivalent of executing

```
M3 := Maximum(Maximum(R, S), T);
```

However, if the assignment to `Bop` had been

```
Bop := Minimum;
```

executing the same statement

```
M3 := Bop(Bop(R, S), T);
```

would have been the equivalent of executing

```
M3 := Minimum(Minimum(R, S), T);
```

This enables the function `Bop` to be selected as either of the two operators (`Maximum` or `Minimum`). The selection decision could be done during the execution of a program and could be dependent on some input value. This is shown in the given example `FunType` in Figure 7.49.

Figure 7.49 Program FunType

```
PROGRAM FunType;  
(* Shows the use of FUNCTION Types *)  
  
TYPE BopType = FUNCTION(X, Y: INTEGER): INTEGER;  
  
FUNCTION Maximum(A, B: INTEGER): INTEGER;  
BEGIN  
  IF A > B THEN  
    Maximum := A  
  ELSE  
    Maximum := B;  
END; { Maximum }  
  
FUNCTION Minimum(A, B: INTEGER): INTEGER;  
BEGIN  
  IF A < B THEN  
    Minimum := A
```

```

    ELSE
        Minimum := B;
    END; { Minimum }

VAR P, Q, R: INTEGER;
    Bop: BopType;
    Reply: CHAR;

BEGIN
    WriteLn('Enter 3 values');
    ReadLn(P, Q, R);
    Write('Enter M for Max or m for min ');
    Read(Reply);
    IF Reply = 'M' THEN
        Bop := Maximum
    ELSE
        Bop := Minimum;
    R := Bop(Bop(P, Q), R);
    IF Bop = Maximum THEN
        WriteLn('The maximum of the 3 values is ', R:4)
    ELSE
        WriteLn('The minimum of the 3 values is ', R:4);
    END. { FunType }

```

A **PROCEDURE type** is similar to a **FUNCTION type**; it is a template for a procedure value. A procedure with two input **INTEGER** parameters and two output **INTEGER** parameters, like `Sort2`, `Swap2`, or `Divide`, could have a type called `Proc2x2Type` and would have the form:

```

TYPE Proc2x2Type = PROCEDURE(  A, B: INTEGER;
                                VAR X, Y: INTEGER);

```

Although the power of **FUNCTION** and **PROCEDURE** types may not be evident yet, the mechanism should be clear. The most powerful use of such types lies in passing functions and procedures as parameters to other functions or procedures. This will be illustrated in the `Solver` program shown in Figure 7.50.

`Solver`, is a general equation solver that was described in Chapter 6 of the Principles book. Given any two equations of a single variable X , such as $F1(X)$ and $F2(X)$, `Solver` finds values of X that satisfy both equations. `Solve`, is the main procedure which does the solving by using random numbers generated by standard function `Random`.

Figure 7.50 The Solver program

```

PROGRAM Solver;
(* Solves algebraic equations *)
(* by Random trial and error *)

TYPE FunType = FUNCTION(X: REAL): REAL;

FUNCTION F1(X: REAL): REAL;

```

```
BEGIN
  F1 := X * X - 1.0;
END; { F1 }

FUNCTION F2(X: REAL): REAL;
BEGIN
  F2 := -2.0 * X + 3.0;
END; { F2 }

PROCEDURE Solve(Left : FunType; Right: FunType; Low :
                REAL; High : REAL);
CONST Many = 100; { Number of points }
VAR I: INTEGER;
    X, Guess, BestVal, MinErr, Err: REAL;

BEGIN { Solve procedure }
  MinErr := 30000.0;
  FOR I := 1 TO Many DO BEGIN
    Guess := Low + Random * (High - Low);
    X := Guess;
    Err := ABS(Left(X) - Right(X));
    IF MinErr > Err THEN BEGIN
      MinErr := Err;
      BestVal := Guess;
    END;
  END;
  Write(' A value for X is ');
  Write(BestVal:7:4);
  Write(' with an error of ');
  Write(MinErr:7:4);
END; { Solve }

VAR Hi, Lo: REAL;
    I: INTEGER;
BEGIN { Main }
  WriteLn('Enter 2 boundary values ');
  WriteLn('input the smaller first ');
  Read(Lo, Hi);
  FOR I := 1 to 5 DO BEGIN
    Solve(F1, F2, Lo, Hi);
    WriteLn;
  END;
END. { Solver }
```

FunType is declared to be a FUNCTION type with one REAL input parameter and a REAL result. Procedure Solve is declared with four input parameters: two functions of type FunType, and two boundary values of type REAL. In the main body of Solver, the user is prompted for the two boundary values, Lo and Hi. Solve is then called by the statement

```
Solve(F1, F2, Lo, Hi);
```

which passes the two functions F1 and F2 to Left and Right, and the boundary values to Low and High. Solve then tries Many points to determine the value of X for which the difference of the two functions is minimum. In doing this, it calls standard function Random that generates a sequence of pseudo random numbers for use as trial values for the points. Solve is run 5 times over the given range of values and produce the results shown at the top of Figure 7.51. The results can then be used to reduce the range and run the program again to decrease the error, as shown at the bottom of the figure.

Figure 7.51 Typical execution of Solver

```
Enter 2 boundary values
input the smaller first
-10 10
A value for X is -3.0502 with an error of 0.7968
A value for X is 1.2297 with an error of 0.0285
A value for X is -3.2298 with an error of 0.0280
A value for X is -3.2383 with an error of 0.0100
A value for X is 1.2183 with an error of 0.0792

Enter 2 boundary values
input the smaller first
-3.5 -3
A value for X is -3.2350 with an error of 0.0047
A value for X is -3.2387 with an error of 0.0119
A value for X is -3.2360 with an error of 0.0003
A value for X is -3.2351 with an error of 0.0043
A value for X is -3.2354 with an error of 0.0031
```

Since procedure Solve is general enough, we could encapsulate it into a library, say SolveLib. The two functions F1 and F2 would still need to be external to this library and passed to Solve.

Note that we could have avoided to declare a function type in Solver, as the Solve procedure heading could have been written as:

```
PROCEDURE Solve(FUNCTION Left(X: REAL): REAL;
                FUNCTION Right(X: REAL): REAL;
                Low : REAL; High : REAL);
```

This is possible only in the case where functions or procedures are passed as arguments to other functions or procedures. In particular, function types are necessary to write programs like FunType.

7.14 Top-Down Development

Pay Again

In the Principles book a seven-step problem solving method was introduced, that encouraged the top-down design and development of a computer solution. In the coding and testing step of this method, a top-down approach should be used as well. Such an approach involves first coding the main body of the program as a skeleton with procedures and functions represented by *stubs*. A stub is a reduced procedure whose declaration comprises only its heading and an almost empty BEGIN-END pair. At first the stub might have no parameters, no local variables, and its actions might only be the simple output of a message. The stubs simply display the fact that control has reached them and they do nothing more. The trace that is produced by the execution of a program with stubs is a verification of the control flow of the program. The program `MainPay1` is our familiar `Main Pay` algorithm from Chapter 7 of the Principles book (Fig. 7.28) in an early stage of development with all procedures replaced by stubs.

Figure 7.52 Skeleton of program `MainPay`

```
PROGRAM MainPay1;

VAR TotalCount:  INTEGER;
    TotalAmount: REAL;
    IdNumber:    INTEGER;

PROCEDURE NetPay();
PROCEDURE GrossPay();
BEGIN
    WriteLn('STUB: GrossPay ');
END; { GrossPay }

PROCEDURE Deduct();
BEGIN
    WriteLn('STUB: Deduct ');
END; { Deduct }

BEGIN
    WriteLn('STUB: NetPay ');
    GrossPay();
    Deduct();
END; { NetPay }

BEGIN
    TotalAmount := 0.00;
    Write('Enter number of employees ');
```

```

Read(TotalCount);
FOR IdNumber := 1 TO TotalCount DO BEGIN
  NetPay();
  WriteLn;
END;
WriteLn('The total paid out is ',
TotalAmount:7:2);
END. { MainPay1 }

```

Figure 7.53 shows the trace produced by a typical run of MainPay1.

Figure 7.53 Output of MainPay1

```

Enter number of employees 4
STUB: NetPay
STUB: GrossPay
STUB: Deduct

STUB: NetPay
STUB: GrossPay
STUB: Deduct

STUB: NetPay
STUB: GrossPay
STUB: Deduct

STUB: NetPay
STUB: GrossPay
STUB: Deduct

The total paid out is  0.00

```

The program's control flow is verified by examination of the trace produced by the program version with stub procedures. Once this is done, the stubs can be refined by filling in the details: first the parameter lists are added. The output remains the same, but the various procedure interfaces are checked. Then the stubs are filled one by one, and after each addition the program is tested. The program MainPay2 shows the final result of this process.

Figure 7.54 Program MainPay2

```

PROGRAM MainPay2;
(* The Payroll program *)

PROCEDURE NetPay(VAR Accumulated: REAL);
VAR Gross, Net, Deductions: REAL;

PROCEDURE GrossPay(VAR Gross: REAL);
CONST Break    = 40;
      OTimeRate = 1.5;

```

```
VAR Hours, Rate: REAL;
BEGIN
  Write('Enter hours worked ');
  Read(Hours);
  Write('Enter rate of pay ');
  Read(Rate);
  IF Hours < Break THEN
    Gross := Hours * Rate
  ELSE
    Gross := Break * Rate +
      OTimeRate * Rate * (Hours - Break);
END; { GrossPay }

PROCEDURE Deduct( Gross: REAL; VAR Deductions: REAL);
CONST TaxRate = 0.20;
VAR Taxes, Misc: REAL;
BEGIN
  Taxes := Gross * TaxRate;
  Write('Enter misc deductions ');
  Read(Misc);
  Deductions := Taxes + Misc;
END; { Deduct }

BEGIN { NetPay }
  GrossPay(Gross);
  Deduct(Gross, Deductions);
  Net := Gross - Deductions;
  WriteLn('Pay the amount of ', Net:7:2);
  Accumulated := Accumulated + Gross;
END; { NetPay }

VAR TotalCount: INTEGER;
    TotalAmount: REAL;
    IdNumber: INTEGER;
BEGIN
  TotalAmount := 0.00;
  Write('Enter number of employees ');
  Read(TotalCount);
  FOR IdNumber := 1 TO TotalCount DO BEGIN
    NetPay(TotalAmount);
    WriteLn;
  END;
  Write('The total paid out is ');
  Write(TotalAmount:7:2);
END. { MainPay2 }
```

Figure 7.55 shows the output produced by a typical run of MainPay2.

Figure 7.55 Execution of MainPay2

```
Enter number of employees 4
Enter hours worked 38
Enter rate of pay 12.5
Enter misc deductions 10.25
Pay the amount of 369.75

Enter hours worked 42
Enter rate of pay 10.25
Enter misc deductions 17.4
Pay the amount of 335.20

Enter hours worked 10
Enter rate of pay 10.25
Enter misc deductions 8.63
Pay the amount of 73.37

Enter hours worked 51
Enter rate of pay 25
Enter misc deductions 99.21
Pay the amount of 1030.79

The total paid out is 2430.75
```

7.15 Chapter 7 Review

This chapter considered how the Subprogram Form is realized in Pascal as either a `PROCEDURE` or a `FUNCTION`. It also discussed how these may be grouped together with declarations of constants and types to form libraries. The syntax and use of these forms were illustrated with diverse examples. The main consideration of the chapter essentially involved packaging both actions and data into bigger and better building blocks.

Procedures and functions were first considered in their most general form. Many examples were presented to demonstrate various concepts. The three kinds of parameters, input, output and input-output, and the nesting of procedures were treated in detail. Recursion was also treated briefly.

The Pascal `UNIT` form with public `INTERFACE` and private `IMPLEMENTATION` parts was studied. Units were used as libraries of constants, types, procedures and functions. Many examples of libraries were introduced: `ShortSortLib`, `DateLib`, `BitLib` and `CharLib`.

The chapter also considered the way in which libraries can interact, when several of them are being used in the construction of a large program. The importance of the order of compilation of libraries was also stressed.

The concept of `FUNCTION` and `PROCEDURE` types was considered, especially the power of generalization through their use in parameters.

Top-down design and development was revisited, and its approach was applied to coding and testing. The use of stubs for procedures and functions was also briefly illustrated.

7.16 Chapter 7 Problems

1. ReadDate

Create a procedure, `ReadDate(Year, Month, Day)`, that reads in and validates a date value.

2. WriteDate

Create a procedure, `WriteDate(Year, Month, Date)` that outputs a date in a standard form.

3. IsVowel

Create a function, `IsVowel(Ch)`, that returns `True` if character `Ch` is an upper or lower case vowel, and `False` otherwise.

4. Decimal

Create a function, `Decimal(Binary): INTEGER`, that converts any given positive binary number (expressed as a `INTEGER`) into its corresponding decimal form (base 10).

5. Quadrant

Create a procedure, `Quadrant(X, Y, Quad)`, that takes the `REAL` coordinates `X` and `Y` of any point and indicates which quadrant, 1, 2, 3 or 4, the point falls into. If the point falls on any axis the returned value is 0.

6. Resistor

Electrical resistors have a resistance that is specified by three bands of colors on the body of the resistor. The numeric digit corresponding to each color is given below. If the numeric digits for the three colors are `X`, `Y`, and `Z` (in that order) then the value `R` of the resistor (in ohms) is given by

$$R = (10 \times X + Y) \times 10^Z$$

For example, if the bands are Red, Yellow and Orange, the corresponding numeric digits are 2, 4 and 3, so the resistance is:

$$R = (10 \times 2 + 4) \times 10^3 = 24 \times 10^3 = 24,000 \text{ ohms.}$$

Create a procedure `Resistor(Code)` that requests a color (input as a three character sequence, RED, BLU etc.) and returns the corresponding Value as an INTEGER decimal digit. This procedure could be used to determine the resistance.

Resistor

Color	Code
Black	0
Brown	1
Red	2
Orange	3
Yellow	4
Green	5
Blue	6
Violet	7
Gray	8
White	9

7. Next Month

Create a procedure `NextMonth(Month)` to convert any given Month into its following month. Do this for both subrange and enumerated definitions of `MonthType`.

8. Round Off

Create both a function `Rounded(RealNum)` and a procedure `Roundoff(RealNum)` that round off any REAL to its closest integer value. If the fraction part is 0.5, then the value is rounded to an even number. For example, 12.5 is rounded down to 12 and 13.5 is rounded up to 14. Why?

9. Cubes

Create a function `Cubed(R)`, and two procedures `Cube(R)` and `CubeOf(R, S)` for finding the cube of any REAL value R.

10. Binary Conversion

Create a procedure `Bin(Dec)` that accepts an INTEGER Dec, and outputs the binary equivalent as a sequence of 0 and 1 character digits.

11. Romanum

Create a procedure `Roman(Num)` to accept any positive `INTEGER Num` (less than 300), and output the corresponding Roman number. Assume that four consecutive occurrences of a symbol are proper.

12. Cashin

Suppose that the input to a program (such as `CHANGE`) must have exactly the form

`$D.DD`

where each `D` is one of the ten decimal digits.

For example, valid sequences are

`$5.00`, `$1.75` and `$0.25`

and invalid sequences are

`$.25`, `$0.255`, `$3`, and `$12.34`.

Create a procedure `Cashin(Cents)` that inputs a sequence of characters (assumed to be in the valid form), and converts it to the corresponding `INTEGER` number of `Cents`.

13. Tell Time

Create a procedure `Time(MilTime)` to accept the Military time `MilTime` as an `INTEGER`, and output the time in the appropriate one of the four formats (H O'clock, Half past H, M minutes after H, and M minutes before H) where H and M are the Hours and Minutes.

14. Break

Create a procedure `Break(N, LSD, MSD, R)` that takes a given `INTEGER N` and “breaks off” its least significant digit `LSD`, its most significant digit `MSD`, and returns also the remaining integer `R` (or a negative 1 if nothing remains).

7.17 Chapter 7 Programming Problems

Random Projects;

The following projects require random numbers which may be created in many ways: using the function `Random` as we did in the program `Solve` in this chapter.

1. Simulate
Simulate the dice game called `Dice`:¹

First, two dice are thrown.
 If their sum is 7 or 11, you win,
 and if the sum is 2, 3 or 12, you lose,
 otherwise remember the sum, the “point count”,
 and keep throwing until either:
 the point count comes up and you win or
 a 7 comes up and you lose.

For example: consider the sequences

7	you win
6, 7	you lose
4, 2, 11, 3, 4	you win
9, 3, 4, 12, 2, 8, 3, 7	you lose

2. Evaluate

Modify the above program to play any number of games, say 100, and find the probability of winning (the ratio of wins to number of times played).

3. Observe

The game of dice poker involves the throwing of five dice, their values forming a “hand”, and the evaluation of the hands in the following order:

- a. *Five of a kind*, means all five dice have the same value,
- b. *Four of a kind*, means four dice have the same value,
- c. *Full house*, means that three dice have one value in common and the other two have another value in common, in other words: three of a kind and a pair.
- d. *A Straight*, means the five values are in consecutive order,
- e. *Three of a kind*, means that three dice have one value in common,
- f. *Two pairs*, means two dice have one value in common and two other dice have another value in common,
- g. *One pair*, means that only two dice have the same value,
- h. *No pairs*, means none of the above.

Create a program to evaluate the hands in Dice Poker, as described above. Then generate many such hands randomly and evaluate them, ultimately printing out the number of each type of hand.

4. Analyze

There are many algorithms for computing grades. Grades are allocated according to the following schedule:

Grades:
A score of 90 to 100 gets a grade of 'A'
A score of 80 to 89 gets a grade of 'B'
A score of 60 to 79 gets a grade of 'C'
A score of 50 to 59 gets a grade of 'D'

A score of less than 50 gets a grade 'F'

The following four algorithms² for solving this problem are examples of the several possibilities.

Method 1:

```
Input Percent
If Percent ≥ 90
    Output 'A'
Else
    If Percent ≥ 80
        Output 'B'
    Else
        If Percent ≥ 60
            Output 'C'
        Else
            If Percent ≥ 50
                Output 'D'
            Else
                Output 'F'
```

This method first tests the largest percentage range and keeps testing the ranges in decreasing order, until the proper range is found and the corresponding grade is output.

Method 2:

```
Input Percent
If Percent < 50
    Output 'F'
Else
    If Percent < 60
        Output 'D'
    Else
        If Percent < 80
            Output 'C'
        Else
            If Percent < 90
                Output 'B'
            Else
                Output 'A'
```

This one similar to Method 1 but starts from the smallest percentage range.

Method 3:

```
Input Percent
Set TestValue to Percent - 50
If TestValue < 0
    Output 'F'
Else
    Set TestValue to TestValue - 10
    If TestValue < 0
        Output 'D'
```



```

Else
  Set TestValue to TestValue - 20
  If TestValue < 0
    Output 'C'
  Else
    Set TestValue to TestValue - 30
    If TestValue < 0
      Output 'B'
    Else
      Output 'A'

```

This method makes the same test ($\text{TestValue} < 0$) at each stage. This is useful for machine level programming since machines can compare values to zero very easily.

Method 4:

```

Input Percent
Set Grade to 'A'
If Percent < 90
  Set Grade to 'B'
If Percent < 80
  Set Grade to 'C'
If Percent < 60
  Set Grade to 'D'
If Percent < 50
  Set Grade to 'F'
Output Grade

```

involves a series of choices, unlike all the above methods which involve nested choices. This method is often simpler to program in older languages which have a limited IF structure. Notice that this method requires the assignment of characters; none of the other methods uses assignment of characters.

Each grade traces a path through the algorithm and involves a number of comparisons. A given grade distribution encounters a fixed average number of comparisons. Compare this average for the various algorithms if the grade distribution is assumed to be: 15% A's, 20% B's, 40% C's, 15% D's and 10% F's.

DateLib

A miniature version of `DateLib`, a library that manipulates dates (year, month, day) was given in this chapter. Its interface part showed that it had three actions:

```

INTERFACE
  FUNCTION IsLeap(Year: INTEGER): BOOLEAN;
  FUNCTION DaysInMonth(Year, Month: INTEGER)
    : INTEGER;
  FUNCTION DayOfYear(Year, Month, Day: INTEGER)
    : INTEGER;

```

Build your own copy of this UNIT and test it with the program `DaysToXmas`. Then extend this UNIT with some of the following actions. Create also a program that tests those that you implement.

1. `Elapsed` calculates the number of days between any two given dates.
2. `Age` determines the integer age of a person, given the birth date.
3. `FirstDate` determines the weekday of January 1 of any year, knowing that January 1 of 1901 was a Tuesday.
4. `WeekDay` finds the weekday that a given date, including the year, falls on. It makes use of `FirstDate`.
5. `WeekDayBorn` finds the weekday a person was born, given the birth date.
6. `ThanksDate` finds the date of Thanksgiving in the USA, for any year, knowing that it occurs on the fourth Thursday of November.
7. `MidWay` determines the date (or dates) half way between two given dates of the same year.
8. `UnLucky` determines, for a given year, the months on which Friday falls on the 13th day.
9. `Calendar` creates a calendar for any month of any year.
10. `BioRhythm` plots out a sine curve supposedly indicating the rise and fall of a certain ability, such as physical strength, endurance. The sine wave has a given period of repetition (say 23 days) starting with 0 at birth. Create such a plot starting at any given day, and continuing for one whole cycle.
11. `BigBio` plots a number of biorhythms, each having a different period, on one graph. For example, the intellectual cycle has a period of 33 days, and the sensitivity cycle has a period of 28 days.

Create Libraries

There are a number of smaller libraries that could be useful. Create some of these and use them.

1. `ConstLib`
If there are a number of physical constants that you often use, e.g., π , e , the mass of an electron, and many others, put them into a Library.
2. `ConverLib`
Conversion of quantities from one measurement system to another could be conveniently done with a Library. The names of the actions should be easy to remember, e.g. `PoundsToKilograms`, `FeetToInches`, etc. Would you do this with functions or procedures?
3. `InOutLib`
The standard procedures of Pascal do not satisfy all input/output needs. Create your own input/output Library with procedures such as:

`WriteIntLen(I)`, which computes the width of `I` and prints that width.

`ReadBool(B)` which reads `BOOLEAN` values `TRUE` and `FALSE`.

`ReadDay(D)` which inputs the day of the week.

`WriteDollars(D)` which prints dollar amounts such as \$1,234.56.

4. **MiscLib**

If you are familiar with other languages you may wish to incorporate some of these features into Pascal. You may also wish to use other names for some of the functions of Pascal (such as `RealToInt` for `entier`).

5. **ArithLib**

Create a library of arithmetic “black boxes”, `DivInt`, `SubInt`, `MulInt`, `Square`, etc.

6. **CharLib**

Complete the `CharLib` library that was given in this chapter.

7. **BitFunctionLib**

Redo `BitLib` using functions instead of procedures for the logical gate operations. For example: DeMorgan's result would have the form:

`Not (And(A, B)) = Or(Not(A), Not(B))`

8. **FileLib**

Create a Library involving Files with actions of:

`OpenSourceFile`, which prompts, checks and opens

`EncryptFile`, which encodes a file for privacy

`DecryptFile`, which reconverts an encrypted file to its original state

FinanceLib

Create a library of the following five financial functions. They may be used to solve either savings problems or loan problems. For savings problems the payment `Payt` is positive; for loan problems it is a negative value. This Library could be used with a program where any four values are given and the fifth value is to be determined.

Fvalue(Rate, Durn, Payt, Pval)

is the future value at a compounded interest Rate, of Durn payment periods, of constant amount Payt, and present value Pval.

Pvalue(Rate, Durn, Payt, Fval)

is the present value needed to reach a given future value Fval with a given interest Rate in Durn periods of amount Payt.

Duration(Rate, Payt, Pval, Fval)

is the number of payments necessary to reach a given future value Fval given the present value Pval with periodic amount of Payt at a compounded Rate per period.

Payment(Rate, Durn, Pval, Fval)

is the periodic payment amount needed to reach the future value Fval from the present value Pval in Durn payments at a given interest Rate.

IntRate(Durn, Payt, Pval, Fval)

is the interest rate per period that is required to reach the future value Fval from the present value Pval in Durn payments of amount Pay.

Another parameter Init could be used as an initial guess for the rate.

Change Again: Done Properly with Procedures

ChangeMaker is to be done properly using subprograms (functions or procedures) with proper passing of parameters and maximum hiding of procedures. Some procedure names follow; you are to specify the parameters and their passing methods. Show the structure of this program by data flow diagrams and contour diagrams.

Instruct

asks if you wish instructions or help, and if so, it provides a brief statement.

InCost

a friendly procedure, prompts for the Cost and checks for proper input.

InTend

like InCost, is a procedure to input the amount tendered; it uses InReal.

InReal

enters a value in decimal form (dollars and cents) and converts it to INTEGER number of cents.

EnterInput

contains the previous procedures (and possibly a Proper Value check).

Changer

makes the change, and uses a Divide procedure.

SpellOut

Spells the count corresponding to its input value.

Plural

appends the character S to any written plural denomination

Size is a function, useful for formatting the output, which determines the number of digits of a Cardinal.

MeanLib

Create the IMPLEMENTATION part of a library corresponding to the following INTERFACE part. This UNIT consists of a number of procedures that compute various mean values of a sequence of real values. The first and last items of the sequence are terminators and are not to be included in the mean values.

```
UNIT MeanLib;
```

```
(* Provides various mean values *)

INTERFACE

  PROCEDURE ArithMean(VAR Count: INTEGER;
                      VAR AMean: REAL);
  (* Computes arithmetic mean by summing values *)
  (* and dividing by the number of values *)

  PROCEDURE GeoMean(VAR Count: INTEGER;
                    VAR GMean :REAL);
  (* Computes the geometric mean by multiplying *)
  (* the N values and then taking the Nth root *)

  PROCEDURE HarmMean(VAR Count: INTEGER;
                     VAR HMean : REAL);
  (* Computes the harmonic mean by summing the *)
  (* reciprocals of all the values and then *)
  (* taking the reciprocal of the result *)

END.
```

7.18 Chapter 7 Programming Projects

DMT: DeMilitarizeTime Lab with Procedures

The goal of this project is to create various procedures that can be used to manipulate Military time. A program `MinDiff` uses these procedures to find the minimum time difference between any three given times in one day.

DATA

Time can be expressed in Military form (or 24-hour form) as an `INTEGER` such as 730 or 1604. Time can also be represented in a Normal form as 7:30 am or 4:04 pm. Improper Military times are 2604 and 1670.

ACTIONS

Create all of the following procedures using exactly the parameter names given, with parameters passed properly.

Divide(Num, Den, Quot, Rem)

divides Num by Den to yield Quot and Rem.

Use `DIV` and `MOD` to implement Divide, just to be different this time.

ReadMilTime(MilTime)

prompts for a single military time `MilTime` and checks that it is in the proper form. If it is not in the proper form it outputs an error message and requests another time to be entered; this continues until the time entered is in the proper form.

At first just check for the proper range of time (0...2400); later return to refine this. Use the above procedure `Divide`.

ReadMilTime3(First, Second, Third)

calls the `ReadMilTime` three times to enter the three values.

WriteNormalTime(MilTime)

writes out a given military time in the normal mode. For example, 0730 is written as "7:30 am" and 1604 is written as "4:04 pm".

Convert(MilTime, Minutes)

converts a given military time to the corresponding minutes since midnight. For example, the time 1240 is 760 minutes past midnight.

Used `Divide` again.

TimeDiff(T1, T2, Diff)

Computes the minutes elapsed between the two given times `T1` and `T2` of one day. For example, the difference between 0250 and 2030 is 1060 minutes.

Minimize3(A, B, C, M)

computes the minimum value `M` of three values `A`, `B`, `C`.

The program *MinDiff*

This program uses these procedures to find the minimum time difference of three times. For example, when the input values are 1200, 1300 and 1800, the minimum time between them is 60 minutes. Use the following program (grown in stages).

```

    WriteLn('Input three values ');
    ReadMilTime3(X, Y, Z);
    TimeDiff(X, Y, P);
    TimeDiff(X, Z, Q);
    TimeDiff(Y, Z, R);
    Write('The minimum time difference between ');
    WriteNormalTime(X);
    WriteNormalTime(Y);
    WriteNormalTime(Z);
    Write(' is');
    Minimize3(P, Q, R, S);
    WriteLn(S: 3);

```

Time Permitting:

Indicate which two times have this minimum difference (the above 60 minute difference is between the first two values, 1200 and 1300).

Finally: Put these procedures in a `UNIT MilTimeLib` and `USE` them for the `MinDiff` program.

SLL: Small Library Project

In this project we will be creating Libraries in Pascal and we will be using these Libraries in some programs.

1. Create a small Library, called `CharLib`, to work with the `CHAR` type. This Library should contain some:

Constants such as

Hyphen, Space, Bell, Buzz, Escape,

Procedures, with verb names such as

Capitalize(CH), EnterDigit(D)

Functions, with noun names such as

MaximumC3(A, B, C), Uncap(C), CharToCard(C),

Boolean Functions, with adjective names such as

Capitalized(C) or equivalently IsCap(C)

Earlier in this chapter, we described the beginnings of this library together with a small example.

2. Create a small program, called `CharTest`, which uses and tests the new `CharLib` that you have just constructed. This program need not be useful in any other way.

3. `RunCharTest` and show some runs of this test program.

Time Permitting:

4. Create your own small Library of things of interest to you, and a program to use it. For example, you may wish to “encapsulate” the previous project involving 24-hour Time conversion.
5. ReUse: Show and Tell
Share your Library with some other students in the class. Have them test yours and you test theirs.

¹ This is figure 3.3 from Principles.

² These algorithms come from Principles Chapter 5 around page 26

Chapter 8 Pascal Data Structures

In this chapter we consider the three structured data types: arrays, records, and sets, and how they are used in Pascal. The main concepts introduced in Chapter 8 of the Principles book and “Abstract Data Type” (or ADT), will be further developed. ADTs will also be used to create three libraries: IntArrayLib, ComplexLib, and a Matrix Library

Chapter Overview

8.1	Preview.....	314
8.2	Arrays in Pascal.....	314
	ChangeMaker and Variance.....	317
	Parallel Arrays: Part Inventory.....	319
	A Tiny Data Base: Retrieving Strings from Arrays.....	321
	Arrays as Parameters.....	323
	IntArrayLib: Integer Array Library.....	325
8.3	Two Dimensional Arrays in Pascal.....	328
	Arrays of Arrays—Two Dimensional Arrays.....	328
8.4	N-Dimensional Arrays.....	331
	More Dimensions.....	331
8.5	Records in Pascal.....	333
	ComplexLib: A Library for Complex Numbers.....	337
	Records of Records: Nested Records.....	339
	Arrays of Records in Pascal.....	342
	Records of Arrays.....	345
	Matrix Library.....	347
8.6	Sets in Pascal.....	352
	More Sets (Optional).....	355
8.7	Dynamic Variables and Pointers in Pascal.....	359
8.8	Chapter 8 Review.....	361
8.9	Chapter 8 Problems.....	361
8.10	Chapter 8 Programming Projects.....	363
	Gas Project.....	363
	Library Projects.....	364
	BSL: Big Stat Lab.....	364

8.1 Preview

The main concepts developed in Chapter 8 of the Principles book will be discussed here in the context of Pascal and used to create programs. Arrays will be considered in detail, and their method of use will be illustrated through a variety of examples.

The concept of an “Abstract Data Type” will be developed further and applied through the use of libraries. Libraries created in this chapter include a Integer Array Library, a Complex Number Library, and a Matrix Library.

The main capabilities developed in this chapter involve the use of data structures, arrays, records and sets, to create larger data items from smaller ones. Arrays and records came into programming languages from different areas of application; arrays from scientific applications and records from commercial applications. Here we shall see that the two techniques can be combined with powerful results. Sets are still not as widely used as the other structures.

Finally, by this time, you should be able to read programs with some ease, so less of the text is devoted to describing the programs given. If you encounter problems, go back to the algorithm development in Chapter 8 of the Principles book.

8.2 Arrays in Pascal

An array is an ordered collection of values, all of the same data type. The simplest kind of array is an ordered sequence of `INTEGERS`, such as the number of hours worked on each of the seven days of a week. More complex arrays, of two and more dimensions, will be considered later in this chapter.

The declaration of an array type in Pascal has the general form:

```
TYPE array-name = ARRAY index-type OF item-type;
```

such as

```
TYPE WeekArray = ARRAY [1..7] OF REAL;
```

where `item-type` represents the type of the elements stored in the array, and can be any valid Pascal data type, either built-in or programmer defined. The second type, `index-type`, represents the type used to index the individual elements in the array, and is usually an enumeration or subrange type, but may be a Boolean or Character type. The `index-type` cannot be `REAL`.

Once an array type has been declared, it may be used in the declaration of variables in the usual way, as for example:

```
VAR Hours, OverTime, TotalHours: WeekArray;
```

Such an array `TYPE` declaration describes a general template for a typical array of that type, showing the number of values and their type. On the other hand the `VAR` declaration specifies a particular instance of that type. It would

be valid but less general, and therefore not preferred, to combine the two declarations in one as in:

```
VAR Hours: ARRAY [1..7] OF REAL;
```

This should be avoided, as it might create type incompatibilities later on, and prevent you to use the array as a procedure or function argument.

In fact it would be even better to be more general and declare yet another type

```
TYPE WeekRange = 1..7;
```

and use that as the index-type in the declaration of the array type WeekArray:

```
TYPE WeekArray = ARRAY [WeekRange] OF REAL;
```

for this new type WeekRange could also be used to declare an index variable used in the program to access values in the array. Consistent changes to the program can then be made in a single place by changing the declaration of WeekRange. A further generalization can be obtained by declaring the upper and lower values of the index-type as named constants.

The following examples of array declarations show something of the diversity available in Pascal for the definition of arrays.

```
CONST
    MOST = 100;
TYPE
    Range = 0..MOST;
    IntArray = ARRAY [Range] OF INTEGER;
VAR
    Age, Grade: IntArray;

CONST
    LOW = 100;
    HIGH = 200;
TYPE
    IdRange = LOW..HIGH;
    RealList = ARRAY [IdRange] OF REAL;
VAR
    Price, Quantity: RealList;

TYPE
    WeekDay = (Sun, Mon, Tue, Wed, Thur, Fri, Sat);
    WeekType = ARRAY [WeekDay] OF REAL;
VAR
    Hours, OverTime: WeekType;

TYPE
    Countdown = ARRAY [-10..10] OF REAL;
    NameString = ARRAY [0..20] OF CHAR;
    BitSequence = ARRAY [0..15] OF 0..1;
```

The individual elements of an array are accessed by using the array name followed by an index value enclosed in square brackets, such as

```
Grade[95]
Price[Part]
Hours[Day]
Name[IdNumber]
```

The index value inside the brackets may be a constant, a variable, or an expression, which is evaluated at the time of access. This index value represents the position of the element in the array. Each of the above references to an individual element in an array is treated as a variable that can be assigned, compared, input, output, and used in expressions. It is sometimes called a *subscripted variable*.

The short program in Figure 8.1 illustrates the use of an array to store a sequence of INTEGERS representing temperatures. The list of input values is sandwiched between sentinel values as we have done in the past. However, to keep things simple, the program does not prompt the user: you can easily add these to improve the program.

Figure 8.1 Program FindMaximumTemperature

```
PROGRAM FindMaximumTemperature;
(* Find the maximum of a sequence of temperatures *)

CONST MaxSeqSize = 10;

TYPE Sequence = ARRAY [1..MaxSeqSize] OF INTEGER;

VAR Vector: Sequence;
    Maximum, Sentinel, Value, Count,
    Position, Index: INTEGER;

BEGIN
    Read(Sentinel);
    Count := 0;
    Read(Value);
    WHILE Value <> Sentinel DO BEGIN
        Inc(Count);
        Vector[Count] := Value;
        Read(Value);
    END;
    Maximum := Vector[1];
    Position := 1;
    FOR Index := 2 TO Count DO BEGIN
        Value := Vector[Index];
        IF Maximum < Value THEN BEGIN
            Maximum := Value;
            Position := Index;
        END;
    END;
END;
```

```

    Write('Maximum temperature was ', Maximum:3);
    WriteLn(' in position ', Position:2);
END. { FindMaximumTemperature }

```

Notice that, even though the actual number of values that will be read in is determined by the sentinel that marks the end of the data, this number may not exceed the value of `MaxSeqSize`, in this case 10. If an attempt is made to input more than 10 numbers, the Pascal programming system will stop the program with an “index/range error” message indicating that an attempt has been made to read or write a value outside the bounds of the array. In other words, the size of an array is fixed when the program is compiled and cannot be changed dynamically. In order to change the size, the program must be modified and recompiled; this emphasizes the advantages of using named constants for bounds as it reduces the number of places where changes have to be made to a program.

ChangeMaker and Variance

In Chapter 8 of the Principles book, we developed yet another version of the Change Maker algorithm, one that used an array to loop through denominations in the order 25, 10, 5, and 1. For each denomination, the algorithm decreases the change due and outputs the denomination value, until the change due is down to zero.

Figure 8.2 Program ChangeWithArrays

```

PROGRAM ChangeWithArrays;
(* Making change using arrays *)

CONST NumOfDenominations = 5;

TYPE Range = 1..NumOfDenominations;
    IntSeq = ARRAY[Range] OF INTEGER;

VAR Cost, Tendered, Remainder, Denomination,
    Index: INTEGER;
    Den: IntSeq;
BEGIN
    { Set up Denominations array }
    Den[1] := 1; { Pennies }
    Den[2] := 5; { Nickels }
    Den[3] := 10; { Dimes }
    Den[4] := 25; { Quarters }
    Den[5] := 50; { Half-dollars }

    { Get input values and compute the change }
    WriteLn('Enter the cost in cents ');
    Read(Cost);
    WriteLn('Enter the amount tendered in cents ');

```

```
Read(Tendered);
Remainder := Tendered - Cost;
WriteLn;
WriteLn('Your change is ');

{ Compute the change }
FOR Index := NumOfDenominations DOWNTO 1 DO
BEGIN
    Denomination := Den[Index];
    WHILE Remainder >= Denomination DO BEGIN
        WriteLn(Denomination:4);
        Dec(Remainder, Denomination);
    END;
END;
END. { ChangeWithArrays }
```

The program `ChangeWithArrays`, in Figure 8.2, is based on this algorithm. Notice the choice of names in the program. Also notice that the values in the array were in decreasing order in the original algorithm, and are in increasing order in the program. This enables the program to grow more easily to higher denominations such as 100, 200, 500, etc.

In Chapter 8 of the Principles book, two methods of calculating variance were shown. The second method computed the sum of all the values, as well as the sum of their squares, before applying a simple formula.

Figure 8.3 shows a Pascal program that is based on this algorithm.

Figure 8.3 Program Variance

```
PROGRAM Variance;
(* Compute the mean and variance of a sequence of values *)

CONST MaxListSize = 10;

TYPE IntSeq = ARRAY[1..MaxListSize] OF INTEGER;

VAR Vector: IntSeq;
    Sum, SumSquares, Sentinel,
    Value, Count, Index: INTEGER;
    Mean, Variance: REAL;

BEGIN
    Read(Sentinel);
    Count := 0;
    Read(Value);
    WHILE Value <> Sentinel DO BEGIN
        Inc(Count);
        Vector[Count] := Value;
        Read(Value);
    END;
```

```

Sum := 0;
SumSquares := 0;
FOR Index := 1 TO Count DO BEGIN
    Sum := Sum + Vector[Index];
    SumSquares := SumSquares +
        Vector[Index] *
Vector[Index];
END;
Mean := Sum / Count;
Variance := SumSquares / Count - Mean * Mean;
Write('Mean = ', Mean:5:2);
WriteLn(' Variance = ', Variance:5:2);
END. { Variance }

```

Notice the way that references to elements of `Vector`, such as `Vector[Index]`, form part of expressions in just the same way as variables do.

Parallel Arrays: Part Inventory

The term *parallel arrays* refers to a number of arrays that are of the same size but may contain different types of value. For example, the inventory of chair parts, that was mentioned in the Principles book, could be represented by the following four parallel arrays, as each attribute (Price, Quantity, etc.) is of a different type. The first element of each array corresponds to the same part, a Leg, the second element of each of the parallel arrays corresponds to a Seat, etc. as shown in Figure 8.4.

Figure 8.4 Parallel arrays

	I	Price[I]	Quant[I]	ReOrder[I]	Status[I]
Leg	1	1.12	200	FALSE	A
Seat	2	2.50	100	TRUE	C
Rung	3	0.75	400	FALSE	B
Back	4	3.00	300	FALSE	A

The program `PartArray`, shown in Figure 8.5, illustrates how the parts inventory of a chair could be represented using parallel arrays. Each of the arrays has a range that extends over the list of parts forming the chairs; this listing of parts is declared as an enumerated type, which is then used as the range in the declaration of the arrays. Four types of array are declared, one for each type of item (`IntArray` for INTEGERS, `RealArray` for REALS, etc.) As usual, these types are templates.

Figure 8.5 Program `PartArray`

```

PROGRAM PartArray;
(* An inventory of parts with parallel arrays *)

TYPE PartsType = (Leg, Seat, Rung, Back);

```

```
    IntArray = ARRAY [PartsType] OF INTEGER;
    RealArray = ARRAY [PartsType] OF REAL;
    BoolArray = ARRAY [PartsType] OF BOOLEAN ;
    CharArray = ARRAY [PartsType] OF CHAR;
VAR Price: RealArray;
    Quantity: IntArray;
    Reorder: BoolArray;
    Status: CharArray;
    Worth: REAL;
    Part : PartsType;

BEGIN
    WriteLn('Enter price and quantity ');
    WriteLn('for leg, seat, rung, back');
    Read(Price[Leg ]); Read(Quantity[Leg ]);
    Read(Price[Seat]); Read(Quantity[Seat]);
    Read(Price[Rung]); Read(Quantity[Rung]);
    Read(Price[Back]); Read(Quantity[Back]);
    Worth := 0;
    FOR Part := Leg TO Back DO
        Worth := Worth +
            Price[Part] * Quantity[Part];
    Write('The total worth is ');
    Write(Worth:8:2);
END. { PartArray }
```

The Price, Quantity, Reorder and Status arrays are all declared of different types. Actually, Reorder and Status are not used in the program for simplicity reasons, and can thus be ignored.

First, the Price and Quantity of the four parts are input, and then the total Worth of the parts is computed. This is done by multiplying the Price of each item by its Quantity, and summing these products. For example, if there are 200 legs with a price of \$1.12 each then the worth of legs is $20 \times 1.12 = \$22.40$. For 100 seats at a price of \$2.50 the worth is \$250.00. If we use the values shown in the diagram of Figure 8.4, we get a total Worth of \$1,674.

The output from a typical run of the PartArray program is:

```
Enter price and quantity
for leg, seat, rung, back
1.12 200
2.5 100
0.75 400
3.0 300
The total worth is 1674.00
```

This program could be extended to use the other arrays, Reorder and Status. There are many ways to use these arrays, for example, to decide whether to reorder Legs:

```
IF (Quantity[Leg] < Critical) AND
    (NOT ReOrder[Leg]) THEN
```



```
{ Order Legs }
```

The same Chair inventory problem will be solved another way with records later in this chapter.

A Tiny Data Base: Retrieving Strings from Arrays

Recall the tiny data base you have used in Chapter 2, as we'll look here into the way one of its operations can be implemented. Data of type `STRING` are particularly useful for the storage of text information so it should not be surprising that arrays of `STRING`s would be a common way for storing lists of information. For example, the following is a list of people and their phone numbers.

```
Cetera, Ed,      (818) 885-3398
DeLion, Dan     (405) 349-6400
Dover, Ben      (213) 987-6543
Druff, Dan      (818) 213-4567
Funt, Ella      (818) 349-2134 X5678
Gone, Polly     (818) 548-5948 AM
Gone, Polly     (818) 439-4393 PM
Ho, Gung        (818) 543-7652
Stein, Frank N. (213) 456-7890
Wood, Holly     (818) 349-6417 by 9am
```

If this list were stored in an array, it could be searched to find the entry containing the pattern `Funt` and it would retrieve:

```
Funt, Ella      (818) 349-2134 X5678
```

Similarly, it can be searched for all occurrences of the pattern `(213)` to find all those having the area code `(213)`. This search would yield the two entries:

```
Dover, Ben      (213) 987-6543
Stein, Frank N. (213) 456-7890
```

Notice that searching for the pattern `213` would have yielded two more strings:

```
Druff, Dan      (818) 213-4567
Funt, Ella      (818) 349-2134 X5678
```

The program `Retrieve` shown in Figure 8.6, was created to make such searches. The strings that form the information list, `Info`, in this case, the phone list, are read from a file that is referred to in the program as `DataFile` but whose actual name is entered by the user before making the search. The list `Info` is declared to be of type `InfoList`, which was defined to be a string array of some fixed maximum size, `MaxSize`.

Figure 8.6 Program Retrieve

```
PROGRAM Retrieve;
(* Retrieve a pattern from an array of strings *)

CONST MaxSize = 25;
```

```
TYPE ListRange = 1..MaxSize;
    InfoList = ARRAY [ListRange] OF STRING;

VAR Pattern, FileName: STRING;
    DataFile: TEXT;
    Info: InfoList;
    Index, Size: ListRange;
BEGIN
    { Get File of strings & Put in array }
    Write('Enter the file name ');
    ReadLn(FileName);
    Assign(DataFile, FileName);
    Reset(DataFile);
    Index := 1;
    ReadLn(DataFile, Info[Index]);
    WHILE Info[Index] <> 'END' DO BEGIN
        Inc(Index);
        ReadLn(DataFile, Info[Index]);
    END;
    Size := Index - 1;

    { Find occurrences of a pattern }
    Write('Enter Search pattern: ');
    Read(Pattern);
    FOR Index := 1 TO Size DO
        IF Pos(Pattern, Info[Index]) <> 0 THEN
            WriteLn(Info[Index]);
    WriteLn('That''s all ', #7);
    Close(DataFile);
END. { Retrieve }
```

During the early development of the program, once the strings have been put into the array, they can be output to verify that the input worked properly. The program then continues by requesting a `Pattern` to search for in `Info`. The search process consists of applying the standard function `Pos` to each item of the `Info` array to see whether that `Pattern` is in that item and, if so, the item is output.

When the search is completed, there is a final line of output:

```
WriteLn('That''s all', #7);
```

to tell the user that the search is over. There are two points to notice about this statement. First, the doubled quote mark in `'That''s all'` shows how a single quote mark can be included in a character string; when output it appears as `That's all`. The `#7` specifies that a character with ASCII value 7 is to be output; this character, called “BELL” causes the terminal to make a sound—in the early days of teletypes, it was an actual bell that sounded, nowadays it’s some kind of electronic sound. It is useful to the user, who is perhaps waiting for

a long file to be searched, to be possibly woken up and told that the search is over. The `Close` statement is required to close the input file.

There are other possible applications for this simple data retrieval program, such as getting times from a Schedule file, tasks from a TO-DO file, or items from an Inventory file, etc.

This program could also be modified in many different manners. It could be made to search for fields (separated by commas) within each record, to output only the phone number field, or only the address field. It could be changed to print names and addresses, but not the phone numbers, of certain zip codes in the form of labels for a mailing list. It could be used to count the numbers of items having various properties, an address in California for example, and then analyze these statistics.

Arrays as Parameters

In principle, the use of an array as a parameter to a procedure should not be different from the use of other variables. There are two main methods of passing parameters: pass by value, used for passing input parameters and pass by reference, used for output and input-output parameters, and these methods are both used for arrays. However, the choice of which one to use is sometimes influenced by efficiency considerations in the case of arrays.

When a parameter is passed by value, the value of the actual parameter is copied into the formal parameter, which is a local variable. Thus, passing an array parameter by value results in making a copy of the entire array, which takes both time (to make the copy) and space (to store the copy). If the array is large, or the computer memory space is small, then the array should be passed by reference, even if it is a passed-in parameter. In this case, the programmer must make certain that the called procedure does not modify the array, because it is not protected as it would have been if passed by value.

The program `Extrema`, shown in Figure 8.7, contains two general procedures, `InSeq` and `Extremes`, both of which may be useful in other programs. The main body of the program calls `InSeq` to read a list of ages into an array and to count the number of ages. The other procedure, `Extremes` is then invoked to find the oldest and youngest. Finally, the values of `Count`, `Oldest` and `Youngest` are output. The array `Age`, in which the ages are stored, is of type `Sequence`, which is defined as having at most 100 `INTEGER`s.

Figure 8.7 Program Extrema

```
PROGRAM Extrema;  
(* Find the extreme values in an array *)  
  
CONST Most = 100;  
  
TYPE Sequence = ARRAY [1..Most] OF INTEGER;  
  
VAR Age: Sequence;  
    Oldest, Youngest, Count: INTEGER;
```

```
PROCEDURE InSeq(VAR Info: Sequence; (* output *)
                VAR Size: INTEGER); (* output *)
VAR Index, value, term: INTEGER;
BEGIN
    Write('Enter terminal value ');
    Read(term);
    WriteLn('Now enter the values ');
    Read(value);
    Index := 0;
    WHILE value <> term DO BEGIN
        Index      := Index + 1;
        Info[Index] := value;
        Read(value);
    END;
    Size := Index;
END; { InSeq }

PROCEDURE Extremes(Data: Sequence;      (* input  *)
                   Number: INTEGER;     (* input  *)
                   VAR Max, Min: INTEGER); (* output *)
VAR Index: INTEGER;
BEGIN
    Max := Data[1];
    Min := Data[1];
    FOR Index := 2 TO Number DO BEGIN
        IF Max < Data[Index] THEN
            Max := Data[Index];
        IF Min > Data[Index] THEN
            Min := Data[Index];
    END;
END; { Extremes }

BEGIN { Extrema }
    InSeq(Age, Count);
    Extremes(Age, Count, Oldest, Youngest);
    WriteLn('The Number of people is ', Count:3);
    WriteLn('The Oldest is ', Oldest:2);
    WriteLn('The Youngest is ', Youngest:2);
END. { Extrema }
```

The procedure `InSeq` reads a sequence of values sandwiched by a terminating value as we have done in the programs `FindMaximumTemperature` and `Variance` earlier in this chapter. `InSeq` has two parameters: `Info`, which is the array that is to be assigned the values read in, and `Size`, which is used to return the number of values read. Both of these are output parameters and therefore must be passed by reference. For this reason, `VAR` precedes each of the names `Info` and `Size` in the formal parameter list. Were they both of the same type, a single `VAR` would have sufficed.

The procedure `Extremes` accepts a sequence, `Data` and an `INTEGER` `Number`, and returns the extreme values `Max` and `Min`. In this case, `Number` is an input parameter and is passed by value, and `Max` and `Min` are output parameters and must be passed by reference. The array `Data` could be passed by value, which would protect its values, or it could be passed by reference to save execution time and memory space. Here we chose protection, but a `VAR` could have preceded `Data` in the formal parameter list.

The identifiers used in the main body of the program differ considerably from those in the procedures. The procedures are general, and apply to any values, so the variables have general names, such as `Data`, `Max` and `Min`. The main program deals with a specific problem and has names associated with that problem such as `Age`, `Oldest`, and `Youngest`. The proper choice of names can make programs very readable.

IntArrayLib: Integer Array Library

As our various examples have shown, arrays have wide application in computing, so it would be useful to “encapsulate” some basic declarations and actions into a library. Use of this library spares the programmer from having to recreate the various common operations on arrays, including input-output and sorting, from scratch each time they are needed. This library illustrates all of the aspects of libraries, including the initialization of variables.

Figure 8.8 Library `IntArrayLib`

```
UNIT IntArrayLib;
(* Library of Arrays of Integers *)

INTERFACE
    CONST MaxSize = 100;

    TYPE Range      = 0..MaxSize;
         IntArray = ARRAY [Range] OF INTEGER;

    VAR Term: INTEGER; { input and array
terminating value }

    PROCEDURE ReadIntArray(VAR Info: IntArray);
    (* Read input values into Info *)
    PROCEDURE WriteIntArray(Info: IntArray; Width: INTEGER);
    (* Display values from Info in Width field *)
    FUNCTION  SizeArray(Info: IntArray): INTEGER;
    (* Return size of Info *)
    PROCEDURE SortIntArray(VAR A: IntArray);
    (* Sort Info array *)

IMPLEMENTATION

    PROCEDURE ReadIntArray(VAR Info: IntArray);
```

```
VAR Ind: Range;
    Value: INTEGER;
BEGIN
    WriteLn('Enter values ending with terminal value ');
    Ind := 0;
    Read(Value);
    WHILE Value <> Term DO BEGIN
        Inc(Ind);
        Info[Ind] := Value;
        Read(Value);
    END;
    Info[Ind+1] := Term;
END; { ReadIntArray }

PROCEDURE WriteIntArray(Info: IntArray; Width: INTEGER);
VAR Ind: Range;
BEGIN
    FOR Ind := 1 TO SizeArray(Info) DO
        WriteLn(Info[Ind]:Width);
    END; { WriteIntArray }

PROCEDURE SizeArray(List: IntArray):INTEGER;
VAR Ind: Range;
BEGIN
    Ind := 0;
    WHILE List[Ind+1] <> Term DO
        Inc(Ind);
    SizeArray := Ind;
END; { SizeArray }

PROCEDURE SortIntArray(VAR A: IntArray);
VAR Ind, J, N: Range;
    Temp: INTEGER;
BEGIN
    N := SizeArray(A);
    FOR Ind := 1 TO N-1 DO
        FOR J := 1 TO N-1 DO
            IF A[J] < A[J+1] THEN BEGIN
                Temp := A[J];
                A[J] := A[J+1];
                A[J+1] := Temp;
            END;
        END;
    END; { SortIntArray }

BEGIN
    Write('Enter Terminal value ');
    Read(Term);
END. { IntArrayLib }
```

The library `IntArrayLib`, shown in Figure 8.8, is a library that defines a constant, a data type, a variable, a number of procedures and a function. These are all described in the publicly available `INTERFACE` part. In particular, the library defines the data type `IntArray` as an array of `INTEGERs` having a `MaxSize`, which is currently specified as constant 100 but could be modified easily. There is also a variable called `Term`, which represents the terminating value that marks the end of the values during input, and also in the arrays. Also included are the procedures `ReadIntArray`, `WriteIntArray` and `SortIntArray`, and the function `SizeArray`. Other procedures such as `SearchIntArray` could be added later. Notice that the size of arrays is not passed to these procedures because the terminator, which is recorded in the arrays, determines that size. Function `SizeArray` returns the actual size of an array, which is less than or equal to `MaxSize`.

The program `IntArrayProg`, shown in Figure 8.9, represents a very simple use of this Library. A typical run of the program is given at the right of the figure with the user's responses in bold.

Figure 8.9 Program IntArrayProg and output

<code>PROGRAM IntArrayProg;</code>	Output from typical run
<code>USES IntArrayLib;</code>	Enter Terminal value 0
	Enter values End with term
<code>VAR A: IntArray;</code>	123 456 789 987 654 321 0
<code>BEGIN</code>	987
<code>ReadIntArray(A);</code>	789
<code>SortIntArray(A);</code>	654
<code>WriteIntArray(A, 4);</code>	456
<code>END. { IntArrayProg }</code>	321
	123

The implementation of the various procedures for reading and writing an array has been demonstrated previously as pieces of programs, and we'll cover sorting in the next chapter (a little anticipation will do you good). Here, these pieces are collected and packaged in the implementation part of this library `UNIT`. Notice that, at its very bottom this `UNIT` has a body, or block, that requests the terminating value and stores it into the publicly available variable `Term`. This initialization is performed as the very first action in the use of this Library, even before the actions of the programs that use it.

This library could be modified and certainly extended, for instance to search for a given value, or to find the maximum value of an array, etc. A similar library could be created for `REAL` values, or for `CHAR` values.

8.3 Two Dimensional Arrays in Pascal

Arrays of Arrays—Two Dimensional Arrays

Thus far, we have only considered arrays of one dimension, i.e. with one index. Arrays of more dimensions, especially two dimensions, are also very common in computing.

A two-dimensional array can be viewed as a rectangular grid with two indices, one specifying a row and the other a column.

Figure 8.10 A calendar

1993 May						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
						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					

For example, a calendar like the one in Figure 8.10 can be seen as a two-dimensional table having rows, known as weeks, and columns known as days. However, a calendar can also be viewed as an array of arrays; a month is an array of weeks, and a week is an array of days. In Pascal this declaration is written as:

```
TYPE DayType    = INTEGER;
    DayNames    = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
    WeekType    = ARRAY [DayNames] OF DayType;
    MonthType   = ARRAY [1..6] OF WeekType;
```

Note that type MonthType could also have been declared as:

```
TYPE MonthType = ARRAY [1..6] OF
    ARRAY [DayNames] OF DayType;
```

which shows that it is an array whose elements are themselves arrays. It could even have been declared using a shortcut as:

```
TYPE MonthType = ARRAY [1..6, DayNames] OF DayType;
```

Variables, such as the number of hours worked in any day of the month may be declared as in the following:

```
VAR Hours, Pay: MonthType;
    DayIndex: DayNames;
    WeekIndex: 1..6;
    Sum: INTEGER;
    Average: REAL;
```


There are two different ways to access the element values of variables of type `MonthType`:

```
Hours[2, Mon] := 8;
```

or equivalently

```
Hours[2][Mon] := 8;
```

as `Hours[2]` represents an array that can be indexed.

The choice of which way to declare a two dimensional array will depend upon how the array is viewed in the context of the application. In our example here, it is natural to think of a month as being a sequence of weeks and therefore it is better to make the declaration in two stages, first the `WeekType` and then the `MonthType` as a sequence of elements of `WeekType`. In contrast, if the rows and columns of the array were of equal importance, for example, as in a chess board, then a declaration of the form:

```
TYPE
    ChessMen = (King, Queen, Bishop, Knight, Rook, Pawn,
    Empty);
    BoardType = ARRAY [1..8, 1..8] OF ChessMen;
```

would be the more natural. This can be extended to more dimensions as will be shown on the following pages.

The program `TableProg`, in Figure 8.11, shows a program involving a table, which is an array of arrays. This program reads in values that are entered on a grid similar to the calendar of Figure 8.10, and outputs the hours corresponding to a given week.

Figure 8.11 Program TableProg

```
PROGRAM TableProg;
{ Build a table of hours worked for a calendar month }

TYPE DayType      = INTEGER;
    DaysOfWeek = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
    WeekType    = ARRAY[DaysOfWeek] OF DayType;
    MonthType   = ARRAY[1..6] of WeekType;

VAR Hours: MonthType;
    Day: DaysOfWeek;
    Week: 1..6;
    Sum: INTEGER;
    Average: REAL;
BEGIN
    { Input of data for a month }
    WriteLn('Enter hours, week by week ');
    WriteLn('If day doesn''t exist, enter -1');
    WriteLn('          Sun Mon Tue Wed Thu Fri Sat');
    FOR Week := 1 TO 6 DO BEGIN
        Write( 'Week ' , Week:2 , ' ' );
        FOR Day := Sun To Sat DO
```

```
        Read(Hours[Week][Day]); { NOTE access }
    END;
    WriteLn;

    { Find data for any given week }
    Write('Enter the week ');
    Read(Week);
    WriteLn('The hours worked are: ');
    FOR Day := Sun TO Sat DO
        IF Hours[Week, Day] < 0 THEN
            Write(' - ')
        ELSE
            Write(Hours[Week, Day]:3); { NOTE access }
    END. { TableProg }
```

The first part of the program enters the hours worked week by week. As can be seen from the calendar sample shown earlier, six weeks have to be allowed and not all days exist in the first and last week. The input part of `TableProg` takes that into account by having the user enter value -1 for those days. The second part of the program first requests the week number and then provides the number of hours worked in each day of that week.

Notice that the two dimensions of the array `Hours` have a different type: `range 1..6` and enumeration `DaysOfWeek`. Also, the first part accesses a day by `Hours[Week][Day]` whereas the second part accesses it by `Hours[Week, Day]`; both access methods are equivalent.

The output of a typical run for the month shown in the calendar of Figure 8.10 is the following.

```
Enter hours, week by week
If day doesn't exist, enter -1
      Sun  Mon  Tue  Wed  Thu  Fri  Sat
Week 1   -1   -1   -1   -1   -1   -1    4
Week 2    0    8    8    8    9    8    0
Week 3    0    8    0    0    9    8    4
Week 4    4    8    8    8    8   10    0
Week 5    0    8    8    8    8    8    0
Week 6    0    8   -1   -1   -1   -1   -1

Enter the week 1
The hours worked are:
- - - - - 4
```

8.4 N-Dimensional Arrays

More Dimensions

The definition of arrays in Pascal lends itself conveniently to extension of three or more dimensions. Three-dimensional arrays are defined and used in a manner similar to the one and two-dimensional arrays seen previously. They can be viewed as extensions of two dimensional arrays, which are, in turn, extensions of one dimensional arrays. They are conveniently viewed as a new entity declared as the type

```
TYPE name = ARRAY[range1, range2, range3] OF item-type;
```

with items selected by

```
array-name[index1, index2, index3]
```

For example, a three-dimensional array of the *Hours* worked during a particular *Month*, *Week* and *Day* can be described by the declaration:

```
TYPE Calendar = ARRAY[1..12, 1..6, 0..6] OF REAL;
```

and the hours worked on the first Saturday of March selected by

```
Hours[3, 1, 6];
```

The program `PayArray3D`, shown in Figure 8.12, presents another declaration of such a `Calendar` data structure. This program shows how the hours worked can be stored in a calendar, which has three dimensions, first a month, then a week and then a day. As we have seen, there could be six weeks in a month, but here we have just assumed four. Notice that this program contains a stub procedure `EnterMonth`, which always returns the value `Feb`. This is another example of a stub procedure used in the top-down development of a program. Later in the development process, this stub will be replaced by an actual procedure that obtains the month from the user. In the meantime, this stub allows the rest of the program to be tested.

Figure 8.12 Program PayArray3D

```
PROGRAM PayArray3D;

TYPE MonthRange = (Jan, Feb, Mar, Apr, May, Jun,
                  Jul, Aug, Sep, Oct, Nov, Dec);
   WeekRange = 1 .. 6;
   DayRange  = (Sun, Mon, Tue, Wed, Thur, Fri, Sat);
   Calendar  = ARRAY[MonthRange, WeekRange,
DayRange] OF REAL;
VAR Hours: Calendar;
    Gross, Rate: REAL;
    Month: MonthRange;
    Week : INTEGER;
    Day : DayRange;
```

```
PROCEDURE ReadMonth(VAR Month: MonthRange); (* Output *)
(* This is a stub procedure *)
BEGIN
    Month := Feb;
END; { ReadMonth }

PROCEDURE EnterWorkWeek(  Month: MonthRange; (* Input *)
                          Week: WeekRange;   (* Input *)
                          VAR Hours: Calendar); (* Output *)
VAR Day: DayRange;
BEGIN
    WriteLn('Enter hours for 7 days ');
    FOR Day := Sun TO Sat DO
        Read(Hours[Month, Week, Day]);
    END; { EnterWorkWeek }

PROCEDURE Pay(          Month: MonthRange; (* Input *)
              Week : WeekRange;  (* Input *)
              Hours: Calendar;    (* Input *)
              Rate : REAL;        (* Input *)
              VAR Gross: REAL);    (* Output *)
VAR Day: DayRange;
    Sum: REAL;
BEGIN
    Sum := 0.0;
    FOR Day := Sun TO Sat DO
        Sum := Sum + Hours[Month, Week, Day];
    IF Sum < 40.0 THEN
        Gross := Rate * Sum
    ELSE
        Gross := Rate * 40.0 +
                1.5 * Rate * (Sum - 40.0);
    END; { Pay }

BEGIN
    ReadMonth(Month);
    Write('Enter the pay rate ');
    Read(Rate);
    Write('Enter the hours for the ');
    WriteLn('four weeks of the month ');
    FOR Week := 1 TO 4 DO BEGIN
        EnterWorkWeek(Month, Week, Hours);
        Pay(Month, Week, Hours, Rate, Gross);
        WriteLn('The gross pay is ', Gross:7:2);
    END;

END. { PayArray3D }
```

The procedure `EnterWorkWeek` obtains from the user the hours worked in a week of a given month. Notice that the `Month` and `Week` are passed in, and the `Hours` are passed out.

The gross pay for a given week of a given month at a given rate is computed by the procedure `Pay`. Notice here that the three-dimensional array `Hours` is passed by value; it could have been passed by reference for efficiency purposes in order not to make a copy, and thus save time and memory space.

It should be obvious that the program should be improved, so as to be able to deal with months having more than four weeks (not all months have 28 days!)

8.5 Records in Pascal

Records are groupings of components that may be of different types. Simple RECORDS are declared as types in the form shown at the left of Figure 8.13. An example of a declaration of a part in an inventory stock application is shown at the right of the figure.

Figure 8.13 Record pattern and example

<pre> TYPE name = RECORD field: type; field: type; ... END;</pre>	<pre> TYPE PartType = RECORD Price: REAL; Quantity: INTEGER; Reorder: BOOLEAN; Status: CHAR; END;</pre>
---	--

Following such a type declaration, variables representing values of this type, e.g. chair parts, can be declared as:

```
VAR Leg, Seat, Rung, Back: PartType;
```

To access the fields of a given record we use a dot notation like, for example:

```
Leg.Price := 3.14;
Read(Seat.Quantity);
```

As another example of a RECORD, consider `DateType` below, a very useful record describing a date as having three components: Year, Month and Day.

```

DateType = RECORD
  Year:  1900..2100;
  Month: 1..12;
  Day :  1..31
END;
```

An alternative to `DateType`, without subranges but with INTEGERS, could be specified as:

```

IntDateType = RECORD
  Year:  INTEGER;
  Month: INTEGER;
```

```
Day:    INTEGER;  
END;
```

The Pascal `WITH` statement has been designed only for use with records. It makes it possible to avoid the repetition of a record name, and has the form:

```
WITH RecordName DO BEGIN  
    StatementList  
END;
```

where the statements in `StatementList`, separated by semicolons, involve the field names of the given record. These field names can then be used by themselves (without the dot notation) and appear to be only simple variables. The `WITH` statement attaches the record name to the fields that need it. For example, the following two groups of statements are equivalent.

```
WITH MoonDate DO BEGIN  
    Year  := 1969;  
    Month := 7;  
    Day   := 20;  
END;  
MoonDate.Year  := 1969;  
MoonDate.Month := 7;  
MoonDate.Day   := 20;
```

We'll see an example of this statement in a later example.

The program `PartRecord`, in Figure 8.14, is a program that computes the total Worth of the four components in an inventory of chairs. In a previous example, the `Worth` was computed using parallel arrays; here, it is done with records.

Figure 8.14 Program PartRecord

```
PROGRAM PartRecord;  
(* Compute inventory worth *)  
TYPE PartType = RECORD  
    Price:    REAL;  
    Quantity: INTEGER;  
    Reorder:  BOOLEAN;  
    Status:   CHAR;  
END;  
VAR Leg, Seat, Rung, Back: PartType;  
    Worth: REAL;  
BEGIN  
    { Enter the Values }  
    WriteLn('Enter price and quantity ');  
    WriteLn('for leg, seat, rung, back');  
    Read(Leg.Price);      Read(Leg.Quantity);  
    Read(Seat.Price);     Read(Seat.Quantity);  
    Read(Rung.Price);     Read(Rung.Quantity);  
    Read(Back.Price);     Read(Back.Quantity);  
  
    { Compute the Worth }  
    Worth := Leg.Price * Leg.Quantity +  
            Seat.Price * Seat.Quantity +  
            Rung.Price * Rung.Quantity +  
            Back.Price * Back.Quantity;
```

```

    { Output the Total Worth }
    WriteLn('The total worth is ', Worth:8:2);
END. { PartRecord }

```

First the `PartType` is defined as a record, and four variables, `Leg`, `Seat`, `Rung`, `Back`, of that type are declared. Then the values for the `Price` and `Quantity` fields of each of the parts are input. The total `Worth` is computed, and the final value is output. A typical run of this program is:

```

Enter price and quantity
for leg, seat, rung, back
1.12 200
2.50 100
0.75 400
3.00 300
The total worth is 1674.00

```

The program shown in Figure 8.15, `DateRecordProg`, makes use of a version of the `DateType` record that is a slightly modified version of what we've seen above. The program contains two procedures, `ReadDate` and `WriteDate` that illustrate how records such as `DateType` can be used as parameters. The main block of the program calls these procedures to read in a `BirthDate` and determine one's approximate age by subtracting that year from the present year, given as a constant, `YearNow`, which must be updated each year.

Figure 8.15 Program `DateRecordProg`

```

PROGRAM DateRecordProg;

CONST YearNow = 1993;

TYPE DateType = RECORD
    Year : INTEGER;
    Month: INTEGER;
    Day  : INTEGER;
END;

PROCEDURE ReadDate(VAR Date: DateType);
BEGIN
    Write('Enter a date in the order YYYY MM DD ');
    Read(Date.Year);
    Read(Date.Month);
    Read(Date.Day);
END; { ReadDate }

PROCEDURE WriteDate(Date: DateType);
BEGIN
    WriteLn;
    WriteLn(Date.Year:4, Date.Month:3, Date.Day: 3);
END; { WriteDate }

```

```
VAR BirthDate: DateType;
    Age: INTEGER;

BEGIN
    WriteLn('What is Your Birthdate?');
    ReadDate(BirthDate);
    WriteDate(BirthDate);
    Age := YearNow - BirthDate.Year;
    WriteLn('Your age is around ', Age:2);
END. { DateRecordProg }
```

Notice especially that only one single data item, `BirthDate`, is passed to the procedures, that is to say, the three parts, Year, Month, Day, do not have to be passed separately. These two procedures could be included in a date library.

When creating a data item that is a `RECORD`, it is convenient to use a special procedure that we could call a `Record Constructor`, to initialize all the fields in the `RECORD`. For example, `MoonDate` could be initialized using a `Date Constructor`:

```
DateConstruct(MoonDate, 1969, 7, 20);
```

Similarly a `Chair` part, such as `Leg`, can be initialized by:

```
ConstructChair(Leg, 10, 20, FALSE, 'A');
```

Constructors of this kind can be implemented as procedures like `ConstructChair` in Figure 8.16.

Figure 8.16 Record constructor example

```
PROCEDURE ConstructChair(VAR Part: PartType;
                        P: REAL;
                        Q: INTEGER;
                        R: BOOLEAN;
                        S: CHAR);

BEGIN
    Part.Price      := P;
    Part.Quantity   := Q;
    Part.ReOrder    := R;
    Part.Status     := S;
END; { ConstructChair }
```

This procedure can then be used as an alternative to input the values.

```
ConstructChair( Leg,  1.12, 200, FALSE, 'A' );
ConstructChair( Seat, 2.50, 100, TRUE,  'C' );
ConstructChair( Rung, 0.75, 400, FALSE, 'B' );
ConstructChair( Back, 3.00, 300, FALSE, 'A' );
```


ComplexLib: A Library for Complex Numbers

A major objective of including data structures such as arrays and records in a programming language like Pascal is to allow the simple manipulation of more complex data items. These complex data items can usually be treated in the same way as the basic ones, such as integers or characters. This is usually done by defining abstract data types, that will be implemented through libraries, just as was done previously for simpler types. Such libraries comprise the type definition as well as the operations that apply to items of that type. This is illustrated by the ComplexLib library that defines the complex number abstract data type. Such complex numbers are very useful in electrical engineering, especially for analyzing alternating current circuits.

A complex number z is usually denoted as $x + iy$, where i is the imaginary square root of -1 ($i^2 = -1$). In such a complex number x is called the Real part and y is called the Imaginary part. A complex number data type can thus be implemented as a record composed of two parts.

```
TYPE COMPLEX = RECORD
    RealPart: REAL;
    ImagPart: REAL;
END;
```

Operations on complex numbers are numerous and will include arithmetic operations as well as input-output operations. The input of a complex number C is done with `ReadComplex(C)`, and its output is done with a call to `WriteComplex(C)`. The arithmetic operations include addition, subtraction, multiplication and division. Other operations include finding the conjugate of a complex number, compute its magnitude and the related angle, etc. The ComplexLib library of Figure 8.17 implements some of these and should be completed.

Figure 8.17 The Complex abstract data type

```
UNIT ComplexLib;
(* The Abstract Data Type Complex Number *)

INTERFACE
TYPE Complex = RECORD
    RealPart: REAL;
    ImagPart: REAL;
END;

PROCEDURE ReadComplex(VAR C: Complex);
PROCEDURE WriteComplex(C: Complex);
PROCEDURE AddComplex(A, B: Complex; VAR C: Complex);
PROCEDURE MultComplex(A, B: Complex; VAR C: Complex);

IMPLEMENTATION
PROCEDURE ReadComplex(VAR C: Complex);
BEGIN
    Write('Enter Real Part ');
    Read (C.RealPart);
```

```
        Write('Enter Imag Part ');
        Read (C.ImagPart);
    END;

    PROCEDURE WriteComplex(C: Complex);
    BEGIN
        Write(C.RealPart: 6: 2);
        Write(' ', ' ');
        Write(C.ImagPart: 6: 2);
    END;

    PROCEDURE AddComplex(A, B: Complex; VAR C: Complex);
    BEGIN
        C.RealPart := A.RealPart + B.RealPart;
        C.ImagPart := A.ImagPart + B.ImagPart;
    END;

    PROCEDURE MultComplex(A, B: Complex; VAR C: Complex);
    BEGIN
        C.RealPart := A.RealPart * B.RealPart -
                     A.ImagPart * B.ImagPart;
        C.ImagPart := A.ImagPart * B.RealPart +
                     A.RealPart * B.ImagPart;
    END;

END. { ComplexLib }
```

The procedure `AddComplex(A, B, C)` adds the two complex numbers `A` and `B` to yield a third complex number `C`, by adding the corresponding real and imaginary parts. The multiplication of two complex numbers is more ...er... complex. If complex number `A` is represented as $AR + iAI$ (which is short for `A.Real + i × A.Imag`) and `B` is $BR + iBI$ then the product is:

$$\begin{aligned} A*B &= (AR + iAI)*(BR + iBI) \\ &= AR*(BR + iBI) + iAI*(BR + iBI) \\ &= AR*BR + iAR*BI + iAI*BR + i^2AI*BI \quad (i^2 = -1) \\ &= AR*BR + i(AR*BI + AI*BR) - AI*BI \\ &= (AR*BR - AI*BI) + i(AR*BI + AI*BR) \end{aligned}$$

The procedure `MultComplex(A, B, C)` implements this multiplication operation.

The program `ComplexProg`, in Figure 8.18, shows the use of this small `ComplexLib` library. It reads two complex numbers `X`, `Y`, computes their sum and product, and outputs the two results.

Figure 8.18 Program ComplexProg

```
PROGRAM ComplexProg;
USES ComplexLib;

VAR X, Y, Z: Complex;
```

```
BEGIN
  WriteLn('Enter a complex number ');
  ReadComplex(X);
  WriteComplex(X);
  WriteLn;

  WriteLn('Enter a complex number ');
  ReadComplex(Y);
  WriteComplex(Y);
  WriteLn;

  Write('The sum of these is ');
  AddComplex(X, Y, Z);
  WriteComplex(Z);
  WriteLn;

  Write('The product of these is ');
  MultComplex(X, Y, Z);
  WriteComplex(Z);
  WriteLn;

END. { ComplexProg }
```

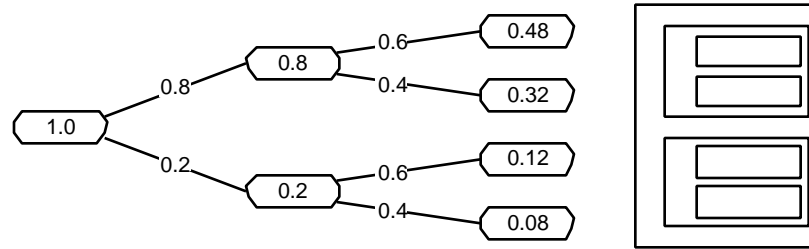
A typical output of ComplexProg is:

```
Enter a complex number
Enter Real Part 3.0
Enter Imag Part 4.0
3.00 , 4.00
Enter a complex number
Enter Real Part 5.0
Enter Imag Part 6.0
5.00 , 6.00
The sum of these is 8.00 , 10.00
The product of these is -9.00 , 38.00
```

As we've indicated, ComplexLib should be extended further to include the subtraction and division of complex numbers, to determine the conjugate of a complex number, its magnitude and angle. It could even draw complex numbers.

Records of Records: Nested Records

In some applications, it is very useful to nest one RECORD within another. This is quite possible as a record field can be of any type, in particular another record. As an example, let's look at a "tiny" tree that consists of two smaller sub-trees as shown in Figure 8.19.

Figure 8.19 Tree and sub-trees

That tiny tree describes a situation involving two stages of decisions, with the possibility of success or failure, “pass” or “fail”, at each decision. This leads to four possible outcomes, each corresponding to a path from the *root* of the tree at the left to a *leaf* at the right. The top path corresponds to both trials succeeding, the bottom path corresponds to both trials failing, and the middle paths correspond to one or the other trial succeeding. Suppose that the probability of success on the first trial is 0.8, and on the second trial is 0.6 (lower because of exhaustion perhaps). The right part of the figure is another way of representing the tree, using records as we explain in more detail below.

When trials are independent then the probability of each path is determined by the product of the individual probabilities along the path. For example, the probability of both trials being successful is $0.8 \times 0.6 = 0.48$. When the probability of success is P then the probability of failure is $(1 - P)$, so the probability of both trials failing is:

$$(1 - 0.8) \times (1 - 0.6) = 0.2 \times 0.4 = 0.08$$

The program `NestedRecords`, shown in Figure 8.20, is an implementation of this tiny tree structure. It consists of a record `Trial`, having itself two records, `Pass` and `Fail`, nested within it. The top record, `Pass`, has a probability field, and also a `Pass` record and a `Fail` record within it. Similarly, the bottom record `Fail` has a probability field and another `Pass` and `Fail` within it. Many field names are repeated (`Pass`, `Fail` and `Prob`) but the nesting puts them at different levels, which removes all ambiguity.

Figure 8.20 Program NestedRecords

```
PROGRAM NestedRecords;
(* Illustrate tree represented by nested records *)
TYPE Trial = RECORD
    Pass: RECORD
        Prob: REAL;
        Pass: RECORD
            Prob: REAL;
        END;
        Fail: RECORD
            Prob: REAL;
        END;
    END;
    Fail: RECORD
        Prob: REAL;
```

```

                                Pass: RECORD
                                    Prob: REAL;
                                END;
                                Fail: RECORD
                                    Prob: REAL;
                                END;
                                END;
                                END;
VAR Tree: Trial;
    P1, P2: REAL;
BEGIN
    Write('Enter probability of first pass ');
    Read(P1);
    Tree.Pass.Prob := P1;
    Tree.Fail.Prob := 1.0 - P1;
    Write('Enter probability of second pass ');
    Read(P2);
    WriteLn;

    (* Compute all the path probabilities*)
    Tree.Pass.Pass.Prob := P1 * P2;
    Tree.Pass.Fail.Prob := P1 * (1.0 - P2);
    Tree.Fail.Pass.Prob := (1.0 - P1) * P2;
    Tree.Fail.Fail.Prob := (1.0-P1) * (1.0-P2);

    (* Draw the tree *)
    WriteLn(Tree.Pass.Pass.Prob:30:2);
    WriteLn(Tree.Pass.Prob:20:2);
    WriteLn(Tree.Pass.Fail.Prob:30:2);
    WriteLn(1.00:10:2);
    WriteLn(Tree.Fail.Pass.Prob:30:2);
    WriteLn(Tree.Fail.Prob:20:2);
    WriteLn(Tree.Fail.Fail.Prob:30:2);
END. { NestedRecords }

```

The body of this program references the various paths using the dot notation. For example, the lowest path involving two failures, has a probability (given on the last line of the program) denoted by:

Tree.Fail.Fail.Prob

The program computes the four probabilities of each path. Then it outputs the tree in a semi-graphical representation. To do that, the output values are spaced both vertically and horizontally in the form of a tree, in a manner similar to Figure 8.19. The first value, 0.48, is placed first far to the right, as are all the leaf values. The tree root is at the left. This is a typical output.

Enter prob of first pass 0.8

Enter prob of second pass 0.6

0.48

```

                                0.80
                                0.32
1.00
                                0.12
                                0.20
                                0.08
```

Larger trees, that have more than three levels could be represented in a similar way. However, when the number of levels increases very much, there are other more convenient ways to represent trees. These involve pointers which are discussed later in this chapter.

Arrays of Records in Pascal

When we first introduced arrays in this chapter, we treated them as groupings of simple types, `INTEGERs`, `REALs`, etc. We then expanded this consideration to arrays of arrays, i.e. multi-dimensional arrays. Now, we expand the scope of arrays to include arrays of records, which are particularly useful data structures.

For example, consider an array `Employees[1..20]` where each of the 20 `Employees` is described by two values: the number of `Hours` worked and the `Rate` of pay. The `Hours` and `Rate` could be grouped into a `Worker` record as follows:

```
TYPE Worker = RECORD
    Hours: REAL;
    Rate : REAL
END;
WorkForce = ARRAY [1..20] OF Worker;
VAR Employees: WorkForce;
```

The time worked by the third employee is then accessed by

```
Employees[3].Hours
```

The program `PayRecords`, given in Figure 8.21, shows a further refinement of the type `Worker` that includes `IdNum`, the employee number, `Birth` of type `DATE`, another record that gives the `Worker's` date of birth and is a component of the `Worker` record, and `Division`, another complementary field.

Figure 8.21 Program PayRecords

```
PROGRAM PayRecords;
(* Computes payroll information *)
(* using arrays of records      *)

CONST NumberOfEmployees = 5;
TYPE DATE = RECORD
    Year:  INTEGER;
    Month: 1..12;
    Day:   1..31;
END;
```

```

Worker = RECORD
    IdNum: INTEGER;
    Hours: REAL;
    Rate: REAL;
    Birth: DATE;
    Division: CHAR;
END;
Range = 1..NumberOfEmployees;
WorkForce = ARRAY[Range] OF Worker;
VAR I: Range;
    Employees: WorkForce;
    MaxHours: REAL;

BEGIN
    { Enter worker information }
    Write('Enter ID, hours, rate ');
    WriteLn('and the year of birth ');
    FOR I := 1 TO NumberOfEmployees DO
        WITH Employees[I] DO BEGIN
            Write('Next ');
            Read(Idnum);
            Read(Hours);
            Read(Rate);
            Read(Birth.Year);
        END; { WITH }

    { Find maximum hours }
    MaxHours := 0.0;
    FOR I := 1 TO NumberOfEmployees DO
        WITH Employees[I] DO
            IF MaxHours < Hours THEN
                MaxHours := Hours;
    WriteLn('Maximum hours = ', MaxHours:3:1);

    { PROJECT:
    {   Finish this by adding:
    {       Find the minimum wage
    {       Find the maximum age
    {       Accumulate all hours
    {       Count persons over 30
    {       Compute the net pay
    {       Make into procedures

END. { PayRecords }

```

This program also shows how information about the employees could be entered. Notice that all the information is not supplied; for instance, only the year of birth is input. Notice also the convenience of using the `WITH` statement.

Also included in the program is a part that computes the maximum hours worked by an employee. The program could be extended, as listed in the comment, to compute a number of things like maximum age, minimum wage, etc.

It is also possible to nest records directly in the declaration rather than making the nested record a separate type. Thus, instead of declaring the type `DATE`, the `Worker` record could be declared as:

```
Worker = RECORD
    IdNum: INTEGER;
    Hours: REAL;
    Rate: REAL;
    Birth: RECORD
        Year: INTEGER;
        Month: 1..12;
        Day: 1..31;
    END;
    Stage: CHAR;
END;
```

This form of declaration is not as general as the previous form because it does not create a separate template of the `DATE` record which can be used elsewhere.

If the list of `Employees` were to include both hourly and salaried employees, there would be slightly different requirements for the two different kinds of employees: for an hourly worker, the `Hours` worked and `Rate` of pay both have to be recorded while, for salaried workers, only the `Salary` is needed. Thus, it would be convenient if the actual layout of the record could be different for the two kinds of employees, and at the same time all the records were still grouped into a single array. This can be achieved in Pascal by using *variant records*. These records include a special field, the *tag field*, which specifies which layout is used for a particular record. In the declaration of a variant record, the discrimination between the various records is defined through a `CASE` form as illustrated by the following type declaration:

```
Worker = RECORD
    IdNum: INTEGER;
    Birth: Date;
    Stage: CHAR;
    CASE Hourly: BOOLEAN OF
        TRUE:
            (Hours: REAL;
             Rate: REAL; )
        FALSE:
            (Salary: REAL; )
    END;
```

The tag field is `Hourly` which determines whether to use the two fields `Hours` and `Rate`, or the single field `Salary`.

The exact syntax of the declaration of variant records is rather complicated so the following should be noted:

1. A record declaration can have only one variant part and it must follow the fixed part of the record.

2.

All the field definitions comprising a particular variant are enclosed in parentheses.
3.

All field names must be distinct—even if they occur in different variants.
4.

The structure of the CASE form in the definition of variant records differs from the CASE statement in that there is no ELSE or OTHERWISE clause and there is no END—since the variant part is at the end of the RECORD definition, the record END serves the purpose.
5.

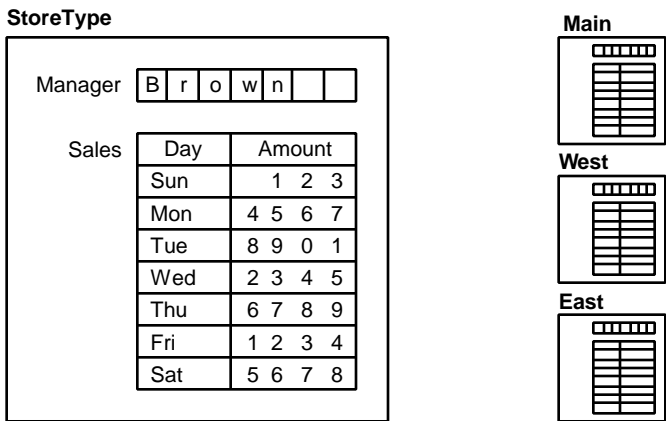
Every value of the type of the tag field—Hourly in our example—should be explicitly listed even if such a value would never be used; for such values an empty pair of parentheses must be shown.
6.

A variant may itself be a record that contains a variant part.

Records of Arrays

In a previous section, we have discussed how RECORDs can be nested within a RECORD. It should probably come as no surprise that ARRAYs can be nested within a RECORD. For example, a grocery store could be described for some purposes by the name of the manager, stored as an array of characters, and an array of Amounts of money, daily sales perhaps, for each day of the week. The diagram of Figure 8.22 shows the structure of StoreType as a RECORD of ARRAYs; notice that one of the arrays, Manager is shown horizontally and the other, Amount, is shown vertically. The type at the left of the figure is the template of a store; three of these stores, Main, West and East, are shown at the right.

Figure 8.22 Record of arrays



Main

West

East

There could be many instances of such representations of stores; there are three, named Main, West, East, shown in the figure. These stores could be defined by the declaration:

VAR Main, West, East: StoreType;

The program `RecordsOfArrays`, shown in Figure 8.23, illustrates how the record `StoreType` is declared in Pascal. It is simply a `RECORD` that has two arrays: a `Manager` array of characters, and an `Amount` array of integers.

Figure 8.23 Program `RecordsOfArrays`

```
PROGRAM RecordsOfArrays;

CONST NameSize = 20;
TYPE WeekDayType = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
   StoreType = RECORD
       Manager: ARRAY[0..NameSize] OF CHAR;
       Amount:  ARRAY[WeekDayType] OF INTEGER;
   END;

PROCEDURE EnterAmount(VAR Store: StoreType);
VAR Day: WeekDayType;
BEGIN
    WriteLn('Enter amounts for 7 days. ');
    WriteLn('Begin with Sunday. ');
    WITH Store DO
        FOR Day := Sun TO Sat DO
            Read(Amount[Day]);
    END; { EnterAmount }

PROCEDURE SumAmount(Store: StoreType; VAR Amount:
INTEGER);
VAR D: WeekDayType;
BEGIN
    Amount := 0;
    FOR D := Sun TO Sat DO
        Inc(Amount, Store.Amount[D]);
    END; { SumAmount }

VAR Main, West, East: StoreType;
    Total: INTEGER;
BEGIN
    EnterAmount(East);
    SumAmount(East, Total);
    Write('The total amount is ', Total:4);
END. { RecordsOfArrays }
```

The procedure `EnterAmount` is used to enter the `Amount` for a specified `Store` for each of the seven days of the week.

The procedure `SumAmount` sums the amounts for the specified `Store` and returns this through the second parameter. Notice the increment statement; it uses `Amount` in two ways, once as a variable and then elsewhere, with the dot notation, to refer to the array field within the record. There is no ambiguity as the names inside a record are totally independent of the rest of the program.

The body of the program calls `EnterAmount` to enter the amounts for the East Store and `SumAmount` to obtain a `Total`, which is then output, as shown in the following test run of the program.

```
Enter amounts for 7 days.
Begin with Sunday.
1 2 3 4 5 6 7
The total amount is 28
```

Matrix Library

Computations in science and engineering frequently use *matrices*, rectangular arrays of numbers on which a number of operations are defined. Because they are so common it is convenient to create a matrix library, that defines the abstract data type `Matrix`. The library `MatrixLib`, shown in Figure 8.24, consists of the type definition for `Matrix` and five operations: `CreateMatrix`, `ReadMatrix`, `WriteMatrix`, `AddMatrix` and `MultMatrix`.

Figure 8.24 The MatrixLib library

```
UNIT MatrixLib;
(* Abstract Data Type Matrix *)

INTERFACE

CONST MaxRow = 10;
      MaxCol = 10;

TYPE RowRange = 1..MaxRow;
      ColRange = 1..MaxCol;
      GridType = ARRAY [RowRange, ColRange] OF REAL;
      Matrix = RECORD
          Grid : GridType;
          HiRow: RowRange;
          HiCol: ColRange;
      END;

PROCEDURE CreateMatrix(VAR Mat: Matrix;
                      Rows: RowRange;
                      Cols: ColRange);
PROCEDURE ReadMatrix(VAR Mat: Matrix );
PROCEDURE WriteMatrix(VAR Mat: Matrix );
PROCEDURE AddMatrix(  Mat1, Mat2: Matrix;
                      VAR Mat3: Matrix);
PROCEDURE MultMatrix(  Mat1, Mat2: Matrix;
                      VAR Mat3: Matrix);

IMPLEMENTATION

PROCEDURE CreateMatrix(VAR Mat: Matrix;
```

```

                                Rows: RowRange;
                                Cols: ColRange);

BEGIN
    Mat.HiRow := Rows;
    Mat.HiCol := Cols;
END; { CreateMatrix }

PROCEDURE ReadMatrix(VAR Mat: Matrix );
VAR R: RowRange;
    C: ColRange;
BEGIN
    WriteLn('Enter by rows ');
    FOR R := 1 TO Mat.HiRow DO BEGIN
        FOR C := 1 TO Mat.HiCol DO BEGIN
            Write('Enter a value ');
            Read(Mat.Grid[R, C]);
        END;
        WriteLn;
    END;
END; { ReadMatrix }

PROCEDURE WriteMatrix(VAR Mat: Matrix );
VAR R: RowRange;
    C: ColRange;
BEGIN
    FOR R := 1 TO Mat.HiRow DO BEGIN
        FOR C := 1 TO Mat.HiCol DO
            Write(Mat.Grid[R, C]: 9: 2);
        WriteLn;
    END;
END; { WriteMatrix }

PROCEDURE AddMatrix(    Mat1, Mat2: Matrix;
                        VAR Mat3: Matrix);
VAR R: RowRange;
    C: ColRange;
BEGIN
    FOR R := 1 TO Mat1.HiRow DO
        FOR C := 1 TO Mat1.HiCol DO
            Mat3.Grid[R, C] :=
                Mat1.Grid[R, C] + Mat2.Grid[R, C];
        END;
    END; { AddMatrix }

PROCEDURE MultMatrix(    Mat1, Mat2: Matrix;
                        VAR Mat3: Matrix);
VAR I, K: ColRange;
    J: RowRange;
    Sum: REAL;
BEGIN
    FOR I := 1 TO Mat1.HiRow DO
```

```

        FOR J := 1 TO Mat2.HiCol DO BEGIN
            Sum := 0.0;
            FOR K := 1 TO Mat2.HiRow DO BEGIN
                Sum := Sum +
                    Mat1.Grid[I, K] * Mat2.Grid[K, J];
            END;
            Mat3.Grid[I, J] := Sum;
        END;
    END; { MultMatrix }

END. { MatrixLib }

```

The type definitions for Matrix:

```

TYPE RowRange = 1..MaxRow;
     ColRange = 1..MaxCol;
     GridType = ARRAY[RowRange, ColRange] OF REAL;
     Matrix = RECORD
         Grid : GridType;
         HiRow: RowRange;
         HiCol: ColRange;
     END;

```

show that a Matrix is represented by a RECORD that contains an array of REALs having a maximum number of rows and columns defined by the two constants MaxRow and MaxCol both of which are set at the value 10, but that can easily be changed. When a particular Matrix is created, the CreateMatrix procedure specifies the actual number of rows and columns by assigning values to the fields HiRow and HiCol. A better version of CreateMatrix could also interactively ask for the sizes of matrices, and could specify the actual number of rows and columns from the response. It should also check whether the values given exceed the maximum values MaxRow and MaxColumn.

The input of matrices is carried out by the procedure ReadMatrix, one row at a time. The input could also have been done a column at a time. ReadMatrix provides a prompt reminding the user to enter a row at a time.

The algorithm for two-dimensional array addition was described in Chapter 8 of the Principles book. It is very simple as it adds the corresponding elements of the two matrices to produce the elements of the new matrix.

The procedure AddMatrix(X, Y, Z) in MatrixLib implements this algorithm, adding matrix X to matrix Y to produce resulting matrix Z. Figure 6.24 illustrates this process.

Figure 8.25 Matrix addition

<table style="border-collapse: collapse;"> <tr><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">2</td></tr> <tr><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">4</td></tr> </table>	1	2	3	4	+	<table style="border-collapse: collapse;"> <tr><td style="padding: 2px 10px;">5</td><td style="padding: 2px 10px;">6</td></tr> <tr><td style="padding: 2px 10px;">7</td><td style="padding: 2px 10px;">8</td></tr> </table>	5	6	7	8	=	<table style="border-collapse: collapse;"> <tr><td style="padding: 2px 10px;">8</td><td style="padding: 2px 10px;">11</td></tr> <tr><td style="padding: 2px 10px;">50</td><td style="padding: 2px 10px;">62</td></tr> </table>	8	11	50	62
1	2															
3	4															
5	6															
7	8															
8	11															
50	62															

A needed improvement to `AddMatrix` is a check that the two matrices `X` and `Y` are the same size.

The multiplication of two matrices is performed by the algorithm described in Chapter 8 of the Principles book. The first matrix is taken row by row while the second matrix is taken column by column. A sum is computed from the products of the elements of the first matrix row by the second matrix column. This sum is stored in the resulting matrix.

The procedure `MultMatrix(X, Y, Z)` given in `MatrixLib` implements this algorithm in Pascal. It computes the product of matrices `X` and `Y` to produce the result matrix `Z`, which will have the same number of rows as `X` and the same number of columns as `Y`. The number of columns of `X` must equal the number of rows of `Y`, and a better version of `MultMatrix` should check this. An example of this matrix multiplication is given in Figure 8.26.

Figure 8.26 Matrix multiplication

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \times \begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 11 \\ 50 & 62 \end{bmatrix}$$

Other operations, including transposing and inverting matrices, should also be implemented in order to have a complete Abstract Data Type, but are not shown here.

The program `MatrixProg` shown in Figure 8.27, illustrates the use of `MatrixLib`. The output obtained from a typical run, corresponding to the examples of Figures 8.24 and 8.25, is shown at the right of the figure.

Figure 8.27 Program MatrixProg

```

PROGRAM MatrixProg;
USES MatrixLib;

VAR A, B, C, D, E: Matrix;
BEGIN
    CreateMatrix(A, 2, 2);
    CreateMatrix(B, 2, 2);
    CreateMatrix(C, 2, 2);
    ReadMatrix(A);
    WriteLn('Matrix A');
    WriteMatrix(A);

```

Enter by rows
Enter a value 1
Enter a value 2

Enter a value 3
Enter a value 4

Matrix A

1.00
2.00

3.00
4.00

Enter by rows
Enter a value 5
Enter a value 6

```

WriteLn;
ReadMatrix(B);
WriteLn('Matrix B');
WriteMatrix(B);
WriteLn;
AddMatrix(A, B, C);
WriteLn('Matrix C');
WriteMatrix(C);
WriteLn;
CreateMatrix(D, 2, 3);
CreateMatrix(E, 3, 2);
ReadMatrix(D);
WriteLn('Matrix D');
WriteMatrix(D);
WriteLn;
ReadMatrix(E);
WriteLn('Matrix E');
WriteMatrix(E);
WriteLn;
MultMatrix(D, E, C);
WriteLn('Matrix C');
WriteMatrix(C);
WriteLn;
END. { MatrixProg }

```

Enter a value 7
 Enter a value 8
 Matrix B
 5.00
 6.00
 7.00
 8.00
 Matrix C
 6.00
 8.00
 10.00
 12.00
 Enter by rows
 Enter a value 0
 Enter a value 1
 Enter a value 2
 Enter a value 3
 Enter a value 4
 Enter a value 5
 Matrix D
 0.00
 1.00
 2.00
 3.00
 4.00
 5.00
 Enter by rows
 Enter a value 6
 Enter a value 7
 Enter a value 8
 Enter a value 9
 Enter a value 0
 Enter a value 1
 Matrix E

```
6.00
7.00

8.00
9.00

0.00
1.00

Matrix C

8.00
11.00

50.00
62.00
```

8.6 Sets in Pascal

A set is a collection of distinct items, all of the same type, with no duplication or significance in ordering. Pascal provides sets, along with arrays and records, as another way of structuring data. In Pascal, the members of a SET are chosen from some base-type which must be a subrange or an enumerated type. The maximum number of elements in a set is 256.

The declaration of sets takes the form:

```
TYPE set-name = SET OF base-type;
```

For example

```
TYPE CharSet = SET OF CHAR; { a set of characters }
DigitSet = SET OF 0..9; { a set of digits }
```

or in stages as

```
TYPE
  WeekDayType = (Sun, Mon, Tue, Wed, Thur, Fri, Sat);
  WeekDaySet = SET OF WeekDayType;
  UpperChar = 'A'..'Z';
  UpCharSet = SET OF UpperChar;
```

A SET constant is simply specified by enclosing constant values in square brackets as in [1, 3, 5, 7]. Assignment of such a set constant to a SET variable of the appropriate type follows the normal format of an assignment statement. For example, *WorkDays*, declared to be of type *WeekDaySet* could be assigned a value by:

```
WorkDays := [Mon, Tue, Wed, Thur, Fri];
```

Membership in a set is tested by the IN operator, which returns a BOOLEAN value. The combination *e* IN *S* is TRUE when element *e* is a member of set *S*. For example, if the set of vowel characters is assigned:


```
VowelSet := ['A', 'E', 'I', 'O', 'U'];
```

then a test for occurrence of a vowel could use this as named set as:

```
IF Ch IN VowelSet THEN
  Inc(VowelCount);
```

There are three operations on sets:

Intersection: $S1 * S2$
contains elements that are both in $S1$ and in $S2$.

Union: $S1 + S2$
contains elements that are either in $S1$ or in $S2$.

Difference: $S1 - S2$
contains elements that are in $S1$ but not in $S2$.

The following relational operators apply to sets:

Equality: $S1 = S2$
true if set $S1$ equals set $S2$

Inequality: $S1 <> S2$
true if set $S1$ is not equal to set $S2$

Subset: $S1 \leq S2$
true if set $S1$ is a subset of set $S2$

Superset: $S1 \geq S2$
true if set $S1$ is a superset of set $S2$

Note that the relational operators $<$ and $>$ do not apply to sets.

The operations of inclusion and exclusion of an element are performed using the above operations:

Inclusion: $S1 := S1 + [e]$

Exclusion: $S1 := S1 - [e]$

The program `SetsOfPeople`, shown in Figure 8.28, concerns a set of people, designated by two-letter names. It counts the number of elements in various sets, illustrating many of the above operations.

Figure 8.28 Program SetsOfPeople

```
PROGRAM SetsOfPeople;

TYPE PersonType = (AB, BO, RA, JO, MO, DE, ED, FA, KA, MA);
   PersonSet = SET OF PersonType;
VAR Tall, Old, Male, Female, Married,
    Rich, BlueEyed : PersonSet;
CONST Universe = [AB, BO, RA, JO, MO, DE, ED, FA, KA, MA];
    First = AB;
    Last = MA;

FUNCTION Size(group: PersonSet): INTEGER;
(* Counts the number of elements in group *)
VAR item: PersonType;
```

```
        count: INTEGER;
BEGIN
    count := 0;
    FOR item := First TO Last DO
        IF item IN group THEN
            Inc(count);
        Size := count;
    END; { Size }

BEGIN
    Male      := [AB,JO,ED,MO];
    Tall      := [AB,BO,JO,MO,MA,RA]; (* over 6 feet *)
    Old       := [AB,JO,KA,ED,FA,MA,RA]; (* over age 21 *)
    Rich      := [BO,KA,RA,JO,AB,MO]; (* millionaire *)
    Married   := [AB,KA,ED,MA];
    Female    := Universe - Male;
    Write('The total number of people is ');
    WriteLn(Size(Universe):2);
    Write('The number of tall males is ');
    WriteLn(Size(Tall * Male):2);
    Write('The number of tall females is ');
    WriteLn(Size(Tall * Female):2);
    Write('The number of tall or rich is ');
    WriteLn(Size(Tall + Rich):2);
    Rich := Rich + [RA];
    Write('The number of rich people is ');
    WriteLn(Size(Rich):2);
    IF Married <= Old THEN
        Write('All the married people are old');
    END. { SetsOfPeople }
```

The program is straightforward and easy to follow. Note how function `Size` operates. It checks all possible persons and increments the count for those who are in the set. Also note the use of an enumerated type loop control variable, `item`. The output from this program is the following.

```
The total number of people is 10
The number of tall males is 3
The number of tall females is 3
The number of tall or rich is 7
The number of rich people is 6
All the married people are old
```

More Sets (Optional)

In Chapter 8 of the Principles book, a simplified school timetable construction problem involving sets was discussed in considerable detail, and the pseudocode algorithms were developed. The following program shows the implementation of those algorithms in Pascal. It incorporates many of the techniques that were

discussed in the previous sections. As we mentioned before, it is a difficult problem, but a good deal can be learned by studying it carefully and understanding how it works.

During school registration, each student makes a selection of courses. The problem is to construct a timetable where certain courses are scheduled concurrently but subject to the constraint that every student's desired selection of courses can be taken without requiring a student to be in more than one class at a time. Program TimeTable, in Figure 8.29, accomplishes just that.

Figure 8.29 Program TimeTable

```
PROGRAM TimeTable;
(* Construct a time table so that students can take all the
   courses they chose without having to be in two courses
   at the same time *)
CONST NumberOfStudents = 5;
      NumberOfCourses  = 6;
      MaxScheduleSize  = 10;
TYPE CourseType  = 1..NumberOfCourses;
      StudentType = 1..NumberOfStudents;
      ScheduleType = 1..MaxScheduleSize;
      CourseSet    = SET OF CourseType;
      StudentArray = ARRAY [StudentType] OF CourseSet;
      CourseArray  = ARRAY [CourseType] OF CourseSet;
      SessionArray = RECORD
        NumberOfSessions: 0..MaxScheduleSize;
        SessionList: ARRAY [ScheduleType] OF CourseSet;
      END;
PROCEDURE ListCourseSet(GivenSet: CourseSet);
(* Output a given set of courses *)
VAR CourseIndex: CourseType;
    MemberOutput: BOOLEAN;
BEGIN
  Write('[');
  MemberOutput := FALSE;
  FOR CourseIndex := 1 TO NumberOfCourses DO
    IF CourseIndex IN GivenSet THEN BEGIN
      IF MemberOutput THEN
        Write(', ');
      Write(CourseIndex: 1);
      MemberOutput := TRUE;
    END;
  Write(']');
END; { ListCourseSet }

PROCEDURE BuildConflicts(Registration: StudentArray;
                        VAR ConflictList: CourseArray);
(* Build conflict sets showing for each course what
   courses cannot be run concurrently with the course *)
VAR StudentID: StudentType;
```

```
        CourseNum: CourseType;
BEGIN
    FOR CourseNum := 1 TO NumberOfCourses DO
        ConflictList[CourseNum] := [];
    FOR StudentID := 1 TO NumberOfStudents DO
        FOR CourseNum := 1 TO NumberOfCourses DO
            IF CourseNum IN Registration[StudentID]
                THEN ConflictList[CourseNum] :=
                    ConflictList[CourseNum] +
                    Registration[StudentID];
END; { BuildConflicts }

PROCEDURE NextPossibleSession(Remaining: CourseSet;
                               ConflictList: CourseArray;
                               VAR Session: CourseSet);
(* Find the next possible session that contains as
   many classes as possible that do not conflict *)
VAR CourseNum, TestCourse: CourseType;
    TrialSet: CourseSet;
BEGIN
    CourseNum := 1;
    WHILE NOT(CourseNum IN Remaining) DO
        Inc(CourseNum);
    Session := [CourseNum];
    TrialSet := Remaining - ConflictList[CourseNum];
    FOR TestCourse := 1 TO NumberOfCourses DO
        IF TestCourse IN TrialSet THEN
            IF (ConflictList[TestCourse] * Session) = []
                THEN Session := Session + [TestCourse];
END; { NextPossibleSession }
PROCEDURE BuildSchedule( ConflictList: CourseArray;
                          VAR Schedule: SessionArray);
(* Build the time table from the conflicts sets *)
VAR Remaining: CourseSet;
    Session: CourseSet;
BEGIN
    WITH Schedule DO BEGIN
        NumberOfSessions := 0;
        Remaining := [1..NumberOfCourses];
        WHILE Remaining <> [] DO BEGIN
            NextPossibleSession(Remaining,
                                ConflictList, Session);
            Remaining := Remaining - Session;
            Inc(NumberOfSessions);
            SessionList[NumberOfSessions] := Session;
        END;
    END;
END; { BuildSchedule }

VAR Students: StudentArray;
```

```

    CourseNum:   CourseType;
    Conflicts:   CourseArray;
    Timetable:   SessionArray;
    SessionNum:  1..MaxScheduleSize;
BEGIN
    { Set Registration data      }
    Students[1] := [1, 2];
    Students[2] := [2, 3];
    Students[3] := [2, 3, 4];
    Students[4] := [1, 5, 6];
    Students[5] := [3, 6];

    { Build and output Conflict list }
    BuildConflicts(Students, Conflicts);
    FOR CourseNum := 1 TO NumberOfCourses DO BEGIN
        Write('Conflicts[', CourseNum: 1, ''] = ');
        ListCourseSet(Conflicts[CourseNum]);
        WriteLn;
    END;
    WriteLn;

    { Build and output Timetable }
    BuildSchedule(Conflicts, TimeTable);
    WITH TimeTable DO
        FOR SessionNum := 1 TO NumberOfSessions DO BEGIN
            Write('Session[', SessionNum: 2, ''] = ');
            ListCourseSet(SessionList[SessionNum]);
            WriteLn;
        END;
    END. { TimeTable }

```

The data obtained from registration can be represented as an array of sets of courses, *Students*. Here, we've simplified the problem by restricting the number of students to 5, and the number of possible courses to 6. The first part of the body of the program sets up the "registration data", which corresponds to the choices of Figure 8.30.

Figure 8.30 Registration data for the TimeTable program

Student	Set of chosen courses
1	{1, 2}
2	{2, 3}
3	{2, 3, 4}
4	{1, 5, 6}
5	{3, 6}

The next step is to build a conflict list, which shows for each course the set of courses that cannot be run concurrently with that course, because there is at least one student that wants to take both. The conflict list is built by the procedure `BuildConflicts`, in which the element `ConflictList[i]` is the set of courses with which course *i* conflicts. Courses conflict with course *i* because one or more students have also selected those courses, and `ConflictList[i]` is thus the set of courses that cannot be scheduled concurrently with course *i*. Thus, for example, courses 1, 2, 5 and 6 all conflict and cannot be run concurrently—because Student 1 has chosen courses 1 and 2, and Student 4 has chosen courses 1, 5 and 6. The procedure first sets all conflict sets to empty. Then for each student it examines all the courses, and add those chosen by the student to the conflict list.

The conflict list is then output so that it can be checked. The time table consists of a number of “sessions”, sets of courses that can be given simultaneously because no two students want to attend both. Since the number of sessions is unknown at first, the time table is implemented as an array of sets of courses that constitute the sessions, whose size is specified by constant `MaxScheduleSize`, together with a count of the number of sessions required, and grouped in a `RECORD`. The time table is built by procedure `BuildSchedule` which starts by putting all the courses in `Remaining`, and repeatedly getting a session by calling `NextPossibleSession` and decreasing `Remaining` until it is empty.

`NextPossibleSession` finds the next session containing as many classes that do not conflict as possible. For the first course in `Remaining` we create a `TrialSet` by removing its conflict set from `Remaining`. Then, other courses from `Remaining` that do not conflict with any of the courses already in `TrialSet` are added to `TrialSet`, until no more courses can be added.

After the timetable has been constructed, it is output. The output obtained from running this program is the following.

```
Conflicts[1] = [1, 2, 5, 6]
Conflicts[2] = [1, 2, 3, 4]
Conflicts[3] = [2, 3, 4, 6]
Conflicts[4] = [2, 3, 4]
Conflicts[5] = [1, 5, 6]
Conflicts[6] = [1, 3, 5, 6]

Session[ 1] = [1, 3]
Session[ 2] = [2, 5]
Session[ 3] = [4, 6]
```

8.7 Dynamic Variables and Pointers in Pascal

The concepts of pointers and dynamic variables were introduced in Chapter 8 of the Principles book. The idea of a dynamic variable is that it is created during execution, and it is referenced through a pointer variable, which stores the location of the dynamic variable.

A list whose length is not known before execution can be built by creating each of the elements of the list during execution, and linking them together through pointers. The algorithm developed in Chapter 8 of the Principles book for the creation of a list repeatedly allocates a dynamic variable, connects it to the last one in the list, inputs information and stores it in that variable.

In Pascal, a pointer type is defined by the pointer symbol `^` followed by the type of the dynamic variables that can be referenced by pointer variables of this type. Suppose we want to construct a linked list of the sort created by the algorithm mentioned above, where each element would consist of a single `INTEGER` variable, which contains the element's value, and a pointer to the next element in the list. The last pointer in the chain would have the special value `NIL`, which points to nothing, and can be tested for. The following type declarations will give us the types that we need to create such a list.

```
TYPE ListPointer = ^ListElement;
   ListElement = RECORD
                       Value: INTEGER;
                       Next:  ListPointer;
                   END;
```

We also define pointer variables through which we can reference elements of the list with the following declarations.

```
VAR List1, Current: ListPointer;
```

Before it makes any sense to use either of these variables, we must give them a value by creating a `ListElement`. This is done through a call to the standard Pascal procedure `New`. The procedure has one parameter, which must be a pointer to dynamic variables of the type we want to create. Thus a new `ListElement` can be created by

```
New(List1);
```

which has the effect of allocating sufficient storage for a `ListElement`, and setting the value of `List1` to point to it. Once this is done, we can use `List1` to reference the fields of the newly created dynamic variable. `List1` is a pointer to the dynamic variable that was just created, while `List1^` is the dynamic variable, i.e. the element pointed to by `List1`. Since `List1^` is a record, the references to its two fields are written:

```
List1^.Value
```

and

```
List1^.Next
```

Thus, the `Value` field in that dynamic variable can be assigned the value 3 by

```
List1^.Value := 3;
```

The only operations that are permitted on pointer values are assignment, and the comparisons based on the two relational operators `=` and `<>`.

The first part of the program `CreateList`, shown in Figure 8.31, is an implementation of the pseudocode algorithm found in Chapter 8 of the Principles book.

Figure 8.31 Program CreateList

```
PROGRAM CreateList;
(* Create a dynamic list *)
CONST NumberOfElements = 5;

TYPE ListPointer = ^ListElement;
    ListElement = RECORD
        Value: INTEGER;
        Next: ListPointer;
    END;
VAR List1, Current: ListPointer;
    Index: INTEGER;

BEGIN
    New(List1);
    Current := List1;
    Write('Enter an integer ');
    Read(Current^.Value);
    FOR Index := 2 TO NumberOfElements DO BEGIN
        New(Current^.Next);
        Write('Enter an integer ');
        Read(Current^.Next^.Value);
        Current      := Current^.Next;
        Current^.Next := NIL;
    END;

    Write('List of elements: ');
    Current := List1;
    WHILE Current <> NIL DO BEGIN
        Write(Current^.Value:3);
        Current := Current^.Next;
    END;
    WriteLn;
END. { CreateList }
```

After creating the first element by the call `New(List1)`, other elements are added and connected to the previous element by the call `New(Current^.Next)`, and the list is created as shown in Figure 8.40 of the Principles book. The second part of the `CreateList` program outputs the values stored in the list, as seen in this output from a typical run:

```
Enter an integer 12
Enter an integer 23
Enter an integer 34
Enter an integer 45
Enter an integer 56
List of elements:  12 23 34 45 56
```


8.8 Chapter 8 Review

The data structures discussed in this chapter are the three structures: arrays, records, and sets. These structures are used to define new types that are illustrated by many Pascal examples.

Arrays, especially those of one dimension, are covered in detail. Arrays of two or more dimensions are treated in less detail, but with sufficient examples to show how they can be used.

Records, nested records, arrays of records and records containing arrays are also covered with many examples. Variant records are also considered briefly.

Sets and operations on sets are illustrated in two example programs, involving counting the elements in various sub-sets, and the not so simple construction of simple school time tables.

The chapter also introduced a number of libraries including a general library for integer arrays, `IntArrayLib`, and implementations of abstract data types like `ComplexLib` and `MatrixLib`.

The use of dynamic variables and the way in which they are referenced through pointer variables is described and illustrated with the example of the creation of a dynamic list. Later, in following courses on data structures, more details on such objects and others will be considered.

8.9 Chapter 8 Problems

1. Normalize

Create part of a program to convert a list of event occurrence counts, `NumEvents`, into a list of probabilities, by summing the values and then dividing each count by this sum.

For example, when two dice are thrown, the results range from 2 to 12 dots. The frequencies of each of these results can be observed and tabulated to compare the particular dice throws to ideal dice (whose probabilities should be: $\frac{1}{36}, \frac{2}{36}, \dots, \frac{6}{36}, \dots, \frac{2}{36}, \frac{1}{36}$).

2. Horizontal Histogram

Create part of a program to enter any number of values (say percentages) and to plot these out horizontally as bars that have a length proportional to the values input. The size of the bars may need to “grow” by some `Factor`. For example, the above frequencies of dice throws could be used as follows.

```
Enter the number of bars      11
Enter the growth factor      3
```

```

Enter values of the bars
1 2 3 4 5 6 5 4 3 2 1

```

```

***
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****

```

3. Intersection

Given two `INTEGER` arrays A, and B, create a third array C, having the values that are common to both of the arrays A and B.

4. KBIG

Create a program to find the Kth largest value of an array A, of N different values.

Hint: the fifth largest value has 4 items larger than itself.

8.10 Chapter 8 Programming Projects

Gas Project

The fuel consumption of a vehicle is to be analyzed. Each time fuel is obtained, the tank is filled and a record is made of the date, present mileage, gallons used, total cost and brand name as shown:

<u>Date</u>	<u>Miles</u>	<u>Gallons</u>	<u>Cost</u>	<u>Brand</u>
93/11/7	50123	10.2	\$10.00	X
93/11/8	50432	15.3	\$15.00	TURBO
93/11/11	50732	13.0	\$14.50	TEXAN
93/11/11	51001	12.5	\$16.24	GULP

etc.

This data is to be analyzed in the following ways.

1. Compute the overall miles per gallon (MPG).
2. Compute the MPG at each refill.
3. Determine the cumulative MPG at each fill up by dividing the mileage to fill up, by the accumulated gallons.
4. Compute the running average over the 3 previous refills.
5. Compute the maximum MPG and the minimum MPG.
6. Determine which brand produces the maximum MPG between fill ups.
7. Compute the MPG for each brand.
8. Determine which brand produces the minimum cost per mile.
9. Determine which brand has the minimum cost per gallon.
10. Compute the average fuel cost per day traveled.
11. Determine the days of minimum mileage.
12. For each brand, compute the total number of gallons used, and the overall average MPG, cost per gallon.
13. Convert some of the above to other units (kilometers per liter, marks per kilometer, liters per 100 kilometers).
14. Produce plots of some of the above results.

Library Projects

1. `IntArrayLib2`
Create another `IntArrayLib` where a terminating value is stored in the first position of the array and also in the last position.

2. **Complete ComplexLib**
Complete the Complex Library by providing procedures for Division, Subtraction, Conjugation, and functions to provide the Magnitude and Angle of the complex numbers.
3. **New DateLib**
Re implement the DateLib of Chapter 7 using a Date-type that is a record.
4. **AccountLib**
Create a library of Accounts (in a bank, organization, etc.) where each account is a record consisting of an IdNumber, a Balance, and various transactions of Withdrawal and Deposit.

BSL: Big Stat Lab

In this lab we will reuse library `IntArrayLib`, which operates on a list of `INTEGER` values as shown in this chapter. You will create some of the following additional procedures involving arrays, and use them and reuse them to do some of the statistical operations.

1. `MaxIntSeq(S, M, P)` is a procedure that computes the maximum value `M` of a sequence `S` of Integers and also returns the position `P` of this maximum value.
2. `SelectSort(S)` is a procedure that sorts the values of `S` into decreasing order. Use `MaxIntSeq` to create `SelectSort`.
3. `MidIntSeq(S)` is a function that returns the middle value of a sequence `L`; if the sequence has an even number of items then the middle returned is the average of the two middle values.
4. `CountIntSeq(S, V, C)` is a procedure that counts the number of occurrences `C` of the value `V` in the sequence `S`. For example, it could count the number of values of zero rainfall, or the number of days that had the maximum rainfall.
5. `MeanIntSeq(S)` is a function procedure that returns the average value of any sequence `S`.
6. `Variance(S)` is a function that returns the amount of deviation from the mean value of a sequence. The algorithm for this is shown earlier in this chapter; make use of `MeanIntSeq`.

Write a program that uses all these procedures. It first prompts a user to enter a sequence and then presents a menu that requests an action in the form:

```
Enter an action:
  A: to determine the Average value of a sequence
  C: to Count the occurrences in a sequence
  L: to find the Largest value of a sequence
  M: to find the Middle value of a sequence
  S: to Sort the sequence into decreasing order
  V: to compute the Variance
  E: to Exit this menu
```

Enter a character

Time Permitting:

Put these procedures (including a Menu) into a Library, called StatLib and use this Library and IntArrayLib in a program called StatProg. Also the data involved should be stored in files.

Chapter 9 Algorithms to Run With

The primary purpose of this chapter is to provide more extensive examples of the data structures introduced in the preceding chapter. Algorithms to sort and search data structures will be discussed, along with the various ways of implementing stacks, queues, and trees. Also, this chapter will further develop the concept of an “Abstract Data Type” (or ADT) and create the StackLib, QueueLib, and SetLib libraries.

Chapter Overview

9.1	Preview.....	368
9.2	Sorting Algorithms.....	369
	A Context for Sorting.....	369
	Count Sort.....	373
	Bubble Sort.....	375
	Select Sort.....	376
9.3	Improving sorts (Optional).....	377
	Recursive Sort: Merge Sort.....	378
	Another Merge Sort: the Von Neumann Sort.....	380
9.4	Searching.....	387
	Binary Search.....	387
9.5	Implementing Stacks and Queues.....	391
	StackLib: Stack as an Abstract Data Type.....	391
	Dynamic Stacks.....	395
	QueueLib.....	397
	Big Cardinals.....	400
	SetLib: Stack of Strings.....	404
9.6	Trees.....	407
9.7	Chapter 9 Review.....	412
9.8	Chapter 9 Problems.....	412
9.9	Chapter 9 Programming Projects.....	415
	Queue Abstract Data Type.....	415
	PES: Performance Evaluation of Sorts.....	416
	VS: Visual Sorts.....	418

9.1 Preview

The primary purpose of this chapter is to provide more extensive examples of the use of the data structures introduced in the preceding chapter. Essentially, no new features of Pascal are introduced, and most of the algorithms shown here are discussed in detail in Chapter 9 of the Principles book. For this reason, many of the examples are presented with only minimal commentary. The Pascal implementations of the algorithms have been written to make them as readable as possible and hence much of the text of this chapter appears in the form of Pascal programs.

The first two tasks studied: sorting—rearranging the data into a specific sequence, e.g. alphabetic order—and searching—information retrieval—are concerned with arrays. However, before any of the sorting or searching algorithms are presented, we need some tools to help us. The first section is devoted to the development of an environment that contains procedures that create data to work with and to test that they are correctly sorted.

As is common with many tasks, there are several ways to perform sorting and searching. For example, although there are only four basic ways to sort an array, there are dozens of variations of each of these ways. Only a few of the variations will be considered here, however, their individual advantages and disadvantages are discussed.

As another example of the use of data structures, various ways of implementing stacks, queues and trees are also introduced. Some of these techniques are based on arrays, possibly of records, as the underlying data structure, while others are based on dynamic variables and pointers. Many of the examples are based on arrays, which are not only used to hold the data for the sorting algorithms but are also used to create other structures, such as stacks. The structures in turn will be “encapsulated” in a library and used as basic building blocks to make even more complex systems.

The concept of an “Abstract Data Type” (or ADT) will be developed further and used to create libraries. Libraries created in this chapter include a Stack Library, a Queue Library and a Set Library.

Probably the most important capability demonstrated in this chapter is the idea of creating larger data items from smaller ones. For example, once we have developed an integer array library, we are able to develop a library of routines for the manipulation of stacks based on the routines available in the array library. These libraries demonstrate the utility of creating an Abstract Data Type through the library mechanism introduced in the previous chapters.

9.2 Sorting Algorithms

A Context for Sorting

Before we can experiment with the many different algorithms for sorting, we need some data to work with. If each time we run the program, we have to enter all the test data through the keyboard, algorithm development will be very slow and tedious and the sizes of the sets of test data will be small. If we are to be able to get any idea of relative efficiency of algorithms, we need large sets of data. The trouble with large sets of test data is that we can no longer check by eye that they have been correctly processed. Therefore, we also need procedures that will check that the data have been correctly sorted. The library in Figure 9.1, `SortEnvLib`, is not only useful for our present discussion of sorting algorithms but it also provides a model of the kinds of libraries that are needed whenever a programming project of any size is undertaken.

Figure 9.1 The `SortEnvLib` library

```

UNIT SortEnvLib;
(* Sorting environment tools *)
INTERFACE

    CONST MaxData = 100;

    TYPE DataArrayType = ARRAY [1..MaxData] OF INTEGER;
       SequenceOrder = (Up, Down);

    FUNCTION  RandInt(): INTEGER;
    (* Generate random integer < 1013 *)
    PROCEDURE SetSeed(NewSeed: INTEGER);
    (* Set seed of random number generator *)
    PROCEDURE RandomData(VAR DataList: DataArrayType);
    (* Fill DataList with random integers *)
    PROCEDURE ListData(DataList: DataArrayType);
    (* List data from DataList 10 values per line *)
    FUNCTION  IsMonotonic(Data: DataArrayType;
        Direction: SequenceOrder)
        : BOOLEAN;
    (* Check if data in DataList is ordered *)

IMPLEMENTATION
    CONST InitialSeed = 777;
    VAR Seed: INTEGER;

    PROCEDURE SetSeed(NewSeed: INTEGER);
    BEGIN
        Seed := NewSeed;

```

```
END; { SetSeed }

FUNCTION RandInt(): INTEGER;
CONST Modu = 1013;
      Coef = 28;
BEGIN
  Seed := (Seed * Coef) MOD Modu;
  RandInt := Seed;
END; { RandInt }

PROCEDURE RandomData(VAR DataList: DataArrayType);
VAR Index: INTEGER;
BEGIN
  FOR Index := 1 TO MaxData DO
    DataList[Index] := RandInt;
  END; { RandomData }

PROCEDURE ListData(DataList: DataArrayType);
CONST ElementsPerRow = 10;
VAR Index, ElemCount: INTEGER;
BEGIN
  ElemCount := 0;
  FOR Index := 1 TO MaxData DO BEGIN
    Write(DataList[Index]: 5);
    Inc(ElemCount);
    IF ElemCount >= ElementsPerRow THEN BEGIN
      WriteLn;
      ElemCount := 0;
    END;
  END;
  IF ElemCount <> 0 THEN
    WriteLn;
END; { ListData }

FUNCTION IsMonotonic(Data: DataArrayType;
                    Direction: SequenceOrder)
: BOOLEAN;
VAR Index: INTEGER;
    InOrder: BOOLEAN;
BEGIN
  InOrder := TRUE;
  Index := 2;
  WHILE (Index <= MaxData) AND InOrder DO
    IF Direction = Up THEN
      IF Data[Index - 1] > Data[Index] THEN
        InOrder := FALSE
      ELSE
        Inc(Index)
      ELSE

```

```

        IF Data[Index - 1] < Data[Index] THEN
            InOrder := FALSE
        ELSE
            Inc(Index);
            IsMonotonic := InOrder;
        END; { IsMonotonic }

BEGIN
    Seed := InitialSeed;
END. { SortEnvLib }

```

The Pascal UNIT `SortEnvLib`, shown in Figure 9.1, defines an array type, `DataArrayType`, that is used to store INTEGER data to be used for testing sorting algorithms. The size of the array is set by the value of constant `MaxData`, which is currently set to 100, but could easily be changed. The library also provides five functions and procedures that are useful in creating test data and checking the results of the sorting algorithms.

RandInt

An INTEGER function that returns a sequence of apparently random numbers; it is defined here to give an example of random number generation. Remember, Pascal offers the standard function `Random`.

SetSeed (NewSeed)

This procedure sets a new starting point for the random number sequence produced by `RandInt`.

RandomData (DataList)

Fills the given `DataList` with a set of random numbers produced by `RandInt`.

ListData (DataList)

Lists the values contained in `DataList`, ten values per line.

IsMonotonic (DataList, Direction)

A BOOLEAN function that checks that the data in the `DataList` is monotonic, either all increasing or decreasing, the direction specified by the parameter `Direction` to be either `Up` or `Down`. This function is used to check that the data have been sorted correctly—much more reliably than visually scanning the output produced by `ListData`.

Figure 9.2 Program TestSortEnv

```

PROGRAM TestSortEnv;
USES SortEnvLib;
VAR Data: DataArrayType;
    Count: INTEGER;
BEGIN
    RandomData(Data);
    WriteLn('With default seed');
    ListData(Data);
    WriteLn;
    SetSeed(527);

```

```
RandomData(Data);
WriteLn('With new seed');
ListData(Data);
FOR Count := 1 TO MaxData DO
    Data[Count] := Count;
WriteLn;
IF IsMonotonic(Data, Up) THEN
    WriteLn('IsMonotonic passed test 1')
ELSE
    WriteLn('IsMonotonic failed test 1');
Data[27] := 67;
IF IsMonotonic(Data, Up) THEN
    WriteLn('IsMonotonic failed test 2')
ELSE
    WriteLn('IsMonotonic passed test 2');
END. { TestSortEnv }
```

Program TestSortEnv, in Figure 9.2, tests the operation of the procedures and functions of SortEnvLib. The output obtained from running it is the following.

With default seed

483	355	823	758	964	654	78	158	372	286
917	351	711	661	274	581	60	667	442	220
82	270	469	976	990	369	202	591	340	403
141	909	127	517	294	128	545	65	807	310
576	933	799	86	382	566	653	50	387	706
521	406	225	222	138	825	814	506	999	621
167	624	251	950	262	245	782	623	223	166
596	480	271	497	747	656	134	713	717	829
926	603	676	694	185	115	181	3	84	326
11	308	520	378	454	556	373	314	688	17

With new seed

574	877	244	754	852	557	401	85	354	795
987	285	889	580	32	896	776	455	584	144
993	453	528	602	648	923	519	350	683	890
608	816	562	541	966	710	633	503	915	295
156	316	744	572	821	702	409	309	548	149
120	321	884	440	164	540	938	939	967	738
404	169	680	806	282	805	254	21	588	256
77	130	601	620	139	853	585	172	764	119
293	100	774	399	29	812	450	444	276	637
615	1012	985	229	334	235	502	887	524	490

IsMonotonic passed test 1

IsMonotonic passed test 2

Count Sort

The simplest sort algorithm is Count Sort, which finds the rank of all N values in an array; if all values are different the largest value has a rank of 1, the smallest has a rank of N . If some of the data values are the same, they will have the same rank and there will be some ranks missing. The algorithm is simple: for each value it counts the number of values that are greater than or equal to it, and stores that in the corresponding rank array. Figure 9.3 shows the implementation of this sorting algorithm as procedure CountSort in testing program TestCountSort.

Figure 9.3 Program TestCountSort

```
PROGRAM TestCountSort;
USES SortEnvLib;

CONST ElementsPerRow = 10;

VAR Data, Rank: DataArrayType;
    Index, IndRank, ElemCount: INTEGER;

PROCEDURE CountSort( Data: DataArrayType;
                     VAR Rank: DataArrayType);
VAR Count, Pass, Index, CurrentValue: INTEGER;
BEGIN
    FOR Pass := 1 TO MaxData DO BEGIN
        CurrentValue := Data[Pass];
        Count := 0;
        FOR Index := 1 TO MaxData DO
            IF Data[Index] >= CurrentValue THEN
                Inc(Count);
        Rank[Pass] := Count;
    END;
END; { CountSort }

BEGIN
    RandomData(Data);
    WriteLn('Original data');
    ListData(Data);
    WriteLn;
    CountSort(Data, Rank);
    WriteLn('Ranking data');
    ListData(Rank);
    WriteLn;
    WriteLn('Sorted data');
    ElemCount := 0;
    FOR Index := 1 TO MaxData DO BEGIN
        IndRank := 1;
        WHILE Index <> Rank[IndRank] DO
            { find element whose rank is Index }
            Inc(IndRank);
```

```

Write(Data[IndRank]:5);
Inc(ElemCount);
IF ElemCount >= ElementsPerRow THEN BEGIN
    WriteLn;
    ElemCount := 0;
END;
END;
IF ElemCount <> 0 THEN
    WriteLn;

END. { TestCountSort }

```

The body of the program makes use of the procedures of `SortEnvLib` to produce test data and list the rankings. However, for that kind of sort, a list of the rankings is not very useful to check that the values are in order. For that reason, we have had to develop and add a small algorithm to display the array values in the order given by the rankings. We look for the index value in the `Rank` array and, when found, output the corresponding `Data` array value. With this addition, the output from running this program is as shown in Figure 9.4.

Figure 9.4 Output from program `TestCountSort`

Original data

483	355	823	758	964	654	78	158	372	286
917	351	711	661	274	581	60	667	442	220
82	270	469	976	990	369	202	591	340	403
141	909	127	517	294	128	545	65	807	310
576	933	799	86	382	566	653	50	387	706
521	406	225	222	138	825	814	506	999	621
167	624	251	950	262	245	782	623	223	166
596	480	271	497	747	656	134	713	717	829
926	603	676	694	185	115	181	3	84	326
11	308	520	378	454	556	373	314	688	17

Ranking data

47	60	12	17	4	29	94	84	58	68
8	61	21	27	69	37	96	26	51	78
93	71	49	3	2	59	79	36	62	53
85	9	89	44	67	88	41	95	14	65
38	6	15	91	55	39	30	97	54	22
42	52	75	77	86	11	13	45	1	33
82	31	73	5	72	74	16	32	76	83
35	48	70	46	18	28	87	20	19	10
7	34	25	23	80	90	81	100	92	63
99	66	43	56	50	40	57	64	24	98

Sorted data

999	990	976	964	950	933	926	917	909	829
825	823	814	807	799	782	758	747	717	713

711	706	694	688	676	667	661	656	654	653
624	623	621	603	596	591	581	576	566	556
545	521	520	517	506	497	483	480	469	454
442	406	403	387	382	378	373	372	369	355
351	340	326	314	310	308	294	286	274	271
270	262	251	245	225	223	222	220	202	185
181	167	166	158	141	138	134	128	127	115
86	84	82	78	65	60	50	17	11	3

Bubble Sort

Bubble Sort is probably the simplest of the actual sort routines where the data are rearranged. Its simplest version should become so well known to you, that you can write the program for doing it without having to refer to any notes. Its Pascal implementation is shown in Figure 9.5: all adjacent values are compared and swapped if not in the right order.

Figure 9.5 Procedure BubbleSort

```
PROCEDURE BubbleSort(VAR Data: DataArrayType);
VAR Pass, Index, Temp: INTEGER;
BEGIN
  FOR Pass := 1 TO (MaxData - 1) DO
    FOR Index := 1 TO (MaxData - 1) DO
      IF Data[Index] < Data[Index + 1] THEN BEGIN
        Temp := Data[Index];
        Data[Index] := Data[Index + 1];
        Data[Index + 1] := Temp;
      END;
    END;
  END; { BubbleSort }
```

In this form, the algorithm is quite adequate for sorting small amounts of data, up to about 100 items, without enough loss of efficiency to merit the effort of either finding, or writing one of the more complicated algorithms. One of the exercises given at the end of this chapter investigates this point.

A slightly more complicated version of this simple BubbleSort algorithm is contained in the program TestBubbleSort, which is in Figure 9.6.

Figure 9.6 Program TestBubbleSort

```
PROGRAM TestBubbleSort;
USES SortEnvLib;

VAR Data: DataArrayType;

PROCEDURE BubbleSort(VAR Data: DataArrayType);
VAR Posn, Temp: INTEGER;
    Done: BOOLEAN;
BEGIN
```

```
        Done := FALSE;
        WHILE NOT Done DO BEGIN
            Done := TRUE;
            FOR Posn := 1 TO MaxData - 1 DO
                IF Data[Posn] < Data[Posn + 1] THEN BEGIN
                    Temp          := Data[Posn];
                    Data[Posn]    := Data[Posn+1];
                    Data[Posn + 1] := Temp;
                    Done          := FALSE;
                END;
            END;
        END; { BubbleSort }

BEGIN
    RandomData(Data);
    WriteLn('Original data');
    ListData(Data);
    WriteLn;
    BubbleSort(Data);
    WriteLn('Sorted data');
    ListData(Data);
    WriteLn;
    IF IsMonotonic(Data, Down) THEN
        WriteLn('Data are correctly sorted')
    ELSE
        WriteLn('Data are NOT correctly sorted');
END. { TestBubbleSort }
```

In this version of the algorithm, the idea is to detect when a complete pass is made through the data without finding anything out of order. When this occurs, Done has the value TRUE and the algorithm terminates early.

Select Sort

The group of select sort algorithms have the general principle that, at each pass, an extreme value of the data still to be sorted is selected and moved to its proper place in the target array. In this version of the algorithm, the assumption is made that all the data values are greater than zero. At each pass, the maximum value in the data array is selected, copied to the target area and replaced by zero in the data array so that it will no longer be considered.

A Pascal procedure that performs this algorithm is given in Figure 9.7. It is based on the same assumption, because it replaces the chosen value by a zero.

Figure 9.7 Procedure SelectSort

```
PROCEDURE SelectSort(    Data: DataArrayType;
                        VAR Result: DataArrayType);
VAR Pass, Index, Maximum, Position: INTEGER;
```

```

BEGIN
  FOR Pass := 1 TO MaxData DO BEGIN
    Maximum := 0;
    FOR Index := 1 TO MaxData DO
      IF Maximum < Data[Index] THEN BEGIN
        Maximum := Data[Index];
        Position := Index;
      END;
    Result[Pass] := Maximum;
    Data[Position] := 0;
  END;
END; { SelectSort }

```

Note that this procedure has two parameters, the original array as input parameter, and the sorted array as output parameter.

9.3 Improving sorts (Optional)

The DistantBubbleSort algorithm that was introduced in Chapter 9 of the Principles book is a major improvement on BubbleSort. Instead of comparing adjacent values, values that are situated at a given distance apart are compared. The idea is that when two distant values are swapped, this possibly saves doing the same thing by several individual swaps. The algorithm for doing this is not very different from the BubbleSort algorithm.

If this DistantBubbleSort algorithm is applied repeatedly with ever decreasing values for the distance apart, the whole data array will eventually be sorted. This method of using DistantBubbleSort in this way was first proposed by David Shell and is generally known as “Shell Sort”. A Pascal implementation of this algorithm is contained in the procedure ShellSort shown in Figure 9.8.

Figure 9.8 Procedure ShellSort

```

PROCEDURE ShellSort(VAR Data: DataArrayType);
VAR Distance: INTEGER;

  PROCEDURE DistantBubbleSort(      Distance: INTEGER;
                                VAR Data: DataArrayType);
VAR Index, Pass, Start, Temp: INTEGER;
    Finished: BOOLEAN;
BEGIN
  Finished := FALSE;
  Pass := 1;
  WHILE (Pass < MaxData) AND NOT Finished DO BEGIN
    Finished := TRUE;
    FOR Start := 1 TO Distance DO BEGIN
      Index := Start;
      WHILE Index <= MaxData - Distance DO BEGIN
        IF Data[Index] < Data[Index + Distance] THEN

```

```
BEGIN
    Finished      := FALSE;
    Temp          := Data[Index];
    Data[Index]   := Data[Index + Distance];
    Data[Index + Distance] := Temp;
    END;
    Inc(Index, Distance);
    END; { WHILE }
    END; { FOR }
    Inc(Pass);
    END; { WHILE }
    END; { DistantBubbleSort }

    BEGIN { ShellSort }
        Distance := MaxData DIV 2;
        WHILE Distance >= 1 DO BEGIN
            DistantBubbleSort(Distance, Data);
            Distance := Distance DIV 2;
        END;
    END; { ShellSort }
```

In this procedure, the initial value for `Distance` is half the number of data items in the list, and this is halved at every iteration. A thorough analysis of the improvement obtained by this technique is well beyond the scope of this book. Look at the `DistantBubbleSort` procedure and compare it with the `BubbleSort` procedure of Figure 9.6. There are differences, and the amount of detail is larger in `DistantBubbleSort`, but on the whole the algorithms are also very similar.

Recursive Sort: Merge Sort

As an example of a recursive procedure, `Merge Sort` was introduced in Chapter 9 of the *Principles* book. Here, sorting is achieved by splitting the array into two halves, sorting each half, and then merging the two sorted halves. Of course, each of the two halves is sorted using the same process, and this is repeated until each half has only a single element (base case).

The merging of the two subarrays is done simply with an index “sliding down” each subarray. For each value of the index the two indexed values are compared, and their maximum is copied into the array `Result`. The index of the copied value is incremented and the process is continued until one of the two subarrays has been entirely copied. The remaining elements in the other subarray are then copied in the resulting array.

The detailed working of this algorithm is a little difficult to understand at first and is fully explained in Chapter 9 of the *Principles* book.

Figure 9.9 Procedure MergeSort

```
PROCEDURE MergeSort( First, Last: INTEGER;
                    VAR Table: DataArrayType);
```

```

PROCEDURE Merge(  First, Last: INTEGER;
                  VAR Table: DataArrayType);
VAR Index, Middle, Bottom, Top: INTEGER;
    Result: DataArrayType;
BEGIN
    Index  := First;
    Middle := (First + Last) DIV 2;
    Top    := First;
    Bottom := Middle + 1;
    WHILE (Top <= Middle) AND (Bottom <= Last) DO BEGIN
        IF Table[Top] > Table[Bottom] THEN BEGIN
            Result[Index] := Table[Top];
            Inc(Top);
        END
        ELSE BEGIN
            Result[Index] := Table[Bottom];
            Inc(Bottom);
        END;
        Inc(Index);
    END;
    WHILE Top <= Middle DO BEGIN
        Result[Index] := Table[Top];
        Inc(Top);
        Inc(Index);
    END;
    WHILE Bottom <= Last DO BEGIN
        Result[Index] := Table[Bottom];
        Inc(Bottom);
        Inc(Index);
    END;
    Index := 1;
    FOR Index := First TO Last DO
        Data[Index] := Result[Index];
END; { Merge }

VAR Middle: INTEGER;
BEGIN { MergeSort }
    IF First <> Last THEN BEGIN
        Middle := (First + Last) DIV 2;
        MergeSort(First, Middle, Table);
        MergeSort(Middle + 1, Last, Table);
        Merge(First, Last, Table);
    END;
END; { MergeSort }

```

The Pascal implementation of this algorithm is shown in Figure 9.9 which shows procedure MergeSort, which contains Merge as an internal procedure. The fact that it is a recursive procedure requires no special indication in the Pascal program.

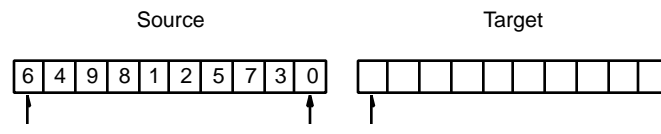
Another Merge Sort: the Von Neumann Sort

Let's now look at another sorting algorithm based upon the idea of merging but, this time, without recursion. This will allow us to introduce a couple of new features of Pascal. This algorithm was originally developed on some ideas by John von Neumann, the inventor of the stored program concept, and so we will name it Von Neumann Sort.

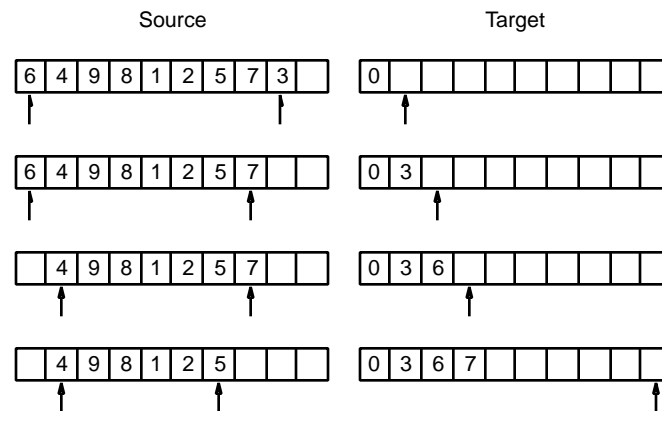
The Von Neumann Sort makes use of an auxiliary storage array of the same size as the original data array. During each major iteration, the data are copied from one data area, the "source", to the other, the "target", the direction of copy reversing between iterations. Thus, during the first iteration, the data are copied from the original data area, the "source", to the auxiliary area, which is the "target". During the second iteration, the roles of the two areas are reversed and the auxiliary area is the source, and the original data area is the target.

We can imagine each data array to be laid out as a row of cells from left to right with element 1 at the left and element n at the right. In the source area, the data are treated as two ascending sequences one starting at the left and going to the right and the other starting at the right and going to the left. The diagram of Figure 9.10 represents the state at the beginning of an example sort of ten digits.

Figure 9.10 Start of a Von Neumann sort

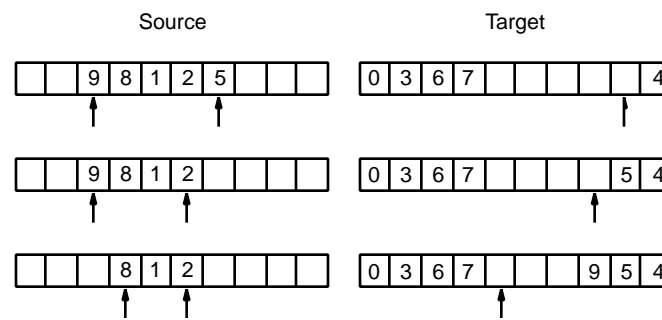


The two arrows in the source area mark the beginnings of the data sequences to be merged, and the arrow in the target area marks the cell into which the first element will be copied. Merging continues until the two next available data elements in the source are both less than the last copied element in the target. The four diagrams of Figure 9.11 show successive steps in the merging process up to the end of the first sequence.

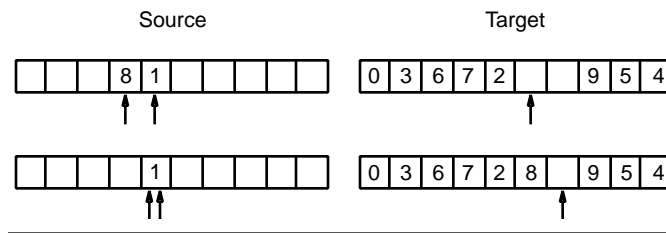
Figure 9.11 The Von Neumann sort

The building of the first sequence stops because the next two available items in the source area, 4 and 5, are both less than the last item of the sequence that has been built in the target area, 7. Although the data is really copied, for clarity, the diagrams show it as being moved. At this point, the next available elements are treated as the beginnings of new sequences and merging continues, forming a sequence that is built at the opposite end of the target area. In the last diagram of Figure 9.11, the arrow in the target area has been moved to show where the next sequence will be put.

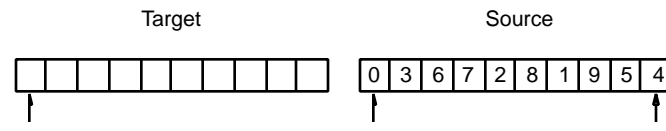
This merging continues until the end of the next sequence is reached, as shown in the three diagrams of Figure 9.12. As with the first sequence, when no more merging is possible, a new sequence is started at the other end of the target area as shown by the arrow.

Figure 9.12 Von Neumann sort

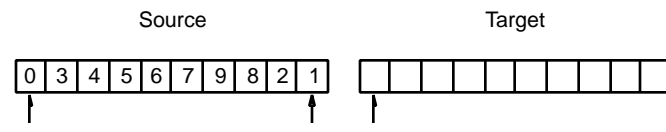
Sequences are formed in the target area in this way until all the data have been copied, at which point the source and target designations are interchanged. This is shown in the three diagrams of Figure 9.13.

Figure 9.13 End of Von Neumann sort

In the example, four sequences, (0, 3, 9, 7), (4, 5, 9), (2, 8), and (1) were constructed in the target area. The process continues by merging these four sequences to form two sequences in the newly designated target area. The situation at the end of the second iteration is as shown in Figure 9.14 with two sequences (0, 3, 4, 5, 6, 7, 9), and (1, 2, 8).

Figure 9.14 Result of first merge

These remaining two sequences are merged in the third iteration to produce the last sequence of Figure 9.15.

Figure 9.15 End of Von Neumann sort

Since there is only one sequence, the sort is complete. However, since the sorted data are in the auxiliary data array, they must be copied back into the original data array.

The sorting process will thus repeatedly copy sequences of data from the source area to the target area until only one sequence was formed in the target, and therefore the data is sorted. Finally, the data are copied back into the original data area if necessary. Thus, we can sketch the algorithm in pseudocode as:

```

Von Neumann Sort
  Repeat
    Merge Sequences
  Until only one sequence copied
  If sorted data is in auxiliary store
    Copy data into original data array
  End Von Neumann Sort

```

In the merging of data to form sequences, each iteration constructs a single sequence in the target area, and looping continues until the end of the source data is reached.

```

Merge Sequences
  Repeat
    Merge One Sequence
  Until data all copied from source to target
  Count sequences
End Merge Sequences

```

In the process to generate one sequence, each iteration copies a data element and the loop terminates when the source data elements cannot be part of the current sequence.

```

Merge One Sequence
  While data being copied forms a sequence
    Copy element from source to target
  End Merge One Sequence

```

When we implement this in Pascal, we'll naturally make three separate procedures: `VonNeumannSort`, `MergeSequences`, and `MergeOneSequence`, as shown in the complete program of Figure 9.16.

Figure 9.16 Program Von Neumann Sort

```

PROGRAM TestVonNeumannSort;
(* Test von Neumann sort *)
USES SortEnvLib;

TYPE DataArrayPointer = ^DataArrayType;
      IncDecProc       = PROCEDURE(VAR X: INTEGER);

VAR Data: DataArrayType;

PROCEDURE Incr(VAR X: INTEGER);
(* Increment X *)
BEGIN
  X := X + 1;
END; { Incr }

PROCEDURE Decr(VAR X: INTEGER);
(* Decrement X *)
BEGIN
  X := X - 1;
END; { Decr }

PROCEDURE MergeOneSequence(      Source:
DataArrayPointer;                Target:
                                  DataArrayPointer;
                                  VAR LeftSource:  INTEGER;
                                  VAR RightSource: INTEGER;
                                  VAR TargetIndex: INTEGER;
                                  IncDec: IncDecProc);
(* Merge one ordered increasing sequence in Target from

```

```
        Source left and right *)
VAR PrevValue: INTEGER;
    Done: BOOLEAN;
BEGIN
    Done := FALSE;
    IF Source^[LeftSource] <= Source^[RightSource] THEN
BEGIN
    Target^[TargetIndex] := Source^[LeftSource];
    INC(LeftSource);
    END
ELSE BEGIN
    Target^[TargetIndex] := Source^[RightSource];
    DEC(RightSource);
    END;
    PrevValue := Target^[TargetIndex];
    IncDec(TargetIndex);
    WHILE (LeftSource <= RightSource) AND NOT Done DO BEGIN
        IF (Source^[LeftSource] < PrevValue) AND
            (Source^[RightSource] < PrevValue) THEN
            Done := TRUE { end of current sequence }
        ELSE BEGIN
            IF (Source^[LeftSource] >= PrevValue) AND
                (Source^[RightSource] >= PrevValue) THEN
                IF Source^[LeftSource] <= Source^[RightSource]
THEN BEGIN
                    Target^[TargetIndex] := Source^[LeftSource];
                    INC(LeftSource);
                    END
                ELSE BEGIN
                    Target^[TargetIndex] := Source^[RightSource];
                    DEC(RightSource);
                    END
            ELSE IF Source^[LeftSource] >= PrevValue THEN BEGIN
                Target^[TargetIndex] := Source^[LeftSource];
                INC(LeftSource);
                END
            ELSE BEGIN
                Target^[TargetIndex] := Source^[RightSource];
                DEC(RightSource);
                END;
            PrevValue := Target^[TargetIndex];
            IncDec(TargetIndex);
        END; { IF }
    END; { WHILE }
END; { MergeOneSequence }

PROCEDURE MergeSequences(Source: DataArrayPointer;
                        Target: DataArrayPointer;
                        VAR Count: INTEGER);
(* Repeatedly create ordered sequences from Source
```



```

        into Target alternatively left and right *)
VAR LeftSource, RightSource: INTEGER;
    LeftTarget, RightTarget: INTEGER;
BEGIN
    Count      := 0;
    LeftSource  := 1;
    RightSource := MaxData;
    LeftTarget  := 1;
    RightTarget := MaxData;
    WHILE LeftSource < RightSource DO BEGIN
        IF (Count MOD 2) = 0 THEN { left in Target }
            MergeOneSequence(Source, Target, LeftSource,
                             RightSource, LeftTarget, Incr)
        ELSE { right in Target }
            MergeOneSequence(Source, Target, LeftSource,
                             RightSource, RightTarget, Decr);
        Inc(Count);
    END;
END; { MergeSequences }

PROCEDURE VonNeumannSort(VAR Data: DataArrayType);
(* Sort Data in increasing order by merging sequences.
   At each pass Source and Target exchange roles *)
VAR WorkingStore: DataArrayType;
    Source, Target, Temp: DataArrayPointer;
    SequenceCount, Index: INTEGER;
BEGIN
    Source := Addr(Data);
    Target := Addr(WorkingStore);
    MergeSequences(Source, Target, SequenceCount);
    WHILE SequenceCount > 1 DO BEGIN
        Temp := Source;
        Source := Target; { exchange Source and Target }
        Target := Temp;
        MergeSequences(Source, Target, SequenceCount);
    END;
    IF Target <> Addr(Data) THEN
        Source^ := Target^; { result must be in Data }
    END; { VonNeumannSort }

BEGIN { TestVonNeumannSort }
    RandomData(Data);
    WriteLn('Original data');
    ListData(Data);
    WriteLn;
    VonNeumannSort(Data);
    WriteLn('Sorted data');
    ListData(Data);
    WriteLn;
    IF IsMonotonic(Data, Up) THEN

```

```
        WriteLn('Data is correctly sorted')
    ELSE
        WriteLn('Data is NOT correctly sorted');
END. { TestVonNeumannSort }
```

In order to make it easy to interchange source and target designations, pointers are used, even though the two data areas are not dynamic variables created with the procedure `New`. We define a data type:

```
TYPE DataArrayPointer = ^DataArrayType;
```

and then declare two pointers:

```
VAR Source, Target: DataArrayPointer;
```

We now need to make these two pointers reference the source data array and the auxiliary data array. There is a standard Pascal function `Addr`, which takes an ordinary variable as a parameter and returns its memory location (*address*), which can then be assigned to a pointer.

```
Source := Addr(Data);
Target := Addr(WorkingStore);
```

Once this has been done, `Source` and `Target` can be used to address the two storage areas. At the end of each major iteration, the direction of transfer can be reversed by simply interchanging the values of `Source` and `Target`. Note the way in which an element of an array is referenced:

```
Source^[LeftIndex]
```

because `Source` is a pointer, but `Source^` is an array. Another point to note in this sort procedure is that the index in the target is either incremented or decremented after each value is copied depending whether we are working at the left or right end of the target area. This is achieved by passing either procedures `Incr` or `Decr` to the procedure type parameter `IncDec`, another example of the usefulness of the procedure type.

The complete procedures are given in program `TestVonNeumannSort`. Procedure `VonNeumannSort` follows the pseudocode given above. Procedure `MergeSequences` also follows the pseudocode, albeit with much more detail. Procedure `MergeOneSequence` is the only really complicated procedure in the program. Although it corresponds to the pseudocode, the logic to decide what to copy and when to finish is difficult to follow. To understand it, just make sure you identify the various cases. The first `IF` just chooses the first value of the sequence. In the `WHILE`, the first `IF` tests for the end of the sequence, and if it is not, determines what value to copy in the target.

The result from a run of this program is shown in Figure 9.17.

Figure 9.17 Output of program `TestVonNeumannSort`

Original data									
483	355	823	758	964	654	78	158	372	286
917	351	711	661	274	581	60	667	442	220
82	270	469	976	990	369	202	591	340	403
141	909	127	517	294	128	545	65	807	310

```

576  933  799   86  382  566  653   50  387  706
521  406  225  222  138  825  814  506  999  621
167  624  251  950  262  245  782  623  223  166
596  480  271  497  747  656  134  713  717  829
926  603  676  694  185  115  181   3   84  326
 11  308  520  378  454  556  373  314  688   17

```

Sorted data

```

 3   11   17   60   65   78   82   84   86  115
127 128 134 138 141 158 166 167 181 185
202 220 222 223 225 245 251 262 270 271
274 286 294 294 308 310 314 326 340 351
355 369 372 373 378 382 387 403 406 442
454 469 480 483 497 506 517 520 521 545
556 566 576 581 591 596 603 621 623 624
653 654 656 661 667 676 688 694 706 711
713 717 747 758 782 799 807 814 823 825
829 909 917 926 933 950 964 976 990 999

```

Data is correctly sorted

9.4 Searching

Binary Search

In Chapter 8, we showed a tiny data-base program, `Retrieve`, that searched an array to find a key or pattern. It did this search in a linear way, beginning at the first item of the array, and ending at the last item. If, however, data in the array had already been sorted, then the binary search method could have been used, as described in Chapter 9 of the Principles volume. This binary search, or bisection method, first tests the value at the midpoint of the array to determine in which half of the array to carry on the search. It continues this way until the key is found, and its position is returned in the parameter `Mid`, or until it is known that the key cannot be matched in the array and 0 is returned in `Mid`.

The program `TestBinarySearch` in Figure 9.18, contains a procedure `BinSearch`, which performs a binary search on an array of sorted data. In order to obtain some test data, the procedure `RandomData` from `SortEnvLib` is first used to produce the data and then the procedure `BubbleSort`, seen earlier in this chapter, is used to sort the data.

Figure 9.18 Program TestBinarySearch

```

PROGRAM TestBinarySearch;
(* Test binary search procedure *)
USES SortEnvLib;

TYPE Range = 0..MaxData;

```

```
PROCEDURE BubbleSort(VAR Data: DataArrayType);
(* Sort array Data in decreasing order *)
VAR Posn, Temp: INTEGER;
    Done: BOOLEAN;
BEGIN
    Done := FALSE;
    WHILE NOT Done DO BEGIN
        Done := TRUE;
        FOR Posn := 1 TO MaxData - 1 DO
            IF Data[Posn] < Data[Posn + 1] THEN BEGIN
                Temp          := Data[Posn];
                Data[Posn]    := Data[Posn+1];
                Data[Posn + 1] := Temp;
                Done          := FALSE;
            END;
        END;
    END;
END; { BubbleSort }

PROCEDURE BinSearch(A: DataArrayType; Low, High: INTEGER;
                    Key: INTEGER; VAR Mid: INTEGER);
(* Search decreasingly ordered array A for Key *)
BEGIN
    Mid := (Low + High) DIV 2;
    WHILE Low < High DO BEGIN
        IF A[Mid] < Key THEN { first half }
            High := Mid - 1
        ELSE
            IF A[Mid] > Key THEN { second half }
                Low := Mid + 1
            ELSE BEGIN { found }
                Low  := Mid;
                High := Mid;
            END;
        Mid := (Low + High) DIV 2;
    END;
    IF A[Mid] <> Key THEN { item not found }
        Mid := 0;
END; { BinSearch }

VAR Data: DataArrayType;
    SearchKey, Index: INTEGER;
BEGIN
    RandomData(Data);
    BubbleSort(Data);
    WriteLn('Sorted data');
    ListData(Data);
    WriteLn;

    Write('Enter the search key: ');
```

```

Read(SearchKey);
BinSearch(Data, 1, MaxData, SearchKey, Index);
If Index = 0 THEN
    Write('It does not occur ')
ELSE
    Write('It is at position ', Index:3);
END. { TestBinarySearch }

```

The binary search algorithm is considerably faster than the linear search; it takes an average time of $\log_2 N$ compared to $\frac{N}{2}$. So when the size N is 1,000, a binary search takes a time of 10 compared to 500 for a linear search. When N is a million, a binary search takes a time of 20 compared to a linear search with a time of 500,000. However, a binary search requires a sorted array to begin with. So, if the array must be sorted each time a search is made, then a binary search may not be the best.

In Chapter 9 of the Principles book, algorithm Find First Match, which finds the first occurrence of a character string pattern in another character string, was introduced and discussed. The algorithm is simple as it compares repeatedly the characters of the string to the characters of the pattern. However, it only compares the characters of the pattern to the characters of the string until it finds a mismatch. Once a mismatch is found, the pattern is moved forward in the string for another try. In order for the algorithm to stop when finding the first match, we use a Boolean flag to indicate this condition as soon as it happens.

This algorithm has the same effect as the standard Pascal function Pos. A Pascal implementation of this algorithm is contained in program TestFirstMatch in Figure 9.19.

Figure 9.19 Program TestFirstMatch

```

PROGRAM TestFirstMatch;

    FUNCTION FindFirstMatch(Source, Pattern: STRING):
    INTEGER;
    VAR Found, Equal: BOOLEAN;
        S_Index, P_Index: INTEGER;
    BEGIN
        Found := FALSE;
        S_Index := 1;
        WHILE (S_Index <= (Length(Source) - Length(Pattern)))
            AND NOT Found DO BEGIN
                Equal := TRUE;
                P_Index := 1;
                WHILE (P_Index <= Length(Pattern)) AND Equal DO
                BEGIN
                    IF Pattern[P_Index] <>
                        Source[S_Index + P_Index - 1] THEN
                        Equal := FALSE;
                    Inc(P_Index);
                END
            END
        END
    END

```

```
        END;
        Found := Equal;
        Inc(S_Index);
    END;
    IF Found THEN
        FindFirstMatch := S_Index - 1
    ELSE
        FindFirstMatch := 0;
    END; { FindFirstMatch }

VAR Text, Key: STRING;
    Position: INTEGER;
BEGIN
    WriteLn('Enter string to be searched');
    ReadLn(Text);
    WriteLn('Enter string to search for');
    ReadLn(Key);
    Position := FindFirstMatch(Text, Key);
    IF Position = 0 THEN
        WriteLn('Pattern not in text')
    ELSE BEGIN
        WriteLn('Pattern starts at position ', Position: 3);
        WriteLn('Contained pattern = |',
            Copy(Text, Position, Length(Key)), '|');
    END;
    WriteLn('Enter second string to search for');
    ReadLn(Key);
    Position := FindFirstMatch(Text, Key);
    IF Position = 0 THEN
        WriteLn('Pattern not in text')
    ELSE BEGIN
        WriteLn('Pattern starts at position ', Position: 3);
        WriteLn('Contained pattern = |',
            Copy(Text, Position, Length(Key)), '|');
    END;
END. { TestFirstMatch }
```

Program `TestFirstMatch` asks for a string, and then for a couple of patterns to search for in that string. When a pattern is found, it is output from the string using standard function `Copy`. Note that when outputting the value of a substring for checking during program development, it is a good idea to bound it with some marker, for example `|`, so that the presence of a blank can be easily seen.

In procedure `FindFirstMatch` note that we use two Boolean variables, `Equal` to signal a mismatch when checking for the pattern, and `Found` to signal the end of the process. We use another standard function, `Length`, to compute the limits of the loops. Note that a Pascal `STRING` is really an array of characters and therefore that its individual characters can be accessed using normal subscripting techniques as in:

```

IF Pattern[P_Index] <>
    Source[S_Index + P_Index - 1] THEN

```

The output from a typical run of program TestFirstMatch is the following.

```

Enter string to be searched
Now is the time for all good people to come to the aid
of the party—it's party time!
Enter string to search for
good
Pattern starts at position 25
Contained pattern = |good|
Enter second string to search for
aiid
Pattern not in text

```

9.5 Implementing Stacks and Queues

StackLib: Stack as an Abstract Data Type

Stacks are used so often in computing that a Stack Abstract Data Type is usually defined and implemented as a Pascal UNIT. This Abstract Data Type, defined in Chapter 8 of the Principles book, is implemented here as UNIT StackLib, shown in Figure 9.20. This UNIT essentially extends the Pascal language by providing the StackType and actions Create, Push, Pop, Empty and Full.

Figure 9.20 The StackLib unit

```

UNIT StackLib;

INTERFACE
    CONST Height = 30;    { maximum size }
    TYPE RANGE   = 1..Height;
        ItemType = CHAR;
        StackType = RECORD
            Top : RANGE;
            Item: ARRAY[RANGE] OF ItemType;
        END;

    PROCEDURE Create(VAR Stack: StackType );
    (* Sets up stack, initially *)

    PROCEDURE Push(VAR Stack: StackType;
                   X: ItemType );
    (* Puts object X onto Stack *)

    PROCEDURE Pop(VAR Stack: StackType;
                  VAR Y: ItemType );
    (* Takes object Y off Stack *)

```

```
FUNCTION Empty(Stack: StackType): BOOLEAN;  
(* Shows if Stack is empty *)
```

```
FUNCTION Full(Stack: StackType ): BOOLEAN;  
(* Shows if Stack is full *)
```

IMPLEMENTATION

```
PROCEDURE Create(VAR Stack: StackType);  
BEGIN  
    Stack.Top := Height;  
END; { Create }
```

```
PROCEDURE Empty(Stack: StackType): BOOLEAN;  
BEGIN  
    IF Stack.Top = Height THEN  
        Empty := TRUE  
    ELSE  
        Empty := FALSE;  
END; { Empty }
```

```
PROCEDURE Full(Stack: StackType): BOOLEAN;  
BEGIN  
    IF Stack.Top = 1 THEN  
        Full := TRUE  
    ELSE  
        Full := FALSE;  
END; { Full }
```

```
PROCEDURE Push(VAR Stack: StackType;  
               X: ItemType);  
BEGIN  
    IF Full(Stack) THEN  
        WriteLn('FULL ')  
    ELSE BEGIN  
        DEC(Stack.Top);  
        Stack.Item[Stack.Top] := X;  
    END;  
END; { Push }
```

```
PROCEDURE Pop(VAR Stack: StackType;  
              VAR Y: ItemType);  
BEGIN  
    IF Empty(Stack) THEN  
        WriteLn('EMPTY ')  
    ELSE BEGIN  
        Y := Stack.Item[Stack.Top];  
        Inc(Stack.Top);  
    END;  
END; { Pop }
```

```
END. { StackLib }
```

The `INTERFACE` part of `StackLib` describes the stack concisely. A stack consists of two parts: an index indicating the top element position, and an array of elements. Notice that the `ItemType` is declared to be a character by:

```
ItemType = CHAR;
```

If, at some later time, the items in a stack are to be `INTEGERS`, `REALS` or other types, this stack definition could still be used with a minor modification of this one declaration. Of course after such a modification this `UNIT` and all other units using `StackLib` would need to be re-compiled.

The `IMPLEMENTATION` part of `StackLib` contains the procedures describing the actions on Stacks. These procedures are all self explanatory. Figure 9.21 presents program `StackProg`, that uses the Stack Abstract Data Type.

Figure 9 21 Program StackProg

```
PROGRAM StackProg;

USES StackLib;
VAR InStack, OutStack: StackType;
    Ch: CHAR;
BEGIN
    Create(InStack);
    Create(OutStack);

    (* Input characters onto stack *)
    Write('Enter letters end with $ ');
    Read(Ch);
    WHILE Ch <> '$' DO BEGIN
        Push(InStack, Ch);
        Read(Ch);
    END;

    (* Pour characters into another stack *)
    WHILE NOT Empty(InStack) DO BEGIN
        Pop(InStack, Ch);
        Push(OutStack, Ch);
        Write(Ch);
    END;
END. { StackProg }
```

Program `StackProg` reads characters and stores them in `InStack`. Then the characters are transferred from `InStack` to `OutStack`, and also output. The characters are output in reverse order of their input.

Other operations on stacks are possible. One rather useful action is `Pour`, a procedure to move the contents from one Stack into another, which is shown in Figure 9.22.

Figure 9.22 Operation Pour

```

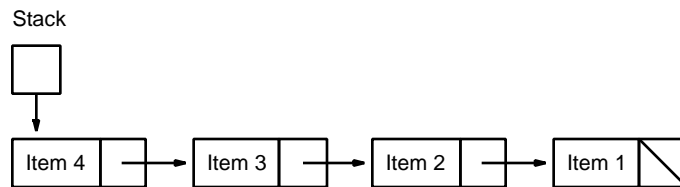
PROCEDURE Pour(    S1: StackType;
                  VAR S2: StackType);
VAR E: ItemType;
BEGIN
    WHILE NOT Empty(S1) DO BEGIN
        Pop(S1, E);
        { Actions could go here }
        Push(S2, E);
    END;
END; { Pour }

```

Notice that the contents of the second stack are in reverse order from what they were in the first stack. A second operation of `Pour` is required if the order is to be maintained. Notice also that the contents of the first stack are destroyed in this pouring process: pouring is not copying.

Dynamic Stacks

One of the problems with implementing a stack with an array is that the maximum size of the stack has to be fixed when the program is compiled. We can escape from this problem by using dynamic variables and pointers to implement our stack abstract data type. In this case, a stack will be represented as shown in Figure 9.23.

Figure 9.23 A dynamic stack

The pointer variable `Stack` will always point to the top element of the stack. All our operations will be redefined, as they were in Chapter 9 of the Principles book. The Pascal unit `DynamicStackLib` implements the redefined operations, and is shown in Figure 9.24.

Figure 9.24 Dynamic implementation of stacks

```

UNIT DynamicStackLib;

INTERFACE
    TYPE ItemType      = CHAR;
        StackType      = ^StackElement;
        StackElement = RECORD
            Value: ItemType;
            Prev:  StackType;
        END;

```

```
PROCEDURE Create(VAR Stack: StackType );
(* Sets up stack, initially *)

PROCEDURE Push(VAR Stack: StackType;
               X: ItemType );
(* Puts object X onto Stack *)

PROCEDURE Pop(VAR Stack: StackType;
              VAR Y: ItemType );
(* Takes object Y off Stack *)

FUNCTION Empty(Stack: StackType): BOOLEAN;
(* Shows if Stack is empty *)
```

IMPLEMENTATION

```
PROCEDURE Create(VAR Stack: StackType);
BEGIN
    Stack := NIL;
END; { Create }

PROCEDURE Empty(Stack: StackType): BOOLEAN;
BEGIN
    IF Stack = NIL THEN
        Empty := TRUE
    ELSE
        Empty := FALSE;
    END; { Empty }

PROCEDURE Push(VAR Stack: StackType;
               X: ItemType);
VAR Next: StackType;
BEGIN
    New(Next);
    Next^.Prev := Stack;
    Next^.Value := X;
    Stack := Next;
END; { Push }

PROCEDURE Pop(VAR Stack: StackType;
              VAR Y: ItemType);
VAR PrevTop: StackType;
BEGIN
    IF Empty(Stack) THEN
        WriteLn('EMPTY ')
    ELSE BEGIN
        Y := Stack^.Value;
        PrevTop := Stack^.Prev;
        Dispose(Stack);
        Stack := PrevTop;
    END;
```

```
        END;  
    END; { Pop }  
  
END. { DynamicStackLib }
```

The changes to `Create` and `Empty` are obvious. Note that operation `Full` has been eliminated, as the static limit on the size of the stack has disappeared. Procedure `Push` is now extremely simple, adding an element in front of the others. Operation `Pop` is as simple.

QueueLib

An abstract data type library for Queues can be constructed from arrays in much the same way as the library for Stacks was constructed in the previous section. However, it could also be constructed in another manner.

Since Queues are very similar to Stacks, it may be worth “inheriting” from stacks some of the basic implementation for queues. This is done by using stacks as the foundation on which to build an implementation of Queues. The main difference between stacks and queues is in the way in which items are “inserted” and “deleted”. If a queue is represented as a stack with the most recently inserted item on the top, then the item to be removed by the `ExitQ` operation is at the bottom of the stack. In order to remove this item, the stack must be “poured” into a temporary stack. The item to be removed is now at the top of the stack and is popped. Finally, the temporary stack is poured back into the original stack, to restore its original order. The procedure `ExitQ` in library `QueueLib`, shown in Figure 9.25, works in this way.

Figure 9.25 The abstract data type Queue

```
UNIT QueueLib;  
  
INTERFACE  
  
USES StackLib;  
  
    TYPE QueueType = StackType;  
  
    PROCEDURE CreateQ(VAR Queue: QueueType);  
        (* Create a queue initially *)  
    PROCEDURE EnterQ(VAR Q: QueueType; X: ItemType);  
        (* Add element X to queue Q *)  
    PROCEDURE ExitQ(VAR Q: QueueType; VAR Y: ItemType);  
        (* Eliminate first element in queue Q and save in Y *)  
    FUNCTION EmptyQ(Queue: QueueType): BOOLEAN;  
        (* Indicate if queue is empty *)  
    FUNCTION FullQ(Queue: QueueType): BOOLEAN;  
        (* Indicate if queue is full *)  
  
IMPLEMENTATION
```

```

PROCEDURE CreateQ(VAR Queue: QueueType);
BEGIN
    Create(Queue);
END; { CreateQ }

PROCEDURE EnterQ(VAR Q: QueueType; X: ItemType);
BEGIN
    Push(Q, X);
END; { EnterQ }

FUNCTION EmptyQ(Queue: QueueType): BOOLEAN;
BEGIN
    EmptyQ := Empty(Queue);
END; { EmptyQ }

FUNCTION FullQ(Queue: QueueType): BOOLEAN;
BEGIN
    FullQ := Full(Queue);
END; { FullQ }

PROCEDURE ExitQ(VAR Q: QueueType; VAR Y: ItemType);
VAR Temp: QueueType;
    I: ItemType;
BEGIN
    Create(Temp);
    WHILE NOT Empty(Q) DO BEGIN
        Pop(Q, I);
        Push(Temp, I);
    END;
    Pop(Temp, Y);
    WHILE NOT Empty(Temp) DO BEGIN
        Pop(Temp, I);
        Push(Q, I);
    END;
END; { ExitQ }

END. { QueueLib }

```

All of the other actions are virtually unchanged from their Stack counterpart: `CreateQ` is really stack `Create`, and `EnterQ` is actually `Push`. In the same manner, the two functions `EmptyQ` and `FullQ` are implemented by their stack counterparts, `Empty` and `Full`. Only procedure `ExitQ` is built differently, but uses `Create`, `Push` and `Pop` from `StackLib`.

The use of `QueueLib` is illustrated by program `QueueProg` in Figure 9.26.

Figure 9.26 Program 9.26

```

PROGRAM QueueProg;

USES QueueLib;

```

```
VAR InQueue, OutQueue: QueueType;
    Ch: CHAR;
BEGIN
    CreateQ(InQueue);
    CreateQ(OutQueue);

    (* Input characters onto queue *)
    Write('Enter letters ');
    Write(' end with $ ');
    WriteLn;
    Read(Ch);
    WHILE CH <> '$' DO BEGIN
        EnterQ(InQueue, Ch);
        Read(Ch);
    END;

    (* Pour characters into another queue *)
    WHILE NOT EmptyQ(InQueue) DO BEGIN
        ExitQ(InQueue, Ch);
        EnterQ(OutQueue, Ch);
        Write(Ch);
    END;
END. { QueueProg }
```

Queues can also be implemented with dynamic variables and pointers in much the same way as was done for Stacks in the previous section. Procedure `ExitQ` will be different as before, but this time instead of pouring the stack to get at the item to be removed from the queue, the chain of pointers could be followed. This new algorithm is shown in Figure 9.27.

Figure 9.27 Dynamic implementation of `ExitQ`

```
PROCEDURE ExitQ(VAR Q: QueueType; VAR Y: ItemType);
VAR Current, NextOlder: QueueType;
BEGIN
    IF EmptyQ(Q) THEN
        WriteLn('EMPTY ')
    ELSE BEGIN
        IF Q^.Prev = NIL THEN BEGIN
            Y := Q^.Value;
            Dispose(Q);
            Q := NIL;
        END
        ELSE BEGIN
            Current := Q;
            WHILE Current^.Prev^.Prev <> NIL DO
                Current := Current^.Prev;
            Y := Current^.Prev^.Value;
            Dispose(Current^.Prev);
            Current^.Prev := NIL;
        END
    END
END;
```

```

      END;
    END;
  END; { ExitQ }

```

Here, if the queue is not empty, a test is made to see whether there is only one item in the queue, $Q^{\wedge}.Prev = NIL$, if this is the case, the value is taken from that item, the item is disposed of, and the queue is set to NIL to show that it is empty. If there is more than one item, the chain of pointers is followed back until *Current* points to the last but one element. Notice the double pointer reference of:

```
WHILE Current^.Prev^.Prev <> NIL DO
```

When the last but one element is found, then the value in the last element is reached and the last element is disposed of through $Current^{\wedge}.Prev$ by the following statements.

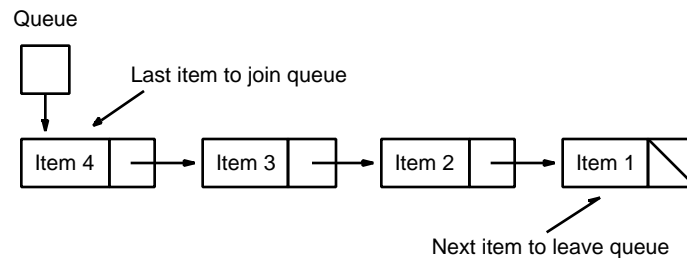
```

Y := Current^.Prev^.Value;
Dispose(Current^.Prev);
Current^.Prev := NIL;

```

To help you understand this, you should follow the *ExitQ* algorithm through to remove an item from the queue, using the diagram of Figure 9.28.

Figure 9.28 A dynamic queue



Big Cardinals

The standard `INTEGER` type available in Pascal has values that are limited to the range -32768 to 32767 . Cardinal numbers are integers that are greater than or equal to zero. If we attempted to represent a cardinal by an `INTEGER`, its value would be limited to the range 0 to 32767 . However, we can define much larger cardinals together with operations on them: this will be the `BigCard` abstract data type, defined by the `BigCardLib` unit of Figure 9.29.

Big cardinal numbers, consisting of 100 or more decimal digits can be represented in different ways. One way is to use arrays, and build the various `BigCard` operations based on the arrays. Another way is to use already existing libraries. We have followed this second approach, and have used stacks to implement `BigCards`.

Figure 9.29 The BigCardLib unit

```
UNIT BigCardLib;

INTERFACE
USES IntStackLib; { Integer stacks }

TYPE BigCard = StackType;
  PROCEDURE CreateBigCard(VAR BC: BigCard);
  PROCEDURE ReadBigCard(VAR BC: BigCard);
  PROCEDURE WriteBigCard(BC: BigCard);
  PROCEDURE AddBigCard( BC1, BC2: BigCard;
                        VAR BC3: BigCard);

IMPLEMENTATION
  PROCEDURE CreateBigCard(VAR BC: BigCard);
  BEGIN
    Create(BC);
  END; { CreateBigCard }

  PROCEDURE ReadBigCard(VAR BC: BigCard);
  CONST EndLine = #13;
  VAR Ch: CHAR;
      Val: INTEGER;
  BEGIN
    Read(Ch);
    WHILE (Ch <> EndLine) AND (NOT Full(BC)) DO BEGIN
      { Convert only the Char digits }
      IF ('0' <= Ch) AND (Ch <= '9') THEN BEGIN
        Val := ORD(Ch) - ORD('0');
        Push(BC, Val);
      END;
      Read(Ch);
    END;
  END; { ReadBigCard }

  PROCEDURE WriteBigCard(BC: BigCard);
  VAR C: INTEGER;
      T: BigCard;
  BEGIN
    Create(T);
    Pour(BC, T);
    WHILE NOT Empty(T) DO BEGIN
      Pop(T, C);
      Write(C: 1);
    END;
  END; { WriteBigCard }

  PROCEDURE AddBigCard( BC1, BC2: BigCard;
                        VAR BC3: BigCard);
  VAR Carry, T: BigCard;
```



```

    A, B, C, S, Sum: INTEGER;
BEGIN
    Create(Carry);
    Create(T);
    Push(Carry, 0);
    WHILE (NOT Empty(BC1)) OR (NOT Empty(BC2)) DO BEGIN
        IF Empty(BC1) THEN
            A := 0
        ELSE
            Pop(BC1, A);
        IF Empty(BC2) THEN
            B := 0
        ELSE
            Pop(BC2, B);
        Pop(Carry, C);
        Sum := A + B + C;
        IF Sum < 10 THEN BEGIN
            S := Sum;
            C := 0;
        END
        ELSE BEGIN
            S := Sum - 10;
            C := 1;
        END;
        Push(T, S);
        Push(Carry, C);
    END;
    Pop(Carry, C);
    IF C = 1 THEN
        Push(T, C);
    Clear(BC3);
    Pour(T, BC3);
END; { AddBigCard }

END. { BigCardLib }

```

The unit `BigCardLib`, shown in Figure 9.29, indicates it uses `IntStackLib`, which was derived from `StackLib` by modifying the definition of `ItemType` to be `INTEGER`, and by adding operations `Clear` (to empty a stack) and `Pour`. In `BigCardLib`, the type `BigCard` is defined to be type `StackType`. Similarly, `CreateBigCard` is a renaming of the `Create` procedure in `StackLib`.

Reading very large cardinals can be inconvenient, so it would be useful to be able to separate every three digits with a comma or a blank as in the following 60 digit prime number:

108,488,104,853,637,470,612,961,399,842,972,948,409,834,611,
525,790,577,216,753

Procedure `ReadBigCard` accepts such sequences of characters as input, beginning with the most significant digits. The characters representing non digits

(commas or blanks) are ignored, leaving only the numeric digits in stack BC with the least significant value at the top, ready to be added.

WriteBigCard is a procedure that first reverses the order of the digits of a given BigCard BC by pouring it into a temporary stack T. This puts the most significant digit of the number at the top of the stack; it then outputs this stack digit by digit, putting them back on the original stack as it does so.

AddBigCard is an implementation of the diagram shown in Chapter 8 of the Principles book (Fig. 8.44). It uses four stacks BC1, BC2, BC3 and Carry. It is assumed that the BigCards have been read into BC1 and BC2 by ReadBigCard, which leaves the least significant digits at the top of each stack. Carry is a small stack that is initialized to a value of zero.

The way in which this adder works is to pop the values A and B from the top of the two input stacks, and to add these together with the value C from the Carry stack. If the Sum of these three is less than 10, then this Sum is sent to the output stack BC3 otherwise a carry value of 1 is pushed onto the Carry stack and (Sum - 10) is pushed onto the output stack BC3. This process of popping from the input stacks, and creating an output stack continues until both input stacks are empty.

If the BigCard operands to the adder are of different lengths, then one stack will become empty before the other. If the empty stack is still popped an error message would be output. For this reason a test is made before each pop, and a value of zero is used in the addition if the stack is empty. This corresponds to representing a number by appending any number of zeros before that number.

An alternative way to handle this problem of different length BigCards would be to modify the IntStackLib procedure Pop to produce the value 0 when the stack is empty. This change would make the AddBigCard algorithm slightly shorter, but the modification of a general stack ADT for solving a particular problem is not good programming practice.

Figure 9.30 shows an example program using BigCardLib.

Figure 9.30 Program BigCardProg

```
PROGRAM BigCardProg;

USES BigCardLib;

VAR X, Y, Z: BigCard;

BEGIN
    CreateBigCard(X);
    CreateBigCard(Y);
    CreateBigCard(Z);
    Write('Enter a big value: ');
    ReadBigCard(X);
    Write('Enter a big value: ');
    ReadBigCard(Y);
    AddBigCard(X, Y, Z);
```

```

    WriteBigCard(Z);
END. { BigCardProg }

```

Other operations on BigCards must be added to BigCardLib. These include the remaining arithmetic operations, as well as various moves and comparisons. Such new operations are like `CopyBigCard(A, B)` to copy the value of A onto B, and `Equal(A, B)`, a `BOOLEAN` function that compares any two BigCards and returns a value of `TRUE`, when the BigCards are of the same size and have identical digits. A procedure `Assign(S, L, V)` can be written to assign the content of string S having length L to the BigCard variable V. It provides a way of creating constant BigCards. These and the other arithmetic operations should be added to BigCardLib.

SetLib: Stack of Strings

As a final example of a useful library that involves data structures, we present SetLib. It ties together many concepts of this chapter and the previous one, showing how easy it is to “grow” libraries by making use of other libraries as building blocks.

Previously, sets have been restricted to a fixed number of small enumerated elements. This could be sufficient for some needs, but it is very restrictive for others. In this example, we will consider a more general type of sets—sets of strings. The string set elements could be numbers, words, quotes, or any combinations of these.

String set elements could, for example, involve people with detailed but “unstructured” descriptions such as:

```
"Able, John (818)885-3399 #1234 Zip=91330 Male, etc. "
```

```
"Charlie, Female Very Tall (123)456-7890 Volleyball"
```

```
"Bob Baker #5678 Male Blonde (818)349-4296 Born: 670312"
```

Sets of strings of various lengths could be stored externally in files. Internally they could be implemented in a number of ways: arrays of strings, stacks of strings, arrays of records, etc. The order of occurrence of elements in a set is not significant and the order of items in a stack is also not significant, so an implementation of sets based on stacks seems all right and will be done here. You are encouraged to try other implementations.

Figure 9.31 Unit SetLib

```

UNIT SetLib;
(* Sets of character strings *)
INTERFACE

USES StrStackLib;

TYPE EltType = ItemType; { Items are Strings }
    SetType = StackType;

```

```
PROCEDURE CreateSet(VAR S: SetType);
PROCEDURE ReadSet(VAR S: SetType);
PROCEDURE WriteSet(S: SetType);
PROCEDURE Intersect(X, Y : SetType; VAR Z: SetType);
FUNCTION IsEmpty(S: SetType): BOOLEAN;
```

IMPLEMENTATION

```
PROCEDURE CreateSet(VAR S: SetType);
BEGIN
    Create(S);
END; { CreateSet }

PROCEDURE ReadSet(VAR S: SetType);
VAR A: EltType;
    I: INTEGER;
    Ch: CHAR;
BEGIN
    Create(S);
    Write('How many ? ');
    Read(I);
    Read(Ch);
    WHILE I > 0 DO BEGIN
        Read(A);
        Push(S, A);
        Dec(I);
    END;
END; { ReadSet }

PROCEDURE WriteSet(S: SetType);
VAR A: EltType;
BEGIN
    WHILE NOT Empty(S) DO BEGIN
        Pop(S, A);
        WriteLn(A);
    END
END; { WriteSet }

PROCEDURE Intersect(X, Y : SetType; VAR Z: SetType);
VAR I, J: EltType;
    T: SetType;
BEGIN
    Create(T);
    WHILE NOT Empty(X) DO BEGIN
        Pop(X, I);
        WHILE NOT Empty(Y) DO BEGIN
            Pop(Y, J); Push(T, J);
            IF I = J THEN
                Push(Z, I);
        END;
    END;
```

```

        WHILE NOT Empty(T) DO BEGIN
            Pop(T, J);
            Push(Y, J);
        END;
    END;
END; { Intersect }

FUNCTION IsEmpty(S: SetType): BOOLEAN;
BEGIN
    IsEmpty := Empty(S);
END; { IsEmpty }

END. { SetLib }

```

The unit `SetLib`, shown in Figure 9.31, uses `StrStackLib` which is derived from `StackLib` in the same way that `IntStackLib` was derived from `StackLib`. The type of `ItemType` was changed from `CHAR` to `STRING`. In `SetLib`, `ItemType` is renamed `EltType` (for Element Type), and `StackType` is renamed `SetType`.

Only the operations `CreateSet`, `ReadSet`, `WriteSet`, `Intersect` and `IsEmpty` are described here. Other operations, including `Union`, `Difference`, `Include`, `Exclude`, `SetSize`, `EqualSet`, `IsEltOf`, `IsSetOf`, `IsEmpty`, must be added later.

Figure 9.32 Program SetProg

```

PROGRAM SetProg;

USES SetLib;
VAR R, S, T: SetType;
BEGIN
    CreateSet(R);
    CreateSet(S);
    CreateSet(T);
    ReadSet(R);
    ReadSet(S);
    Intersect(R, S, T);
    Write('Common ones: ');
    WriteLn;
    WriteSet(T);
END. { SetProg }

```

The simple program `SetProg`, shown in Figure 9.32, creates three string sets, reads in two string sets, finds their intersection and outputs the resulting intersection. Notice that there is no reference to strings or stacks that underlie these Sets; such details are hidden. This shows the power of using libraries! We have “adopted” rather than “inherited” the important aspects of other libraries into our own libraries. The output from a typical run of `SetProg` is as follows:

```
How many elements in the set? 3
```

```
first
second
third
How many elements in the set? 2
third
fourth
Elements common to both sets:
third
```

Files of strings like the above three examples could be used to create sets having various characteristics such as:

- A = Set of all those in area code 818
- B = Set of all Males
- C = Set of all Males in area code 818

Another important procedure would be `BuildSet`, that builds a Set of strings that contain a given substring. For example,

```
BuildSet(File1, A, "(818)") adds all strings from File1 having
the given area code

BuildSet(File1, B, "Male") adds all strings from File1 having
the pattern Male
```

From these two sets, the third set C can be determined as the intersection of the other two by the call


```
Intersect(A, B, C);
```

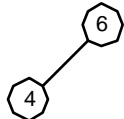
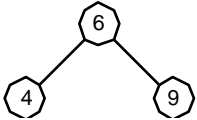
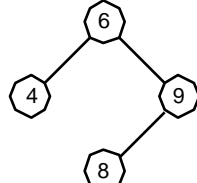
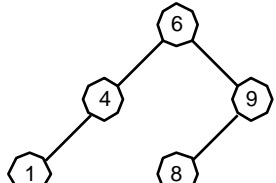
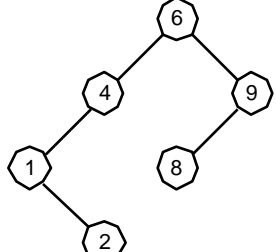
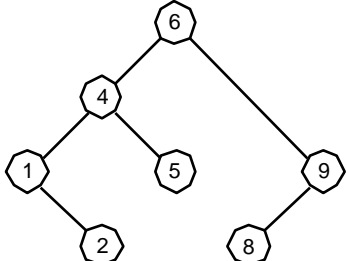
9.6 Trees

In Chapter 9 of the Principles book, we discussed briefly the representation of trees. As we already mentioned in that chapter, the most convenient way to represent a tree is through the use of pointers. We can represent a node in a binary tree by a record that contains the value associated with the node and two pointers, `Left` and `Right`, which point to the left and right sub-trees of the node.

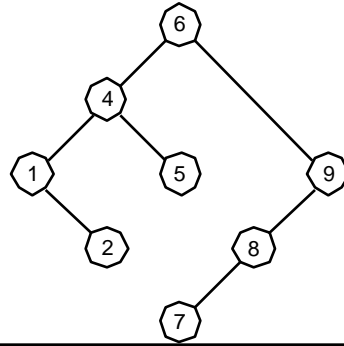
One of the applications of such a binary tree is as a method of sorting. First the items to be sorted are read in and the binary search tree constructed, and then the tree is “traversed” to get the values out in increasing order. The following is a step by step description of how a binary search tree might be built to store the following sequence of digits:

```
6 4 9 8 1 2 5 7 3.
```

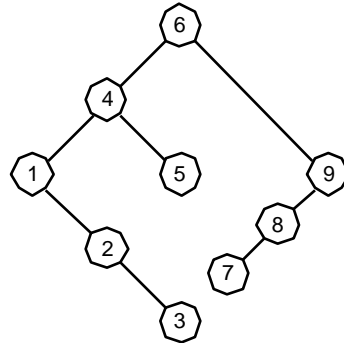
The digit 6 is read; since it is the first, it becomes the root node, at this stage the only node of the tree.	
--	---

<p>The digit 4 is read, since it is less than 6, it is made into a node attached to the left branch of the root node</p>	
<p>The digit 9 is read, since it is greater than 6, it is made into a node attached to the right branch of the root node.</p>	
<p>The digit 8 is read, since it is greater than 6, the right branch is followed; it is less than 9, the value of the next node reached, and is made into a node and attached as the left branch of this node.</p>	
<p>The digit 1 is read, since it is less than 6, the left branch is followed, since it is less than 4, it is put into a node attached to the left branch of this node.</p>	
<p>The digit 2 is read, since it is less than 6, the left branch is followed, since it is less than 4, the left branch is followed, since it is greater than 1, it is made into a node attached as the right branch of this node.</p>	
<p>The digit 5 is read, since it is less than 6, the left branch is followed, since it is greater than 4, it is made into a node and attached as the right branch of this node.</p>	

The digit 7 is read; since it is greater than 6, the right branch is followed; since it is less than 9, the left branch is followed; since it is less than 8, it is made into a node attached as the left branch of this node.



The digit 3 is read; since it is less than 6, the left branch is followed; since it is less than 4, the left branch is followed; since it is greater than 1, the right branch is followed; since it is greater than 2, it is made into a node attached as the right branch of this node.



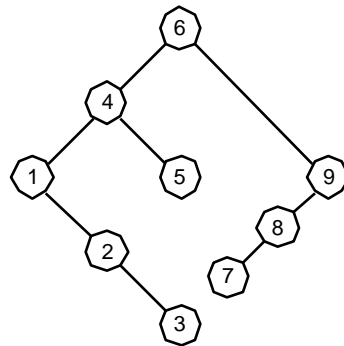
Now, having put the numbers into a binary search tree as we just described, we must bring them out in ascending order. The output algorithm that we will use is recursive and similar to the Traversal algorithm we introduced in Chapter 9 of the Principles book.

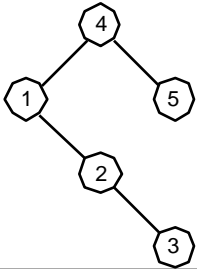
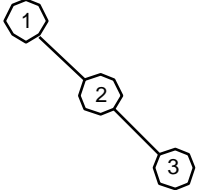
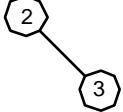

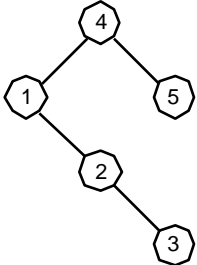

```

 Traverse(Tree)
   If Tree not empty
     Traverse left sub-tree
     Output Node
     Traverse right sub-tree
   End Traverse
  
```

The tree that we have just constructed is output following this algorithm as shown by the following trace.

Traverse left sub-tree



Traverse left sub-tree	
Traverse left sub-tree Nothing to traverse Output node Output 1 Traverse right sub-tree	
Traverse left sub-tree Nothing to traverse Output node Output 2 Traverse right sub-tree	
Traverse left sub-tree Nothing to traverse Output node Output 3 Traverse right sub-tree Nothing to traverse	
Output node Output 4 Traverse right sub-tree	
Traverse left sub-tree Nothing to traverse Output node Output 5 Traverse right sub-tree Nothing to traverse	

The rest of the tree is output continuing to follow the same recursive algorithm. Program `TreeSort`, in Figure 9.33, shows a Pascal implementation of the process just described.

Figure 9.33 Program `TreeSort`

```
PROGRAM TreeSort;

TYPE TreePointer = ^TreeElement;
     TreeElement = RECORD
         Value: INTEGER;
         Left: TreePointer;
         Right: TreePointer;
     END;

PROCEDURE TreeTraverse(CurrentNode: TreePointer);
BEGIN
    IF CurrentNode <> NIL THEN BEGIN
        TreeTraverse(CurrentNode^.Left);
        WriteLn(CurrentNode^.Value: 4);
        TreeTraverse(CurrentNode^.Right);
    END;
END; { TreeTraverse }

PROCEDURE BuildTree(VAR TreeRoot: TreePointer);
VAR Current, Node, Next: TreePointer;
    NewValue, TerminalValue: INTEGER;
BEGIN
    TreeRoot := NIL;
    Write('Enter terminal value ');
    Read(TerminalValue);
    WriteLn('Enter values terminated by terminal value');
    Read(NewValue);
    WHILE NewValue <> TerminalValue DO BEGIN
        New(Node);
        Node^.Value := NewValue;
        Node^.Left := NIL;
        Node^.Right := NIL;
        IF TreeRoot = NIL THEN
            TreeRoot := Node
        ELSE BEGIN
            Next := TreeRoot;
            WHILE Next <> NIL DO BEGIN
                Current := Next;
                IF Node^.Value < Current^.Value THEN
                    Next := Current^.Left
                ELSE
                    Next := Current^.Right;
            END;
            IF Node^.Value < Current^.Value THEN
```

```
        Current^.Left := Node
      ELSE
        Current^.Right := Node;
      END;
      Read(NewValue);
    END;
  END; { BuildTree }

VAR Root: TreePointer;

BEGIN
  BuildTree(Root);
  WriteLn('Sorted list is:');
  TreeTraverse(Root);
END. { TreeSort }
```

Procedure `TreeTraverse` is exactly as described by the pseudocode above. Procedure `BuildTree` is more complex. The outer loop is repeated as long as there are values to input. For each value read a new node is created. This node is then inserted at the right place in the tree. To find the right place, the tree is traversed by following left and right branches depending on the value to be inserted.

9.7 Chapter 9 Review

This chapter offered many illustrations of the use of the data structures described in the previous chapter. These data structures were illustrated with Pascal programs that implemented the algorithms described in Chapter 9 of the Principles book.

In particular, many different sorting algorithms were shown, one of which was recursive. Some searching algorithms, both for linear lists and for patterns within a string were also illustrated.

Libraries of various kinds were created, including `IntArrayLib`, `ComplexLib`, `StackLib`, `QueueLib`, `MatrixLib` and `SetLib`. Different techniques for representing stacks and queues were illustrated, including arrays and dynamic variables, and structures linked by pointers.

Later, in following computer science courses on data structures, you will consider and learn more details of such structures. For instance, stacks and queues will be implemented in other ways. Trees and graphs, and other more “nonlinear” structures will be introduced, implemented and analyzed.

9.8 Chapter 9 Problems

1. StringLib

Implement a StringLib unit that is based on StackLib and whose INTERFACE part is the following.

```
UNIT StringLib;

INTERFACE
    USES StackLib;

    CONST
        MaxLen = 80; (* Longest length      *)
        EndStr = #0; (* End of String        *)
        EndLin = #13; (* Carriage return    *)

    TYPE
        Strng = StackType;

    PROCEDURE ReadStrng(VAR Str: Strng);
        (* Read characters until a Return *)
        (* Destroy any previous contents! *)

    PROCEDURE WriteStrng(Str: Strng);
        (* Display contents of a string    *)

    FUNCTION LenStrng(Str: Strng): INTEGER;
        (* Return the count of characters *)

    PROCEDURE AssignStrng(      Source: Strng;
                           VAR Destin: Strng);
        (* Assign Source string to Destin *)

    PROCEDURE JoinStrng(  Str1, Str2: Strng;
                        VAR Str3: Strng);
        (* Joins 2 strings into a long one *)

    PROCEDURE SearchStrng(      Str, Pat: Strng;
                           VAR Posn, Count: INTEGER);
        (* Search Str for a Pattern Pat  *)
        (* return a count of occurrences *)
        (* return a position of last one  *)
        (* return 0 for no occurrences    *)

    FUNCTION EqualStrng(Str1, Str2: Strng): BOOLEAN;
        (* Compare two strings  *)
        (* Return TRUE if exactly equal    *)
```

```
(* in both size and also content *)
```

```
END. { StringLib }
```

Create other procedures such as:

```
CompareStr(A, B, C)
```

where the result C is:

0 if the two strings A and B are identical

If the strings are not identical, then if the difference in ASCII value for the first differing characters is positive—the first string comes after the second string in ASCII ordering—the value of C is set to 1, otherwise it is set to -1.

```
FromPat(Str, Pat, C)
```

where the string C is the substring of Str starting at the first occurrence of Pat and continuing to the end of Str. If Pat does not occur in Str, C is the null string.

```
AfterPat(Str, Pat, C)
```

where the string C is the substring of Str starting after the first occurrence of Pat and continuing to the end of Str. If Pat does not occur in Str, C is the null string.

```
BeforePat(Str, Pat, C)
```

where the string C is the substring of Str starting at the beginning of Str and continuing up to the first occurrence of Pat. If Pat does not occur in Str, C is the null string.

```
UptoPat(Str, Pat, C)
```

where the string C is the substring of Str starting at the beginning of Str and continuing up to the end of the first occurrence of Pat. If Pat does not occur in Str, C is the null string.

2. Complete SetLib

Complete the Set library shown in this chapter by providing procedures for Union, Difference, Include, Exclude, IsEltOf, IsSetOf, EqualElt, EqualSet, IsEmpty and SetSize.

3. BinTreeLib

Create a binary tree library whose data are binary trees where each node has a value and a left and right branch, and whose actions are ReadTree, WriteTree, CreateNode, SetNodeValue, AttachLeftNode, AttachRightNode, DetachLeftNode, DetachRightNode, MoveToLeftNode and MoveToRightNode.

4. StackUtilLib

Create the IMPLEMENTATION Part of the following Stack Utility Library. Create also a program to test this Library.

```
UNIT StackUtilLib;
(* Useful Utilities for Stacks *)

INTERFACE
    USES StackLib;

    PROCEDURE Clear(VAR S: StackType);
    (* Remove all items from a stack *)

    PROCEDURE Reverse(VAR S: StackType);
    (* Flip the entire contents of the stack *)

    PROCEDURE Pour(    Source: StackType;
                     VAR Destin: StackType);
    (* Dump the Source Stack into the Destination
       Stack *)

    PROCEDURE Cop(Source: StackType; T: ItemType);
    (* Show the item at the top of a given stack *)

    FUNCTION IsEqual(S, T: StackType): BOOLEAN;
    (* Return value TRUE when S and T are identical *)

END. { StackUtilLib }
```

9.9 Chapter 9 Programming Projects

Queue Abstract Data Type

The following `INTERFACE` part describes a Library of Queues. Create the `IMPLEMENTATION` part. Then use this Library to illustrate some potential use of queues.

```

UNIT QueueLib;
(* A Queue implemented as an Array *)

INTERFACE

    CONST Length = 50;

    TYPE RANGE    = 0..Length;
         ItemType = CHAR;

    QUEUEUETYPE = RECORD
        item:  ARRAY [RANGE] OF ItemType;
        Rear:  RANGE;  { Point of entry }
        Front: RANGE;  { Point of exit  }
        Size:  RANGE;  { Object count
    }

    END;

    PROCEDURE CreateQ(VAR Q: QUEUEUETYPE);
    (* Set up a Queue initially *)

    PROCEDURE EnterQ(VAR Q: QUEUEUETYPE; X: ItemType);
    (* Put object X onto Rear *)

    PROCEDURE ExitQ(VAR Q: QUEUEUETYPE; VAR Y:ItemType);
    (* Remove object Y from the front *)

    FUNCTION EmptyQ(Q: QUEUEUETYPE): BOOLEAN;
    (* Indicate no objects in Queue Q *)

    FUNCTION FullQ(Q: QUEUEUETYPE): BOOLEAN;
    (* Indicate that Queue is filled *)

    PROCEDURE Next(VAR Point: RANGE);
    (* Bump pointer to circular array *)

END. { QueueLib }
```

PES: Performance Evaluation of Sorts

The goal of this project is to study the time performance of programs using PerformLib with its two procedures `StartTime` and `TellTime`. These two procedures "sandwich" the part of a program that is to be evaluated. `StartTime` begins the timing and `TellTime` indicates the time in "ticks" ($1/60$ of a second) elapsed since this beginning.

```

UNIT PerformLib;

INTERFACE
  PROCEDURE StartTime();

  FUNCTION TellTime: LONGINT;

IMPLEMENTATION
  USES Events;

  VAR TimeOfStarting: LONGINT;

  PROCEDURE StartTime();
  BEGIN
    TimeOfStarting := TickCount();
  END; { StartTime }

  FUNCTION TellTime(): LONGINT;
  VAR FinishTime: LONGINT;
  BEGIN
    FinishTime := TickCount();
    TellTime := FinishTime - TimeOfStarting;
  END; { TellTime }

END. { PerformLib }

```

Your project is to experiment with sort programs, to gain insights into the time behavior of such algorithms. The following `SwapSort` is good as a starting point. Use random numbers to create the arrays.

```

(* SwapSort *)
FOR I := 1 TO N-1 DO
  FOR J := 1 TO N-1 DO (* ORDER *)
    IF A[J] > A[J+1] THEN (* SWAP *)
      Temp := A[J];
      A[J] := A[J+1];
      A[J+1] := Temp;
    END (* IF *);
  END (* FOR J *);
END (* FOR I *);

```

Experiment with this sort in the following ways:

- a. Determine the time when the array size is varied, from 25 to 50, 100, 200, 400, 800, 1600. Sketch a plot of this. Note

- a. To avoid data pattern dependencies, you should run the sort against many different sets of data.
- b. since, for short lists, the time taken is too short to measure accurately, you should do many runs and count the cumulative time and then allow for the overhead.

The following loop is suggested:

```

StartTime();
FOR Count := 1 TO 100 DO
  BEGIN
    RandomData(A);
    DummyBubbleSort(A);
  END;
Overhead := TellTime();
StartTime();
FOR Count := 1 TO 100 DO
  BEGIN
    RandomData(A);
    BubbleSort(A);
  END;
SortTime := (TickCount() - Overhead) DIV 100;
DummyBubbleSort is a procedure with the same heading as
BubbleSort but with an empty body.

```

- b. Change the inner loop from $N-1$ passes to $N-I$ passes. Notice the effect of this change.
- c. Modify the program to stop when no swaps were made in a pass by, counting the number of swaps in a pass. Note the effect and justify it.

Time Permitting

- d. Make SWAP as a procedure and see if that method of calling the procedure causes much slowdown.
- e. Modify the program to stop when no swaps were made in a pass by using a logical variable to determine if a swap was made.
- f. Modify the program to swap values that are some distance apart (instead of being adjacent). This should produce an incredible effect!
- g. Create one of the other 3 sorts (especially InsertSort, which is not done in the book) and compare times.
- h. Try any other modifications, variations, etc. like creating your own sort (even a SlowSort!).

VS: Visual Sorts

The purpose of this project is to obtain some appreciation for the way in which the various sort algorithms work by showing on the screen the data as it is sorted.

```
UNIT SortViewLib;

INTERFACE
  USES
    SortEnvLib, ScreenIO, Graphics, QuickDraw;

  PROCEDURE MakeWindow;
  PROCEDURE MakeSquare(      X, Y: INTEGER;
                           VAR Square: Rect);
  PROCEDURE DrawDot(X, Y: INTEGER);
  PROCEDURE EraseDot(X, Y: INTEGER);
  PROCEDURE MoveDot(FromX, FromY,
                   ToX, ToY: INTEGER);
  PROCEDURE ShowData(DataList: DataArrayType);
  PROCEDURE CloseWindow;
  PROCEDURE Beep;
  PROCEDURE Freeze;
  PROCEDURE MakeSortViewData(VAR DataList:
                              DataArrayType);

IMPLEMENTATION

CONST
  Min      = 10;
  Max      = 350;
  Left     = 10;
  Bottom   = 10;
  Right    = 310;
  Top      = 310;
  Scale    = (Right - Left) DIV MaxData;
  Border   = 2;

PROCEDURE MakeWindow;
BEGIN
  OpenGraphicWindow(Min, Min, Max, Max,
                   'Sort View');

  ScClear;
  MoveTo(Left - Border, Bottom - Border);
  LineTo(Left - Border, Top + Border);
  LineTo(Right + Border, Top + Border);
  LineTo(Right + Border, Bottom - Border);
  LineTo(Left - Border, Bottom - Border);
END; { MakeWindow }
```

```
PROCEDURE MakeSquare(      X, Y: INTEGER;
                          VAR Square: Rect);
CONST DotSize = 2;
BEGIN
    Square.top      := Scale * Y -
                      DotSize + Bottom;
    Square.bottom := Scale * Y +
                      DotSize + Bottom;
    Square.left  := Scale * X - DotSize + Left;
    Square.right := Scale * X + DotSize + Left;
END; { MakeSquare }

PROCEDURE DrawDot(X, Y: INTEGER);
VAR Sq: Rect;
BEGIN
    MakeSquare(X, Y, Sq);
    PaintOval(Sq);
END; { DrawDot }

PROCEDURE EraseDot(X, Y: INTEGER);
VAR Sq: Rect;
BEGIN
    MakeSquare(X, Y, Sq);
    EraseOval(Sq);
END; { EraseDot }

PROCEDURE MoveDot(FromX, FromY,
                  ToX, ToY: INTEGER);
BEGIN
    EraseDot(FromX, FromY);
    DrawDot(ToX, ToY);
END; { MoveDot }

PROCEDURE ShowData(DataList: DataArrayType);
VAR Index: INTEGER;
BEGIN
    FOR Index := 1 TO MaxData DO
        DrawDot(Index, DataList[Index]);
    END; { ShowData }

PROCEDURE CloseWindow;
BEGIN
    CloseGraphicWindow();
END; { CloseWindow }

PROCEDURE Beep;
BEGIN
    ScBeep(1);
END; { Beep }
```

```
PROCEDURE Freeze;
BEGIN
    ScFreeze;
END; { Freeze }

PROCEDURE MakeSortViewData(VAR DataList:
                           DataArrayType);
VAR Index, Source, Dest, Temp: INTEGER;
BEGIN
    FOR Index := 1 TO MaxData DO
        DataList[Index] := Index;

        FOR Index := 1 TO 2 * MaxData DO BEGIN
            Source := (RandInt() Mod MaxData) + 1;
            Dest := (RandInt() Mod MaxData) + 1;
            Temp := DataList[Source];
            DataList[Source] := DataList[Dest];
            DataList[Dest] := Temp;
        END;
    END; { MakeSortViewData }

END. { SortViewLib }
```

The library `SortViewLib` shown above contains the following procedures that can be used to show the movement of data during a sort:

MakeWindow:

Constructs a graphic window in which the diagram will be drawn

DrawDot(X, Y: INTEGER)

Draws a dot at the position X, Y in the graphic window.

EraseDot(X, Y: INTEGER)

Erases the dot at the position X, Y in the graphic window.

MoveDot(FromX, FromY, ToX, ToY: INTEGER)

Erases the dot at the position FromX, FromY, and draws a dot at position ToX, ToY

ShowData(DataList: DataArrayType)

Plots one dot for each data value in `DataList`; each value is plotted with index of the value in the list as the x-coordinate and the actual value of the item as the y-coordinate. Thus, if the value of the *I*th data item is *J*, it will be represented by a dot at *I*, *J*.

CloseWindow

Closes and deletes the graphic window.

Beep

Sounds a tone on the terminal.

Freeze

Freezes the screen until it is released by depressing any key.

`MakeSortViewData(VAR DataList: DataArrayType)`

The data consists of the integer values 1..MaxData arranged in random order in DataList.

Enter and compile the library. Next enter and compile the following program.

```

PROGRAM SortViewBubble;
USES SortEnvLib, SortViewLib;

VAR Data: DataArrayType;
    Count, Index: INTEGER;

PROCEDURE BubbleSort(VAR Data: DataArrayType);
VAR Posn, Temp: INTEGER;
    Done: BOOLEAN;
BEGIN
    Done := FALSE;
    WHILE NOT Done DO BEGIN
        Done := TRUE;
        FOR Posn := 1 TO MaxData - 1 DO
            IF Data[Posn] < Data[Posn + 1] THEN BEGIN
                MoveDot(Posn, Data[Posn], Posn+1,
                        Data[Posn]);
                MoveDot(Posn+1, Data[Posn+1], Posn,
                        Data[Posn+1]);
                Temp := Data[Posn];
                Data[Posn] := Data[Posn+1];
                Data[Posn + 1] := Temp;
                Done := FALSE;
            END;
        END;
    END; { BubbleSort }

BEGIN
    MakeWindow;
    MakeSortViewData(Data);
    ShowData(Data);
    Beep;
    Freeze;
    BubbleSort(Data);
    Beep;
    Freeze;
    CloseWindow;
END. { SortViewBubble }

```

When the diagram starts, the data is a random set of dots on the screen. After it has been sorted, all the dots lie on a diagonal line from bottom left to top right. The way in which the dots move to reach their final position provides a visualization of how the sort algorithm works. Study the way in which the calls to the procedures in SortViewLib have been put in SortViewBubble and construct similar programs to show the data movements for the other sorting algorithms described in this chapter.

Chapter 10 The Seven Step Method

This chapter implements solutions as computer programs, and then demonstrates how to verify the correctness of solutions through testing. To reach this solution, the seven step problem-solving method—introduced in the Principles book—will be reviewed and discussed. Although we'll concentrate on the implementation in Pascal of an already designed program, we cannot ignore the design.

Chapter Overview

10.1	Method: Part II.....	425
	The Seven Step Method.....	425
10.2	Design Stage: Acme Payroll System.....	426
1.	Problem Definition.....	426
	Problem Definition Application.....	426
2.	Solution Design.....	427
	Solution Design Application.....	427
3.	Solution Refinement.....	428
	Solution Refinement Application.....	428
4.	Testing Strategy Development.....	429
	Testing Strategy Development Application.....	429
10.3	Implementation Stage: Acme Payroll System.....	431
5.	Program Coding and Testing.....	431
	Case Study: The Acme Payroll System.....	432
6.	Documentation Completion.....	444
	Documentation Completion Application.....	444
7.	Program Maintenance.....	445
	Program Maintenance Application.....	445
10.4	An Advanced Case Study: Building a Text Index.....	445
	Design Stage.....	446
1.	Problem Definition.....	446
2.	Solution Design.....	446
3.	Solution Refinement.....	447
	Binary Search Tree Unit.....	449
	Queues Unit.....	451
	Implementation Stage.....	452
4.	Testing Strategy Development.....	452
5.	Program Coding and Testing.....	453
6.	Documentation.....	463
7.	Program Maintenance.....	463
10.5	Chapter 10 Review.....	464
10.6	Chapter 10 Programming Problems.....	464
1.	Editor Application.....	464
2.	Typing.....	470
3.	Calculator Applications.....	472
10.7	Chapter 10 Programming Projects.....	473

10.8	Level 1 — Getting Started.....	473
1-1.	General.....	474
	A Guessing Game.....	474
1-2.	Business.....	474
	Computing a Customer's Change.....	474
1-3.	Scientific.....	475
	A Bouncing Ball.....	475
10.9	Level 2 — Getting Organized with Procedures.....	477
2-1.	General.....	477
	Your Age in Days.....	477
2-2	Business.....	477
	What's the Cost of My Mortgage?.....	477
2-3	Scientific.....	478
	Solving the Quadratic Equation.....	478
10.10	Level 3 — Getting Fancier with Parameters.....	479
3-1.	General.....	479
	Count the Word Occurrences in a Text.....	479
	Processing Personnel Data.....	480
3-3.	Scientific.....	481
	Plotting a Function.....	481
10.11	Level 4 — Getting Your Wings with Units.....	483
4-1	General.....	483
	The Kwic Index.....	483
4-2	Business.....	484
	Information Retrieval.....	484
4-3	Scientific.....	486
	Complex Algebra.....	486

10.1 Method: Part II

In Chapter 10 of the Principles book, you have seen a second and more thorough presentation of the method for solving a problem with a computer, which was introduced in Chapter 2 of the same book.

The preceding chapters have introduced you to a number of practical tools based on the Pascal programming language. These will be helpful when you come to the Coding and Testing step of the method. This step is only the fifth step of the method, and that means that you must do the first four steps before starting programming. We'll review very briefly the seven step method here, as it was already well covered in the Principles book.

When solving large problems with a computer, it is absolutely necessary to design the solutions with great care, since rushing into the coding of a program (step 5) before the solution has been carefully defined is a very common and big mistake. The aim of this chapter is to review the implementation of solutions as computer programs, and the verification of the correctness of the solutions through testing.

The Seven Step Method

In Chapter 2 of the Principles book, we introduced a seven-step problem-solving method, in order to help you make sure your solutions are well designed before you start programming them.

The seven steps of the method are

1. Problem Definition
2. Solution Design
3. Solution Refinement
4. Testing Strategy Development
5. Program Coding and Testing
6. Documentation Completion
7. Program Maintenance

The first four steps are usually grouped together as part of the *Design* stage, while the last three steps belong to the *Implementation* stage. Although this chapter is concerned mainly with Step 5, the Program Coding and Testing, we will briefly review the other steps here so that Step 5 is seen in its proper context. These steps will be illustrated by following through the complete solution of a simple example.

10.2 Design Stage: Acme Payroll System

1. Problem Definition

This step produces a precise description of the problem to solve. The three major parts in the definition of the specifications is to determine

- what the input of the process is going to be,
- what output it is to produce
- and what the relationship between the two is.

Problem Definition Application

To illustrate all the steps, we'll consider the problem of developing a new computer system for the payroll of the Acme Company, as we have done in the Principles book.

The Acme Company payroll system must compute the pay of a list of hourly paid employees based on a line of input data per employee. The input data is kept in a file and each line of data corresponds to one employee and consists of three integers (the number of hours worked in $\frac{1}{100}$ ths of an hour, the hourly rate in cents, the number of dependents), followed by a character string (the employee name). These data will have the following format:

```
3050 1025 8 Gabrotil Michael
```

The end of the data will be indicated by the end of the file.

The computation of the gross pay is done by multiplying the hours worked by the hourly rate. Hours above 40 are to be paid one and a half times the normal hourly rate. For each dependent, the employee gets an exemption of \$20 for tax withholding computation. Federal and state withholdings are computed by applying rates of 18% and 3%, respectively, to the taxable amount. Social security withholdings are computed by applying a 5% rate to the gross pay. Net pay is computed by subtracting the various withholdings from the gross pay. Results must be displayed on one line per employee: name, gross pay, federal withholdings, state withholdings, social security withholdings, and net pay, using the format

```
Gabrotil Michael      312.63  27.47  4.58  7.63  272.95
```

At the end of processing, a summary line with the totals of the various withholdings and pay categories should be displayed, using a similar format, so that the results are aligned with the preceding individual columns.

Input data must be validated before they are used. The following validity ranges should be used.

The number of hours worked cannot be negative or greater than 55.

The hourly rate cannot be less than \$3.50 or more than \$16.50.

The number of dependents cannot be negative or greater than 12.

2. Solution Design

In this step, we decompose the problem we have to solve into its major components. We analyze the problem statement and divide it into a number of subproblems. Each subproblem is itself then divided into smaller subproblems, and this decomposition is continued until we have subproblems whose solutions are simple and are easily converted into a computer program.

The result of the solution design step consists of the basic structural design for the computer solution to the problem.

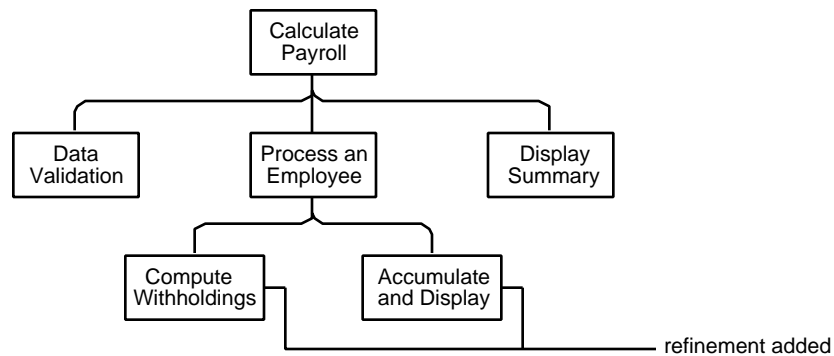
Solution Design Application

Based on the specifications for the Acme Payroll System developed in the previous step, we define all the tasks that have to be done in order to produce the payroll. This first level of decomposition is simple. The system will have to:

- read in and validate the employees' pay data,
- process the data and display the results for each employee
- display a summary of the payroll operation.

This leads us to the structure chart of Figure 10.1.

Figure 10.1 Structure chart for payroll system



This structure chart corresponds to the following very high level pseudocode:

```

Calculate Payroll
  Initialize
  For each employee
    Read and validate data
    Process employee
  Display summary
End Calculate Payroll
  
```

In the structure chart Initialize does not appear, as it is so simple it is part of the main program Calculate Payroll. Process employee is further broken down into three sub-tasks:

```
Process employee
  Compute Gross Pay
  Compute Withholdings
  Accumulate totals and display
    results for employee
End Process employee
```

Here again, Compute Gross Pay does not appear on the structure chart because it is so simple it does not justify a separate box.

3. Solution Refinement

Using the structure chart developed in the previous step as a starting point, we refine this solution by adding more detail. Each box in the structure chart must be converted into a detailed pseudocode algorithm. The specification of our data interfaces between functions is made more detailed by giving precise descriptions of the data, and also by using parameter lists for our algorithms.

Solution Refinement Application

At this stage we define data types for the variables used for input, major processing, and output. We define a record variable, Employee Data, with fields Name, of type String, and Hours, Rate and Dependents of type Integer. We also specify simple Integer variables Gross Pay, State Tax, Federal Tax, Soc Sec Tax, Net Pay, Gross Total, Soc Sec Total, Federal Total, State Total, and Net Total. We also define a Boolean variable Valid Data to indicate the result of the data validation.

Using pseudocode, we develop the algorithms for each functional part of the solution.

```
Calculate Payroll
  Display title and column headings
  Set all totals to zero
  While there are data to read
    Input Employee Data
    Data Validation(Employee Data, Valid Data)
    If Valid Data
      Process Employee(Employee Data, Totals)
    Display Summary(Totals)
  End Calculate Payroll
```

In order to simplify the pseudocode, we use collective names to represent groups of variables. We develop pseudocode algorithms for the following structure chart components:

```
Data Validation(Employee Data, Valid Data)
```

Process Employee(Employee Data, Totals)

*Compute Withholdings(Gross Pay, Employee Data,
Withholdings)*

*Accumulate and Display(Employee Data, Pay Data,
Totals)*

The complete pseudocode for these components was developed in Chapter 10 of the Principles book, and we won't copy it here, as the program code in step 5 below is more accurate.

4. Testing Strategy Development

Although we cannot test a program before it is written, it is important to devise a plan for testing before actually coding the program. If we do this, developing a testing strategy and the test cases that go with it will help us find errors in the design.

Although the approach chosen for testing may be top-down, bottom-up, or a combination of both, we'll concentrate on top-down testing. This means starting the development of the program at the top of the structure chart and working down component by component. This also means that we'll have to use program *stubs* for the lower level components during the initial steps of development. This way, the testing will be done in stages as the program development progresses.

In stubs it is usual that a message indicating that the procedure has been called is displayed. The stubs are later replaced by a complete procedure that actually does the desired processing.

The result of this step is a description of the testing strategy, input data for all test cases along with the corresponding expected output, and pseudocode for stubs when necessary.

Testing Strategy Development Application

For our payroll program, we identify the various cases including abnormal values (cases 1-3 below), and normal values (cases 4-8):

1. Hours worked negative or greater than 55.
2. Rate less than \$3.50 or greater than \$16.50.
3. Number of dependents negative or greater than 12.
4. No overtime.
5. With overtime.
6. No dependents.
7. With dependents.
8. Zero hours.

From these cases, we make up our test data.

Hours	Rate	Dep.	Name
-10	400	2	hours-negative
5510	400	2	hours-too-large
1050	349	2	rate-too-low
1050	1651	2	rate-too-high
1050	400	-1	dependents-negative
1050	400	13	dependents-too-large
6000	100	3	hours-and-rate
6000	400	14	hours-and-dependents
1050	100	14	rate-and-dependents
6000	100	14	hours-rate-dependents
4000	400	0	no-overtime-no-dep
4000	400	2	no-overtime-with-dep
5500	400	0	max-overtime-no-dep
5500	600	12	maximum-hours-and-dep
4000	1650	12	maximum-rate-and-dep
0	500	2	zero-hour-and-dep

The expected results for these test data must be computed manually and given here.

Invalid hours for hours-negative
 Invalid hours for hours-too-large
 Invalid rate for rate-too-low
 Invalid rate for rate-too-high
 Invalid dependents for dependents-negative
 Invalid dependents for dependents-too-large
 Invalid hours for hours-and-rate
 Invalid rate for hours-and-rate
 Invalid hours for hours-and-dependents
 Invalid dependents for hours-and-dependents
 Invalid rate for rate-and-dependents
 Invalid dependents for rate-and-dependents
 Invalid hours for hours-rate-dependents
 Invalid rate for hours-rate-dependents
 Invalid dependents for hours-rate-dependents

Name	Gross	Fed	State	Soc	Net
no-overtime-no-dep	160.00	28.80	4.80	8.00	118.40
no-overtime-with-dep	160.00	21.60	3.60	8.00	126.80
max-overtime-no-dep	250.00	45.00	7.50	12.50	185.00
maximum-hours-and-dep	375.00	24.30	4.05	18.75	327.90
maximum-rate-and-dep	660.00	75.60	12.60	33.00	538.80
zero-hour-and-dep	0.00	0.00	0.00	0.00	0.00
Totals	1605.00	195.30	32.55	80.25	1296.90

10.3 Implementation Stage: Acme Payroll System

5. Program Coding and Testing

The actual computer program is coded from our pseudocode algorithms. Then, using our testing strategy, we test the program. For large programs, using the top-down approach, the coding will be done progressively, the various components being coded with the necessary stubs, so that they can be tested systematically. The programming and coding step is finished when all the coding has been done and when all the test data have been correctly processed.

The results of this step are the program, able to run with a set of test data, along with information on the testing that has been done.

The written program should follow some common guidelines for good programming style.

- Develop programs using a top-down approach so that they are well structured, with procedures of reasonable size (not longer than a page or two) and no large blocks of code.
- Use the basic control structures (sequence, selection, repetition) to write your code. The flow of control should be as straightforward as possible: blocks of code should be entered only at the top and be exited only at the bottom.
- Strive for simplicity and clarity: there should be no “clever” coding that makes the program hard to understand.
- Use parameters to transfer information to and from procedures or functions and do not use global data whenever possible.
- Avoid side effects in functions, such as returning VAR parameters or changing the value of global variables.
- Declare variables as local as possible.
- Use status VAR parameters to signal error conditions during execution of a procedure or a function.
- Use symbolic constants to improve readability and portability.
- Each program should have an extensive preface, including a brief statement of the problem, appropriate references to external documents, name of the original programmer or team, date of the original implementation, and a change log. The change log should have an entry for each significant alteration of the program, including the date of the change, the name of the person who made the change, and a brief description of the change.
- Each procedure or function should be documented in a similar manner.
- Choose meaningful identifiers to improve program readability.
- Use uppercase and lowercase letters to improve program readability.

- Use blank lines to separate components, and use indentation to indicate the flow of control that will be followed at runtime.
- Meaningful comments that explain the program should be included. Difficult sections of code should contain explanatory comments. The pseudocode developed earlier may sometimes be used as comments.

Case Study: The Acme Payroll System

The program of our case study provides an example that adheres to these guidelines. We will develop it using a top-down approach. Our first version will comprise the main program with stubs for the three main functions of Figure 10.1.

```

PROGRAM Payroll;
(* This program computes the weekly pay of hourly paid *)
(* employees. It reads lines of data comprising for each *)
(* employee: hours worked, hourly rate, number of depen- *)
(* dents and name of employee. The validated data is then *)
(* used to compute gross pay, federal withholdings (18% *)
(* of taxable income), state withholdings (3% of taxable *)
(* income), social security withholdings (5% of gross pay) *)
(* and net pay, which are printed for each employee. At *)
(* the end of processing, the program prints the accu- *)
(* mulated totals of each category *)
(* *)
(* Philip Marcmotil December 1993 *)

CONST Blanks = ' ';
      Title = ' Computation of weekly pay';
      Header = 'Gross Fed State Soc Net';

TYPE TotalsType = RECORD
    FederalTotal: LONGINT;
    StateTotal: LONGINT;
    SSTotal: LONGINT;
    GrossTotal: LONGINT;
    NetTotal: LONGINT;
END;

EmpRecordType = RECORD
    Name: STRING;
    Dependents: LONGINT;
    Hours: LONGINT;
    Rate: LONGINT;
END;

PROCEDURE DataValidation( EmployeeData: EmpRecordType;
                          VAR ValidData: BOOLEAN);
(* Validation of hours, rate and dependents data *)
BEGIN

```



```

        ValidData := TRUE;
        WriteLn('DataValidation');
    END; { DataValidation }

PROCEDURE ProcessEmployee( EmployeeData: EmpRecordType;
                           VAR TotalList: TotalsType);
(* Processing of data for one employee *)
BEGIN
    WriteLn('ProcessEmployee');
END; { ProcessEmployee }

PROCEDURE DisplaySummary(TotalList: TotalsType);
(* Print a summary of the payroll computation *)
BEGIN
    WriteLn('DisplaySummary');
END; { DisplaySummary }

VAR Employee: EmpRecordType;
    Valid:    BOOLEAN;
    DataFile: TEXT;
    Totals: TotalsType;

BEGIN
    Assign(DataFile, 'Payroll.data');
    Reset(DataFile);
    WriteLn(Blanks, Title);
    WriteLn(Blanks, Header);

    WITH Totals DO BEGIN
        FederalTotal := 0;
        StateTotal   := 0;
        SSTotal       := 0;
        GrossTotal    := 0;
        NetTotal      := 0;
    END;

    WHILE NOT Eof(DataFile) DO BEGIN
        WITH Employee DO
            ReadLn(DataFile, Hours, Rate, Dependents, Name);
            DataValidation(Employee, Valid);
            IF Valid THEN
                ProcessEmployee(Employee, Totals);
        END;
        DisplaySummary(Totals);
        Close(DataFile);
    END. { Payroll }

```

This version can be compiled and run with our test data. It produces the following output.

Computation of weekly pay

[illegible]

This shows that all the input data were read (16 lines of data) and that the procedures were called in the right order. Our next version will develop code for the three stubs: `DataValidation`, `ProcessEmployee` and `DisplaySummary`, and in so doing introduce three new stubs: `ComputeWithholdings`, `AccumulateAndDisplay` and `PrintDollars`. The last of these, `PrintDollars`, did not appear in the pseudocode version of the program but was found necessary for the printing of monetary values during the elaboration of the previous stubs.

```
PROGRAM Payroll;
(* This program computes the weekly pay of hourly paid    *)
(* employees. It reads lines of data comprising for each *)
(* employee: hours worked, hourly rate, number of depen- *)
(* dents and name of employee. The validated data is then*)
(* used to compute gross pay, federal withholdings (18%  *)
(* of taxable income), state withholdings (3% of taxable *)
```

```

(* income), social security withholdings (5% of grosspay)*)
(* and net pay, which are printed for each employee. At *)
(* the end of processing, the program prints the accu- *)
(* mulated totals of each category *)
(*)
(*) Philip Marcmotil December 1993 (*)

CONST Blanks = ' ';
      Title = ' Computation of weekly pay';
      Header = 'Gross Fed State Soc Net';

TYPE TotalsType = RECORD
      FederalTotal: LONGINT;
      StateTotal: LONGINT;
      SSTotal: LONGINT;
      GrossTotal: LONGINT;
      NetTotal: LONGINT;
    END;
EmpRecordType = RECORD
      Name: STRING;
      Dependents: LONGINT;
      Hours: LONGINT;
      Rate: LONGINT;
    END;
PayRecordType = RECORD
      Name: STRING;
      FederalTax: LONGINT;
      StateTax: LONGINT;
      SSTax: LONGINT;
      GrossPay: LONGINT;
      NetPay: LONGINT;
    END;

PROCEDURE PrintDollars(Amount, Width: LONGINT);
(* Print dollar Amount with period and cents in *)
(* given Width *)
BEGIN
  Write('PrintDollars ');
END; { PrintDollars }

PROCEDURE DataValidation( EmployeeData: EmpRecordType;
      VAR ValidData: BOOLEAN);
(* Validation of hours, rate and dependents data *)
CONST MinHours = 0;
      MaxHours = 5500; (* 55 hours *)
      MinRate = 350; (* $2.50 *)
      MaxRate = 1650; (* $12.50 *)
      MinDep = 0;
      MaxDep = 12;
BEGIN

```

```
WITH EmployeeData DO BEGIN
    ValidData := TRUE;
    IF (Hours < MinHours) OR (Hours > MaxHours) THEN BEGIN
        ValidData := FALSE;
        Writeln('Invalid hours for      ', Name);
    END;
    IF (Rate < MinRate) OR (Rate > MaxRate) THEN BEGIN
        ValidData := FALSE;
        Writeln('Invalid rate for      ', Name);
    END;
    IF (Dependents < MinDep) OR
        (Dependents > MaxDep) THEN BEGIN
        ValidData := FALSE;
        Writeln('Invalid dependents for ', Name);
    END;
END;
END; { DataValidation }

PROCEDURE ProcessEmployee(    EmployeeData: EmpRecordType;
                             VAR TotalList: TotalsType);
(* Processing of data for one employee *)
CONST NormalHours = 4000; { 40 hours      }
      Overtime     =    5; { 0.5 normal rate }

VAR Pay: PayRecordType;

PROCEDURE ComputeWithholdings(    Dependents: LONGINT;
                                 VAR ComputedPay: PayRecordType);
(* Computation of federal, state and *)
(* social security withholdings      *)
BEGIN
    Writeln('ComputeWithholdings');
    WITH ComputedPay DO BEGIN
        FederalTax := 0;
        StateTax   := 0;
        SSTax      := 0;
    END;
END; { ComputeWithholdings }

PROCEDURE AccumulateAndDisplay(    ComputedPay:
                                   PayRecordType;
                                   VAR Totals: TotalsType);
(* Accumulation of totals of data and *)
(* printing of employee data          *)
BEGIN
    Writeln('AccumulateAndDisplay');
END; { AccumulateAndDisplay }

BEGIN { ProcessEmployee }
    Pay.Name := EmployeeData.Name;
```

```

    WITH EmployeeData, Pay DO BEGIN
        GrossPay := Hours * Rate DIV 100;
        { Hours given in 0.01 of hour }
        IF Hours > NormalHours THEN
            GrossPay := GrossPay + Overtime * Rate
                * (Hours - NormalHours) DIV 1000;
            { Overtime/10 and hours/100 }
        ComputeWithholdings(Dependents, Pay);
        NetPay := GrossPay - FederalTax - StateTax - SSTax;
        AccumulateAndDisplay(Pay, TotalList);
    END;
END; { ProcessEmployee }

PROCEDURE DisplaySummary(TotalList: TotalsType);
(* Print a summary of the payroll computation *)
BEGIN
    WITH TotalList DO BEGIN
        WriteLn;
        Write(' Totals ');
        PrintDollars(GrossTotal, 9);
        PrintDollars(FederalTotal, 9);
        PrintDollars(StateTotal, 9);
        PrintDollars(SSTotal, 9);
        PrintDollars(NetTotal, 9);
        WriteLn
    END;
END; { DisplaySummary }

VAR Employee: EmpRecordType;
    Valid:    BOOLEAN;
    DataFile: TEXT;
    Totals:   TotalsType;

BEGIN
    Assign(DataFile, 'Payroll.data');
    Reset(DataFile);
    WriteLn(Blanks, Title);
    WriteLn(Blanks, Header);

    WITH Totals DO BEGIN
        FederalTotal := 0;
        StateTotal   := 0;
        SSTotal       := 0;
        GrossTotal    := 0;
        NetTotal      := 0;
    END;

    WHILE NOT Eof(DataFile) DO BEGIN
        WITH Employee DO
            ReadLn(DataFile, Hours, Rate, Dependents, Name);

```

```

        DataValidation(Employee, Valid);
        IF Valid THEN
            ProcessEmployee(Employee, Totals);
        END;
        DisplaySummary(Totals);
        Close(DataFile);
    END. { Payroll }

```

Note that procedures `ComputeWithholdings` and `AccumulateAndDisplay` have been declared local to procedure `ProcessEmployee`. This is to keep things as local as possible, and also to keep as close as possible to the structure shown in the structure chart. With our input data, this version of the program produced the following output.

	Computation of weekly pay				
	Gross	Fed	State	Soc	Net
Invalid hours for			hours-negative		
Invalid hours for			hours-too-large		
Invalid rate for			rate-too-low		
Invalid rate for			rate-too-high		
Invalid dependents for			dependents-negative		
Invalid dependents for			dependents-too-large		
Invalid hours for			hours-and-rate		
Invalid rate for			hours-and-rate		
Invalid hours for			hours-and-dependents		
Invalid dependents for			hours-and-dependents		
Invalid rate for			rate-and-dependents		
Invalid dependents for			rate-and-dependents		
Invalid hours for			hours-rate-dependents		
Invalid rate for			hours-rate-dependents		
Invalid dependents for			hours-rate-dependents		
ComputeWithholdings					
AccumulateAndDisplay					
ComputeWithholdings					
AccumulateAndDisplay					
ComputeWithholdings					
AccumulateAndDisplay					
ComputeWithholdings					
AccumulateAndDisplay					
ComputeWithholdings					
AccumulateAndDisplay					
Totals	PrintDollars	PrintDollars	PrintDollars	PrintDollars	
	PrintDollars				

This output shows that our validation process works correctly, and that, for normal cases, the procedures are called in the correct order.

Our last version is complete and produces the expected results after some slight adjustments to the spacing in the Header string constant so that the column

names print properly over the columns (addition of constant NameLen). Such adjustments are to be expected after one can see exactly what is being printed out.

```

PROGRAM Payroll;
(* This program computes the weekly pay of hourly paid *)
(* employees. It reads lines of data comprising for each *)
(* employee: hours worked, hourly rate, number of depen- *)
(* dents and name of employee. The validated data is then *)
(* used to compute gross pay, federal withholdings (18% *)
(* of taxable income), state withholdings (3% of taxable *)
(* income), social security withholdings (5% of grosspay) *)
(* and net pay, which are printed for each employee. At *)
(* the end of processing, the program prints the accu- *)
(* mulated totals of each category *)
(* *)
(* Philip Marcmotil December 1993 *)

CONST Blanks = ' ';
      Title = ' Computation of weekly pay';
      Header = 'Gross Fed State Soc Net';
      NameLen = 23;

TYPE TotalsType = RECORD
      FederalTotal: LONGINT;
      StateTotal: LONGINT;
      SSTotal: LONGINT;
      GrossTotal: LONGINT;
      NetTotal: LONGINT;
    END;

EmpRecordType = RECORD
      Name: STRING;
      Dependents: LONGINT;
      Hours: LONGINT;
      Rate: LONGINT;
    END;

PayRecordType = RECORD
      Name: STRING;
      FederalTax: LONGINT;
      StateTax: LONGINT;
      SSTax: LONGINT;
      GrossPay: LONGINT;
      NetPay: LONGINT;
    END;

PROCEDURE PrintDollars(Amount, Width: LONGINT);
(* Print dollar Amount with period and cents in *)
(* given Width *)
VAR Cents: LONGINT;
BEGIN

```

```
Write(Amount DIV 100: Width-3, '.');
Cents := Amount MOD 100;
IF Cents < 10 THEN
    Write('0', Cents: 1)
ELSE
    Write(Cents: 2);
END; { PrintDollars }

PROCEDURE DataValidation(    EmployeeData: EmpRecordType;
                           VAR ValidData: BOOLEAN);
(* Validation of hours, rate and dependents data *)
CONST MinHours =    0;
      MaxHours = 5500; (* 55 hours *)
      MinRate  =  350; (* $2.50 *)
      MaxRate  = 1650; (* $12.50 *)
      MinDep   =    0;
      MaxDep   =   12;
BEGIN
    WITH EmployeeData DO BEGIN
        ValidData := TRUE;
        IF (Hours < MinHours) OR (Hours > MaxHours) THEN BEGIN
            ValidData := FALSE;
            Writeln('Invalid hours for      ', Name);
        END;
        IF (Rate < MinRate) OR (Rate > MaxRate) THEN BEGIN
            ValidData := FALSE;
            Writeln('Invalid rate for      ', Name);
        END;
        IF (Dependents < MinDep) OR
            (Dependents > MaxDep) THEN BEGIN
            ValidData := FALSE;
            Writeln('Invalid dependents for ', Name);
        END;
    END;
END; { DataValidation }

PROCEDURE ProcessEmployee(    EmployeeData: EmpRecordType;
                           VAR TotalList: TotalsType);
(* Processing of data for one employee *)
CONST NormalHours = 4000; { 40 hours      }
      Overtime    =    5; { 0.5 normal rate }

VAR Pay: PayRecordType;

PROCEDURE ComputeWithholdings(    Dependents: LONGINT;
                                VAR ComputedPay:
PayRecordType);
(* Computation of federal, state and *)
(* social security withholdings      *)
CONST FederalRate =  18; (* 18%      *)
```



```

        StateRate   =    3; (* 3% *)
        SSRate      =    5; (* 5% *)
        Exemption   = 2000; (* $20.00 *)
VAR Taxable: LONGINT;
BEGIN
    WITH ComputedPay DO BEGIN
        Taxable := GrossPay - Dependents * Exemption;
        IF Taxable > 0 THEN BEGIN
            FederalTax := Taxable * FederalRate DIV 100;
            StateTax   := Taxable * StateRate DIV 100;
        END
        ELSE BEGIN
            FederalTax := 0;
            StateTax   := 0;
        END;
        SSTax := GrossPay * SSRate DIV 100;
    END;
END; { ComputeWithholdings }

PROCEDURE AccumulateAndDisplay(    ComputedPay:
                                   PayRecordType;
                                   VAR Totals: TotalsType);
(* Accumulation of totals of data and *)
(* printing of employee data          *)
BEGIN
    WITH ComputedPay, Totals DO BEGIN
        FederalTotal := FederalTotal + FederalTax;
        StateTotal   := StateTotal + StateTax;
        SSTotal      := SSTotal + SSTax;
        GrossTotal   := GrossTotal + GrossPay;
        NetTotal     := NetTotal + NetPay;
        Write(Name: NameLen);
        PrintDollars(GrossPay, 9);
        PrintDollars(FederalTax, 9);
        PrintDollars(StateTax, 9);
        PrintDollars(SSTax, 9);
        PrintDollars(NetPay, 9);
        Writeln
    END;
END; { AccumulateAndDisplay }

BEGIN { ProcessEmployee }
Pay.Name := EmployeeData.Name;
WITH EmployeeData, Pay DO BEGIN
    GrossPay := Hours * Rate DIV 100;
    { Hours given in 0.01 of hour }
    IF Hours > NormalHours THEN
        GrossPay := GrossPay +
            Overtime * Rate * (Hours - NormalHours) DIV 1000;
    { Overtime/10 and hours/100 }

```

```
        ComputeWithholdings(Dependents, Pay);
        NetPay := GrossPay - FederalTax - StateTax - SSTax;
        AccumulateAndDisplay(Pay, TotalList);
    END;
END; { ProcessEmployee }

PROCEDURE DisplaySummary(TotalList: TotalsType);
(* Print a summary of the payroll computation *)
BEGIN
    WITH TotalList DO BEGIN
        WriteLn;
        Write(' Totals ': NameLen);
        PrintDollars(GrossTotal, 9);
        PrintDollars(FederalTotal, 9);
        PrintDollars(StateTotal, 9);
        PrintDollars(SSTotal, 9);
        PrintDollars(NetTotal, 9);
        WriteLn
    END;
END; { DisplaySummary }

VAR Employee: EmpRecordType;
    Valid:    BOOLEAN;
    DataFile: TEXT;
    Totals:   TotalsType;

BEGIN
    Assign(DataFile, 'Payroll.data');
    Reset(DataFile);
    WriteLn(Blanks, Title);
    WriteLn(' ': NameLen + 4, Header);

    WITH Totals DO BEGIN
        FederalTotal := 0;
        StateTotal   := 0;
        SSTotal       := 0;
        GrossTotal    := 0;
        NetTotal      := 0;
    END;

    WHILE NOT Eof(DataFile) DO BEGIN
        WITH Employee DO
            ReadLn(DataFile, Hours, Rate, Dependents, Name);
            DataValidation(Employee, Valid);
            IF Valid THEN
                ProcessEmployee(Employee, Totals);
        END;
        DisplaySummary(Totals);
        Close(DataFile);
    END. { Payroll }
```

Program testing should not be too difficult if the program has been well designed, and if the design has been closely followed and coded using good programming style.

However, even if we are very careful, some semantic errors (the program does not produce the desired results) usually occur, and the effort needed to find and correct these errors is never negligible. To make the debugging process less onerous, it is useful to follow some debugging guidelines.

- First, make sure that the results given by the program are really erroneous before investing time to find the error. In particular, be certain that the pre computed results that go with the test cases are right.
- Do a walk through the code with the input data to see if you can locate the error quickly. This is also called “desk checking” or producing a trace of the program execution.
- Either by inserting write statements at key points in the program or by using a symbolic debugger, try to trace the location of the error to a small segment of the program. For instance, if you think that a given procedure may be the source of the error, insert write statements or breakpoints at the beginning of that procedure to examine arguments passed to it, and at the end of the procedure to examine the results it computed.
- If your program is large and if you haven’t been able to isolate the error, reduce your program to a simplified version. Such a version of the program is a copy where some segments of code have been deleted. This will require some planning and also some effort, but with the tools at your disposal (text editor, symbolic debugger) is well worth it.

Your Pascal system most certainly contains a symbolic debugger, and it will be well worth your time to learn how to use it effectively. It will allow you to carefully control the execution of any program and to examine the state of the program. The debugger will display the source program, the procedure calls chain, and the values of variables and parameters so that you will know exactly what the program is doing. Using the source program displayed, you can set breakpoints, that is, places where execution will temporarily halt. You can examine values of any variables visible at that point in the program and you can even modify these values if you wish. You can also change the execution mode of the program to single step, so that one statement is executed at a time. Since a sophisticated symbolic debugger is available on most systems, you should invest some time in learning how to use it, because this will improve your programming efficiency in a noticeable way.

6. Documentation Completion

Documentation is a vital part of any program. In particular the user needs some kind of user’s manual to know how to use the program. But this manual is only a small part of the program’s documentation. In fact all the steps in the problem solving method produce some sort of documentation: program specifications,

structure and modular charts, pseudocode and data definition, test data and results, program code. In this step, you must collect all the documentation pieces, and integrate them in what is called *external documentation*. *Internal documentation* is part of the program code in the form of comments, but also of well chosen variable and constant names, and of indentation to show clearly the structure of the program.

Documentation Completion Application

The payroll system documentation will include the problem definition, the design documents, a description of the testing performed, a history of the program development and its different versions, and a user's manual. Such a manual was developed in the Principles book. We'll show that manual here again to be complete.

Acme Payroll User's Manual

Payroll is a program to compute and display the weekly pay of hourly paid employees. For each employee it will read a series of four data items separated by at least one blank: number of hours worked during the week (0-55), hourly rate of pay (\$3.50-\$16.50), number of dependents (0-12), and name of employee (20 characters). If the data are valid, the program will compute and display federal (18% of taxable income), state (3% of taxable income), and social security (5% of gross pay) withholdings, as well as gross and net pay for the employee.

- The program will read data for each employee from the file Payroll.data until it reaches the end of the file. It will produce results on the screen. Input data format is such that three integer values precede a character string, as in
4500 600 3 Allan Mackenzie
- Output data will be written one line at a time, each line corresponding to an employee. A normal output line will consist of a 20-character string followed by five values. The last line will contain the word "Totals" followed by five values and will be separated from the previous output line by a blank line. The output will be preceded by the title lines:

```
Computation of Weekly Pay
Gross      Fed   State  Soc.   Net
```

- A normal output line will look like
Allan Mackenzie 285.00 40.50 6.75 14.25 223.50
- Erroneous data will produce error messages of the form
Invalid hours for Robert A. Verner

```
Invalid rate for      Simon J. C. W. Surry  
Invalid dependents for T. Guy Rimmer
```

- These messages will appear if the values read are not within the given limits. A single employee data line with erroneous data can generate from one to three error messages. The erroneous data have to be corrected and resubmitted.

To run the program, enter the payroll data in file Payroll.data and execute Payroll.

7. Program Maintenance

As you already know, program maintenance is not directly part of the original implementation process. Many large programs have long lifetimes that often exceed the lifetime of the hardware they run on. Maintenance includes all activities that occur after a program first becomes operational, and in particular:

- the discovery and elimination of program errors,
- the modification of the current program,
- the addition of new features, and
- the updating of the documentation.

The documentation must be complete and accurate: don't forget that most program maintenance is done by someone not involved in the original design.

Program Maintenance Application

The changes and improvements to the payroll program might have the program actually produce the employees' paychecks, or add a unit to compute tax withholdings in a more flexible way, or even add an interactive way of fixing the erroneous data.

10.4 An Advanced Case Study: Building a Text Index

We'll now look at a somewhat larger case study, involving several units. The design of this case study was done in Chapter 10 of the Principles book, so we'll only give a summary of it here, and concentrate on the implementation.

Design Stage

1. Problem Definition

We are designing a program that reads in a text stored in a given file, that collects all the significant words of that text together with the page numbers where the words occur, and that displays an alphabetical index of the words with their page numbers.

The program prompts the user for the name of the file of trivial words not to be included in the index, for the name of the text file, and the name of the new index file.

```
Give name of trivial words file:
Give name of text file:
Give name of output file:
Index complete
```

The format of the index will be the following:

June	1	8						
Karine	1	2	3	4	5	6	7	8
	9	10						
Kludge	5	9						

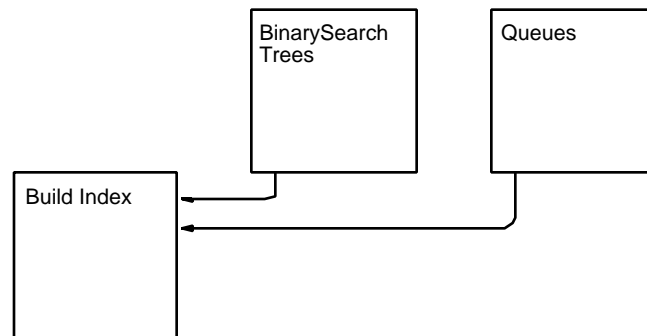
2. Solution Design

The program will keep the index in a *binary search tree* (illustrated in Chapter 9). This structure has the property of keeping its elements ordered, which makes it possible to display its contents easily in order. We will use a binary search tree unit, which will encapsulate its representation and operations.

The page numbers associated with a word will be kept in a *queue*, where each page number is unique and will be output in the order of insertion. We'll use a separate unit for the ADT queue, similar to `QueueLib` seen in Chapter 9.

Thus, our solution is based on two library units, one for binary search trees and the other for queues, as shown in Figure 10.2.

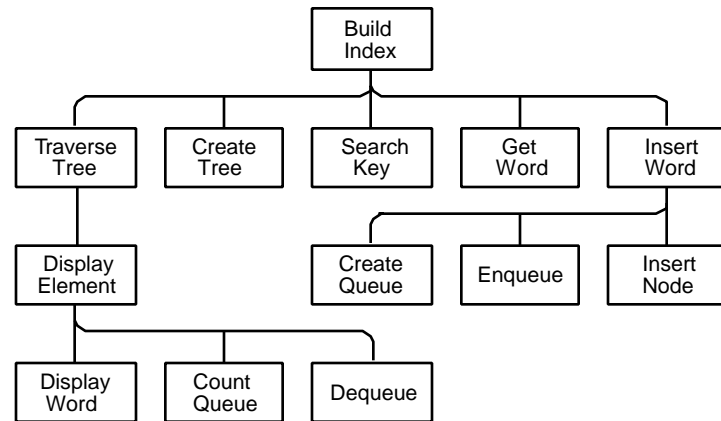
Figure 10.2 Modular design chart for building a text index



We'll use two binary search trees, one for the trivial words, and another one for the significant words. From the binary search tree operations, we'll use the initialization of a tree, the search for a word in a tree, the insertion of an element in a tree, and the traversal of a tree to display its contents.

We will use a queue structure to store all the page numbers associated with a word, and the queue operations to initialize a queue, to insert an element in a queue, to eliminate an element from a queue, and to count the elements in a queue. Figure 10.3 shows the complete structure chart for the Build Index program.

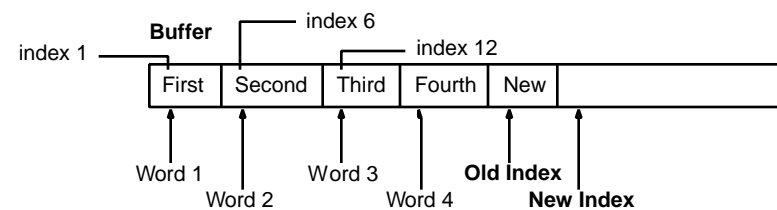
Figure 10.3 Structure chart for building a text index



3. Solution Refinement

The words from the text are kept in a long buffer (storage area). The buffer is a big array of characters, and each word in it is identified by the index of its first character, as shown in Figure 10.4.

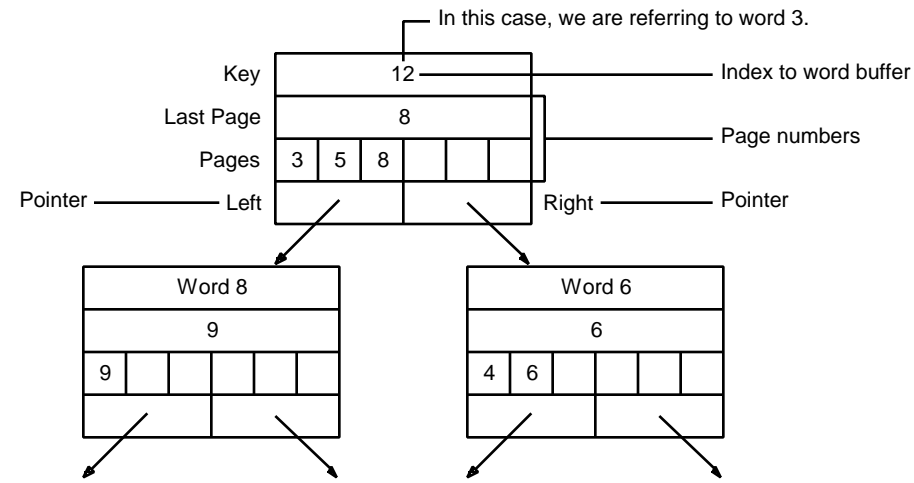
Figure 10.4 Word buffer



Associated with the buffer are two indices, Old Index and New Index. Normally, Old Index always points to the next free space in the buffer. However, when a new word is read, it is added to the buffer by advancing New Index which then points to the next free space after the new word. When it is decided to keep that new word Old Index is updated to the value of New Index. Otherwise the next word read is stored over the word that is not kept.

All the information for the index is kept in a binary search tree whose nodes have the structure shown in Figure 10.5.

Figure 10.5 Binary search tree nodes



Each indexed word is represented by a node, which has five parts: Key, an index to the word in the buffer, a Last page number (to avoid keeping duplicate page numbers), a Pages queue, and a left and a right pointer for the tree structure.

Using these data structures the Build Index program first initializes various variables, then reads the trivial words file and builds the trivial words tree. Then it reads the text and inserts the non-trivial words in the word index tree. Finally, it traverses the word index tree and displays the word index. We won't repeat here the pseudocode solution which was given in Chapter 10 of the Principles book. Let's just give the main processing loop for the building of the index binary search tree.

```

Input Character
While text not finished
  Select Character
  End of line:
    Terminate current line
    Increment and display line number
  Letter:
    Get Word(Text File, Character,
              Old Index, New Index)
    Search Word in trivial words tree
    If NOT trivial
      Insert Word(Index, Page number,
                  New Index, Old Index)
  End of page:
    Increment Page number
Input Character
  
```


The text is read character by character, end of line characters update the Line number, end of page characters update the Page number, while letters invoke Get Word, check to see if the word is significant and if it is, invoke Insert Word to insert it in the Index tree.

Get Word reads in a word, character by character until it finds a non digit or non letter character, and stores it temporarily in the big character buffer. To separate words in the buffer, Get Word also stores the word length in the first character of the word, in the format of Pascal Strings. Get Word does not modify Old Index but changes New Index.

Insert Word checks to see whether the word is already in the binary search tree. If it is, it only updates its element in the tree by adding, if necessary, a new page number to its associated queue. It also doesn't update Old Index so that the duplicate word is not kept in the buffer. For a new word, the associated queue is initialized. In the trivial words tree, trivial words don't need a page queue, and have a zero page number. The new word element is inserted into the tree and Old Index is updated to keep the word in the buffer.

An element is displayed by first displaying the associated word, followed by a number of spaces to align all index entries. Then, the page numbers are removed one by one from the Pages queue and displayed. They are also counted and, if they take more than one line, are displayed on the next lines after skipping the space underneath the word.

A word is displayed by displaying all its characters from the buffer, and then padding the rest of the word space with blanks.

Binary Search Tree Unit

A tree is defined as a pointer to a tree node, whose structure was just described (Figure 10.5). The way in which a binary search tree is constructed is shown in Chapter 9.

The BinSearchTrees unit implements the ADT binary search tree. It includes the operations that appear in the unit interface below.

```
UNIT BinSearchTrees;
(* This unit offers type BinarySearchTree and the      *)
(* operations that apply to it. The user can create and *)
(* build binary search trees with elements of type      *)
(* ElementType in the nodes.                            *)
(*                                     P. J. Gabrini    December 1993 *)
INTERFACE
USES Queues;

TYPE KeyType = INTEGER;
      ElementType = RECORD
                           Key:      KeyType;
                           LastPage: INTEGER;
                           Pages:    Queue;
                        END;
      BinarySearchTree = ^TreeNode;
```

```
TreeNode = RECORD
    Element: ElementType;
    Left, Right: BinarySearchTree;
END;

TraverseProc = PROCEDURE(E: ElementType);
CompProc      = FUNCTION(K1, K2: KeyType): INTEGER;
DisplayProc    = PROCEDURE(K: KeyType);

PROCEDURE DisplayKey(Element: KeyType);
(* Display a key value *)

PROCEDURE InitializeTree(VAR Tree: BinarySearchTree);
(* Must be called first, it initializes a Tree. *)

PROCEDURE DeleteTree(VAR Tree: BinarySearchTree);
(* All information about the Tree and the data
   records it contains are deleted. *)

PROCEDURE InsertNode(VAR Tree: BinarySearchTree;
    Element: ElementType;
    Comp: CompProc);
(* Insert Element at proper position into the Tree
   according to the Element's key. Use Comp for key
   comparisons. If a node already existed with same
   key, it is updated with value Element. *)

PROCEDURE DeleteNode(VAR Tree: BinarySearchTree;
    Key: KeyType;
    Compare: CompProc);
(* Find the element with this Key and delete it from
   the tree. Use Compare for key comparisons. If no
   node has this Key, Tree is unchanged. *)

PROCEDURE TraverseTree(VAR Tree: BinarySearchTree;
    Process: TraverseProc);
(* Apply Process to each node of Tree in order. *)

FUNCTION SearchKey(    Tree: BinarySearchTree;
    Key: KeyType;
    VAR Element: ElementType;
    Compare: CompProc): BOOLEAN;
(* Search for an Element identified by Key in the Tree.
   Use Compare for key comparisons. *)

PROCEDURE DisplayTree(Tree: BinarySearchTree;
    Indentation: INTEGER;
    DisplayKey: DisplayProc);
(* Display Tree keys with indentations to show Tree
   structure. The tree traversal operation is recursive
   (it calls itself on the left and right subtrees) and
```

very simple. *)

Note that unit `BinSearchTrees` uses unit `Queues` and that it defines the elements to be used as nodes of the binary search tree.

Queues Unit

The Queue Abstract Data Type will be defined in a slightly different way from what we defined in `QueueLib` at the end of Chapter 9. We'll use different, and more commonly used names for the two major operations, and add an operation to count the queue elements and an operation to examine the front element of the queue. These operations are described in the `Queues` unit interface below. A queue is implemented as a record with two indices, a counter, and an array of elements.

```
UNIT Queues;
(* This unit offers type Queue and all operations that *)
(* normally apply to queues. The user can create and *)
(* manipulate queues of elements of type QElementType. *)
(* P. J. Gabrini December 1993 *)
INTERFACE
  CONST MaxQueue = 50;
  TYPE QElementType = INTEGER;
  Queue = RECORD
    Head, Tail, Number: 0..MaxQueue;
    Data: ARRAY [1..MaxQueue] OF
QElementType;
  END;
  VAR QueueError: BOOLEAN; { Result of operation:
                             QueueError is always false
                             unless the operation failed }

  PROCEDURE InitializeQueue(VAR Q: Queue);
  (* Create an empty queue Q, must be called first *)

  PROCEDURE Enqueue(VAR Q: Queue; Item: QElementType);
  (* Insert element Item at the end of queue Q *)

  PROCEDURE Dequeue(VAR Q: Queue; VAR Item: QElementType);
  (* Delete first element of queue Q and return it in Item *)

  FUNCTION CountQueue(Q: Queue): INTEGER;
  (* Return the current number of elements in queue Q *)

  PROCEDURE QueueHead(Q: Queue; VAR Item: QElementType);
  (* Return value of first element of queue Q in Item *)
```

Note that this unit defines `Queues` of integers, and that it can be easily changed by redefining `QElementType`.

Implementation Stage

4. Testing Strategy Development

In Chapter 10 of the Principles book, we have identified the following test cases.

1. Empty trivial words file: the index will include all the words in the text.
2. Only one trivial word: the only trivial word will not appear in the index.
3. One page text file: the page numbers are all the same, and appear only once for each word.
4. Text file with several pages: general case.
5. Text file with only trivial words: the index will be empty.
6. Text file with no trivial words: the index will include all the words in the text.
7. Word found on more pages than fit on a single line: necessary to test the splitting of the page numbers over several lines.

Three trivial words file are needed to test all cases.

Trivial 1: an empty file for case 1

Trivial 2: a file with a single word for case 2

kernel

Trivial 3: a general file including the list given in the Principles book.

for On on Had a is this At If To as her much the when
up was we Am Did He Its That With an by do did be but
has his like nor ours there will you with whom A An
But For Do Has His I In My Much She The We You all and
at from have hers if it me my of or so their these
those to us whose All As By From Have No Our So This
Will Your am are etc had he him in its mine no not off
our she that them they who your yours

Three special text files are also needed.

Text 1: a one page text file with words in alphabetical order

albatross beauty cartography demon earl fugitive gross
helicopter indeed joy kernel lullaby mammoth nerd
opera possible quintessence refrigeration subtlety ton
utilitarian vampire wapiti xylophone yak zero

Text 2: the same file as Text 1 but with an end of page mark between every word

albatross\ beauty\ cartography\ demon\ earl\ fugitive\
gross\ helicopter\ indeed\ joy\ kernel\ lullaby\
mammoth\ nerd\ opera\ possible\ quintessence\
refrigeration\ subtlety\ ton\ utilitarian\ vampire\ wapiti\ xylophone\ yak\ zero\


```
EndOfLine      = CHR(13);

VAR Buffer: ARRAY [1..BufferLength] OF CHAR;
                                { Global word buffer }
Words, TrivialWords: BinarySearchTree;
OldIndex, NewIndex, PageNumber, LineNumber: INTEGER;
Ch: CHAR;
Word: ElementType;
TriviaName, TextName: STRING; { file names }
TriviaFile, TextFile: TEXT; { file variables }

PROCEDURE DisplayWord(Index: INTEGER);
(* Display a word from global buffer in WordLength width
*)
VAR CharNumber: INTEGER;
BEGIN
  FOR CharNumber := Index+1 TO
    Index+ORD(Buffer[Index]) DO
    Write(Buffer[CharNumber]); { output word }
  FOR CharNumber := ORD(Buffer[Index]) TO
    WordLength-1 DO
    Write(Space);           { pad with blanks }
END; { DisplayWord }

PROCEDURE DisplayElement(Elt: ElementType);
(* Display an index element: word and page references *)
VAR Index, NumbersDisplayed, Item: INTEGER;
BEGIN
  DisplayWord(Elt.Key);
  NumbersDisplayed := 0;
  WHILE CountQueue(Elt.Pages) <> 0 DO BEGIN
    IF NumbersDisplayed = ItemsPerLine THEN BEGIN
      WriteLn;   { line is full }
      NumbersDisplayed := 0;
      { skip space under word }
      FOR Index := 1 TO WordLength DO
        Write(Space);
      END;
      Dequeue(Elt.Pages, Item); { get next page number }
      Write(Item:6);           { output page number }
      Inc(NumbersDisplayed);
    END;
    WriteLn;
  END; { DisplayElement }

FUNCTION WordCompare(First, Second: KeyType): INTEGER;
(* Compare two words stored in global Buffer, and return
the difference computed between the two words. *)
VAR Continue: BOOLEAN;
    Last1, Last2: INTEGER;
```

```

BEGIN
  Continue := TRUE;
  Last1 := First + ORD(Buffer[First]);
  Last2 := Second + ORD(Buffer[Second]);
  INC(First); INC(Second);
  WHILE Continue AND
    (First <= Last1) AND (Second <= Last2) DO BEGIN
    IF Buffer[First] <> Buffer[Second] THEN BEGIN
      WordCompare := ORD(Buffer[First]) -
        ORD(Buffer[Second]);
      Continue := FALSE; { not identical, stop }
    END
  ELSE
    IF (First = Last1) THEN
      IF (Second = Last2) THEN { identical }
        WordCompare := 0
      ELSE { second is longer }
        WordCompare := -1
    ELSE { first is longer }
      WordCompare := 1;
    Inc(First);
    Inc(Second);
  END;
END; { WordCompare }

PROCEDURE GetWord(VAR InputFile: TEXT;
  VAR Ch: CHAR;
  VAR New, Old: INTEGER);
(* Read a word from input file and store it into
   global buffer *)
BEGIN
  New := Old + 1; { index of first character }
  REPEAT { for all letters including accented }
    Write(Ch);
    Buffer[New] := Ch;
    Inc(New);
    Read(InputFile, Ch);
  UNTIL (Ch < '0') OR (Ch > '9') AND (Ch < 'A') OR
    (Ch > 'Z') AND (Ch < 'a') OR
    (Ch > 'z') AND (Ch < #128) OR (Ch > #159) OR
    Eof(InputFile);
  Buffer[Old] := CHR(New - Old - 1); { word length }
END; { GetWord }

PROCEDURE InsertWord(VAR Root: BinarySearchTree;
  Page: INTEGER;
  VAR New, Old: INTEGER);
(* Insert a word in index tree if not already there.
   Add new reference to its page number queue *)
VAR Word: ElementType;

```

```
        AlreadyIn: BOOLEAN;
        Last: INTEGER;
BEGIN
    AlreadyIn := SearchKey(Root, Old, Word, WordCompare);
    IF AlreadyIn THEN BEGIN
        IF Word.LastPage <> Page THEN BEGIN
            { add only new page references }
            Enqueue(Word.Pages, Page);
            Word.LastPage := Page;
            { update existing node }
            InsertNode(Root, Word, WordCompare);
        END;
    END
    ELSE BEGIN { new word }
        Word.Key := Old;
        InitializeQueue(Word.Pages);
        IF Page <> 0 THEN BEGIN
            { page is zero for trivial words }
            Enqueue(Word.Pages, Page);
            Word.LastPage := Page;
        END;
        InsertNode(Root, Word, WordCompare);
        Old := New; { keep word in global buffer }
    END;
END; { InsertWord }

BEGIN { BuildIndex }
    InitializeTree(Words);
    InitializeTree(TrivialWords);
    OldIndex := 1;
    PageNumber := 1;
    LineNumber := 1;

    {**** read in file of trivial words and create tree ****}
    Write('Give name of trivial words file: ');
    Read(TriviaName);
    Assign(TriviaFile, TriviaName);
    Reset(TriviaFile);
    WriteLn('List of trivial words: ');
    Read(TriviaFile, Ch);
    WHILE NOT Eof(TriviaFile) DO
        CASE Ch OF
            'A'..'Z', 'a'..'z':
                BEGIN
                    GetWord(TriviaFile, Ch, NewIndex, OldIndex);
                    InsertWord(TrivialWords, 0, NewIndex, OldIndex);
                END
            ELSE BEGIN
                Write(Ch);
                Read(TriviaFile, Ch);
            END
        END
    END
```



```

        END
    END;
    Close(TriviaFile);
    WriteLn; WriteLn; WriteLn;

    {**** read in text file and create word tree ****}
    Write('Give name of text file: ');
    Read(TextName);
    Assign(TextFile, TextName);
    Reset(TextFile);
    WriteLn; WriteLn;
    Write(LineNumber:6, Space);
    Read(TextFile, Ch);
    WHILE NOT Eof(TextFile) DO {read text and create index}
        CASE Ch OF
            EndOfLine:
                BEGIN
                    WriteLn; Read(TextFile, Ch);
                    Inc(LineNumber);
                    Write(LineNumber: 6, Space);
                END;
            'A'..'Z', 'a'..'z':
                BEGIN
                    GetWord(TextFile, Ch, NewIndex, OldIndex);
                    IF NOT SearchKey(TrivialWords, OldIndex,
                                    Word, WordCompare) THEN
                        InsertWord(Words, PageNumber,
                                    NewIndex, OldIndex);
                    END;
                END;
            EndOfPage:
                BEGIN
                    Inc(PageNumber);
                    Write(Ch);
                    Read(TextFile, Ch);
                END;
            ELSE
                BEGIN
                    Write(Ch);
                    Read(TextFile, Ch);
                END;
            END;
        END;
    Close(TextFile);
    WriteLn; WriteLn;
    WriteLn('Text word index');
    WriteLn;
    TraverseTree(Words, DisplayElement); { output index }
    WriteLn;
    WriteLn('Index complete');
END. { BuildIndex }

```

To run this program we need to complete the IMPLEMENTATION parts of the two units it uses. The IMPLEMENTATION part of unit Queues corresponding to the INTERFACE part given earlier follows.

IMPLEMENTATION

```
{ Empty queue:   Tail = 0, Head = MaxQueue, Number = 0
  Full queue:    Tail = Head, Number = MaxQueue
  Average queue: Tail indicates actual last element
  Head indicates previous first element }
```

```
PROCEDURE InitializeQueue(VAR Q: Queue);
(* Create queue Q *)
```

```
BEGIN
```

```
  Q.Tail      := 0;
  Q.Number    := 0;
  Q.Head      := MaxQueue;
  QueueError := FALSE;
```

```
END; { InitializeQueue }
```

```
PROCEDURE Enqueue(VAR Q: Queue; Item : QElementType);
(* Insert element Item in queue Q *)
```

```
BEGIN
```

```
  IF NOT (Q.Number = MaxQueue) THEN BEGIN
    WITH Q DO BEGIN
      Tail      := (Tail MOD MaxQueue) + 1;
      Data[Tail] := Item;
      Inc(Number);
    END;
    QueueError := FALSE;
  END
```

```
  ELSE
    QueueError := TRUE; { full queue }
```

```
END; { Enqueue }
```

```
PROCEDURE Dequeue(VAR Q: Queue; VAR Item : QElementType);
(* Retrieve in Item and delete head of queue Q *)
```

```
BEGIN
```

```
  IF NOT (Q.Number = 0) THEN BEGIN
    WITH Q DO BEGIN
      Head := (Head MOD MaxQueue) + 1;
      Item := Data[Head];
      Dec(Number);
      IF Number = 0 THEN BEGIN { queue is now empty}
        Tail := 0;
        Head := MaxQueue;
      END;
    END;
    QueueError := FALSE;
  END
```

```
  ELSE
```

```

        QueueError := TRUE; { no element to retrieve }
    END; { Dequeue }

FUNCTION CountQueue(Q: Queue): INTEGER;
(* Return number of elements in Q *)
BEGIN
    CountQueue := Q.Number;
END; { CountQueue }

PROCEDURE QueueHead(Q: Queue; VAR Item: QElementType);
(* Return value of first element of queue Q in Item *)
BEGIN
    IF Q.Number <> 0 THEN BEGIN
        Item      := Q.Data[Q.Head MOD MaxQueue + 1];
        QueueError := FALSE;
    END
    ELSE
        QueueError := TRUE; { no queue }
    END; { QueueHead }

END. { Queues }

```

The IMPLEMENTATION part of UNIT BinSearchTrees corresponding to the INTERFACE part given earlier follows. The procedures and functions correspond to the algorithms given in Chapter 10 of the Principles book.

IMPLEMENTATION

```

PROCEDURE DisplayKey(Element: KeyType);
(* Display a key value *)
BEGIN
    Write(Element: 6);
END; { DisplayKey }

PROCEDURE InitializeTree(VAR Tree: BinarySearchTree);
(* The first procedure to call, which initializes a tree
   *)
BEGIN
    Tree := NIL;
END; { InitializeTree }

PROCEDURE DeleteTree(VAR Tree: BinarySearchTree);
(* All information about the tree and the data records it
   contains are deleted *)
BEGIN
    IF Tree <> NIL THEN BEGIN
        DeleteTree(Tree^.Left);
        DeleteTree(Tree^.Right);
        Dispose(Tree);
    END;
END; { DeleteTree }

```

```
PROCEDURE InsertNode(VAR Tree: BinarySearchTree;
                    Element: ElementType;
                    Compare: CompProc);
(* Insert Element into the Tree. The key is in the
   Element.*)
VAR Diff: INTEGER;
BEGIN
  IF Tree = NIL THEN BEGIN { insert at current position}
    New(Tree); { create new node }
    Tree^.Element := Element;
    Tree^.Left := NIL;
    Tree^.Right := NIL;
  END
  ELSE BEGIN
    Diff := Compare(Element.Key, Tree^.Element.Key);
    IF Diff < 0 THEN { look left }
      InsertNode(Tree^.Left, Element, Compare)
    ELSE IF Diff > 0 THEN { look right }
      InsertNode(Tree^.Right, Element, Compare)
    ELSE { already in Tree update node }
      Tree^.Element := Element;
    END;
  END; { InsertNode }

PROCEDURE DeleteNode(VAR Tree: BinarySearchTree;
                    Key: KeyType;
                    Compare: CompProc);
(* Find the Element with this Key and delete it from
   the Tree *)

PROCEDURE FindPredecessor(  Tree: BinarySearchTree;
                          VAR Node: BinarySearchTree);
(* Find rightmost node in left subtree *)
BEGIN
  Node := Tree^.Left;
  WHILE Node^.Right <> NIL DO
    Node := Node^.Right;
  END; { FindPredecessor }

VAR Diff: INTEGER;
    Node: BinarySearchTree;
BEGIN
  IF Tree <> NIL THEN BEGIN
    Diff := Compare(Tree^.Element.Key, Key);
    IF Diff = 0 THEN { found node to delete }
      IF Tree^.Left = NIL THEN BEGIN { empty left
                                     branch }
        Node := Tree;
        Tree := Tree^.Right;
        Dispose(Node);
```

```

        END
    ELSE
        IF Tree^.Right = NIL THEN BEGIN
            { empty right branch }
            Node := Tree;
            Tree := Tree^.Left;
            Dispose(Node);
            END
        ELSE BEGIN
            { no branch empty, find inorder
              predecessor }
            FindPredecessor(Tree, Node);
            Tree^.Element := Node^.Element;
            DeleteNode(Tree^.Left,
                      Tree^.Element.Key, Compare);
            END
    ELSE
        IF Diff < 0 THEN { try left }
            DeleteNode(Tree^.Left, Key, Compare)
        ELSE { try right }
            DeleteNode(Tree^.Right, Key, Compare)
        END;
    END; { DeleteNode }

PROCEDURE TraverseTree(VAR Tree: BinarySearchTree;
                      Process: TraverseProc);
(* Call Process for each node in order.
   Don't call any procedures that modify links
   while in mid traversal, like InsertNode,
   DeleteNode, DeleteTree. *)
BEGIN
    IF Tree <> NIL THEN BEGIN
        TraverseTree(Tree^.Left, Process);
        Process(Tree^.Element);
        TraverseTree(Tree^.Right, Process);
    END;
END; { TraverseTree }

FUNCTION SearchKey(    Tree: BinarySearchTree;
                      Key: KeyType;
                      VAR Element: ElementType;
                      Compare: CompProc): BOOLEAN;
(* Search for "Key" in the Tree. If found then return
   true and Element will contain the node information,
   otherwise return false. *)
VAR Diff: INTEGER;
BEGIN
    IF Tree = NIL THEN
        SearchKey := FALSE
    ELSE BEGIN

```

```

Diff := Compare(Key, Tree^.Element.Key);
IF Diff = 0 THEN BEGIN { found }
    Element := Tree^.Element;
    SearchKey := TRUE;
END
ELSE IF Diff < 0 THEN { look left }
    SearchKey := SearchKey(Tree^.Left, Key,
                           Element, Compare)
ELSE { look right }
    SearchKey := SearchKey(Tree^.Right, Key,
                           Element, Compare);
END;
END; { SearchKey }

PROCEDURE DisplayTree(Tree: BinarySearchTree;
                     Indentation: INTEGER;
                     DisplayKey: DisplayProc);
(* Print Tree with indentations to show structure *)
VAR Indent: INTEGER;
BEGIN
    IF Tree <> NIL THEN BEGIN
        DisplayTree(Tree^.Right, Indentation+1, DisplayKey);
        FOR Indent := 1 TO Indentation DO
            Write(' ');
        DisplayKey(Tree^.Element.Key);
        WriteLn;
        DisplayTree(Tree^.Left, Indentation+1, DisplayKey);
    END;
END; { DisplayTree }

END. { BinSearchTrees }

```

The program BuildIndex was run with the files identified in the testing strategy section. It gave the expected results. Here is part of the index produced for a normal text.

though	4				
thought	1	2	5		
threatening	4				
three	8				
through	3	6	9		
thus	9				
time	1	2	4	5	8
	9				
together	1	2			
told	5				
tonight	5	8			
too	2	3			
took	6	9	10		
tooth	10				

Use your copy of the program to build the index of other texts. If you find bugs, you must remove them! Note that to compile your program you need to compile first the `Queues` unit, then the `BinSearchTrees` unit, and then the `BuildIndex` program. This is because units used by a program must always be compiled before their users.

6. Documentation

All the results of the previous steps should be integrated in the documentation: problem specifications, solution design and refinement, testing strategy, program code, and testing results. The user's manual was already done in the Principles book, we repeat it here for the sake of completeness.

User's Manual for the Build Index Program

The Build Index program builds the index of a text. The program prompts you for a file of trivial words (words of the text that must not be part of the index). The program then prompts you for the name of the text file, and the name of the output index file. To run it, select *Execute* in the menu and double-click on Build Index. The program will start executing and will start prompting you. When the execution is over, the program will display the message:

Index complete

- You can examine the index and print it if need be. The index file comprises the input text with added line numbers, and an alphabetical list of all the words in the text (except for the trivial words) followed by a list of page numbers as in the following:

June	1	8						
Karine	1	2	3	4	5	6	7	8
	9	10						
Kludge	5	9						

7. Program Maintenance

With a larger program the risk of bugs is higher, despite all the precautions that were taken. Part of that program maintenance includes bug removal, but also making improvements. We can improve the program by giving the option to the user to list the trivial words used as part of a better documentation of the index. We can also make the program recognize true end of pages instead of an arbitrary sign. We can also modify the program so that it displays page *and line* numbers as part of the index. Etc. Etc.

10.5 Chapter 10 Review

In this chapter we have reviewed the seven step problem solving method which was introduced in Chapter 2 of the Principles book, and illustrated in various other chapters of that book. Here, we have concentrated on only one of these seven steps—the implementation in Pascal of an already designed program. However, the implementation cannot be done in a vacuum, divorced from the design, and we have therefore included each of the seven steps so that the implementation can be seen in its proper context.

It is important for you to remember that a lack of method will be disastrous for you when you start developing programs on a larger scale, maybe even before. The complexity of a given application grows quickly, as the number of interactions between the various parts of a system increases. The examples we have presented were aimed at showing that. The complete example to build a text index is still small, but is big enough to show that a larger system is harder to understand, even under good conditions.

10.6 Chapter 10 Programming Problems

A programmer spends a great deal of time modifying already written programs, generally written by somebody else. We will therefore start with a series of program modification exercises. The following programs were introduced in Chapter 2 of this book, when you used them to get a beginning experience in actually using programs. The following problems show you the programs and ask you to change their behavior by modifying them.

1. Editor Application

TED is a Tiny Editor used to create and modify files of text. This editor acts on a line at a time. It requests operations by displaying a question mark “?” and the following commands may be given (by the first letter of the command: A, B, D, etc. in upper or lower case):

- B** to go to the **B**eginning of the file
- D** to **D**elete the line at the given position
- T** to **T**ype out the entire file
- P** to move the Now pointer to the **P**revious line
- N** to move the Now pointer to the **N**ext or following line
- I** to **I**nsert one line after given line
- F** to **F**ind a given string, starting at the present line
- M** to **M**odify the present line
- H** to provide **H**elp, by listing all commands
- R** to replace a given line by another
- S** to **S**ave a file
- L** to **L**oad a previous file which was created and saved
- E** to **E**nd or exit the edit session

The Pascal program for TED is rather short because it makes use of data structures from the UNIT StackLib2. The source code for StackLib2 is:

```

UNIT StackLib2;

INTERFACE
  TYPE ItemType      = STRING;
       StackType      = ^StackElement;
       StackElement = RECORD
                           Value: ItemType;
                           Prev:  StackType;
                         END;

  PROCEDURE Create(VAR Stack: StackType);
    (* Sets up stack, initially *)

  PROCEDURE Push(VAR Stack: StackType;
                 X: ItemType);
    (* Puts object X onto Stack *)

  PROCEDURE Pop(VAR Stack: StackType;
                VAR Y: ItemType);
    (* Takes object Y off Stack *)

  FUNCTION Empty(Stack: StackType):BOOLEAN;
    (* Shows if Stack is empty *)

IMPLEMENTATION
  PROCEDURE Create(VAR Stack: StackType);
  BEGIN
    Stack := NIL;
  END; { Create }

  PROCEDURE Empty(Stack: StackType): BOOLEAN;
  BEGIN
    IF Stack = NIL THEN
      Empty := TRUE
    ELSE
      Empty := FALSE;
    END; { Empty }

  PROCEDURE Push(VAR Stack: StackType;
                 X: ItemType);
  VAR Next: StackType;
  BEGIN
    New(Next);
    Next^.Prev := Stack;
    Next^.Value := X;
    Stack      := Next;
  END; { Push }

```

```
PROCEDURE Pop(VAR Stack: StackType;
              VAR Y: ItemType);
VAR PrevTop: StackType;
BEGIN
  IF Empty(Stack) THEN
    WriteLn('EMPTY ')
  ELSE BEGIN
    Y      := Stack^.Value;
    PrevTop := Stack^.Prev;
    Dispose(Stack);
    Stack := PrevTop;
  END;
END; { Pop }

END. { StackLib2 }
```

The Pascal program for TED is:

```
PROGRAM TED;
(* A line editor that uses stacks of lines, *)
(* which are strings.                      *)

USES StackLib2;

VAR PStack, NStack: StackType;
    Line: STRING;
    Command, Return: CHAR;

PROCEDURE Help();
BEGIN
  WriteLn('The commands are:');
  WriteLn('B to go to the Beginning of the file');
  WriteLn('D to Delete the line at the given
position');
  WriteLn('T to Type out the entire file');
  WriteLn('P to move to the Previous line');
  WriteLn('N to move to the Next or following
line');
  WriteLn('I to Insert one line after given
line');
  WriteLn('F to Find a given string');
  WriteLn('M to Modify the present line');
  WriteLn('H to provide Help, by listing all
commands');
  WriteLn('R to replace a given line by another');
  WriteLn('S to Save a file');
  WriteLn('L to Load a previously created file');
  WriteLn('E to End or exit the edit session');
END; { Help }
```

```
PROCEDURE Insert();
(* Inserts a single line after this one *)
BEGIN
    ReadLn(Line);
    Push(PStack, Line);
END; { Insert }

PROCEDURE Delete();
(* Deletes the single present line *)
VAR Line: STRING;
BEGIN
    IF Empty(PStack) THEN
        WriteLn('What!?!')
    ELSE BEGIN
        Pop(PStack, Line);
        WriteLn('Deleted: ', Line);
    END;
END; { Delete }

PROCEDURE TypePage();
(* Types all lines in page and stops at end *)
VAR Line: STRING;
BEGIN
    { Pour Present stack to Next one }
    WHILE NOT Empty(PStack) DO BEGIN
        Pop(PStack, Line);
        Push(NStack, Line);
    END;

    { Pour Next stack while printing }
    WHILE NOT Empty(NStack) DO BEGIN
        Pop(NStack, Line);
        WriteLn(Line);
        Push(PStack, Line);
    END;
END; { TypePage }

PROCEDURE Replace();
BEGIN
    Delete();
    Write('Replace:');
    Insert();
END; { Replace }

PROCEDURE GoBegin();
(* Goes to the first line of the page *)
VAR Line: STRING;
BEGIN
    WHILE NOT Empty(PStack) DO BEGIN
        Pop(PStack, Line);
```

```
        Push(NStack, Line);
    END;
    Pop(NStack, Line);
    Push(PStack, Line);
    WriteLn(Line);
END; { GoBegin }

PROCEDURE NextLine();
(* Goes to the following line      *)
VAR Line: STRING;
BEGIN
    IF Empty(NStack) THEN
        WriteLn('At the end of page')
    ELSE BEGIN
        Pop(NStack, Line);
        Push(PStack, Line);
        WriteLn(Line);
    END;
END; { NextLine }

PROCEDURE Previous();
(* Go to previous line            *)
BEGIN
    WriteLn('This command not yet implemented');
END; { Previous }

PROCEDURE Modify();
(* Modify the present line        *)
BEGIN
    WriteLn('This command not yet implemented');
END; { Modify }

PROCEDURE Find();
(* Find given string starting at present line *)
BEGIN
    WriteLn('This command not yet implemented');
END; { Find }

PROCEDURE Save();
(* Save page in file              *)
BEGIN
    WriteLn('This command not yet implemented');
END; { Save }

PROCEDURE Load();
(* Load previously saved file into page      *)
BEGIN
    WriteLn('This command not yet implemented');
END; { Load }
```

```

PROCEDURE Capitalize(VAR Ch: CHAR);
(* Procedure capitalizes any passed letter *)
CONST UpperLowerDiff = ORD('a') - ORD('A');
BEGIN
    IF ('a' <= Ch) AND (Ch <= 'z') THEN
        CH := CHR(ORD(Ch) - UpperLowerDiff);
    END; { Capitalize }

BEGIN
    Create(PStack);
    Create(NStack);
    WriteLn('Enter a command');
    Write('?');
    Read(Command);
    Read(Return);
    Capitalize(Command);

    WHILE Command <> 'E' DO BEGIN
        IF Command = 'B' THEN GoBegin()
        ELSE IF Command = 'D' THEN Delete()
        ELSE IF Command = 'T' THEN TypePage()
        ELSE IF Command = 'P' THEN Previous()
        ELSE IF Command = 'N' THEN NextLine()
        ELSE IF Command = 'I' THEN Insert()
        ELSE IF Command = 'F' THEN Find()
        ELSE IF Command = 'M' THEN Modify()
        ELSE IF Command = 'H' THEN Help()
        ELSE IF Command = 'R' THEN Replace()
        ELSE IF Command = 'S' THEN Save()
        ELSE IF Command = 'L' THEN Load()
        { More commands can go here }
        ELSE WriteLn('What!?');

        WriteLn;
        Write('?');
        Read(Command);
        Read(Return);
        Capitalize(Command);
    END;
END. { TED }

```

You will notice that not all the commands in TED have been implemented; there are just place holders for them. Complete the implementation of TED and then:

- Add more commands, such as Append, to insert more than one line
- Output with line numbers for reference
- Provide more detailed help or instructions

2. Typing

TypeTimer is an application program that presents a line of text to be typed in, and then indicates how quickly this line was typed. Accuracy is not measured because it is assumed that errors can easily be corrected. A typical run of this program is shown below; the part of the dialog that the user typed is shown in bold.

```
Typing Speed Test
You are to type the following line
Type Return when you are ready, and
type Return when you are finished
A quick brown fox jumps over the lazy dog
A quick brown fox jumps overf the laxy dog
The time taken is 50 units.
```

Time is measured by units, which are not seconds, but some arbitrary units that are consistent and serve to compare times to measure progress. The Pascal code for TypeTimer is:

```
PROGRAM TypeTimer;
(* Evaluates time to type a line of text  *)
USES Events;

CONST Return = #13;

VAR TimeOfStarting: LONGINT;

PROCEDURE StartTime();
BEGIN
    TimeOfStarting := TickCount();
END; { StartTime }

FUNCTION TellTime(): LONGINT;
VAR FinishTime: LONGINT;
BEGIN
    FinishTime := TickCount();
    TellTime   := FinishTime - TimeOfStarting;
END; { TellTime }

VAR ElapsedTime: LONGINT;
    Ch: CHAR;
    Text: STRING;

BEGIN
    WriteLn('Typing Speed Test');
    WriteLn('You are to type the following line');
    WriteLn('Press return when you are ready, and');
    WriteLn('press return when you are finished');
    WriteLn('A quick brown fox jumps over the lazy
dog');
    WriteLn;
```

```

Read(Ch);
StartTime;
Read(Text);
ElapsedTime := TellTime();
WriteLn('The time taken was ', ElapsedTime: 8, '
units');
END. { TypeTimer }

```

Typ2 is another application program that presents a number of lines of various kinds of text and indicates whether the typed line is correct or has errors.

```

PROGRAM Typ2;
(* Typing test to determine accuracy *)

CONST Return = #13;

VAR Ch: CHAR;
    Line, InLine, Response: STRING;
    GoodResponse, Same: BOOLEAN;

BEGIN
    WriteLn('Typing Accuracy Test');
    WriteLn('What do you wish to try?');
    WriteLn('Silly sentences or serious statements?');
    WriteLn('Enter "silly" or "serious"');
    ReadLn(Response);
    GoodResponse := FALSE;
    WHILE NOT GoodResponse DO
        IF Response = 'silly' THEN BEGIN
            GoodResponse := TRUE;
            Line := 'exquisite farm wench gives body jolt
to prize stinker';
        END
        ELSE
            IF Response = 'serious' THEN BEGIN
                GoodResponse := TRUE;
                Line := 'a quick brown fox jumps over the
lazy dog';
            END
            ELSE BEGIN
                WriteLn('Try again--Enter "silly" or
"silly"');
                ReadLn(Response);
            END;
    WriteLn('Type the following');
    WriteLn(Line);
    ReadLn(InLine);
    IF Line = InLine THEN
        WriteLn('CORRECT!')
    END;

```

```
ELSE
    WriteLn('ERROR!!!');
    WriteLn('Try again soon');
END. { Typer2 }
```

Typer2, as it stands has only two sentences, neither of them really serious. Extend the program to draw on a richer repertoire of sentences from a file. Other extensions to be made are:

- to combine both speed and accuracy tests into one,
- to keep track of your progress after each exercise.
- to enter yet a third kind of file, “semi-serious”,
- to count the number of errors.

3. Calculator Applications

Calculate is a calculator that is simulated by a computer. It provides the typical four arithmetic functions (add, subtract, multiply, divide). Entering the letter “q” or “Q” causes the calculation to quit. The Pascal program for Calculate is

```
PROGRAM Calculate;
(* Four Function Calculator of Real Values *)

VAR Value, Result: REAL;
    Ch, Action, Return: CHAR;

BEGIN
    Write('Calculate Real Numbers ');
    WriteLn;
    Write('End with 'Q' for quit ');
    WriteLn;
    Write(' Enter a value: ');
    WriteLn;
    Read(Result);
    Read(Return); { to "eat" Carriage Return }
    Write(' Enter an action: ');
    WriteLn;
    Read(Action);
    Read(Return); { to "eat" Carriage Return }

    Action := UpCase(Action);
    WHILE (Action <> 'Q') DO BEGIN
        Write(' Enter a value ');
        WriteLn;
        Read (Value);
        Read(Return); { to "eat" Carriage Return }
        IF (Action = '+') OR (Action = 'A') THEN
            Result := Result + Value
```



```

ELSE IF (Action = '-') OR (Action = 'S') THEN
    Result := Result - Value
ELSE IF (Action = '*') OR (Action = 'M') THEN
    Result := Result * Value
ELSE IF (Action = '/') OR (Action = 'D') THEN
    Result := Result / Value
ELSE
    Write('Error ')
{ END IF };
Write(' The Result is ');
Write(Result: 10:3); WriteLn;
Write(' Enter an action ');
WriteLn;
Read(Action);
Read(Return); { to "eat" Carriage Return }
Action := UpCase(Action);
END { WHILE };

WriteLn('End of Calculation ');
END { Calculate }.

```

Extend this program to:

- Compute squares and powers
- Include trigonometric functions
- Output results in scientific notation

Make another version of this calculator that uses `ComplexLib` from Chapter 8 to work with complex numbers. Extend this version to:

- Compute complex conjugates
- Compute magnitudes and angles
- Output results in polar notation.

10.7 Chapter 10 Programming Projects

The following is a series of problems and projects to be solved in the way in which we have done the two case studies in this chapter, that is, all the steps of the seven step process except for the actual implementation must be followed. These problems are presented in order of increasing difficulty and complexity, and cover various domains of application: general, business and scientific. They were already described in Chapter 10 of the Principles book. If you did the design then, only the implementation remains to be done!

10.8 Level 1 — Getting Started

The first-level problems require the development of your first programs.

1-1. General

A Guessing Game

You are on vacation at home and planning to enjoy your free time. Alas! your parents ask you to take care of your little sister, and she is a real pest. In order not to see your vacation time slowly wasted, you decide to have the computer entertain your little sister. To do this, you want to develop a simple game program that will pick randomly an integer number between 1 and 1,000. The program will ask your little sister to guess that number in a maximum of ten tries, and will produce an appropriate message when the end of the game is reached.

Obviously, the program needs to be interactive: It will display a message at the start of the game, prompt your sister for a guess, and each time she makes a guess it will have to indicate whether the guess was between the limits or was high or low, or detect that the guess was right. At the end of a game the program will allow your little sister to decide to continue to play or to stop.

The input format is simply that of an integer number, or a character for yes or no. The output formats are mostly messages.

Start-of-game message:

```
Let's play a guessing game.  
I pick a number between 0 and 1,000. You have to guess  
it. But you have only 10 tries to guess my number.
```

End-of-game messages:

```
Congratulations! 999 is right.  
You lose! My number was 999  
Do you want another game? (Y/N)
```

Game messages:

```
Make a guess:  
Wait a minute! My number is greater than 0!  
You wasted a guess! My number is 1,000 or less.  
Well ... your number is too small.  
Sorry, but your number is too big.
```

Don't forget! You have seen this example in Chapter 4.

1-2. Business

Computing a Customer's Change

Your cousin just opened a small store and does not have the funds to buy one of those sophisticated cash registers that compute the change to return to a customer. Since he still possesses his old personal computer, he asks you to develop a program that, given an amount due and a payment, computes the

change. This way he will be able to make sure that whoever he hires won't make a mistake on the change to give back to the customer. The program will compute the change repeatedly until a zero value is given to indicate termination.

The change must be computed in dollar bills, quarters, dimes, nickels, and pennies, with the smallest number of coins. The clerk will enter the amount due in cents, the payment also in cents, and the program will return the number of dollars, quarters, dimes, nickels, and pennies to give back. The clerk will be prompted to enter the amount due and the payment by the following messages:

```
Enter amount due in cents (negative or zero to stop):
Enter payment in cents:
```

The change will be indicated in the following way:

```
Dollars      1
Quarters     1
Dimes        1
Nickels      1
Pennies      1
```

Don't forget! We have seen examples of change making in Chapters 3, 4, 5, 7 and 8.

1–3. Scientific

A Bouncing Ball

While waiting for your date to show up, you idly bounced a tennis ball on the sidewalk. This gave you the idea to develop a program to compute and display some data on the bounces a ball will make when dropped from a given height. Forgetting your late date, you went home to solve this interesting problem.

To simplify the problem, you assume the ball bounces in place, that is, it remains bouncing on the same spot and does not have any forward motion. The program will prompt the user for the initial height of the ball, the number of bounces to consider, and the ball's elasticity (which must be positive). It will compute the height of each bounce based on the initial height and the ball's elasticity, and display it. The program will also compute the total distance traveled, which is the sum of the up and down bounces, for the given number of bounces, and display it. The program will repeat this process until the user tells it to stop.

If the ball is at height h , when it bounces, it reaches new height h' , which is computed by the formula

$$h' = h \times \text{resilience}$$

where resilience is expressed as the elasticity coefficient raised to the n th power, if n is the bounce number:

$$\text{resilience} = \text{elasticity}^n$$

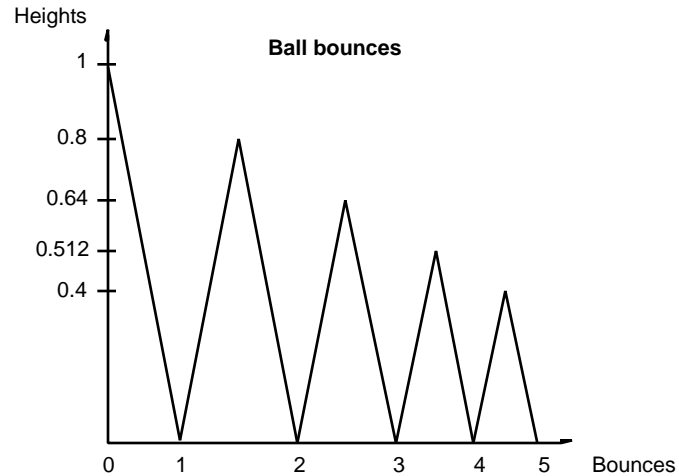
If the original height is H , then the first bounce height will be

$$h_1 = H \times \text{elasticity}$$

Note that, in your simplified conditions, an elasticity coefficient greater than one means that the ball will never stop bouncing, and that each bounce will be higher than the preceding one.

The ball will travel the distance shown in Figure 10.6, which we have drawn to help you, and where we have represented a forward motion for the sake of clarity (the ball bounces on the same spot).

Figure 10.6. Distance traveled by ball as it bounces



Each time the ball bounces, it travels twice the height of the bounce, so the total distance traveled is

$$H + 2h_1 + 2h_2 + 2h_3 + 2h_4 + \dots$$

The user will be prompted for the initial height in this way:

Give the height in inches from which the ball is dropped:

Then the user will be asked the number of bounces:

Give the number of times the ball will bounce:

And finally the user will be asked the elasticity of the ball:

Give the elasticity of the ball (between 0 and 1 if the ball is to stop bouncing):

For each bounce the program will display the following:

On bounce 9 the ball rises to a height of 99.9 inches

After the last bounce the program will display the following:

The total distance traveled by a ball with an elasticity of 0.999 when dropped from a height of 99.9 inches

```
and after bouncing 9 times is 999 inches.  
The user will be asked if he wants to continue:  
Another try?
```

10.9 Level 2 — Getting Organized with Procedures

The second level of problems requires the use of procedures and functions to manage larger programs.

2-1. General

Your Age in Days

Your problem is to write a program that will read in a person's birth date and the current date, and compute and display the person's age in days. The program will have to be interactive, to validate the dates it reads and to display results in a clear manner. When the dates given are incorrect, precise messages should be displayed. The program should be set up in such a way that it can compute repeatedly a number of ages and let the user decide when to terminate execution.

The various output formats are mostly user's prompts and error messages.

```
Enter birth date in the form 11 21 91:  
Enter current date in the form 11 21 91:  
Today you are 2059 days old  
Want to compute another age?  
Incorrect value for month.  
Incorrect value for day.  
Incorrect value for year.  
Incorrect values for day and month.  
Incorrect values for month and year.  
Incorrect values for day and year.  
Incorrect values for day, month and year.  
You are not born yet!
```

2-2 Business

What's the Cost of My Mortgage?

The loan department of your bank still uses silly tables to determine what the monthly payment of a mortgage loan is going to be. The consumer association wants you to design a program to compute interactively monthly mortgage payments as well as the cost of such a loan over the first 5 years of the loan and the total cost of that loan. In order to validate the input data, the loan

amounts must be between \$1,000 and \$500,000, the loan interest must be between 3 and 20%, and the loan length must always be between 1 and 35 years.

Although banks are a little secretive about the formulas they use to compute mortgage loans, we know that they base their computations on a monthly rate, that can be computed from the annual mortgage rate compounded twice a year. Loan rate tables show you that

$$(1 + \text{Monthly Rate})^{12y} = (1 + \text{Annual Rate}/2)^{2y}$$

which leads to

$$(1 + \text{Monthly Rate})^6 = (1 + \text{Annual Rate}/2)$$

or

$$6 \times \log(1 + \text{Monthly Rate}) = \log(1 + \text{Annual Rate}/2)$$

and finally

$$\text{Monthly Rate} = e^{\log(1 + \text{Annual Rate}/2)/6} - 1$$

Using this, we can compute the monthly payment.

$$\text{Monthly Payment} = \text{Monthly Rate} \times \text{Loan} \times \frac{(1 + \text{Monthly Rate})^{12y}}{(1 + \text{Monthly Rate})^{12y} - 1}$$

The program will use these formulas to compute and display the monthly payment of a given mortgage loan and to produce a detailed report of the payments over the first 5 years if needed.

The output format used will be the following:

```
Amount of mortgage loan:
Annual interest rate:
Length of mortgage loan (in years):
Monthly payment: 999.99
Do you want a detailed report?
Interest paid in 5 years: 12345.67
5 year balance: 23456.78
Total cost of mortgage loan: 34567.89
```

The detailed report format will be the following:

Payment#	Interest	Capital	Cum. Int.	Balance
1	97.59	44.21	97.59	9955.79
2	97.16	44.65	194.75	9911.14

2-3 Scientific

Solving the Quadratic Equation

The problem you have to solve now is the well-known quadratic equation. Remember its form?

$$ax^2 + bx + c = 0$$

Your program will accept the values of the three coefficients a , b , and c and compute the roots of the equation. The program will prompt the user for the three coefficients repeatedly, and stop when the user enters three zeroes for the coefficients. The program will distinguish between the various solutions and display the results with an appropriate message: one root, double root, real roots, complex roots, as well as an error message if coefficients a and b are zero while c is not.

The output formats will be the following:

```
Give values of three coefficients:
Contradiction    2.0 = 0
One root        = -25.0
Double root     = 120.0
Root 1         = -2.0
Root 2         = -1.0
Complex roots   = -18.0 +/- 12.4i
```

10.10 Level 3 — Getting Fancier with Parameters

The third level of problems requires the use of procedures and functions to manage larger programs and provides practice in how to parameterize them.

3-1. General

Count the Word Occurrences in a Text

We want a program to read a text, to extract the various words from the text, and to count the number of occurrences of each word. The program will output a list of all the words in the text in alphabetical order (ascending or descending) with their number of occurrences. A word is defined to be a sequence of characters between two separators, and the separators will include all punctuation signs, as well as all available special characters. In fact, a separator will be any character other than a letter (upper case or lower case) or a digit.

The program will prompt the user for the name of a text file to use as input file. It will read the entire file, separate the words and count their occurrences, and finally display the words in alphabetical order, one per line, with their corresponding number of occurrences.

The input and output formats can be summed up by the following messages and examples of output.

```
Give the name of your text file:
In what order do you want the word list? (A/D):
Number of occurrences for the 96 words found in
fffff.ttt
Word : Occurrences
A 18
```

```
At 1
Words table is full, no room for xxxxxx
3-2. Business
```

Processing Personnel Data

Your chum Arnie, from the personnel department of the good old municipal services, has just given you a frantic call: he needs, right now, all sorts of employee lists, and his information systems department just told him it would take three to six months for them to produce a *feasibility study* for a needed program to read, sort, and display data on municipal employees. Obviously, he will get nowhere with his own services, and has the OK to contract out to you the writing of this program.

Further prodding on your part elicits a little more information on the program: it must be interactive; it must run on a Macintosh; it will be used by his boss in the personnel department; it should be able to read various employee files; and it should be able to sort employee records extremely quickly by name, by age, or by seniority, and to display these records in four different simple formats. The employee files all have the same format: one line per employee with family name, first name, employee number, hiring date, birth year, and various informations including the social security number.

The program will display a short menu to the user, and read and validate the user's choice. The menu format will be the following:

```
1. Read data from file
2. Sort by age
3. Sort by name
4. Sort by seniority
5. Display name and birth year
6. Display name and first name
7. Display name and hired date
8. Display all information
9. Exit program
Please enter your selection and press return:
```

The program will have to make sure that operations 2 to 8 are not usable until operation 1 has been used at least once. The program will display the following message:

```
No data has been read yet
```

After each sort, a message will be displayed:

```
Employees sorted by age
Employees sorted by name
Employees sorted by seniority
```

The output formats for operations 5, 6, 7, and 8 are quite simple: the employee name will be followed by a single value, or all the values:

```
Berger 1956
Berger Antonia
```



```
Berger      710221
Berger      Antonia BERA0 710221 1956 198-39-9739 Middle-managemen
```

The personnel department foresees a new format for employee records which would double or even triple their size. The sorting of employee records must be designed so that the change in size of the employee records does not unduly affect the sorting time (a variation of Count Sort —see Chapter 9— seems in order).

In order to read in the employee data, the program will prompt the user in the following manner:

```
Please give name of employees file:
```

Once the data have been read, a message indicating the number of records read will be displayed:

```
Number of employee records read:  99
```

In the case of a large file, the program will check that the data can be stored in the employees table; if not, it will display the following message and skip the rest of the file.

```
File too large, input truncated
```

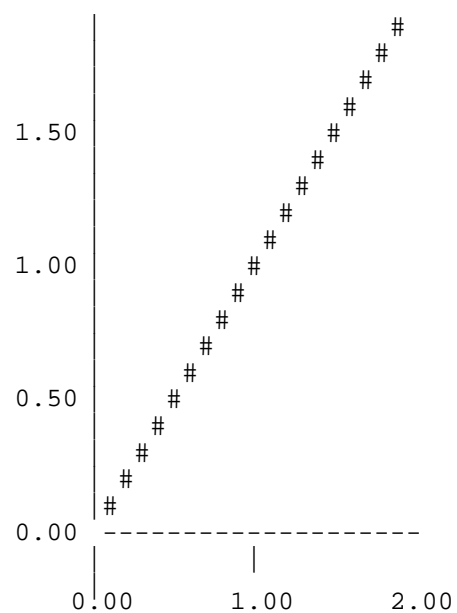
3–3. Scientific

Plotting a Function

In this graphic era, we want to be able to plot a given function $y = f(x)$ between two values of x . The plot of such a function will be graphical and will appear in the usual manner, that is, assuming a vertical y axis and a horizontal x axis. We want the plot to show parallels to the x and y axis for the minimum values of x and of y , with some indication of the x and y values. We also want the user to be able to plot any function of one variable, and to define the range of x values for the plot, as well as the size of the plot.

A typical output would look like Figure 10.7.

Figure 10.7. Plot for $y = x$



10.11 Level 4 — Getting Your Wings with Units

The fourth level of problems requires the creation and use of separately compilable units to construct a complete program.

4–1 General

The Kwic Index

Suppose we were interested in programming languages and looking for books or papers on the subject. It will be easy to find promising titles in a listing provided that the authors have been considerate enough to put the words *Programming Language* at the beginning of the title. Thus the book *Programming Language Concepts* will be where we expect to find it in the alphabetical ordering. However, the paper *A Comparative Study of Programming Languages* will be in another part of the listing and, unless we think of looking under *Comparative*, we will be unlikely to find it without a sequential scan through the catalogue—something that rapidly exceeds our attention span making us very likely to miss items.

The Key Word in Context (KWIC) index tries to solve this problem by listing each title several times, once for each of its keywords—“noise words” such as *a*, *the*, *and*, *of*, and so on are not counted as keywords. We might define a KWIC index as being produced by taking each title, generating *circularly shifted* copies, each with a different keyword at the beginning and then sorting the newly generated list alphabetically. A circularly shifted copy is formed by moving one or more words from the beginning of the title to the end. The title *A Comparative Study of Programming Languages* would appear four times as:

Comparative Study of Programming Languages. A
Study of Programming Languages. A Comparative
Programming Languages. A Comparative Study of
Languages. A Comparative Study of Programming

This is not very easy to read and in its final form of the index, part of the listing might be rearranged as:

A Comparative Study of Programming Languages
Programming Language Concepts
Programming Language Landscape: ...
Programming Language Structures
A Comparison of Programming Languages for Softw...
Programming Languages: Design a...
Principles of Programming Languages: Design, ...
...ion to the Study of Programming Languages
Concepts of Programming Languages
Concurrency and Programming Languages
Fundamentals of Programming Languages

...cture and Design of Programming Languages

where the titles have been aligned on the word that is being used for the alphabetical ordering and sufficient other words are provided to give some context. Also appearing would be a citation to allow the reader to find the work.

The KWIC program accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Each line consists of two parts, a Title-Part and a Reference-Part. The Reference-Part is enclosed in brackets. It is assumed that brackets never occur in either the Title-Part or the Reference-Part. An example of input data is:

```
Software Engineering with Ada [Booch 1983]
The Mythical Man Month [Brooks 1975]
An Overview of JSD [Cameron 1986]
Nesting in Ada is for the Birds [Clark et al. 1980]
Object Oriented Programming [Cox 1986]
Social Processes and Proofs of Theorems and Programs
                                [DeMillo et al. 1979]
Programming Considered as a Human Activity [Dijkstra 1965]
```

A Title-Part may be *circularly shifted* by removing the first word and appending it at the end of the line to form a new Title-Part. From each line in the input data, new lines are constructed by appending a copy of the Reference-Part from the original line to all distinct circularly shifted Title-Parts. The first word of each such line is the *keyword*. Those Title-Parts that begin with the keywords: a, an, and, as, be, by, for, from, in, is, not, of, on, or, that, the, to, with, and without are ignored. The output of the KWIC program is a listing of all the titles constructed in this way and displayed so that the title is presented with its original word ordering with all keywords lined up vertically in the center of the page. Where a line must be truncated at the beginning or end in order to fit on the printed line with the keyword aligned the truncation is shown with an ellipsis, "...".

4-2 Business

Information Retrieval

The board of directors of the Piranha Club, of which you are a member, have commissioned you to implement a small data base system for the members of the club. Members will give information that will be stored in the system and will be retrieved for the benefit of other members. Members' data will be kept in a permanent file and the system will allow the addition of new members, as well as the deletion of departing members, and also the updating of member information. Some security checks will be implemented in the system by keeping a password with each member's data, and requiring that password for certain operations. The password will be kept in the system in a ciphered form in order to improve system security. The system will display a menu of the

possible operations to the user who will then choose the desired operation. On program exit, the members table will be automatically stored in a new file.

After discussions with the board of directors, the following formats are agreed upon. The format of the menu will be the following:

1. Add a new member
2. Check membership
3. Get a member's name
4. Get a member's address
5. Get a member's phone number
6. Get information on a member
7. Change member's password
8. Remove member
9. Show member list
10. Exit program

Please enter your selection and press return:

The various messages of the query system will be the following:

Initialization from a data file

Initializing members table. Give file name:

Addition of a new member

Give ID of employee to add:
Give member name:
Give member address:
Give member phone number:
Give member password:
Member added

Checking a membership

Give ID of employee to look for:
XXXXXXX is a member
XXXXXXX is not a member

Information query on a member

Give ID of employee:
The member's name is:
The member's address is:
The member's phone number is:
Sorry but this ID does not belong to a member.

Changing a member's password

Give ID whose password must be changed:
Give new password:
Give your old password:
Give new password again:
Password has been changed
Wrong password.
You don't seem to be sure.
Password has not been changed

Removal of a member

```
Give ID of employee to eliminate:
Give Password of member you want to eliminate:
Wrong password, member could not be removed
Member removed
```

Display of the member table

```
Give administrative password:
Sorry but you do not have access to the list.
```

Copy of members table into a data file at end of execution

```
Saving members table. Give file name:
```

4-3 Scientific**Complex Algebra**

Several methods are known to find the roots of an equation. One of the most efficient of these methods is the Newton-Raphson method, developed by Isaac Newton around 1685, and refined by Joseph Raphson in 1690. We can design a Pascal program to apply the method without much difficulty. However, now that we have reached level 4, we can be a little more ambitious. We will design and implement a Pascal program to apply the Newton-Raphson method to find the root of a polynomial equation in x of a given degree *whose coefficients are complex numbers*.

The program will read in the degree of the equation, the corresponding coefficients, the requested accuracy, and the starting point for the Newton-Raphson method; it will apply the method and return a result for the root or an explicative message when the root can not be computed. The program will repeatedly prompt the user for data, and stop when the user wants to stop.

The following is the format of the dialogue messages that will be used by the program.

```
Give the degree of the polynomial:
Give the polynomial coefficients defining the function
Coefficient of degree 9 ; real part:
Coefficient of degree 9 ; imaginary part:
Indicate the precision you wish to achieve:
Indicate the maximum number of iterations:
Now give the starting value for x(real part):
Now give the starting value for x(imaginary part):
The root is 9.9999999999 + 9.9999999999i
Newton-Raphson took 9 iterations
For a precision of 9.999999E-09
More?
```