INVITATION · TO

PASCAL

FOR · THE

TRS-80

Lawrence L. McNitt

INVITATION TO **PASCAL** FOR THE **TRS-80**

INVITATION TO PASCAL FOR THE TRS-80

Lawrence L. McNitt

PETROCELLI BOOKS
Princeton, New Jersey

Designed by Diane L. Backes
Typesetting by Backes Graphics

Printed in the U.S.A.
First Printing

Library of Congress Cataloging in Publication Data

McNitt, Lawrence L.
Invitation to Pascal for the TRS-80.

Includes Index.
1. TRS-80 computers--Programming. 2. PASCAL (Computer program language) I. Title.
QA76.8.T18M374 1984 001.64'2 84-25590
ISBN 0-89433-253-8

Contents

Preface

Pascal is the language of choice for introductory programming courses. Computer science teachers choose Pascal to establish good programming habits. Most programmers retain the programming style they developed with their first programming language. BASIC and FORTRAN are primitive languages compared to Pascal. Those who initially learn to program in the more primitive languages are less likely later to use the more flexible control structures available in COBOL, PL/I, and Pascal.

Pascal allows more data types and user-defined data structures than the more primitive languages, but it is not as forgiving as BASIC. Its strong-typing feature requires that all data be fully defined and properly used. Data values should be checked for proper type at run time. When errors are found, the run is aborted. Pascal is primarily a teaching tool and secondarily a production language. Its reason for existence is to introduce programming concepts and develop skill in their use.

Pascal is available as an option for many microcomputers and will become popular for advanced high school courses and for college-level computer science courses. BASIC is the primary language for microcomputers and is the first language taught to high school students.

Pascal will gain in popularity as a scientific programming language as the number of computer science graduates mounts. FORTRAN has been the primary language for scientific computing and will remain strong because of the wealth of available software.

Pascal is less likely to make inroads into the COBOL application area since many Pascal implementations are weak in the area of disk file manipulation. COBOL is the predominant language for business file processing. The newer Pascal-like languages ADA and MODULA should continue the assault on FORTRAN and COBOL.

This book introduces the Pascal language with the Alcor Pascal system sold by Radio Shack for its TRS-80 computers. This Pascal implementation has a good reputation for completeness and efficiency of execution. This book serves as a study guide with numerous example programs. The material is presented in a logical order designed to aid the learning process. "Invitation to Pascal for the TRS-80" complements the "TRS-80 Pascal" manual, an indispensible reference tool that comes with the software package.

Part I contains an introduction to Pascal covering its major features, its philosophy, and how to use it on the TRS-80. Part II includes several applications chapters illustrating the use of Pascal for practical problem situations.

I

Introduction to Pascal

1 Introduction

OVERVIEW Pascal is the language of choice for introductory programming courses in many computer science departments. Writing style is as important in computer programming as it is in writing short stories or books. Computer science teachers choose Pascal because it establishes good programming habits. Most programmers retain the programming style they develop with their first programming language.

Those who initially learn to program in the more primitive languages of BASIC or FORTRAN are less likely to use the more flexible control structures available in Pascal, PL/I, or COBOL. In fact, those who learn Pascal as their first language are more likely to write better, more readable BASIC or FORTRAN programs than those who first learned to program with one of the more primitive languages.

Does this mean that the person who learns programming with BASIC or FORTRAN is destined to mediocrity? Not at all. Anyone who resists the inate aversion to change can study Pascal with the goal of improving his programming style. This may involve thinking new thoughts and dreaming new dreams. The good programmer continues to grow. He or she reads programs written by others, tries new languages, compares alternative approaches, and polishes his or her programming style.

Niklaus Wirth developed Pascal in the late 1960s and published the report in 1971. The name Pascal was chosen in honor of the French mathematician Blaise Pascal who lived in the 17th century. Most programming languages are acronyms and are capitalized, e.g., FORTRAN, COBOL, BASIC, APL. The programming language Pascal is an exception. Since it is not an acronym, the name Pascal is not capitalized.

"Invitation to Pascal for the TRS-80" introduces the Pascal language with the Alcor Pascal system sold by Radio Shack for its TRS-80 computers. This implementation has a good reputation for completeness and efficiency of execution. This book serves as a study guide and contains numerous example programs. It is designed to complement the language reference manual by presenting the material in a logical order to aid the learning process.

□

1.1 PHILOSOPHY

Niklaus Wirth developed Pascal because of his disenchantment with teaching introductory computer programming courses using the languages that existed in the 1960s. Computer scientists were just beginning to define the principles of their discipline. It was becoming a science known as software engineering.

LANGUAGES AVAILABLE FOR INSTRUCTION

Until the 1970s most introductory computer programming courses used FORTRAN, the predominant scientific programming language. The commands controlling program flow were primitive, resulting in unnecessary dependence on the GO TO construct.

ALGOL had much better tools for the flow of control but computer science instructors had to contend with balky compilers. These were difficult to use and inefficient in a college environment in which large numbers of students were using limited computer resources.

John Kemeny and Thomas Kurtz at Dartmouth College developed BASIC (Beginners All-purpose Symbolic Instruction Code). This language has had a tremendous impact on computing. Compared to other computer programming languages, BASIC is simple to learn and easy to use. Like FORTRAN, the flow of control commands are too primitive to satisfy the needs of computer science instruction.

PURPOSE OF PASCAL

Niklaus Wirth had two goals in developing Pascal. First, he wanted a language that would be suitable for teaching computer programming as a systematic discipline. By the late 1960s the science of computer programming was emerging and computer scientists were polishing its fundamental concepts and methods.

Second, Wirth wanted to develop an implementation of the language that would be reliable and make efficient use of computer resources in an academic environment. In a college environment the introductory computer programming courses place an immense load on the compiler as students debug their programs. The time spent executing the program is almost inconsequential. Compilation time is the bottleneck.

The needs of government and industry differ from those of academia. Computer programs in the professional world must have efficient run-time characteristics. When the program has been debugged, tne compiled object code is placed in a library. It is run many times without having to be recompiled. In an academic environment, the student debugs a program, submits it to the teacher, and begins a new program.

Pascal is a teaching tool, and is designed for fast compilation. Although some implementations of Pascal do provide fast execution speed, this was not the primary purpose. Some implementations of Pascal are weak in manipulating external files, therefore, not all Pascal implementations work well in a production environment.

ONE-PASS COMPILATION

A compiler is a computer program that takes a source program as input and produces an object program as output. A Pascal compiler translates a Pascal source program into an appropriate object program.

Compilers for most computer programming languages require at least two passes through the source program text. The first pass creates a symbol table and object program skeleton. The second pass fills in the object program skeleton with memory addresses from the symbol table.

A properly implemented Pascal compiler requires only one pass through the source program. This means that the symbol table entries must be completed before generating the object code requiring those entries. All variables and location names must be defined before they are used by the program procedure. Subroutines are defined before they are used. A one-pass Pascal compiler can provide very fast compilations.

Although Pascal appears to be a comprehensive language when compared to BASIC, it does not have all the "bells and whistles" of PL/I, ADA, and other comprehensive production languages. Compared to the more comprehensive languages, Pascal compilers can be both lean(small) and fast (compilation speed). The result is a system that makes efficient use of computer resources in an academic environment.

1.2 STRUCTURED PROGRAMMING

Structured programming requires self-discipline. It controls the way in which the programmer writes the program. It restricts the programmer's choices of commands specifying the flow of control within the program for most programming languages. The GO TO statement is just too primitive to use as a standard flow of control statement.

Pascal is designed to make structured programming as natural as possible. It employs those structured control statements proven successful. It makes the use of the GO TO statement inconvenient enough to hinder its unbridled use. The other control statements available make the GO TO statement almost unnecessary.

HARDWARE/SOFTWARE COSTS

The relative cost of hardware and software has changed dramatically during the last 30 years. Only large institutions could afford computers in the 1950s. Now

computers have not only invaded the business world but the home. The drastic increase in computer use results from lower hardware costs.

Programmer productivity has not kept pace with falling hardware prices. Software development costs have risen with programmers' salaries. The two broad categories of computer programming are systems programming and applications programming. The systems programmer writes the compilers and supervisor routines needed for the daily operation of the computer. The applications programmer writes the computer programs, such as payroll, developed specifically for the operational needs of the organization.

Programmer productivity has remained at an average 10-15 debugged, documented statements per day for applications programmers regardless of language. Only 10 percent of the programmer's time is spent coding the program in the selected language. Ninety percent is spent on other steps of the programming process. Because of the complexities of their task, systems programmers average only 2-3 debugged statements per day. These productivity figures have remained remarkably constant for the last 20 years.

During the early years of computing, hardware costs were paramount. Energy was devoted to make effective use of the computer. People, including programmers, took a back seat. The marked drop in hardware cost—combined with increasing salary costs—resulted in a shift in emphasis. Efficient use of people is now the byword. Part of this shift in emphasis is the move to improve programmer productivity. Structured programming is one of the tools to increase that productivity.

SOFTWARE PROBLEMS

Software engineering identifies three main problem areas in the software development process:

1. programmer productivity
2. software reliability
3. software maintenance.

Programmer productivity (lines of debugged statements per day) remains constant, therefore, salary increases cause software development cost increases. Software engineers seek to provide the programmer with better tools to lower software costs.

These tools include more powerful languages for which each language statement provides more functional capability. The higher-level languages of FORTRAN and COBOL replaced lower-level assembly languages for applications programming. Even higher-level database retrieval and manipulation languages further increase the power of each debugged statement, thus increasing effective programmer productivity.

Other tools from software engineering address the problem of improving programmer productivity with existing languages. Structured programming is one of these tools. By following a carefully disciplined approach, the programmer can increase the number of debugged statements per day.

Structured programming also addresses the problem areas of reliability and software maintenence. The careful, disciplined approach results in programs having fewer errors. This not only reduces the time spent debugging the program before its release, it also increases the reliability of the final program. A program is unreliable if it produces inaccurate results. It is also unreliable if it crashes (fails during run time) because of internal logical errors.

If all programmers follow the same, disciplined approach, their programming styles will be similar. The result is a library of programs that will be easy to maintain. Failure to enforce standardization results in programs of widely differing styles which are time-consuming and difficult to maintain.

FLOW OF CONTROL CONSTRUCTS

Corrado Bohm and Guiseppe Jacopini, in a highly theoretical paper, proved that a few simple control structures can be used to express any programming logic, regardless of how complex it is. The program has one entry point, one exit point, no infinite loops, and no unreachable program segments. The three types of control structures consist of sequence (natural order from top to bottom), alternation (If-then-else), and repetition (looping). Primitive GO TO statements are not essential for those languages that allow flexible nested IF (alternation) statements.

ALGOL, PL/I, and COBOL permitted flexible IF statements and programmers discovered, independently of Bohm and Jacopini, that large, complex programs could be written without use of the GO TO statement. The resulting programs were often clearer and easier to understand than the corresponding programs using traditional methods requiring the GO TO statement.

THE "GO TO" CONTROVERSY

Edsgar Dijkstra, in a letter to the editor of the ***Communications of ACM*** entitled "GO TO Statement Considered Harmful," March, 1968, advocated the position that the GO TO statement was both unnecessary and potentially harmful. His observation was that the quality of a programmer is in decreasing function of the density of GO TO statements that he used in the programs. Dijkstra suggested abolishing the GO TO statement from all higher-level languages as it is too primitive and is an invitation to make a mess of the program.

To GO TO or not to GO TO became one of the chief coffee-break topics among programmers. Some advocates of GO-TO-less programming became self-righteous zealots. The GO TO advocates became scoffers of the whole idea. For-

tunately, many programmers took a saner middle ground, observed, compared programs, experimented with the new approach, and began adopting those features permitted in their language.

The Pascal language was spawned from these early structured programming considerations, but COBOL and FORTRAN also profited. Now structured programming constructs are appearing in these traditional languages. It is becoming easier to write clear, readable programs. Dijkstra said some things that needed to be said to attract the attention of the programming community. The programming world is a better place for his efforts.

1.3 MODULAR ORGANIZATION

Concurrent with the structured control constructs came the concept of dividing a program into modules. It is easier for the finite human mind to develop a program one piece at a time. This process is known as iterative refinement, top down design, or divide and conquer.

SUBROUTINE LIBRARIES

The beginnings of modular programming predate the GO TO controversy by 10 years. Early compilers such as FORTRAN included the subroutine concept in which frequently used routines were separately compiled and placed in a library. If subroutines in the library of general interest are documented and placed in a library accessible to the programming community, they become a powerful tool to increase the productivity of programmers.

The use of subroutines requires an extra step between the source program and the final object program. The source programs are compiled into object programs that are not yet ready for execution. The main program and its subroutines must be linked together so that they can communicate properly. The resulting linked object program must then be loaded into the computer and its execution supervised by the system software developed for that task. All of this system overhead typical of the COBOL and FORTRAN environment has resulted in language implementations almost totally inappropriate for the academic community with its heavy demand for compile-debug activities.

BOTTOM-UP OR TOP-DOWN

A programmer does not have to reinvent the wheel if a subroutine to perform the desired task is already in the library. Many FORTRAN programs consist of calls to the library subroutines embedded in a framework of control statements. The subroutines form a language of a higher level than the primitive statements themselves. Programmer productivity is improved by a factor of 10 to 100. The result is a method of programming now called the bottom-up approach to modular programming.

In this approach, programmers develop the set of language tools in the form of subroutines. They then design their application programs around the tools. This method has been very successful in scientific computing for which the programs remain relatively stable over time. However, the bottom-up approach has not proven as successful in systems programming and in the business environment. These programs are constantly being modified to include enhanced abilities or changing governmental regulations.

In a business or systems programming project more emphasis is placed on local efficiency rather than on general-purpose reusability. The method of approach recommended by software engineering has become that of top-down design, or iterative refinement. Define the overall tasks first. Subdivide the overall task into a few subtasks. Tackle each subtask in turn, dividing it into subsubtasks, etc.

HIERARCHICAL ORGANIZATION

When implemented in a programming language, the overall task becomes the main routine which calls subroutines defining the subtasks. With large programs the subroutines forming the subtasks call subroutines on a still lower level defining the subsubtasks. The result is an hierarchical tree of subroutines with the main routine as its root.

Regardless of which approach is used, bottom-up or top-down, the result is a similar tree of subroutines rooted in the main calling program. With the bottom-up approach the programmer adapts the program to fit the subroutines available in the library. In the top-down approach he creates the necessary subroutines level by level until finished with the lowest level. The top-down approach does not rule out the use of predefined subroutines, but they are used on the lowest level only.

In practice, programmers use a combination of top-down and bottom-up approaches. However, practice is now moving to top-down, iterative refinement development.

ADVANTAGES OF MODULARIZATION

The complexity of a program, or program module, is not proportional to its size. Complexity may be proportional to the square of the size. A program containing 400 statements may take from four to eight times as long to understand as a similar program containing 200 statements. The larger program has more variables to monitor and more alternative paths to consider in its flow of control. The program quickly becomes more than the human mind can cope with.

Modularizing a program into an hierarchical tree of subroutines brings order out of chaos. The programmer focuses on one module at a time. Communication between calling program and subroutine is through data in the form of a parame-

ter list. The large, incomprehensible program has been subdivided into manageable chunks, each of which can be analyzed separately.

Assuming the square law is appropriate, a 400-line program will be 100 times as complex as a 40-line program. What is the result of dividing the 400-line program into 10 modules of 40 lines each? Does this mean that the modularized program is 10 times as easy to understand as the single 400-line program? Not exactly. The programmer must still remember the communication among the modules through their argument lists.

Modularization helps the maintenance programmer locate the areas of the program that need changing. It also assists in understanding the necessary inner workings of the program by isolating the module of interest from the rest of the program.

TEAMWORK

Many large programming projects include teams of programmers working together. Having a team of several programmers working on one large monolithic, disorganized program is more than difficult. Subdividing the program into modules results in easier assignment of programming tasks. The programmers work on the modules in parallel. The communication between programmers coincides exactly with the communication that takes place among the modules. The programmers are vitally concerned with the parameters used to communicate with the other modules. However, they are not concerned with the inner workings of the other modules.

INPUT-PROCESS-OUTPUT

Each module of a program accepts one or more inputs, processes them, and produces one or more outputs. The inner workings of the module can be kept hidden. The module is a black box which performs its task converting inputs to outputs as specified. Only when the module is suspected of behaving improperly, or if changes are necessary, does the box need to be opened.

MODULE AS MATHEMATICAL FUNCTION

Two programmers faced with the task of modularizing the same program may define different sets of modules. They will modularize the program in different ways. Computer scientists suggest using the mathematical concept of a function as a guide when modularizing a program. The function

$w = f(x, y, z)$

states that the result w is a function of the variables x, y, and z. There are three inputs to this function, but only one output. This is the key.

Each module should have only one output; that is, it should perform only one task. If necessary, divide complex modules performing more than one task into two or more modules, each performing one task.

On the highest levels of the subroutine tree, a module may have one or more files as input and create one file as output. At intermediate levels the module may have one or more vectors as input and produce one vector as output. At the lowest module levels each module may have one or more values as input and generate one value as output.

1.4 STRONG TYPING

Pascal requires that each variable be explicitly defined and properly used. Some programmers view this as a blessing; others view it as a curse. Computer science instructors love it because it forces beginning programming students to be consistent in the definition and use of variables. A variable associates a name with a value. The value is assigned to the named variable by placing it in memory at the location reserved for that variable by the compiler.

STRONG TYPING AS A BLESSING

One facet of human nature is the ability to redefine our view of the world. The programmer begins writing the program with one view and the data required by that view. After discussing the program with the customer (who will later be using the program), the programmer revises his view of the data. Forgetting about the impact on the portions of the program already written, the programmer begins to use the new view.

The result is a program that does not fit together well. If the programmer is lucky, the inconsistency will come to light during testing and be corrected. The unlucky outcome occurs when the program goes into production giving results that look plausible but that are still wrong. Or, the results are usually correct, but under rare conditions the inconsistency causes incorrect results.

Strong typing enforces consistency in data definition and use. Most Pascal programs do type checking at run time; many other programming languages do not. Run-time type checking is expensive because the programs run more slowly. However, they are more likely to give notice of erroneous results. Execution speed is not of primary importance to introductory computer science students, therefore, those learning Pascal should use the type-checking feature.

Some Pascal systems are designed to produce programs that perform well in a production environment. Run-time type checking is an option for these systems. The careful programmer will use type checking unless there is an overpowering reason not to use it.

STRONG TYPING AS A CURSE

Most systems programmers don't like strong typing. They often need to work on a level only slightly removed from the inner workings of their computer. They are working with internal codes that are not meaningfully defined in Pascal. Pascal is not necessarily efficient at handling these codes. The systems programmer turns off the type checking to do what needs to be done without being razzed by the compiler or the executing program.

Many Pascal systems produce programs that are not very "user friendly." If the user types a letter of the alphabet when the program expected a numeric digit, the program aborts (stops running) and the user has to begin again. To circumvent this problem programmers may turn off the run-time type checking feature and put in their own type checking routines that handle the user more gently.

1.5 PROGRAMMING PROCESS

The programming process consists of several steps:

1. define the problem
2. develop the algorithm
3. code the program
4. debug and test the program
5. document the program
6. place program in production
7. maintain program.

Beginning programmers tend to focus on step 3. As programs get larger and more complex, an inordinate amount of time is spent with step 4. To prevent wasting time on step 4 they must turn additional attention to step 2. Developing the algorithm consists of designing the step-by-step approach with its data organization.

Most programmers write programs for other people. To prevent building the "wrong" program, the programmer must spend more time with the user defining the problem (step 1) and preparing documentation (step 5). Program maintenance includes correcting bugs that escaped step 4. Maintenance also includes adding new features to the program. During the eight to twelve-year life of a typical program, many programmers find they will have devoted more time to ongoing maintenance than to the initial programming effort indicated by steps 1 through 6.

WHAT CONSTITUTES A GOOD PROGRAM?

Software engineering is the science of software development. Software consists of the computer programs that make the computer work. A computer program

is a set of instructions for the computer to execute one at a time. The Pascal compiler is a software product (program) that translates Pascal source programs into object programs usable by the computer system.

Why is one program better than another one? What is a good program? Is there such a thing as a bad program? Is it possible to label programs good, better, and best? Is it possible to label programmers? Is a programmer who writes good programs a "good" programmer?

The following is a list of desirable qualities that a computer program may have:

1. work correctly
2. clear, readable, easy to understand
3. fast execution speed
4. compact (fit in small amount of memory)
5. easy for beginners to learn
6. efficient for regular use by experienced operators
7. quickly programmed
8. portable (easy to convert to another computer model).

This list is not exhaustive. What type of computer program is appropriate for the task at hand? Some of these qualities are mutually conflicting. Optimizing the program execution speed may increase its memory requirements and limit its portability. Finishing the program as quickly as possible does not leave much time for optimizing any of the other qualities. A program that is easy for a beginner to learn is often very annoying to the experienced operator.

Almost everyone agrees that the program must work correctly. That is why it is number 1 on the list. Software engineering has elevated quality 2 to the same level; every program should be clear, readable, and easy to understand. The programmer chooses the other qualities on the basis of the problem situation. The program should be appropriate to its task and to the people who will be using it.

EGO-LESS PROGRAMMING

Gerald Weinberg popularized the concept of ego-less programming. After investing many hours in their work programmers perceive their programs as extensions of themselves. They conceive the inner workings of the program in their minds, design and build them using creative talents similar to those of a sculptor. They nurse the program through the testing and debugging stage like a mother takes care of her child. Is it any wonder that the programmer's ego is so tied up with his or her program?

What happens to a programmer's ego if a bug is found after the program is placed in production? His ego is deflated. What if the user doesn't like the program? Another blow to his ego. Other programmers may comment about his dumb program. He may lash out in anger. The programming manager recommends changes in his programming style. He starts looking for a position with a company that is more appreciative.

Gerald Weinberg suggests divorcing the ego from the program. The program is an object designed to perform a task. The programmer should accept advice and criticism without suffering a bruised ego. Programmers should be encouraged to review each other's work at regular intervals. They should study others' methods and programming styles with the goal of improving them to the benefit of all.

Most programmers must make a conscious effort to remove the ego factor from their programming. They must work at taking an objective view of the programs they write. Above all, they must be willing to adapt, to try new approaches.

BUILDING ON THE EXPERIENCES OF OTHERS

The serious programmer soon learns to build on the experiences of others. The following few books are rich in personal experiences, immensely informative, and delightful to read.

"The Psychology of Computer Programming" by Gerald Weinberg does more than any other book to present the programming life as it is rather than as it ought to be. This book is worth five years of on-the-job experience to a budding programmer. To be forewarned is to be forearmed for those who must relive these traumatic incidents.

"The Mythical Man Month" by Frederick P. Brooks provides a rare opportunity to sit with a man who writes about what happened, including mistakes, without trying to cover up. This Biblical honesty is a far cry from the rulers of ancient Egypt who tried to erase any record of their failures, leaving only glowing accounts of their successful military campaigns. This book is for those who unknowingly are about to become embedded in the tar pit of missed deadlines and cost overruns of large software projects.

"How to Manage Structured Programming" by Edward Yourdon points out some of the pitfalls facing the person who thinks he is going to bring about a revolution in the way programmers, managers, or users operate. Yourdon includes advice about how to bring about change. He discusses modern programmer productivity tools in the light of corporate politics.

All three books are entertaining and authoritative. Computer programming should not be a lone wolf activity between a computer and a programmer. Computer programming is a social activity carried on by real people who are often much too human.

2 Primitive programs

OVERVIEW The first step in learning Pascal is to become familiar with the computer, its keyboard, disk drive operation, and setting up the printer. With this comes the need to learn how to use a text editor for creating and modifying Pascal source programs. Another important duty is the selection of software for the major Pascal system disk.

This chapter covers some of the preliminary details needed to use the Radio Shack Alcor Pascal system. This is the Pascal system distributed through Radio Shack stores. It also covers the overall structure of the Pascal language and gives some simple example programs. The goal is to introduce the language and to demonstrate its use on the Radio Shack TRS-80.

□

2.1 PRINTING MESSAGES

Become familiar with your computer before using the Pascal system. Alcor Pascal works well on a Radio Shack TRS-80 Model I, Model III, or Model IV with at least 48K RAM, two disk drives, and a printer. TRSDOS (Tandy Radio Shack Disk Operating System) is the most common operating system for these computers, although Alcor Pascal operates on other operating systems.

OPERATING SYSTEM UTILITIES

The TRSDOS utilities of primary interest include DIR (display directory of files stored on disk), BACKUP (create backup copies of entire disks), and FORMAT (initialize data diskettes). The disk drives are number 0 and 1 for a two-drive Radio Shack system. The diskette in drive 0 should contain the TRSDOS system software for "booting up" the system and providing the system utilities. The diskette in drive 1 can be a data diskette. A data diskette does not contain the TRSDOS system software.

WORKING COPIES

The Alcor Pascal system for the Models I and III comes with five diskettes. Two of the diskettes are for the Model III and already include the TRSDOS 1.3 system

software. The other three diskettes are for the Model I and require that users supply their own TRSDOS 2.3 system software. The TRS-80 Model IV can use the Model III version of Alcor Pascal, but the special Model IV Alcor Pascal makes better use of the faster Z-80 processor.

Use the original diskettes only for preparing working copies. The wise computerist will keep the originals in a safe place, a backup set in a separate safe place, and will use one or more working sets for normal processing. One set of working copies kept near the computer work area can function as the system master for creating customized system diskettes for writing and running Pascal programs.

Model III and IV owners can create backup and working copies of the Alcor Pascal system disks immediately. Turn on the computer before placing a diskette in a disk drive. Let it warm up for a few seconds. Place one of the Pascal system diskettes in drive 0 and then press the orange Reset key. Answer the requests for the date and time in the format specified, pressing the ENTER key after each one.

Call the BACKUP utility by typing the word BACKUP followed by the ENTER key. With the system disk to be copied in drive 0, designate 0 for the SOURCE Drive Number and 1 for the DESTINATION Drive Number. Use PASSWORD as the SOURCE disk master password. The BACKUP utility will format the blank diskette in drive 1 and then copy the contents of the SOURCE diskette to the DESTINATION diskette.

The three diskettes containing the Model I version of Alcor Pascal do not contain the TRSDOS 2.3 system software. Therefore, the Model I owner must call the backup utility from one of his system diskettes before inserting the SOURCE and DESTINATION diskettes for the copy operation.

PASCAL SYSTEM DISKETTES FOR MODELS III AND IV

Many TRSDOS system files are not necessary for Pascal program development. The TRSDOS PURGE command with the system option may be used to remove the files:

BASIC/CMD	XFERSYS/CMD	MEMTEST/CMD
CONVERT/CMD	LPC/CMD	HERZ/BLD

from the master working copies. For Model III and IV owners the two system diskettes are labeled Disk 1 of 2 and Disk 2 of 2. Disk 1 of 2 is the primary system diskette to use when learning Pascal. Diskette 2 of 2 needs a copy of the file ERRORS/DAT from disk 1 of 2 to provide a complete Pascal compiler. Disk 2 of 2 does not contain the text editor for creating Pascal source programs.

The Pascal compiler on Disk 2 of 2 consists of a segmented Pascal compiler allowing additional space for compiling large source programs. The Pascal compiler for either diskette produces object programs consisting of pseudo-machine language which must be interpreted as it is executed. Disk 2 of 2 contains a pro-

gram that generates native Z80 microprocessor code from the pseudo-code. The result is a program that runs up to five times faster but takes up to three times as much internal memory to run.

Usually, Model III and Model IV owners use the Pascal system diskette 1 of 2 containing TRSDOS 1.3 as the development system during the process of creating and debugging the Pascal program. When finished, the source and object programs are saved permanently on other diskettes. After becoming familiar with the editor, delete the files

HELP/HLP	KEY/HLP	CMD/HLP
PATCHER/CMD	DATABASE/PCL	LDOS/PAT

from the working copy of the system disk 1 of 2, leaving more room for saving source and object programs under current development.

PASCAL SYSTEM DISKETTES FOR MODEL I

Model I owners keep their own TRSDOS 2.3 diskette in drive 0 and use the Pascal system diskettes from drive 1. The resulting Pascal source and object programs will reside on the user diskette in drive 0.

LDOS OPERATING SYSTEM

Users of the LDOS operating system will need to patch their operating system. Detailed instructions for doing this are given in the Alcor Pascal Reference Manual.

CREATING AND EDITING PASCAL SOURCE PROGRAMS

The Alcor Pascal system includes a text editor for creating and editing text files. The resulting files are compatible with normal TRSDOS files if they are saved in the ASCII format. Radio Shack's SCRIPSIT word processor is a suitable program editor if the source programs are saved using the S,A program-name format for creating ASCII files.

The text editor consists of a main file ED/CMD and three subsidiary files—HELP/HLP, KEY/HLP, and CMD/HLP used for displaying help messages from within the editor. When editing an old file, the editor creates a temporary work file labeled T011/TMP to hold changes. After the editing session is completed, the system assigns the old name to the working file and deletes the old file.

The text editor maintains a buffer capable of holding up to 13,000 characters in internal memory. For pre-existing files the APPEND command adds more lines of the old file to the buffer and the WRITE command writes lines from the buffer to the new file. Typing Shift/@ (hold shift key down while typing @ symbol) adds a blank line to the buffer. This operation is needed to create space in the buffer when creating a new file.

CURSOR MOVEMENT

Simple cursor movement uses the standard arrow keys. The cursor will move left or right one character and up or down one line at a time. Holding the Clear key down while typing the appropriate cursor movement key will move the cursor to the left or right of the line or scroll the document one page toward the beginning or toward the end of the file.

TEXT INSERTION AND DELETION

Holding the Shift key down while typing the right-hand cursor movement key deletes the character under the cursor. Holding the shift key down while typing the left-hand cursor deletes the entire line. Holding the Clear key down while typing the letter K deletes the text from the cursor to the end of the line.

Holding the Shift key down while typing the @ symbol inserts a blank line into the text buffer at the current location. The normal text editing mode is type-over in which new text replaces existing text. Holding the Clear key down while typing the letter I changes the mode to insertion mode until the next cursor movement key is pressed or until the ENTER key is typed.

COMMAND MODE

Holding the Clear key down while typing the letter C places the editor in command mode. The commands of primary interest include APPEND, WRITE, and EXIT. APPEND adds the specified number of lines to the buffer. WRITE sends the specified number of lines from the buffer to the work file. EXIT writes the file to the disk.

The QUIT command leaves the editor without saving the changes. The HELP command displays help messages. SHOWLINE positions the cursor at a specified line number in the buffer. INSFILE inserts text from a specified external file into the buffer. The commands FIND and REPLACE perform the common text-editing functions. The commands + and - position the cursor a specified number of lines forward or backward in the buffer.

SAMPLE PASCAL PROGRAM

This program and the next include a short excerpt from "She was a Phantom of Delight" by William Wordsworth. The two lines

> And now I see with eye serene,
> The very pulse of the machine

are taken out of context but do reflect the exciting anticipation of the person embarking on new ventures with the computer.

The first exercise is to use the editor to create an initial Pascal source program that displays the first line of the excerpt. Use Shift-0 (zero) to switch between uppercase and lowercase letters. The text of the source program follows:

```
PROGRAM PRINT1;
BEGIN
   WRITELN ('And now I see with eye serene,');
END.
```

Every Pascal program must have the parts PROGRAM, BEGIN, and END. The period after the last END is mandatory. The semicolon after the program name is also mandatory. Semicolons separate Pascal statements.

The file ED/CMD contains the text editor. Typing the command

```
ED
```

loads the editor and the message *EOB (end of buffer) appears at the top of the screen. Hold the Shift key down and type the @ symbol four times (one for each line of the program). Type in the program exactly as it appears including the period and the semicolons. Once the text is entered, hold the Clear key down while typing the letter C to enter command mode. Type the command EXIT and respond with the file name PRINT1/PCL to place the Pascal source program PRINT1/PCL on the system diskette in drive 0.

The Pascal compiler translates the source program into an object program. The command

```
PASCAL PRINT1
```

compiles the source program PRINT1/PCL and generates the object program to be stored in the file PRINT1/OBJ. The compiler sends a listing of the program to the video screen together with any error messages.

Running the Pascal object program requires a routine called RUN/CMD. Typing the command

```
RUN PRINT1
```

runs the object program PRINT/OBJ. The run time routine asks for files for the two logical files called INPUT and OUTPUT. Normally, data entry for the INPUT file comes from the keyboard and output from the OUTPUT file goes to the video display. Pressing the ENTER key for each request gives this usual assignment.

The normal input and output can be redirected. Typing a TRSDOS file name allows input or output to be redirected to a disk file. Typing the letter L sends the output to the printer rather than to the video display.

The output of the run is the one line message

```
And now I see with eye serene,
```

contained in the WRITELN statement of the Pascal program.

STACK AND HEAP

The editor, the compiler, and the resulting Pascal programs identify two areas of memory usage. The STACK contains most of the variables. The HEAP contains dynamic variables and file descriptors. The editor or compiler displays the amount of STACK and HEAP space used out of the amount available. The amount of HEAP and STACK allocated by the system is usually sufficient. If not, more can be allocated.

EDITING AN EXISTING PROGRAM

Now change the program to include the second line. The command

ED PRINT1/PCL

loads the editor and tells it to edit the file PRINT1/PCL. The editor automatically appends the first 100 lines of text to the buffer. Move the cursor to the WRITELN statement and insert a blank line by holding the Shift key down while typing the @ symbol. Now type the line

```
WRITELN ('the very pulse of the machine');
```

giving the following program:

```
PROGRAM PRINT1;
BEGIN
    WRITELN ('And now I see with eye serene,');
    WRITELN ('The very pulse of the machine');
END.
```

Hold the Clear key down while typing the letter C to enter the command mode. Type the command

EXIT

and respond with the ENTER key to the request for the file name to use for the edited version.

SPECIAL EXECUTION-TIME KEYS

The Break key will terminate the current program and will also stop the compiler. If the @ key is pressed during the running or compiling of the program, the computer will pause until the Enter key tells it to resume execution. With extensive output going to the video screen, the @ and Enter keys may be used many times during the program execution or compilation.

ERROR MESSAGES

The compiler will flag parts of the program that contain syntax errors and will attempt to point out the problem with error messages. It is a good idea to become familiar with these messages and the problems they point out by introducing intentional errors into a program.

Use the text editor to remove the right-hand quote from the first WRITELN statement of the program PRINT1/PCL. The resulting program should be as follows:

```
PROGRAM PRINT1;
BEGIN
   WRITELN ('And now I see with eye serene,);
   WRITELN ('The very pulse of the machine');
END.
```

The command

```
PASCAL PRINT1
```

compiles the updated Pascal source program contained in PRINT1/PCL and includes several error messages. The missing quote symbol throws the compiler off the track, causing additional error messages.

MOST-USED EDITOR CONTROL KEYS

↑	Cursor up one line
↓	Cursor down one line
→	Cursor right one character
←	Cursor left one character
Clear ↑	Scroll back one page
Clear ↓	Scroll forward one page
Clear →	To end of line
Clear ←	To beginning of line
Shift →	Delete character under cursor
Shift ←	Delete line
Clear K	Delete to end of line
Shift @	Insert blank line
Clear I	Insert text at cursor position
Clear O	Split line into two lines at cursor

EDITOR COMMAND MODE (Clear C)

APPEND	Append text to buffer
WRITE	Write text to file
EXIT	Exit editor and save file
QUIT	Quit editor without saving file
+ 120	Move cursor forward 120 lines
- 230	Move cursor back 230 lines
SHOWLINE 45	Move cursor to line 45
INSFILE	Insert another file at cursor position
FIND	Search for character string
REPLACE	Replace character string

EXERCISES

1. Write a Pascal program to print the following lines from "The Raven" by Edgar Allan Poe:

   Once upon a midnight dreary
     while I pondered weak and weary,
   Over many a quaint and curious
     volume of forgotten lore,

2. Remove the semicolon at the end of the first WRITELN statement and observe the error messages.
3. Modify the program of problem 1 by removing the period following the END statement.
4. Modify the program of exercise 1 by removing the semicolon after the program name of the PROGRAM statement.

2.2 PARTS OF A PASCAL PROGRAM

The parts of a Pascal program should be as follows:

program heading

label declarations

constant definitions

type definitions

variable declarations

procedure and function definitions

main body of program

Some parts of the program may not be necessary. For example, the simple programs of Section 1.1 did not need to declare labels, constants, types, or variables.

PROGRAM HEADING

The program heading is the first line of the Pascal program. It gives the name of the program and the needed files. The program heading

```
PROGRAM PRINT1 (INPUT,OUTPUT);
```

identifies the program with the name PRINT1 and specifies that the program will use normal input and output channels. Input data comes from the keyboard and output goes to the video display. Alcor Pascal ignores the text contained within the parentheses.

The program heading

```
PROGRAM PRINT1 (OUTPUT);
```

specifies that there will be output only, no input. Most Pascal programs include normal input and output operations. Many Pascal compilers allow the use of these files without explicitly defining them. The program heading

```
PROGRAM PRINT1;
```

is equivalent to the heading

```
PROGRAM PRINT1 (INPUT,OUTPUT);
```

for many Pascal systems.

LABEL DECLARATIONS

Pascal uses GO TO statements which require labels (names) to identify the sections of the program used as the objects of the GO TO statements. A program that uses no GO TO statements needs no labels. The label declaration part of Pascal programs alerts the person reading the program that GO TO statements are present and permits the development of one-pass Pascal compilers.

CONSTANT DEFINITIONS

A constant does not change. Pascal permits the assigning of a name to a quantity that never changes. For example, the symbol PI may be assigned to the numerical quantity 3.14159 for formulas involving the area or circumference of a circle.

TYPE DEFINITIONS

Pascal has the following fundamental data types:

1. Real: Numbers with decimal points.
2. Integer: Whole numbers.
3. Char: Symbols, including letters of the alphabet, numeric digits, and punctuation used by the computer to communicate with the outside world.
4. Boolean: Logical result (true, false, binary 0 or 1).

Using these primitive data types, the Pascal programmer can define more complex types. This is a powerful tool in the hands of an experienced programmer.

PROCEDURE AND FUNCTION DEFINITIONS

The programmer subdivides large programs into smaller sections called modules. Each module is a procedure or function. The main procedure comes at the end of the program. Procedures and functions called by the main procedure must be previously defined if Pascal is to be a one-pass compiler. Short programs, such as PRINT1 in Section 1.1 may have only one procedure. Long programs should be created as a main program calling subroutines in the form of procedures and functions.

MAIN BODY

The main body of the Pascal program consists of a single compound statement formed from elementary statements separated by semicolons. The single compound statement is also called a block. The BEGIN and END give the scope of the block in the same manner that the left parenthesis "(" and the right parenthesis ")" define the scope of a parenthetical expression. Each procedure and function contains its BEGIN and END block definition. The END for the main program block is followed by a period. Subsidiary functions and procedures require semicolons following the END delimiters of their compound statements.

NULL PROGRAMS AND STATEMENTS

Pascal permits the creation of programs and functions that don't do anything. The program

```
PROGRAM DONOTHING;
BEGIN
END.
```

defines the program DONOTHING which has no input or output and does no processing. Its BEGIN . . . END block is empty. In Pascal terminology, it is a null

program. Pascal also permits the definition of empty files which contain no data. They are called null files. A variable normally contains data. A variable that contains no data is a null variable.

SPECIAL SYMBOLS NOT VISIBLE ON THE KEYBOARD

Several Pascal symbols are not visible on the TRS-80 keyboard. The editor generates these symbols as you hold the Clear key down while typing the required numeric digit. There is also an alternate symbol that Pascal compilers accept for those systems having terminals lacking the desired primary symbol.

The following lists the symbol and its Clear key code for the Alcor Pascal editor:

Character	*Clear/digit*
[	Clear/1
]	Clear/2
^	Clear/3
{	Clear/4
}	Clear/5
|	Clear/6
~	Clear/7

Not every terminal has these symbols; therefore, the following alternate symbols are common:

Primary symbol	*Alternate symbol*	*Meaning*
{	(*	Open comment
}	*)	Close comment
^	@	Pointer symbol
[	(.	Open array index
]	.)	Close array index

WRITELN VERSUS WRITE STATEMENTS

There are many ways of writing computer programs. This brings out the creative instincts of the programmer. Consider the line

I came, I saw, I conquered.

attributed to Julius Caesar. The program

```
PROGRAM PRINT2A (OUTPUT);
BEGIN
   WRITELN ('I came, I saw, I conquered.');
END.
```

uses a style similar to the program of Section 1.1. The output of this program consists of the single line

I came, I saw, I conquered.

The WRITELN statement finishes the current line of output so that any subsequent output statements begin a new line. The following program uses a separate WRITELN statement for each phrase of the quotation:

```
PROGRAM PRINT2B (OUTPUT);
BEGIN
    WRITELN ('I CAME,');
    WRITELN ('I SAW,');
    WRITELN ('I CONQUERED.');
END.
```

The output of this program follows:

```
I came,
I saw,
I conquered.
```

The WRITE statement places the indicated item on the output line without the end-of-line signal. Use of the WRITE statement tells the system that more will be added to the line later. The following program uses the WRITE statements for the first two phrases and finishes with the WRITELN statement for the last phrase:

```
PROGRAM PRINT2C (OUTPUT);
BEGIN
    WRITE ('I came,');
    WRITE ('I saw,');
    WRITELN ('I conquered.');
END.
```

The output from this program follows:

```
I came,I saw,I conquered.
```

Now the problem is the lack of control over the spacing of the phrases. There should be a space after each comma and before the beginning of the next phrase. The following program includes an extra space at the end of each of the first two phrases:

```
PROGRAM PRINT2D (OUTPUT);
BEGIN
    WRITE ('I came, ');
    WRITE ('I saw, ');
```

```
    WRITELN ('I conquered.');
END.
```

An alternative approach is to place the extra space at the beginning of each of the last two phrases. The following program uses this approach:

```
PROGRAM PRINT2E (OUTPUT);
BEGIN
    WRITE ('I came,');
    WRITE (' I saw,');
    WRITELN (' I conquered.');
END.
```

In either case, the following output results:

```
I came, I saw, I conquered.
```

SEMICOLONS SEPARATE STATEMENTS

Pascal uses semicolons to separate statements within a BEGIN... END block. Strictly speaking, a semicolon is not needed between the last statement of the BEGIN... END block and the END keyword. The Pascal compiler assumes that a null statement follows the last semicolon.

The program

```
PROGRAM PRINT2F (OUTPUT);
BEGIN
    WRITELN ('I came, I saw, I conquered.')
END.
```

does not need a semicolon after its one WRITELN command. The semicolon after the WRITELN command in the following program is permissible, but results in a null statement between the WRITELN statement and the END.

```
PROGRAM PRINT2G (OUTPUT);
BEGIN
    WRITELN ('I came, I saw, I conquered.');
END.
```

The following program does not use the semicolon before the END delimiter and does not generate the extra null statement:

```
PROGRAM PRINT2H (OUTPUT);
BEGIN
    WRITE ('I came,');
    WRITE (' I saw,');
    WRITELN (' I conquered.')
END.
```

The following program uses the extra semicolon resulting in the extraneous null statement:

```
PROGRAM PRINT2I (OUTPUT);
BEGIN
    WRITE ('I came,');
    WRITE (' I saw,');
    WRITELN (' I conquered.');
END.
```

EXERCISES

1. The line, "It has long been an axiom of mine that the little things are infinitely the most important." from "The Adventures of Sherlock Holmes" by Sir Arthur Conan Doyle is appropriate for computer programming. Write a program that displays this quotation using two or more WRITELN statements.
2. Write a program displaying the quotation in exercise 1 using one or more WRITE statements followed by a WRITELN statement.
3. Remove the semicolons from the program of exercise 1 or 2 and note the error messages that result.

2.3 VARIABLES

A constant is an entity that cannot change. A label is a named section of a program used as an object of a GO TO statement. The variable is a named entity that can be changed during the running of the program.

IDENTIFIERS

An identifier is a program name, constant name, variable name, procedure name, or label. The first character must be a letter of the alphabet. Permissible characters for the rest of the name include the digits 0 through 9, the dollar sign $, and the underline character _. There is no distinction between uppercase and lowercase letters. The names

APPLE Apple apple aPpLe

are equivalent. Most programmers use either all capital letters or all lowercase letters for variable names.

Alcor Pascal allows names longer than eight characters but only the first eight are significant. An identifier cannot contain a space and cannot be continued from one line to the next. Most programmers using Alcor Pascal will limit variable names to eight characters or less.

RESERVED WORDS

Certain words have special meaning for the Pascal compiler and cannot be used as identifiers within the source program. The reserved words are

AND	END	NIL	SET
ARRAY	FILE	NOT	THEN
BEGIN	FOR	OF	TO
CASE	FUNCTION	OR	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	PROCEDURE	VAR
DO	IN	PROGRAM	WHILE
DOWNTO	LABEL	RECORD	WITH
ELSE	MOD	REPEAT	

CONSTANT DEFINITIONS

The constant definition attaches a symbolic name (identifier) to a data constant. The form is

identifier = constant

for which identifier is a user-defined name and constant is a valid integer, real number, or character string. The keyword CONST starts the constant section of the program.

The following program contains several constant definitions and displays the constants:

```
PROGRAM PRINT3A (OUTPUT);
CONST
    NUMBER = 365;
    PI = 3.14159;
    LINE1 = 'The end of a perfect day';
    LINE2 = 'Don't tread on me';
BEGIN
    WRITELN ('NUMBER = ',NUMBER);
    WRITELN ('PI = ',PI);
    WRITELN ('LINE1 = ',LINE1);
    WRITELN ('LINE2 = ',LINE2)
END.
```

The sample run follows

```
Number =       365
PI =  3.14159E+00
The end of a perfect day
Don't tread on me
```

Notice the consecutive quote symbols in LINE2. Most computer programming languages use this method of embedding a quote within a string of characters.

NUMBERS

Numerical quantities are of two types: integer and real. An integer is a whole number and allows no fractional part. A real number permits a fractional part. Real numbers are stored using scientific notation consisting of a mantissa (fraction) and an exponent (power of 10). The scientific notation number

2.74125×10^4

stands for the decimal number 27,412.5. Pascal displays the number as 2.74125E +04. This notation is also useful for defining real constants and for creating data files to be read into the program.

MATCHING DATA TYPES

Pascal requires the use of appropriate data types. Integer variables cannot receive real values. Character variables cannot receive numerical quantities nor can numerical variables receive character data. The data types must match.

ASSIGNMENT STATEMENTS

The Pascal assignment statement is of the form:

identifier := identifier, literal, or expression;

An identifier is a user-supplied name. A literal is a numerical quantity or a character string enclosed in quotes. An expression defines arithmetic operations on literals and identifiers. The assignment statements

identifier := identifier

and

identifier := literal

move the contents of the literal or identifier on the right to the identifer on the left. The assignment statement

identifier := expression

computes the expression on the right and places the result in the identifier on the left of the := symbol.

VARIABLE DECLARATIONS

Variables use symbols to represent the data. The computer reserves a portion of memory for each variable. The data must be defined before it is processed. The constant and variable definitions come before the BEGIN... END block of processing statements.

The keyword VAR signals the start of the variable definition portion of the program. The three data types of interest now are integer, real, and character. The keywords INTEGER, REAL, and CHAR define these data types. There are several ways of organizing the VAR section. The following defines each variable separately:

```
VAR
    NUMBER : INTEGER;
    AGE : INTEGER;
    AREA : REAL;
    AMOUNT : REAL;
    ANSWER : CHAR;
```

When more than one variable is of the same type, common practice is to list the variable names in series with one colon separating the variable list from the type specification. The following section uses this approach:

```
VAR
    NUMBER,
    AGE : INTEGER;
    AREA,
    AMOUNT : REAL;
    ANSWER : CHAR;
```

To conserve space, some programmers will pack several identifier names on the same line as the type definition. The following illustrates this approach:

```
VAR
    NUMBER, AGE : INTEGER;
    AREA, AMOUNT : REAL;
    ANSWER : CHAR;
```

Packing too much information on one line makes the program hard to read. The first approach results in more readable programs.

ASSIGNING VALUES TO VARIABLES

The following program uses literals and constants to assign values to variables:

```
PROGRAM PRINT3B (OUTPUT);
CONST
```

```
    PI = 3.14159;
VAR
    NUMBER,
    AGE : INTEGER;
    AREA,
    AMOUNT : REAL;
    ANSWER : CHAR;
BEGIN
    NUMBER := 25;
    AGE := 43;
    AREA := PI;
    AMOUNT := 23.95;
    ANSWER := "Y";
    WRITELN ('NUMBER IS     ',NUMBER);
    WRITELN ('AGE IS        ',AGE);
    WRITELN ('AREA IS       ',AREA);
    WRITELN ('AMOUNT IS     ',AMOUNT);
    WRITELN ('ANSWER IS     ',ANSWER)
END.
```

Running the program produces the following output:

```
NUMBER IS          25
AGE IS             43
AREA IS       3.14159E+00
AMOUNT IS     2.39500E+01
ANSWER IS     Y
```

EXERCISES

1. Write a Pascal program to display the following line from Robert Burns' "To a Mouse":

   The best laid plans o' mice an' men

2. Write a Pascal program that declares the two integer variables SIZE and ROWS, the three real variables LENGTH, WIDTH, and AREA, and the character variable FLAG. Use assignment statements to place the value 10 in SIZE, 5 in ROWS, 2.5 in LENGTH, 1.0 in WIDTH, 2.5 in AREA, and "N" in FLAG. Use WRITELN statements to display the contents of the variables.

3. Write a Pascal program that defines constants having the same names and values as the program of exercise 2. Use WRITELN statements to display the constants.

2.4 ARITHMETIC

A simple arithmetic expression performs one arithmetic operation. The expression

```
12 + 13
```

forms the sum of the value 12 and 13. The symbol "+" designates the arithmetic operator addition. The result of an arithmetic expression is called its value.

LOCATION OF EXPRESSIONS

Arithmetic expressions are usually associated with assignment statements. The statement

```
ANSWER := VALUE1 + BASE1;
```

assigns the value of the expression to the variable ANSWER. The output statement

```
WRITELN ('ANSWER IS ',12 + 13);
```

displays the message "ANSWER IS" together with the value of the expression (12 + 13).

INTEGER ARITHMETIC

Integers are whole numbers. They may be positive, negative, or zero. They cannot have fractional parts. The literals 12 and 13 are integers. The operator symbol separates the two operands upon which it works. The operator symbols

+	Addition
-	Subtraction
*	Multiplication
DIV	Integer (truncated) quotient
MOD	Modulo (integer remainder)

give integer results when both operands are integers. The negation operator symbol is the minus sign, the same as subtraction, and negates the operand on the right.

The arithmetic expression

```
12 + 13
```

gives the integer result 25 since the literals 12 and 13 are both integers. The expression

```
3 * 4
```

gives the result 12 after multiplying the integers 3 and 4. The expression

```
8 - 5
```

gives the result 3 computed by subtracting the integer 5 from the integer 8.

The expression

```
14 DIV 4
```

gives the integer result 3, and the expression

```
14 MOD 4
```

gives the result 2 since dividing the integer 4 into the integer 14 gives the integer quotient 3 with 2 the remainder. The expression

```
-ANSWER
```

negates the value in the variable called ANSWER.

The following Pascal program computes the area of a rectangle (assuming that all measurements are whole numbers):

```
PROGRAM AREA (OUTPUT);
VAR
    LENGTH,
    WIDTH,
    AREA : INTEGER;
BEGIN
    LENGTH := 40;
    WIDTH := 30;
    AREA := LENGTH * WIDTH;
    WRITELN ('LENGTH     ',LENGTH);
    WRITELN ('WIDTH      ',WIDTH);
    WRITELN ('AREA       ',AREA)
END.
```

The sample output follows:

```
LENGTH     40
WIDTH      30
AREA       1200
```

REAL ARITHMETIC

If both operands of the arithmetic operation are real, then the value of the expression is also real. The operator symbols for real operands include

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division

and give real results.

The expression

2.7 + 4.1

gives the real result 6.8. The expression

12.75 - 9.25

gives the result 3.5. The expression

2.5 * 2.5

gives the product 6.25. The expression

8.8 / 2.0

gives the real result 4.4.

The integer arithmetic operators DIV and MOD are not defined for real numbers. The division operator symbol "/" requires that both operands be real. The expression

14 / 4

has the real value 3.5 rather than the integer quotient 3. Both values are converted to real form before the division takes place.

The following program computes the area and circumference of a circle of radius 3.5 using real values:

```
PROGRAM CIRCLE (OUTPUT);
CONST
    PI = 3.14159;
    RADIUS = 3.5;
VAR
    AREA,
    CIRCUM : REAL;
BEGIN
    AREA := RADIUS * RADIUS * PI;
    CIRCUM := 2.0 * RADIUS * PI;
    WRITELN ('RADIUS              ',RADIUS);
    WRITELN ('AREA                ',AREA);
    WRITELN ('CIRCUMFERENCE       ',CIRCUM)
END.
```

The sample test run follows:

```
RADIUS          3.50000E+00
AREA            3.84845E+01
CIRCUMFERENCE   2.19911E+01
```

MIXED MODE EXPRESSIONS

Pascal permits mixed mode expressions for arithmetic operators other than DIV and MOD. A mixed mode expression contains both integer and real operands. For a mixed mode expression the result will be real and the arithmetic will be carried out in real mode. The mixed mode expression

```
16 / 2.0
```

gives the real result 8.0 since the integer 16 is divided by the real number 2.0.

A falling body accelerates at the constant rate of 32.2 feet per second per second. The formula

$v = at$

gives the velocity v after t seconds of a body accelerating at the constant rate a. The formula

$d = .5at^2$

gives the distance d after t seconds.

The following program computes the velocity in feet per second and distance in feet after $t = 5$ seconds with the integer variable TIME containing the time in seconds, the real variable ACCEL containing the 32.2 feet per sec^2, the real variable VELOCITY containing the velocity in feet per second, and the real variable DISTANCE containing the distance in feet:

```
PROGRAM FALL (OUTPUT);
VAR
    TIME : INTEGER;
    ACCEL,
    VELOCITY,
    DISTANCE : REAL;
BEGIN
    TIME := 5;
    ACCEL := 32.2;
    VELOCITY := ACCEL * TIME;
    DISTANCE := 0.5 * ACCEL * TIME * TIME;
    WRITELN ('TIME                ',TIME);
    WRITELN ('ACCELERATION  ',ACCEL);
    WRITELN ('VELOCITY          ',VELOCITY);
    WRITELN ('DISTANCE          ',DISTANCE)
END.
```

The test run output follows:

```
TIME                5
ACCELERATION    3.22000E+01
```

```
VELOCITY        1.61000E+02
DISTANCE        4.02500E+02
```

PRECEDENCE ORDERING

Pascal arithmetic operators follow the normal precedence of elementary algebra. This ordering follows:

Highest		( )	Parentheses
		-	Negate
	DIV MOD	* /	Multiply, divide
Lowest		+ -	Add, subtract

STANDARD FUNCTIONS

The Pascal language contains a number of standard built-in functions applicable to arithmetic operations. An argument enclosed with parentheses is given to the function which returns a value. The following lists some of these functions:

ABS(X)	Absolute value, same type as argument
ROUND(X)	Round real argument giving integer
TRUNC(X)	Truncate real argument giving integer
SQR(X)	Square integer or real argument giving result of same type
SQRT(X)	Square root of integer or real argument giving real result

The length of the longest side (hypotenuse) of a right triangle is equal to the square root of the sum of the squares of the lengths of the other two sides. To travel from one city to another, a driver may travel 30 miles West on one road and 40 miles North on another. Because of the extensive traffic between the two cities, the road commission is recommending a new road be built to connect the two cities by the most direct route. What will the new travel distance be?

The following program computes the shortest distance as the square root of the sum of the squares of the two sides:

```
PROGRAM SHORTEST (OUTPUT);
VAR
    SIDE1,
    SIDE2,
    SUMSQR,
    DISTANCE : REAL;
BEGIN
    SIDE1 := 30.0;
    SIDE2 := 40.0;
```

```
    SUMSQR := SIDE1 * SIDE1 + SIDE2 * SIDE2;
    DISTANCE := SQRT(SUMSQR);
    WRITELN ('FIRST SIDE     ',SIDE1);
    WRITELN ('SECOND SIDE    ',SIDE2);
    WRITELN ('DISTANCE       ',DISTANCE)
END.
```

The output from the test run follows:

```
FIRST SIDE      3.00000E+01
SECOND SIDE     4.00000E+01
DISTANCE        5.00000E+01
```

EXERCISES

1. Compute the area and length of the perimeter of a rectangle having a length of 36 feet and a width of 22 feet.
2. Compute the velocity in miles per hour and distance in feet of a falling body at the end of 8 seconds.
3. Compute gross pay for 33.4 hours of work for a person earning $9.25 per hour.
4. An automobile travels 431 miles on a tank of gas. The cost of filling the tank is $27.87 and the price is $1.279 per gallon. Compute the gas mileage in miles per gallon and the fuel cost in cents per mile.
5. A sales commission consists of a base amount of $450.00 plus a 7 percent commission on sales of $23,400.00. Compute the amount of the commission.

3 Program organization

OVERVIEW Learning the many intricate details of programming requires time commitment. This chapter reinforces and expands the features introduced in the second chapter. This is the second of several spirals through Pascal, each building on the foundation laid by previous spirals.

Take time to run the example programs and experiment with them. Change the programming style. Try different varieties of uppercase, lowercase, and mixed-case variable names. Experiment, observe, compare.

□

3.1 VARIABLE DECLARATIONS

A variable is a symbolic name referencing a computer memory location. The computer programmer chooses the name, but there are some restrictions. The name cannot be one of the common reserved words. The Radio Shack Alcor Pascal allows long variable names but uses only the first eight characters to differentiate between variables.

VARIABLE NAMES

Pascal does not distinguish between uppercase and lowercase within variable names. The names

ENDOFDAY, endofday, and EndOfDay

are equivalent. Some programmers prefer to type all Pascal commands in uppercase letters. Others type key words such as VAR, BEGIN, and END in caps and use lowercase letters for user-supplied names. Still others prefer mixed-case variable names with a capital letter beginning each subpart of the name.

VARIABLE TYPES

The three primitive types of variables introduced at this stage are real, integer, and character. The Pascal programmer defines more complex data structures from

these primitive types. Pascal is very type-conscious. It becomes very "haughty" if a programmer or ultimate user tries to give the variable a value of the wrong type.

FORMAT-CONTROLLED OUTPUT

Each literal and variable carries with it a default format. Pascal uses the default format for WRITE and WRITELN commands unless the programmer defines another format. The command

```
WRITELN ('answer is', AGE, AVERAGE)
```

contains the character string literal 'answer is' and two variables. Let AGE be an integer and AVERAGE be a real number.

The following program includes the literal and variables in the one WRITELN statement:

```
PROGRAM FORMAT1 (OUTPUT);
VAR
    AGE : INTEGER;
    AVERAGE : REAL;
BEGIN
    AGE := 32;
    AVERAGE := 3.75;
    WRITELN ('ANSWER IS', AGE, AVERAGE)
END.
```

The output of the test run follows:

```
ANSWER IS      32 3.750000E+00
```

The default size for string literals and variables is the length of the string. The string literal 'ANSWER IS' uses 9 character positions. Following the string literal with a colon and number overrides the default field size. The extra character positions are filled with spaces.

Alcor Pascal assigns an 8-digit field to each integer literal or variable. This is the default width, but it can be changed by the programmer. Following an integer variable or literal by a colon and a number overrides the default field width. For example, AGE:4 places the value of the integer variable AGE in a field 4 digits wide.

The forms of display for real variables are fixed and exponential format. The default format is exponential with a width of 12 positions. This format is like scientific notation. The value 3.750000E+03 stands for 3.75 times 10 to the 3rd power, or 3,750. This is the default format. Using the format AVERAGE:8 gives a field size of 8 positions and results in the value 3.75E+03.

Fixed format numbers are not in scientific notation and require two specification numbers separated by colons. The format AVERAGE:8:2 defines a field

width of 8 positions and locates the decimal point two places from the right. The result is in the form 3750.00.

The following program illustrates the use of format-controlled output:

```
PROGRAM FORMAT2 (OUTPUT);
VAR
    AGE : INTEGER;
    AVERAGE : REAL;
BEGIN
    AGE := 32;
    AVERAGE := 3.75;
    WRITELN ('ANSWER IS':12, AGE:5; AVERAGE:6:2)
END.
```

The output from the test run follows:

```
ANSWER IS      32      3.75
```

EXERCISES

1. Write a program to compute a person's gross earnings. The pay rate is $9.25 per hour and the hours worked are 37.6. Display the values for pay rate, hours worked, and gross pay in fixed decimal format.
2. Write a program to compute the average of the following quiz scores formatting the average using the fixed decimal form:

   8 6 9 7 8.

3. Write a program to compute the average miles per gallon for an automobile that travels 317 miles on 12.4 gallons of gas. Format the result as fixed decimal with the decimal point three places from the right.

3.2 TYPE DECLARATIONS

The four fundamental Pascal data types are real, integer, character, and Boolean. Real numbers are in scientific notation with mantissa and exponent. Integers include no fractional part. Characters consist of letters of the alphabet and other printable symbols. Boolean values are true or false depending on the result of a test.

USER-DEFINED TYPES

Pascal gives the programmer a powerful tool for defining specialized data types. The keyword TYPE identifies the part of the Pascal program containing the type definitions. The form of the definition is

identifier = type specification;

with a mandatory semicolon. Note that type specifications and constant definitions use the equals sign "=" while the assignment statement uses the symbol ":=".

A common method of defining a data type is specifying a list of possible values. The type specification

```
WHERE = (NORTH, SOUTH, EAST, WEST);
```

defines the type WHERE as the four directions. In the VAR portion of the program the statement

```
DIRECTION : WHERE;
```

defines the variable DIRECTION to be of the type WHERE. The variable DIRECTION can contain one or more of the four directions defined for the type WHERE. An error condition results if the variable DIRECTION does not contain one of the four listed directions as its data type.

One type may consist of elements that are a subset of another type. The type definition

```
DAYS = (SUN, MON, TUE, WED, THU, FRI, SAT);
```

defines the type DAYS as the names of the seven days of the week. The type definition

```
WEEKEND = (SAT, SUN);
```

defines the type WEEKEND as SAT and SUN. The two types DAYS and WEEKEND are inconsistent. The values within the parentheses enumerate the permissible values and specify their logical order (SUN comes before MON, which comes before TUE, etc.). Types defined by enumeration cannot intersect. The days SAT and SUN of the type WEEKEND also appear in the list enumerating the permissible values of DAYS.

It is possible to define a type as a subrange of another, more inclusive, type. The type definition

```
WORKDAYS = MON .. FRI;
```

defines the type WORKDAYS consisting of the days of the week from MON through FRI. The days MON, TUE, WED, THU, and FRI form a subrange of the seven days of the week. The subrange specification MON... FRI gives the beginning and ending values of the subrange. The relative ordering within the subrange is consistent with the enumerated type.

The type definition

```
DAYSOFMONTH = 1 .. 31;
```

defines the type DAYSOFMONTH consisting of the integers 1, 2, ..., 31. Any value outside this range causes an error message. The type definition

```
LETTERS = 'A' ... 'Z';
```

defines the type LETTERS as the letters of the alphabet from the primitive type CHAR.

ORDINALITY

The type definition

DAYS = (SUN, MON, TUE, WED, THU, FRI, SAT);

enumerates seven permissible values. The seven days have the seven ordinal positions 0, 1, . . . , 6. THU is the fifth day of the week and is in ordinal position 4. WED, which is in ordinal position 3, is the predecessor of THU. FRI is the successor of THU. SUN has no predecessor, and SAT has no successor.

The built-in functions ORD, PRED, and SUCC are of special interest for enumerated types. The function ORD(TUE) returns the ordinal position 2, and the function ORD(SUN) returns the value 0. The function PRED(TUE) returns the predecessor of TUE, which is MON. The function SUCC(TUE) returns the value WED. The function PRED(SUN) is undefined as is SUCC(SAT). For the data type CHAR the built-in function CHR returns the character having the given ordinal position.

EXTERNAL REPRESENTATION

User-defined enumeration types have no external representation. Their values cannot be read in from external files nor can they be written out. The type within a Pascal program retains its definition and type checking internally.

VARIABLE DECLARATIONS

Variable declarations may use primitive types or user-defined types. The statement

DAY : DAYS;

defines the variable DAY to be of the type DAYS consisting of the names of the seven days of the week. The statement

WRKDAY : WORKDAYS;

allows the variable WRKDAY to assume one of the values MON, TUE, WED, THU, or FRI if the type definition

WORKDAYS = MON .. FRI;

defined the type WORKDAYS.

Alternately, the variable declaration

WRKDAY : MON .. FRI;

accomplishes the same task. The variable declaration limits the permissible values to a subrange of the type. The variable declaration

```
MONTH : 1 .. 12;
```

limits the value of the variable MONTH to the integers 1 through 12. Thoughtful use of subrange limits for variable declarations results in more reliable programs since Pascal does internal type checking.

Subrange limits may be set at the variable declaration or at the definition of a named type. The named type approach is preferable when several variables have the same subrange limits. When two variables are compared, they must be of the same named type. Passing parameters to Pascal functions and procedures requires the use of named types for user-defined data structures.

EXAMPLE PROGRAM

The following program illustrates the use of type definitions and variable declarations:

```
PROGRAM TYPE1 (OUTPUT);
TYPE
    DAYS = (SUN, MON, TUE, WED, THU, FRI, SAT);
    LETTERS = 'A' .. 'Z';
VAR
    DAY : DAYS;
    WRKDAY : MON .. FRI;
    INITIAL : LETTERS;
    MONTH : 1 .. 12;
BEGIN
    DAY := SUN;
    WRKDAY := THU;
    INITIAL := 'L';
    MONTH := 4;
    WRITELN ('POSITION OF DAY          ',ORD(DAY) );
    WRITELN ('POSITION OF WORKDAY      ',ORD (WRKDAY) );
    WRKDAY := PRED(WRKDAY);
    WRITELN ('POSITION OF PRED         ',ORD(WRKDAY) );
    WRITELN ('CURRENT LETTER           ',INITIAL);
    WRITELN ('PREDECESSOR              ',PRED(INITIAL) );
    WRITELN ('SUCCESSOR                ',SUCC(INITIAL) );
    WRITELN ('MONTH                    ',MONTH)
END.
```

Sample run

```
POSITION OF DAY          0
POSITION OF WORKDAY      4
```

```
POSITION OF PRED        3
CURRENT LETTER    L
PREDECESSOR       K
SUCCESSOR         M
MONTH                   4
```

The program can print the user-defined types based on integer and character types, but it cannot print the user-defined type DAYS based on enumeration because the enumerated type has no external representation. The integer and character types do have external representations.

EXERCISES

1. Write a program defining the days of the week with one subrange giving the normal work days MON, . . . , FRI, and another subrange defining the days SAT and SUN as the weekend. Use MON rather than SUN as the first day of the week.
2. Write a program that defines the type YEAR to be within the range 71, 72, . . . , 99. Determine the ordinal position for the year 84. Determine the predecessor and successor years. Let CURRENT be a variable of type YEAR containing the value 84 representing the current year. Use the variable PREVIOUS of type YEAR to contain the value of the predecessor year. Use the variable NEXTYEAR of type YEAR to contain the value of the successor year. Display the ordinal position and value for 83, 84, and 85.
3. Write a program that displays the ordinal position of each of the characters 'A', 'B', 'C'. Use this information to determine the ordinal positions of 'D', 'E', 'F'. Use the built-in CHR function to display the characters 'D', 'E', and 'F'.

3.3 BEGIN, END BLOCKS

Pascal often uses the key word END as a terminator. The primary use is as the terminator of a BEGIN END block. Every BEGIN must have a corresponding END. Not every END has a BEGIN, however. Anything contained within the BEGIN and its corresponding END is treated as if it were one statement.

The BEGIN END block is vital in the looping of Chapter 4 and the conditional statements of Chapter 5. The purpose is to form a single compound statement from a sequence of simple statements. The entire Pascal program consists of one large statement enclosed in a BEGIN END block with a period terminating the END.

BEGIN END BLOCK

The following program displays Alfred Lord Tennyson's famous lines from "The Charge of the Light Brigade":

```
PROGRAM CHARGE1 (OUTPUT);
BEGIN
    WRITE ('Half a league,');
    WRITELN (' half a league,');
    WRITELN ('Half a league onward.');
    WRITELN ('All in the Valley of Death');
    WRITELN ('Rode the six hundred.')
END.
```

The program consists of the single compound statement composed of the five simple statements. The first two statements display one line. The first statement uses the WRITE command to display the first part, and the second statement uses the WRITELN command to finish the line.

The following program places the first two statements in a separate BEGIN END block nested within the outer BEGIN END block defining the whole program:

```
PROGRAM CHARGE2 (OUTPUT);
BEGIN
    BEGIN
        WRITE ('Half a league,');
        WRITELN (' half a league,')
    END;
    WRITELN ('Half a league onward');
    WRITELN ('All in the Valley of Death');
    WRITELN ('Rode the six hundred.')
END.
```

The display from the test run follows

```
Half a league, half a league,
Half a league onward.
All in the Valley of Death
Rode the six hundred.
```

DISCIPLINED PROGRAMMING

Large programs are divided into tasks which are divided into subtasks. Subtasks may be divided into subsubtasks, etc. The result is an hierarchical tree of modules with the main program as its root. The nested hierarchy of BEGIN END blocks shows this structure.

COMMENTS

A comment is an explanation about the program that is placed in the program listing for people to read. The braces "{" and "}" enclose the comments which

are ignored by the computer. Comments can appear anywhere within the program. They may give the purpose of a command, give additional information about a variable, or describe the purpose of a BEGIN END block.

Many keyboards, including those for the Radio Shack TRS-80, do not have the left and right braces. The alternate symbol pair "(*" serves as the left-hand delimiter and the symbol pair "*)" serves as the righthand delimiter. Most Pascal compilers recognize both sets of delimiters for comments. The Alcor Pascal editor accepts the special key sequence Clear 4 for the left brace "{" and the sequence Clear 5 for the right brace "}". Hold the Clear key down while typing the appropriate digit.

COMPUTE AREA AND CIRCUMFERENCE

The following program computing the area and circumference of a rectangle illustrates the use of nested BEGIN END blocks and comments:

```
PROGRAM COMMENT1 (OUTPUT);
VAR
    LENGTH,          {LENGTH OF RECTANGLE}
    WIDTH,           {WIDTH OF RECTANGLE}
    AREA,            {AREA OF RECTANGLE}
    CIRCUM           {CIRCUMFERENCE}
        : REAL;
BEGIN
    { Compute the area and circumference
      for a rectangular garden plot. }
    BEGIN
        { Get dimensions of rectangle }
        LENGTH := 23.5;
        WIDTH := 14.0
    END;
    BEGIN
        { Compute area and circumference }
        AREA := LENGTH * WIDTH;
        CIRCUM := 2.0 * (LENGTH + WIDTH)
    END;
    BEGIN
        { Display results }
        WRITELN ('LENGTH              ',LENGTH:9:2);
        WRITELN ('WIDTH               ',WIDTH:9:2);
        WRITELN ('AREA                ',AREA:9:2);
        WRITELN ('CIRCUMFERENCE       ',CIRCUM:9:2)
    END;
END.
```

The output from the test run follows:

```
LENGTH          23.50
WIDTH           14.00
AREA           329.00
CIRCUMFERENCE   75.00
```

VELOCITY AND DISTANCE

The following program computes the velocity in feet per second and distance in feet of an object undergoing constant acceleration:

```
PROGRAM COMMENT2 (OUTPUT);
VAR
    TIME,           (* Time in seconds *)
    ACCEL,          (* Acceleration in feet/sec/sec/ *)
    SPEED,          (* Velocity in feet/sec *)
    DISTANCE        (* Distance in feet *)
        : REAL;
BEGIN
    (* Compute the velocity in feet per second
       and the distance in feet for an object
       accelerating at a constant rate in feet
       per second per second for a given time
       in seconds *)
    BEGIN
       (* Get problem parameters *)
       TIME := 5.0;
       ACCEL := 32.2
    END;
    BEGIN
       (* Compute velocity and distance *)
       SPEED := ACCEL * TIME;
       DISTANCE := 0.5 * ACCEL * TIME * TIME
    END;
    BEGIN
       (* Display results *)
       WRITELN ('TIME                ',TIME:9:1);
       WRITELN ('ACCELERATION        ',ACCEL:9:1);
       WRITELN ('VELOCITY            ',SPEED:9:1);
       WRITELN ('DISTANCE            ',DISTANCE:9:1)
    END;
END.
```

The output from the test run follows:

```
TIME            5.0
ACCELERATION   32.2
```

```
VELOCITY        161.0
DISTANCE        402.5
```

This program uses the alternate symbols "(*" and "*)" to deliniate comments.

EXERCISES

1. Write a program to compute the area and circumference of a circle having a radius of 4.0. Use comments and nested BEGIN END blocks.
2. Write a program to compute a 5 percent commission for a $23,500 sale. Use comments and nested BEGIN END blocks.
3. Write a program to compute the average miles per gallon and the average fuel cost per mile for an automobile that travels 374 miles on fuel costing $21.96 at $1.179 per gallon. Use comments and nested BEGIN END blocks.

3.4 STATEMENTS AND SEMICOLONS

The semicolon follows the program header and is used as a statement separator. It also follows each constant definition, type definition, and variable type declaration. Commas separate the variables declared with the same type declaration statement.

EMPTY OR NULL STATEMENT

A semicolon is not necessary before the END of a compound statement. The BEGIN and END words act as statement parentheses. They are delimiters of the compound statement. A semicolon immediately preceding an END statement results in a null statement between the semicolon and the END. This usually does no harm, but it is unnecessary.

OUTPUT FILE

The display screen is the normal output device for most microcomputers and computer terminals. All Pascal programs have use of the default output file predeclared by the Alcor Pascal compiler. At run time the user may reassign the default file OUTPUT to the printer by responding with the letter L rather than the Enter key to the OUTPUT file request.

The following program displays a well-known remark made by Benjamin Franklin at the signing of the Declaration of Independence:

```
PROGRAM SIGN1 (OUTPUT);
BEGIN
    WRITELN ('Yes, we must, indeed,');
    WRITELN ('all hang together or,');
```

```
    WRITELN ('most assuredly,');
    WRITELN ('we shall all hang separately.')
END.
```

First, run the program typing the Enter key for the request for INPUT and OUTPUT file assignments. The display is as follows

```
Yes, we must, indeed,
all hang together or,
most assuredly,
we shall all hang separately.
```

Now run the program typing the Enter key for the INPUT file assignment and the response :L (line printer) for the OUTPUT file assignment. The output should go to the printer. It is not necessary to recompile the program to reassign the INPUT and OUTPUT files.

TYPES OF FILES

Files may be TEXT or nontext files. TEXT files contain variable values of type REAL, INTEGER, or CHAR. The types can be mixed with REAL, INTEGER, and CHAR values in the same file. The values read from the file must agree in type with the variables into which they are placed. TEXT files contain an external representation of the values of the type variables.

Nontext files contain the contents of the typed variables in internal form. In the case of INTEGER and REAL variables, the cost of converting external into internal forms and vice versa can be very high. Files that are created for the sole purpose of being read in by another program can be in internal form. Files to be sent to the video display or printer should be TEXT files. Data entered from the keyboard is in TEXT form.

INPUT FILE

The keyboard is the default INPUT file. Alcor Pascal automatically declares the TEXT file INPUT for use in the program. TEXT file processing is on a line-by-line basis. A file pointer locates the part of the file to be used by the next READ and READLN statement. The READ command brings the next data item from the file and advances the pointer to the place immediately following the item brought in. The READLN command fetches the next data item and advances the pointer to the beginning of the next line.

Alcor Pascal automatically declares the default INPUT file and positions it properly for use. Other external files require additional commands for their use. Each line of a TEXT file is terminated with a carriage return character generated by the Enter key. Use the READLN command when requesting values one-at-a-time from the keyboard.

OBTAINING DATA FROM THE KEYBOARD

The following program computing the area and circumference of a rectangle uses the READLN command to obtain data from the keyboard:

```
PROGRAM KEYIN1 (INPUT, OUTPUT);
VAR
    LENGTH,
    WIDTH,
    AREA,
    CIRCUM : INTEGER;
BEGIN
    READLN (LENGTH, WIDTH);
    AREA := LENGTH * WIDTH;
    CIRCUM := 2 * (LENGTH + WIDTH);
    WRITELN ('AREA                ',AREA);
    WRITELN ('CIRCUMFERENCE ',CIRCUM)
END.
```

The test run including data entry from keyboard follows:

```
40 30
AREA                1200
CIRCUMFERENCE       140
```

PROMPT MESSAGES

When obtaining data from the keyboard interactively, a good practice is to display a short message describing what is wanted before each READLN command. Using the WRITE command for the prompt message results in the user's response appearing on the same line as the message on the screen.

The following program computing the area and circumference of a rectangle uses the WRITE command to display the prompting message and the READLN command to obtain the data value:

```
PROGRAM KEYIN2 (INPUT, OUTPUT);
VAR
    LENGTH,
    WIDTH,
    AREA,
    CIRCUM : INTEGER;
BEGIN
    WRITE ('LENGTH? ');
    READLN (LENGTH);
    WRITE ('WIDTH ? ');
    READLN (WIDTH);
    AREA := LENGTH * WIDTH;
```

```
    CIRCUM := 2 * (LENGTH + WIDTH);
    WRITELN ('AREA                ',AREA);
    WRITELN ('CIRCUMFERENCE   ',CIRCUM)
END.
```

The test run with sample dialog follows:

```
LENGTH ? 40
WIDTH   ? 30
AREA                    1200
CIRCUMFERENCE            140
```

COMMENTS AND NESTED BEGIN END BLOCKS

The following program adds comments and nested BEGIN END blocks to improve program readability:

```
PROGRAM KEYIN3 (INPUT, OUTPUT);
VAR
    LENGTH,          { Length of rectangle }
    WIDTH,           { Width of rectangle }
    AREA,            { Area of rectangle }
    CIRCUM           { Circumference }
        : INTEGER;
BEGIN
    { Compute the area and circumference
      of a rectangular garden plot. }
    BEGIN
        { Get garden plot dimensions }
        WRITE ('LENGTH OF PLOT ?');
        READLN (LENGTH);
        WRITE ('WIDTH OF PLOT   ?');
        READLN (WIDTH)
    END;
    BEGIN
        { Compute area and circumference }
        AREA := LENGTH * WIDTH;
        CIRCUM := 2 * (LENGTH + WIDTH)
    END;
    BEGIN
        { Display results }
        WRITELN ('AREA                ',AREA);
        WRITELN ('CIRCUMFERENCE   ',CIRCUM)
    END
END.
```

The sample output from this program follows

```
LENGTH OF PLOT ? 40
WIDTH OF PLOT  ? 30
AREA                    1200
CIRCUMFERENCE           140
```

EXERCISES

1. Electricity costs 6.75 cents per kilowatt hour. A surcharge of 10 percent is added to the bill. Write a program using interactive data entry to compute the electric bill for a home that uses 2,340 kilowatt hours of electricity.
2. Write an interactive program that asks for the Fahrenheit temperature and then computes the corresponding Celsius temperature reading.
3. Write an interactive program to compute the dollar amount of an invoice if 12 items are ordered at $96.00 each.

3.5 PROGRAMMING STYLE CONSIDERATIONS

Style is an extremely important tool for readability. Cosmetics have no bearing on the operation of the program, but the appearance of the program listing is important for readability. Programmers should make every effort to make their programs easy to read. There are many ways of writing the same program.

UNREADABLE PROGRAM

Consider the following program that jams as much as possible on one line and includes no comments:

```
PROGRAM STYLE1 (OUTPUT);VAR LENGTH,WIDTH,AREA,
CIRCUM:INTEGER;BEGIN LENGTH:=40;WIDTH:=30;
AREA:=LENGTH*WIDTH;CIRCUM:=2*(LENGTH+WIDTH);
WRITELN('LENGTH',LENGTH);WRITELN('WIDTH',WIDTH);
WRITELN('AREA',AREA);WRITELN('CIRCUMFERENCE',
CIRCUM) END.
```

The following program is equivalent and much easier to read:

```
PROGRAM STYLE2 (INPUT, OUTPUT);
VAR
LENGTH,
WIDTH,
AREA,
CIRCUM : INTEGER;
BEGIN
LENGTH := 40;
WIDTH := 30;
```

```
AREA := LENGTH * WIDTH;
CIRCUM := 2 * (LENGTH + WIDTH);
WRITELN ('LENGTH              ',LENGTH);
WRITELN ('WIDTH               ',WIDTH);
WRITELN ('AREA                ',AREA);
WRITELN ('CIRCUMFERENCE       ',CIRCUM)
END.
```

Indenting subordinate parts of the program adds even more to readability as the following program shows:

```
PROGRAM STYLE3 (OUTPUT);
VAR
    LENGTH,
    WIDTH,
    AREA,
    CIRCUM : INTEGER;
BEGIN
    LENGTH := 40;
    WIDTH := 30;
    AREA := LENGTH * WIDTH;
    CIRCUM := 2 * (LENGTH + WIDTH);
    WRITELN ('LENGTH              ',LENGTH);
    WRITELN ('WIDTH               ',WIDTH);
    WRITELN ('AREA                ',AREA);
    WRITELN ('CIRCUMFERENCE       ',CIRCUM)
END.
```

Dividing the program into blocks and adding comments is helpful for larger programs:

```
PROGRAM STYLE4 (OUTPUT);
VAR
    LENGTH,          { Length of rectangle }
    WIDTH,           { Width of rectangle }
    AREA,            { Area of rectangle }
    CIRCUM           { Circumference }
        INTEGER;
BEGIN
    { Compute the area and circumference
      for a rectangle. }
    BEGIN
        { Get dimensions of rectangle }
        LENGTH := 40;
        WIDTH  := 30
    END;
    BEGIN
```

```
        { Compute area and circumference }
        AREA    := LENGTH * WIDTH;
        CIRCUM := 2 * (LENGTH + WIDTH)
    END;
    BEGIN
        { Display results }
        WRITELN ('LENGTH                ',LENGTH);
        WRITELN ('WIDTH                 ',WIDTH);
        WRITELN ('AREA                  ',AREA);
        WRITELN ('CIRCUMFERENCE         ',CIRCUM)
    END
END.
```

UPPERCASE AND LOWERCASE LETTERS

Many Pascal programmers adopt a style using uppercase letters for Pascal key words and lowercase letters for programmer-supplied terms. The following program uses this approach:

```
PROGRAM style5 (OUTPUT);
VAR
    length,             { Length of rectangle }
    width,              { Width of rectangle }
    area,               { Area of rectangle }
    circum              { Circumference }
      : INTEGER;
BEGIN
    { Compute the area and circumference }
      for a rectangle
    BEGIN
        { Get dimensions of rectangle }
        length := 40;
        width  := 30
    END;
    BEGIN
        { Compute area and circumference }
        area    := length * width;
        circum := 2 * (length + width)
    END;
    BEGIN
        { Display results }
        WRITELN ('Length                ',length);
        WRITELN ('Width                 ',width);
        WRITELN ('Area                  ',area);
        WRITELN ('Circumference         ',circum)
    END
END.
```

CHOICE OF STYLE

The programmer chooses a style that is appropriate for the task at hand. If the program is to be placed in a program library for use by others, it should be more carefully written with the intended users in mind. It should also be written with other programmers in mind. This is the reason for following programming conventions, including style. Each organization will adopt a standard style and insist that programmers use that style for programs to be placed in the library.

EXERCISES

1. Write a program to compute the volume and surface area of a sphere having a diameter of 6 feet. Use comments and nested BEGIN END blocks.
2. Write a program to compute the gross pay for a person earning $9.75 per hour for 36.5 hours. Use comments and nested BEGIN END blocks.
3. Write a program using interactive data entry to compute the surface area, volume, and net weight of a cylinder 12 feet long by 3 feet in diameter. Assume that the cylinder is filled with water. Use comments and nested BEGIN END blocks.

4 Repetition, looping

OVERVIEW Many activities are repetitive. Accumulating the sum of a set of values requires repeated addition. Computing a compound interest table requires repetitious calculation and printing for each line of the table. Searching for the first occurrence of a value requires repeating the test.

A program loop consists of a sequence of one or more statements to be executed repeatedly. All programming languages contain control statements for altering the normal sequential flow of execution within the program. Pascal provides three general looping control structures: WHILE, REPEAT, and FOR.

□

4.1 FOR DO

The Pascal FOR DO statement provides looping control by using a control variable with an automatic updating feature. The information required by the FOR statement includes the name of the control variable, the initial and final values for the control variable, and the statement representing the body of the loop.

The control variable starting and ending expressions must be ordinal, that is, integer, character, or enumerative. The control variable cannot be real. The control variable is incremented or decremented by one ordinal position each time through the loop.

INCREMENTING AND DECREMENTING

The statement

```
FOR I := 1 TO 30 DO WRITE (I);
```

repeats the WRITE statement 30 times. The variable I assumes the value 1 the first time, 2 the second time, etc. The statement

```
FOR I := 30 DOWNTO 1 DO WRITE (I);
```

also repeats the WRITE statement 30 times but with the values 30, 29, . . . , 1 assigned in turn to the variable I.

The keyword TO implies incrementing by one position each time with the ending value normally following the starting value. The keyword DOWNTO implies decrementing by one position with the final value normally preceding the starting value.

The control variable does not have to be an integer. The statement

```
FOR ICHAR := 'B' TO 'K' DO WRITE (ICHAR);
```

repeats the WRITE statement 10 times assigning the character values in the order B, C, . . . , K. The statement

```
FOR ICHAR := 'K' DOWNTO 'B' DO WRITE (ICHAR);
```

assigns the letters in reverse order K, J, . . . , B.

COUNT THE WAYS

One of Elizabeth Barrett Browning's famous verses begins

> How do I love thee? Let me count the ways.
> I love thee to the depth and breadth and height
> My soul can reach, when feeling out of sight.

The following Pascal program makes up in output volume what it lacks in linguistic elegance and taste:

```
PROGRAM LOVE (OUTPUT);
VAR
    COUNT : INTEGER;
BEGIN
    WRITELN ('How do I love thee?');
    WRITELN ('Let me count the ways.');
    FOR COUNT := 1 TO 5 DO
        WRITELN (COUNT)
END.
```

The output of the program follows:

```
How do I love thee?
Let me count the ways.
   1
   2
   3
   4
   5
```

If this program does not count high enough to suit the occasion, a simple change to the final value for the control variable will up the count.

TABLE GENERATION

Charles Babbage, in the early 1800s, conceived the idea of an automatic machine he called the analytical engine which could both calculate and print mathematical tables for navigation and other purposes. His vision was unattainable until the first working computers of the 1940s. What was one of their main applications? Calculating and printing mathematical tables.

The following Pascal program illustrates this application by generating a table giving the velocity in feet per second and distance in feet travelled by an object accelerating at the constant rate of 8 feet per second per second:

```
PROGRAM body (OUTPUT);
VAR
    accel,
    time,
    final,
    velocity,
    distance : INTEGER,
BEGIN
    accel := 8;
    final  := 5;
    WRITELN ('Time      Velocity      Distance');
    FOR time := 1 TO final DO
        BEGIN
            velocity  := accel * time;
            distance := accel * time * time DIV 2;
            WRITELN (time:5, velocity:13, distance:10)
        END
END.
```

The output follows:

Time	*Velocity*	*Distance*
1	8	4
2	16	16
3	24	36
4	32	64
5	40	100

The Pascal FOR statement generates integers in steps of 1 only. It cannot directly generate real values, although the programmer may use the counter variable in expressions generating real values. This program illustrates one of the main uses of the BEGIN END block to form a compound statement under the control of the FOR statement.

SQUARE ROOT TABLE

The following program uses the counter variable to generate the real values 1.0, 1.1, . . . , 2.0 for generating the square root tables of those values:

```
PROGRAM sqroot (OUTPUT);
VAR
    counter : INTEGER;
    value,
    root : REAL;
BEGIN
    WRITELN ('Value      Square root');
    FOR counter := 1 TO 11 DO
        BEGIN
            value := 0.9 + counter / 10.0;
            root := SQRT(value);
            WRITELN (value:5:1, root:10:4)
        END
END.
```

The output follows

Value	*Square root*
1.0	1.0000
1.1	1.0488
1.2	1.0954
1.3	1.1412
1.4	1.1832
1.5	1.2247
1.6	1.2649
1.7	1.3038
1.8	1.3416
1.9	1.3748
2.0	1.4142

The integer control variable may appear in mixed expressions with real variables to generate many possible values.

FOR LOOPS WITH CHARACTER CONTROL VARIABLE

The following program illustrates the use of the FOR looping construct with a nonnumeric control variable:

```
PROGRAM char1 (OUTPUT);
VAR
    letter : CHAR;
BEGIN
    FOR letter := 'B' TO 'E' DO
```

```
        WRITELN (letter)
END.
```

The output follows:

```
B
C
D
E
```

The following program uses the DOWNTO version to decrement the control variable:

```
PROGRAM char2 (OUTPUT);
VAR
    letter : CHAR;
BEGIN
    FOR letter := 'K' DOWNTO 'E' DO
        WRITELN (letter)
END.
```

The output follows:

```
K
J
I
H
G
F
E
```

FOR LOOP WITH ENUMERATED TYPES

Integer, character, and enumerated types are ordinal and can be used for control variables of FOR loops. Enumerated types cannot be used directly for input and output. The following program defines the enumerated type DAYS consisting of the names of the days of the week. It uses the FOR loop with the variable DAY and this enumerated type to generate output values:

```
PROGRAM daylist (OUTPUT);
TYPE
    days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
VAR
    day : days;
    value : integer;
BEGIN
    value := 0;
    FOR day := Sun TO Sat DO
```

```
        BEGIN
            value := value + 1;
            WRITELN (value)
        END
END.
```

The output follows:

```
1
2
3
4
5
6
7
```

INTERACTIVE DATA ENTRY

One use of the FOR loop is as a counter controlling the repetition of the loop. This is useful for repeated data entry and calculation. The following program uses interactive data entry. The program asks for the number of values, and then asks for the values one by one, summing them into an accumulator.

```
PROGRAM sum1 (INPUT, OUTPUT);
VAR
    number,
    counter : INTEGER;
    value,
    sum : REAL;
BEGIN
    WRITELN ('Form the sum of');
    WRITELN ('a set of values.');
    WRITELN (' ');
    WRITE ('Number of values? ');
    READLN (number);
    sum := 0.0;
    WRITELN ('Value');
    FOR counter := 1 TO number DO
        BEGIN
            WRITE ('No.', counter:2,' ? ');
            READLN (value);
            sum := sum + value
        END;
    WRITELN (' ');
    WRITELN ('Sum', sum:10:5)
END.
```

The output of a sample run follows:

```
Form the sum of
a set of values.

Number of values ? 3
Value
No. 1 ? 3.5
No. 2 ? 2.1
No. 3 ? 3.2

Sum     8.80000
```

EXERCISES

1. Write a program that will print all the letters of the alphabet from A to Z on the same line.
2. Write a program that will print all the letters of the alphabet in reverse order from Z to A on the same line.
3. A Fibonacci sequence of numbers begins with the numbers 1 and 2 and forms each successive number as the sum of the previous two numbers. The first few values of the sequence are 1, 2, 3, 5, 8, 11, 19. Write a program that generates the first 15 terms of the Fibonacci sequence.
4. Write a program to compute the average of the following test scores:

   83 79 91 84 85

   Use interactive data entry, asking for the number of test scores and using that number with a FOR loop to control the reading and summing of the test scores.

4.2 WHILE

The three loop control structures in Pascal are the FOR DO loop, the WHILE loop, and the REPEAT UNTIL loop. The FOR DO loop increases or decreases a loop control variable a set number of times. It has major applications in the generation of data values for use within the loop.

The WHILE and the REPEAT UNTIL loops are generalized looping structures. Both structures employ a test using a variable or expression. It is the responsibility of the programmer to initiate and update the variables in the expression used in the loop control test.

The WHILE statement performs its test at the beginning of the loop. The REPEAT UNTIL statement does its test at the end of the loop. The computer executes the body of the WHILE loop if the condition test is true. The computer will loop back and repeat the body of the REPEAT UNTIL loop until the test is satisfied.

RELATIONAL OPERATORS

Relational operators result in outcomes that are either "true" or "false". The common relational operators follow:

=	True if left operand equals right
<	True if left less than right
<=	True if left less than or equal to right
>=	True if left greater than or equal to right
>	True if left greater than right
<>	True if left not equal to right

EXAMPLE RELATIONAL EXPRESSIONS

The relational operator compares the two operands located on each side of the operator symbol. The expression

```
COUNT <= 25
```

is true if the contents of the variable COUNT is less than or equal to 25. The expression is false if the contents is greater than 25.

The expression

```
AMOUNT <> FLAG1
```

is true if the contents of the variable AMOUNT differs from that of the variable FLAG. The expression is false if the contents of the two variables are equal.

The expression

```
(COUNT DIV 2) = (AMOUNT + DIFF)
```

is true if the expression on the left is equal to the expression on the right. The expression is false if they are not equal.

COMPOUND LOGICAL EXPRESSIONS

Compound Boolean expressions are formed from simple expressions using combinations of "and" and "or" operators. The compound expression

```
(COUNT <= NUMBER) AND (AMOUNT > 0)
```

is true if the contents of COUNT is less than or equal to the contents of NUMBER and the contents of AMOUNT is greater than 0. If the contents of COUNT is greater than the contents of NUMBER or if the contents of AMOUNT is less than or equal to 0, the value of the compound logical expression is false.

The compound Boolean expression

(BIG > VERYBIG) OR (AMOUNT = 0)

is true if the contents of BIG is greater than the contents of VERYBIG or the contents of AMOUNT equals 0. The expression is false if the contents of BIG is less than or equal to the contents of VERYBIG and the contents of AMOUNT is not equal to 0.

A compound expression formed with the OR operator is true if at least one of the two logical expressions is true. The compound expression is false only if both are false. A compound expression formed with the AND expression is true only if both logical expressions are true. It is false if at least one of the two is false.

NEGATION

The operator symbol NOT operates only on the logical expression to the right. The expression

NOT AMOUNT <= 0

is true if the contents of the variable AMOUNT is not less than or equal to 0. Too much use of the NOT operator results in unreadable programs. The expression

AMOUNT > 0

is equivalent and easier to understand.

SQUARE ROOT TABLE

The following program uses the WHILE construct to control the looping while generating a table of square roots of the values 1.0, 1.1, . . . , 1.5:

```
PROGRAM sqroot (OUTPUT);
VAR
    value,
    root : REAL;
BEGIN
    value := 1.0;
    WHILE value <= 1.5 DO
        BEGIN
            root := SQRT(value);
            WRITELN (value:5:1, root:10:6);
            value := value + 0.1
        END
END.
```

The sample output follows:

```
1.0   1.000000
1.1   1.048810
1.2   1.095445
1.3   1.140150
1.4   1.183220
```

The program looks like it should have included one more line because of the statement

```
WHILE value <= 1.5 DO
```

which should have executed the body of the loop for the value 1.5. The problem results from the fact that computers that use binary internal numbers cannot represent the decimal fraction .1 exactly. The roundoff error in repeating .1 resulted in a value slightly greater than 1.50000 which triggered the premature exit from the WHILE loop.

The program initializes VALUE before executing the WHILE statement. Each time through the WHILE loop the program computes the square root, prints the line of the table, and adds the value .1 to VALUE to update it for the next time through the loop. The internal representations of the fractional numbers are only approximations to the corresponding decimal fractions.

COMPUTE FACTORIAL

For integer values of ***n*** greater than 0, the mathematical expression ***n***!, called ***n*** factorial, has a value equal to the product of the first ***n*** integers. The value for 2! is 2 since 1 times 2 gives the value 2. The value for 3! is 1 x 2 x 3 = 6. The value for 5! is 1 x 2 x 3 x 4 x 5 = 120.

PROGRAM USING FOR LOOP

The following program uses the FOR TO looping structure to generate the integers needed for generating the product:

```
PROGRAM fact1 (OUTPUT);
VAR
    count,
    number,
    fact : INTEGER;
BEGIN
    number := 5;
    fact := 1;
    FOR count := 1 TO number DO
        fact := fact * count;
    WRITELN (number,' factorial = ',fact)
END.
```

Sample output follows:

```
5 factorial =   120
```

By initializing the variable fact to the value 1, the statement

```
fact := fact * count;
```

in the FOR DO loop forms the product of the values 1, 2, . . . , number.

PROGRAM USING WHILE LOOP

The following program uses the WHILE loop with the programmer controlling the updating of the variable count:

```
PROGRAM fact2 (OUTPUT);
VAR
    count,
    number,
    fact : INTEGER;
BEGIN
    number := 5;
    fact := 1;
    count := 1;
    WHILE count <= number DO
        BEGIN
            fact := fact * count;
            count := count + 1
        END;
    WRITELN (number:2 ' factorial = ',fact)
END.
```

The sample output follows:

```
5 factorial =   120
```

Although the WHILE construct accomplishes the same thing as the FOR statement, it is not as easy to understand. The single FOR statement initializes the control variable, increments it, and makes the necessary test. However, the WHILE construct is more flexible since it is not limited to ordinal variables.

USING TRAILER VALUE TO TERMINATE DATA ENTRY

One method of terminating data entry is to use a trailer value and test for that value every time through the loop. The WHILE statement requires that the variable used by the test have a value in it. This results from the nature of the WHILE statement with its test at the beginning of the loop. The following program reads in a set of real values forming their sum and using the value -999.999 as a trailer value signaling the end of the data:

```
PROGRAM sum (INPUT, OUTPUT);
VAR
    value,
    sum : REAL;
BEGIN
    WRITELN ('Form sum of a set of values');
    WRITELN ('using -999.999 to terminate');
    WRITELN ('data entry.');
    sum := 0.0;
    WRITE ('value ? ');
    READLN (value);
    WHILE value <> -999.999 DO
        BEGIN
            sum := sum + value;
            WRITE ('value ? ');
            READLN (value)
        END;
  WRITELN ('sum = ',sum:10:4)
END.
```

Sample run

```
Form sum of a set of values
using -999.999 to terminate
data entry.
value ? 2.4
value ? 3.8
value ? 1.2
value ? -999.999
sum =    7.4000
```

EXERCISES

1. Write a program to compute the sum of the first 15 integers. Use a WHILE loop to generate the integers.
2. Write a program to compute the sum of the squares of the first 15 integers. Use a WHILE loop to generate the integers.
3. Write a program to compute and print the square root table for the values 10.0, 10.2, . . . , 14.0.
4. Write a program using real variables to compute 30! (30 factorial). Use a WHILE loop to generate the integers.
5. Write a program to compute the average of the following test scores:

   88 73 77 81 92

   Use -1.0 as a trailer value to terminate the data entry loop.

4.3 REPEAT UNTIL

The REPEAT UNTIL looping construct performs its test at the end of the loop and always executes at least once. The body of the WHILE loop does not execute at all if the condition fails on the first try because the test comes at the beginning of the loop. The REPEAT UNTIL statements provide an exit controlled loop.

BODY OF THE LOOP

Both the FOR loop and the WHILE loop control the following Pascal statement only. Compound statements formed with BEGIN END blocks are required if the body of the loop contains more than one statement. The REPEAT statement defines the beginning of the REPEAT UNTIL loop. The UNTIL statement with the relational expression defines the end of the loop. These two Pascal statements define the scope of the loop. There is no need for a BEGIN END block to form a compound statement from several elementary statements.

TEST AT END OF LOOP

The UNTIL statement at the end of the loop includes a relational expression. If the expression is false, the loop is repeated from the REPEAT statement. If the expression is true, control falls to the statement immediately following the UNTIL statement.

The WHILE statement executes the loop if the relational expression is true. The REPEAT UNTIL construct repeats the loop if the expression is false. The WHILE statement performs the test at the beginning of the loop. The REPEAT UNTIL construct performs the test at the end of the loop.

SQUARE ROOT TABLE

The following program uses the REPEAT UNTIL construct in the generation of a square root table for the values 1.0, . . . , 1.5:

```
PROGRAM sqroot (OUTPUT);
VAR
    value,
    root : REAL;
BEGIN
    value := 1.0;
    REPEAT
        root := SQRT(value);
        WRITELN (value:5:1, root:10:6);
        value := value + 0.1
    UNTIL value > 1.501
END.
```

The sample output follows:

```
1.0   1.000000
1.1   1.048810
1.2   1.095450
1.3   1.140180
1.4   1.183220
1.5   1.224750
```

The upper limit of 1.501 allows for roundoff error accumulation of the repeated additions of .1 to the loop control variable. The reason for doing this is that the computer cannot represent the decimal fraction .1 exactly in internal binary form.

INFINITE LOOPS

It is easy to write a WHILE loop or a REPEAT UNTIL loop that never terminates. An infinite loop is a nonterminating loop. To create a loop that will terminate, one of the statements within the loop must modify the loop control variable. The following program does not contain the statement:

```
value := value + 0.1
```

from the previous program:

```
PROGRAM infinite (OUTPUT);
VAR
    value,
    root : REAL;
BEGIN
    value := 1.0;
    REPEAT
        root := SQRT(value);
        WRITELN (value:5:1, root:10:6)
    UNTIL value >= 1.5
END.
```

The beginning few lines of output follow:

```
1.0   1.000000
1.0   1.000000
1.0   1.000000
1.0   1.000000
```

This program is in an infinite loop. This is apparent as it runs. The output does not terminate after six lines as expected. The program does not compute the square roots of the values 1.0, 1.1, 1.2, ... as desired. Press the Break key to terminate the execution of a Pascal program.

STOPPING A RUNAWAY PROGRAM

Normal output from a Pascal program goes to the video screen. Information quickly scrolls off the screen with voluminous output. Pressing the @ key during program execution causes the Pascal program or Pascal compiler to stop execution temporarily. Pressing the Enter key causes the computer to resume where it left off.

Holding the Break key down for a few moments causes the program to stop execution and control passes to the operating system. This is the mode used to stop a program that is in an infinite loop.

If the Break key does not stop the program, pressing the Reset key should reload the operating system. This is a more drastic action and may destroy information on the diskette if done during disk input or output operations. If the Reset key does not stop the stampeding program, turning off the computer is always the last resort. Again, if done during input and output operations with disk, this may damage the information stored on the diskettes.

Try the Break key before resorting to the Reset key. Try the Reset key before turning off the power. If the disk drive lights are off, pressing the Reset key should not damage the data on the diskettes. The Radio Shack manual suggests removing the diskettes from the drives before turning the power off.

POWER FAILURES AND DISKETTE BACKUP POLICIES

The greatest fear of the computer user is not a runaway program although the results can be startling when unexpected. Power failures are the bane of the computerist's existence. Usually, the loss resulting from a power failure is limited only to that work done since the last write operation to the diskette. If the power failure comes during a disk access, the resulting damage to the diskette directory may make that diskette unusable. Do not press the reset key or turn off the power while the disk drive lights are on.

The programs written while first learning Pascal may not have lasting value. They are tools for the learning process. As programming skills increase, so does the value of the programs. The programmer is soon embarking on an adventuresome journey, creating a personal library of Pascal programs that have lasting utility. Backup copies of these diskettes are cheap insurance.

STOPPING DATA ENTRY WITH TRAILER VALUE

One way to make a general-purpose program is to enable it to process a variable amount of data. The processing takes place in a program loop, with one item processed each time through the loop. To stop the program, use a trailer value having an invalid value. The program continues processing the data until it picks up the trailer value.

The following program computes the sum of a set of values obtained from the keyboard with the trailer value -999 indicating the end of the data:

```
PROGRAM sum (INPUT, OUTPUT);
VAR
    value,
    sum : INTEGER;
BEGIN
    sum := 0;
    WRITE ('value ? ');
    READLN (value);
    REPEAT
        sum := sum + value;
        WRITE ('value ? ');
        READLN (value)
    UNTIL value = -999;
    WRITELN ('Sum = ', sum)
END.
```

The sample output follows:

```
value ? 12
value ? 15
value ? 11
value ? -999
Sum =        38
```

The program includes an initial READLN command to establish the initial value for the input variable. This is accumulated into the variable sum. The program reads the next value just before the test at the end of the loop.

EXERCISES

1. Write a program to compute the sum of the first 15 integers. Use a REPEAT UNTIL loop to generate the integers.
2. Write a program to compute the sum of the squares of the first 15 integers. Use a REPEAT UNTIL loop to generate the integers.
3. Write a program to compute and print the square root table for the values 10.0, 10.2, . . . , 14.0. Use a REPEAT UNTIL loop to generate the integers.
4. Write a program to compute the average of the following test scores: 88, 73, 77, 81, 92. Use a REPEAT UNTIL loop and a trailer value of -1.0 to terminate data entry.
5. Write a program to compute the sum of the terms

   $x^n/n!$

   for the value $x = 2$ and $n = 0, 1, 2, \ldots$ until the value of the last term is less than .00001.

5 Conditional execution

OVERVIEW A Boolean variable has only two possible values, "true" and "false". A Boolean expression formed by the relationships "<", "<=", "=", "<>", ">=", and ">" also assumes only the same two values. Of what use is a variable that can assume only two possible values? Consider the WHILE and REPEAT UNTIL looping constructs. Boolean expressions allow the conditional execution of a loop.

Boolean variables and Boolean expressions are among the most useful because they give decision-making ability to the program. Chapter 4 covers the conditional execution of loops. This chapter extends the decision-making ability to the conditional execution of Pascal statements. This is the mechanism that brings flexibility to the computer program which processes one set of data in one manner and a different set of data in another manner. The program adapts its processing method to the specific characteristics of the data.

□

5.1 IF...THEN

The simplest conditional statements are of the form "IF Boolean expression THEN statement". Similar statements abound in everyday life. "If it is raining when I leave for work, I will take an umbrella." The statement, "It is raining when I leave for work," will be either true or false. If it is true, the action statement, "I will take an umbrella," goes into effect.

Buried deep within one of the bank's computer programs is a conditional statement to the effect, "IF customer checking account balance is less than zero, THEN send 'account overdrawn' notice to customer." Within a payroll program is a statement to the effect, "IF hours worked is greater than 40 hours, THEN compute overtime pay at time-and-a-half."

FLOW OF CONTROL

Normally, the flow of control within a Pascal program is from beginning to end. Some describe this flow of control as from top to bottom because the beginning of the program is toward the top of the screen. The looping constructs override

this natural order, but the body of the loop is still sequentially executed from beginning to end. After finishing the looping operation, the program continues in sequential fashion with the statements immediately following the loop.

The simple IF ... THEN statement provides a short single-statement conditional execution. It slips an additional statement into the sequence of statements if the condition is true. By using BEGIN END blocks the extra statement can be a complex structure of Pascal statements including nested looping constructs.

OVERTIME PAY

An organization pays time-and-a-half for overtime beyond 40 hours a week. Gross pay is normally the hourly pay rate times the number of hours worked. Overtime hours earn an additional 50 percent because of the time-and-a-half rule. The following program uses the simple conditional IF statement to add the overtime factor:

```
PROGRAM pay1 (OUTPUT);
VAR
    rate,
    hours,
    gross : REAL;
BEGIN
    rate := 9.25;
    hours := 47.5;
    gross := rate * hours;
    IF hours > 40.0 THEN
        gross := gross + 0.5 * rate * (hours - 40.0);
    WRITELN ('gross pay ',gross:9:2)
END.
```

Sample output from program:

```
gross pay    474.06
```

MAXIMUM VALUE

Another application of the simple IF ... THEN conditional statement is searching for the maximum among a set of values. The statement

```
IF value > maximum THEN maximum := value
```

updates the value of the variable maximum to reflect the maximum value found so far. The contents of "variable" is moved to "maximum" if it is larger than the existing "maximum".

The variable "maximum" must be initialized to an extremely small value or to the first data value. Using a trailer value within the data items to terminate data entry suggests using the WHILE or the REPEAT UNTIL looping constructs.

The following program uses the REPEAT UNTIL looping statements and reads the first data item before starting the looping section of the program:

```
PROGRAM max1 (INPUT, OUTPUT);
VAR
    value,
    maximum : REAL;
BEGIN
    WRITELN ('Determine the maximum value');
    WRITELN ('in a set of data. Use the');
    WRITELN ('value -999.999 to terminate.');
    WRITELN (' ');
    WRITE ('value ? ');
    READLN (value);
    maximum := value;
    REPEAT
        IF value > maximum THEN
            maximum := value;
        WRITE ('value ? ');
        READLN (value)
    UNTIL value = -999.999;
    WRITELN (' ');
    WRITELN ('maximum value = ',maximum:9:2)
END.
```

Sample output from program:

```
Determine the maximum value
in a set of data. Use the
value -999.999 to terminate.

value ? 25.95
value ? 17.95
value ? 31.49
value ? -999.999

maximum =   31.49
```

SALES COMMISSION

A store pays its sales force weekly on a commission basis. The commission is 5 percent of the first $10,000 in sales, 8 percent of the next $10,000 in sales, and 10 percent of all sales beyond $20,000. The following interactive program requests the total sales for each salesperson and computes the resulting commission:

```
PROGRAM sales (INPUT, OUTPUT);
VAR
```

```
    amount,         { Amount of sales }
    comm            { Sales commission }
        : REAL;
BEGIN
    { Compute sales commission for the sales
      force using the rates 5% for the first
      $10,000, 8% for the second $10,000, and
      10% for sales beyond $20,000. }
    BEGIN
        { Initial message }
        WRITELN ('Compute the sales commission');
        WRITELN ('for the sales force.');
        WRITELN (' ');
        WRITELN ('Use sales amount -999.99');
        WRITELN ('to terminate data entry.')
    END;
    BEGIN
        { Process }
        WRITELN (' ');
        WRITE ('Sales amount ? ');
        READLN (amount);
        WHILE amount <> -999.99 DO
            BEGIN
                { Determine commission }
                IF amount < 10000.0 THEN
                    comm := 0.05 * amount;
                IF amount >= 10000.0
                  AND amount < 20000.0 THEN
                  comm := 500.0
                      + 0.08 * (amount -10000.0);
                IF amount >= 20000.0 THEN
                  comm := 1300.0
                      + 0.10 * (amount - 20000.0);
                WRITELN ('Commission = ', comm:9:2);
                { Get next sales amount }
                WRITELN (' ');
                WRITE ('Sales amount ? ');
                READLN (amount)
            END
    END
END.
```

Sample output from this program:

```
Compute the sales commission
for the sales force.
```

```
Use the value -999.99
to terminate data entry.

Sales amount ? 12700.00
Commission =     716.00

Sales amount ? 8275.73
Commission =    413.79

Sales amount ? -999.99
```

ENUMERATED TYPES

One of the problems inherent with enumerated variable types is that they have no external representation. The programmer can supply this external representation through the use of conditional output statements. The following program defines an enumeration type listing the names of the seven days of the week and uses IF ... THEN statements within a FOR loop to display the names of the days:

```
PROGRAM daylist (OUTPUT);
TYPE
    days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
VAR
    day : days;
BEGIN
    FOR day := Sun TO Sat DO
        BEGIN
            IF day = Sun THEN WRITELN ('Sunday');
            IF day = Mon THEN WRITELN ('Monday');
            IF day = Tues THEN WRITELN ('Tuesday');
            IF day = Wed THEN WRITELN ('Wednesday');
            IF day = Thu THEN WRITELN ('Thursday');
            IF day = Fri THEN WRITELN ('Friday');
            IF day = Sat THEN WRITELN ('Saturday');
        END
END.
```

The output from this program follows:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

The output displays the longer form of the names while the internal type names are the three-letter abbreviations.

TEST FOR END OF LINE

One line of input to the program may contain more than one item. The READLN command fetches only the first item and positions the file pointer to the next line. The READ command picks up one item and positions the pointer to the next item.

There are two special terms reserved for describing the status of an input file. The term EOLN stands for end of line, and the term EOF stands for end of file. These are standard Boolean variables having outcomes "true" or "false". Chapter 8 covers external files and use of the EOF variable.

Interactive data entry uses the keyboard for input data. Pressing the Enter key at the end of each line is the signal that the line is completed. Control passes to the computer which processes that line. Data from the keyboard is text data consisting of letters of the alphabet, numeric digits, periods, and other printable symbols.

Nonprintable characters include carriage return, line feed, end of file marker, and many other special control symbols. Many systems use the carriage return or the combination carriage return and line feed to designate the end of the line.

The EOLN and EOF variables are most commonly used with the WHILE and REPEAT UNTIL looping structures. They are also available for reference by IF ... THEN conditional statements.

FREQUENCY COUNT FOR THE LETTER E

The following program uses a REPEAT UNTIL loop to read a one-line message from the keyboard and count number of occurrences of the lowercase letter 'e':

```
PROGRAM letter1 (INPUT, OUTPUT);
VAR
    letter : CHAR;
    count : INTEGER;
BEGIN
    count := 0;
    WRITELN ('Type line for frequency count');
    READ (letter);
    REPEAT
        IF letter = 'e' THEN count := count + 1;
        READ (letter)
    UNTIL EOLN;
    WRITELN ('Frequency for letter e = ',count)
END.
```

Sample output from the program follows:

```
Type line for frequency count
```

```
Come, let us reason together.
Frequency for letter e =          5
```

NULL LINE

The REPEAT UNTIL loop always executes at least once because the test comes at the end of the loop. If the person keying in the data types the Enter command without including any prior data items, the line is empty. It is a null line in the same sense that a semicolon immediately prior to an END results in a null Pascal statement.

The WHILE construct with the test at the beginning of the loop avoids the problem of trying to process data that isn't there if the input line is empty. The following program uses the WHILE looping construct with its test at the beginning of the loop:

```
PROGRAM letter2 (INPUT, OUTPUT);
VAR
    letter : CHAR;
    count : INTEGER;
BEGIN
    WRITELN ('Type line to analyze');
    READ (letter);
    count := 0;
    WHILE NOT EOLN DO
        BEGIN
            IF letter = 'e' THEN count := count + 1;
            READ (letter)
        END;
    WRITELN ('Frequency for letter e = ',count)
END.
```

Sample output from the program:

```
Type line to analyze
Friends, Romans, countrymen, lend me your ears.
Frequency for letter e =          5
```

Now try running the program letter2 with a null line containing no letters at all.

EXERCISES

1. Write a program using pay rate and hours worked to compute gross pay giving time-and-a-half for time over 40 hours. Use interactive data entry.

2. Write a program that computes the minimum of a set of data obtained interactively from the keyboard.
3. Write a program that fetches one line of input from the keyboard and forms the frequency count for each of the vowels 'a', 'e', 'i', 'o', and 'u'.
4. Write a program that reads in the exam scores 88, 83, 75, and 86, computes the average exam score and assigns a letter grade based on the average. Use the following scale:

90-100%	A
80-89.99	B
70-79.99	C
60-69.99	D
0-59.99	F

5.2 IF...THEN...ELSE

The IF ... THEN ... ELSE conditional statement gives the computer a choice of two possible statements. The computer will execute only one of the statements. If the condition is true, the computer executes the first of the two statements. If the condition is false, the computer executes the second statement.

ALTERNATION

Alternation refers to the process of choosing one of two possible courses of action. Either or both of the alternate statements can consist of compound statements formed into BEGIN END blocks.

COMPUTE REGULAR AND OVERTIME PAY

A company pays time-and-a-half for overtime beyond 40 hours a week. The pay is categorized into regular pay and overtime pay. Gross pay is the sum of regular pay and overtime pay. The following program uses an IF ... THEN ... ELSE statement to compute overtime pay if necessary or sets overtime pay to 0.0 if not necessary. Another IF ... THEN ... ELSE statement computes regular pay.

```
PROGRAM pay1 (OUTPUT);
VAR
    rate,
    hours,
    regular,
    overtime,
    grosspay : REAL;
BEGIN
    rate := 9.25;
```

```
    hours := 47.5;
    IF hours <= 40.0 THEN
            regular := rate * hours
        ELSE
            regular := rate * 40.0;
    IF hours <= 40.0 THEN
            overtime := 0.0
        ELSE
            overtime := 1.5 * rate * (hours - 40.0);
    grosspay := regular + overtime;
    WRITELN ('Regular pay   = ',regular:9:2);
    WRITELN ('Overtime pay = ',overtime:9:2);
    WRITELN ('Gross pay     = ',grosspay:9:2)
END.
```

The output from the program follows:

```
Regular pay    =     370.00
Overtime pay   =     104.06
Gross pay      =     474.06
```

IF . . . THEN . . . ELSE WITH BEGIN END BLOCKS

The gross pay calculation involves only the two conditions overtime and no overtime. The following program makes the test once and uses the BEGIN END blocks to define compound statements for alternate execution:

```
PROGRAM pay2 (OUTPUT);
VAR
    rate,
    hours,
    regular,
    overtime,
    grosspay : REAL;
BEGIN
    rate := 9.25;
    hours := 4.75;
    IF hours <= 40.0 THEN
        BEGIN
            regular := hours * rate;
            overtime := 0.0
        END
    ELSE
        BEGIN
            regular := 40.0 * rate;
            overtime := 1.5 * (hours - 40.0) * rate
        END;
```

```
    grosspay := regular + overtime;
    WRITELN ('Regular pay   = ',regular:9:2);
    WRITELN ('Overtime pay  = ',overtime:9:2);
    WRITELN ('Gross pay     = ',grosspay:9:2)
END.
```

The output from this program follows:

```
Regular pay   =    370.00
Overtime pay  =    104.06
Gross pay     =    474.06
```

NESTED IF ... THEN ... ELSE STATEMENTS

A distributor features two types of products and offers quantity discounts at different quantity breaks for each type. The per-unit discount is 20 percent for quantities of 10 or more class A products. The discount is 5 percent for quantities of 100 or more class B products.

The following program computes the total purchase cost for an order given the type of product (class A or class B), the regular per-unit price, and the quantity ordered:

```
PROGRAM discnt1 (INPUT, OUTPUT);
VAR
    class : CHAR;
    price,
    amount : REAL;
    quantity,
    number,
    count : INTEGER;
BEGIN
    WRITE ('Number of orders ? ');
    READLN (number);
    FOR count := 1 to number DO
        BEGIN
            WRITELN (' ');
            WRITE ('Product class ? ');
            READLN (class);
            WRITE ('Per-unit price? ');
            READLN (price);
            WRITE ('Quantity     ? ');
            READLN (quantity);
            IF class = 'A' THEN
                IF quantity < 10 THEN
                    amount := price * quantity
                ELSE
                    amount := 0.8 * price * quantity
```

```
            ELSE
                IF quantity < 100 THEN
                    amount := price * quantity
                ELSE
                    amount := 0.95 * price * quantity;
            WRITELN ('Amount of purchase = ',amount:9:2)
        END
END.
```

Sample output from the program follows:

```
Number of orders ? 4

Product class ? A
Per-unit price ? 3.95
Quantity      ? 25
Amount of purchase =     79.00

Product class ? B
Per-unit price ? 0.14
Quantity      ? 5
Amount of purchase =       .70

Product class ? B
Per-unit price ? 0.69
Quantity      ? 144
Amount of purchase =     94.39

Product class ? A
Per-unit price ? 12.95
Quantity      ? 4
Amount of purchase =     51.80
```

QUANTITY DISCOUNTS USING BOOLEAN EXPRESSIONS

Boolean expressions provide an alternate method to nested IF statements. The following program uses compound Boolean expressions to test the product class and quantity for the quantity discount problem:

```
PROGRAM discnt2 (INPUT, OUTPUT);
VAR
    class : CHAR;
    price,
    amount : REAL;
    quantity,
    number,
    count : INTEGER;
BEGIN
    WRITE ('Number of customers ? ');
```

```
    READLN (number);
    FOR count := 1 TO number DO
        BEGIN
            WRITELN (' ');
            WRITE ('Product class ? ');
            READLN (class);
            WRITE ('Per-unit price ? ');
            READLN (price);
            WRITE ('Quantity      ? ');
            READLN (quantity);
            IF class = 'A' AND quantity < 10 THEN
               amount := price * quantity;
            IF class = 'A' AND quantity >= 10 THEN
               amount := 0.8 * price * quantity;
            IF class = 'B' AND quantity < 100 THEN
               amount := price * quantity;
            IF class = 'B' AND quantity >= 100 THEN
               amount := 0.95 * price * quantity;
            WRITELN ('Amount of purchase = ',amount:9:2)
        END
END.
```

The output is the same as the previous program.

SERIES OF QUANTITY DISCOUNTS

A series of nested IF . . . THEN . . . ELSE statements of the type

```
IF condition THEN statement
   ELSE IF condition THEN statement
      ELSE IF condition THEN statement
         ELSE IF condition THEN statement
            ELSE statement
```

is very useful. The nested IF comparisons continue only until the first true condition. After executing the statement following the THEN clause of the first "true" condition, the rest are no longer needed.

A series of nested IF statements of this type can replace a similar series of IF statements requiring complex Boolean expressions. In some cases the nested IF construct is easier to understand than the corresponding IF statements using Boolean expressions. In other cases, Boolean expressions are easier to understand.

A manufacturer offers progressively higher discounts for a series of order quantities. The regular per-unit price is $19.95. For orders of 20 to 49 the per-unit price is $17.95. For orders of 50 to 199 the per-unit price is $16.95. For

orders of 200 or more the per-unit price is $15.95. The following program uses a series of nested IF . . . THEN . . . ELSE statements:

```
PROGRAM discnt3 (INPUT, OUTPUT);
VAR
    quantity : INTEGER;
    price,
    amount : REAL;
BEGIN
    WRITELN ('Determine quantity discount price');
    WRITELN ('and amount of purchase for several');
    WRITELN ('order quantities. Use quantity of');
    WRITELN ('0 to terminate.');
    WRITELN (' ');
    WRITE ('Quantity ? ');
    READLN (quantity);
    REPEAT
        IF quantity < 20 THEN
            price := 19.95
        ELSE IF quantity < 50 THEN
            price := 17.95
        ELSE IF quantity < 200 THEN
            price := 16.95
        ELSE
            price := 15.95;
        amount := price * quantity;
        WRITELN ('Per-unit price        ',price:9:2);
        WRITELN ('Amount of purchase',amount:9:2);
        WRITELN (' ');
        WRITE ('Quantity ? ');
        READLN (quantity)
    UNTIL quantity = 0
END.
```

Sample output follows:

```
Determine quantity discount price
and amount of purchase for several
order quantities. Use a quantity of
0 to terminate.

Quantity ? 7
Per-unit price          19.95
Amount of purchase     139.65

Quantity ? 25
Per-unit price          17.95
Amount of purchase     448.75
```

```
Quantity ? 50
Per-unit price          16.95
Amount of purchase     847.50
Quantity ? 200
Per-unit price          15.95
Amount of purchase    3190.00
```

IF THEN BLOOPERS

Several problems arise when using IF ... THEN statements, especially those that are deeply nested. A misplaced semicolon will cause a program to perform differently from that intended. The statement

```
IF X < 25 THEN price := 24.95;
```

will assign the value 24.95 to the variable price only if the value of X is less than 25. The statement

```
IF X < 25 THEN; price := 24.95;
```

assigns the value 24.95 to price regardless of the value contained in X. The semicolon after the THEN keyword terminates the IF statement.

Some deeply nested IF... THEN statements are ambiguous to people reading the program, but not to the Pascal compiler. Many of the ambiguities are removed by noting that Pascal always associates an ELSE statement with the immediately preceding open IF statement. It that is not intended, then careful use of BEGIN END blocks can force the compiler to associate the ELSE statement with the appropriate IF ... THEN statement.

The ELSE portion of the IF ... THEN ... ELSE statement can be omitted entirely. It is very difficult to read programs having nested IF statements if some of the IF statements have ELSE statements and some do not. Use extreme care when writing, testing, and debugging nested IF statements.

EXERCISES

1. Write a checkbook balancing program. Read the initial balance first, then read the transactions one at a time. For each transaction, type the letter "D" for a deposit and the letter "C" for a check. The value of the transaction will be positive. Use a transaction type of "S" to stop the data entry. Compute and display the service charge. There is no service charge if the balance never falls below $200.00. The service charge is $.10 per check if the balance does fall below $200.00.
2. Write a program that fetches one line of input from the keyboard and counts the frequencies of each of the vowels "a", "e", "i", "o", and "u". Use a nested IF ... THEN ... ELSE statement.

3. Write a program that reads in a set of 4 exam scores for each of 5 students and computes the average exam score. Assign a letter grade as follows:

90.00-99.99	A
80.00-89.99	B
70.00-79.99	C
60.00-69.99	D
00.00-59.99	F

5.3 CASE...OF

When one of several alternate statements must be selected, it is possible to use IF statements to control the selection. The IF statements are in the form of deeply nested IF statements and are difficult to construct, difficult to test, difficult to debug, and difficult to read. Avoid deeply nested IF statements that are difficult to read.

The CASE selection statement is of the form

```
CASE expression OF
    value1 list : statement1;
    value2 list : statement2;
    value3 list : statement3;
      etc.
    valuen list : statementn
END;
```

The expression must result in values which are integer, character, or Boolean. It may reference a variable of an enumerated ordinal type. The expression cannot result in a real value.

ERROR CONDITIONS

What if the expression results in a value not contained in the list? Many Pascal systems consider this an unrecoverable error and the program stops working. Other Pascal systems, including Alcor Pascal, continue program execution with the statement immediately following the END.

OTHERWISE OPTION

Alcor Pascal defines an OTHERWISE option giving a set of statements for execution if the expression results in a nonlisted value. The format of the CASE statement with the OTHERWISE option is

```
CASE expression OF
    value1 list : statement1;
```

```
    value2 list : statement2;
    value3 list : statement3;
    value4 list : statement4;
    OTHERWISE
    statement 5;
    statement 6;
    statement 7
END;
```

COUNT LETTER FREQUENCIES

The following program uses the CASE selection statement to count the frequencies of the vowels "a", "e", "i", "o", and "u":

```
PROGRAM letter1 (INPUT, OUTPUT);
VAR
    letter : char;
    counta,
    counte,
    counti,
    counto,
    countu,
    countk : INTEGER;
BEGIN
    counta := 0;
    counte := 0;
    counti  := 0;
    counto := 0;
    countu := 0;
    countk := 0;
    WRITELN ('Type line to be analyzed');
    READ (letter);
    WHILE NOT EOLN DO
        BEGIN
            CASE letter OF
                'a', 'A' : counta := counta + 1;
                'e', 'E' : counte := counte + 1;
                'i', 'I'  : counti  := counti  + 1;
                'o', 'O' : counto := counto + 1;
                'u', 'U' : countu := countu + 1;
                OTHERWISE
                countk := countk + 1
            END;
            READ (letter)
        END;
    WRITELN ('Frequency for');
```

```
    WRITELN ('Vowel a',counta);
    WRITELN ('Vowel e',counte);
    WRITELN ('Vowel i',counti);
    WRITELN ('Vowel o',counto);
    WRITELN ('Vowel u',countu);
    WRITELN ('Consonants',countk)
END.
```

Sample output for the program follows:

```
Type line to be analyzed
On the road to Mandalay, where the flying fishes play
Vowel a     5
Vowel e     5
Vowel i     2
Vowel o     3
Vowel u     0
Consonants     37
```

ENUMERATED DATA TYPES

The following program defines a data type consisting of the seven days of the week and uses the FOR loop to enumerate the days. The CASE selection statement displays the full names rather than the abbreviations used in the data type:

```
PROGRAM daylist (OUTPUT);
TYPE
    days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
VAR
    day : days;
BEGIN
    FOR day := Sun TO Sat DO
        CASE day OF
            Sun  : WRITELN ('Sunday');
            Mon : WRITELN ('Monday');
            Tue  : WRITELN ('Tuesday'):
            Wed : WRITELN ('Wednesday'):
            Thu  : WRITELN ('Thursday');
            Fri   : WRITELN ('Friday');
            Sat   : WRITELN ('Saturday')
        END
END.
```

The output from the program follows:

```
Sunday
Monday
```

```
Tuesday
Wednesday
Thursday
Friday
Saturday
```

ASSIGNING LETTER GRADES

The following program assigns and displays a letter grade for an integer test score on an exam having 100 as the maximum possible score:

```
PROGRAM assign (INPUT, OUTPUT);
TYPE
    grades = 'A' .. 'F';
VAR
    score : INTEGER;
    grade : grades;
BEGIN
    WRITELN ('Assign letter grades to');
    WRITELN ('exam scores. Use an exam');
    WRITELN ('score of -99 to terminate.');
    WRITELN (' ');
    WRITE ('Score ? ');
    READLN (score);
    REPEAT
        CASE score DIV 10 OF
            10, 9 : grade := 'A';
            8     : grade := 'B';
            7     : grade := 'C';
            6     : grade := 'D';
            OTHERWISE
                  grade := 'F'
        END;
        WRITELN ('Grade = ',grade);
        WRITELN (' ');
        WRITE ('Score ? ');
        READLN (score)
    UNTIL score = -99
END.
```

Sample output for the program follows:

```
Assign letter grades to
exam scores. Use an exam
score of -99 to terminate.

Score ? 83
Grade = B
```

```
Score ? 69
Grade = D
Score ? 91
Grade = A
Score ? -99
```

The Pascal CASE selection statement is a powerful statement for organizing multiway action problems.

EXERCISES

1. A service station charges $1.319 per gallon for unleaded gas during the week and $1.379 on weekends. Days are numbered sequentially starting with 1 for January 1, 2, for January 2, etc. February 1 is day 32 since January has 31 days. This is a common business practice.

   Look at the calendar for the current year to determine the day of the week for January 1. Using that information, write a program that accepts the number of the day, displays the name of the day, and assigns the proper price per gallon. Use the CASE selection statement.

2. Write a program that computes the average exam score for three exams for each of five students. Assign letter grades using the boundaries given by the program "grades". Convert the average grade (real) to an integer score and use the CASE selection approach.

3. Write a program that accepts an integer value and displays the message "odd" or "even" depending on whether the integer is odd or even. Use CASE selection and a selector expression of the type

   ```
   number MOD 2
   ```

   which gives the remainder after dividing the integer variable number by 2.

5.4 BOOLEAN EXPRESSIONS

Boolean algebra is named in honor of the mathematician George Boole. Objects in Boolean algebra assume one of two values (true or false). The objects are statements or assertions. Two or more are combined into compound statements using Boolean algebra.

TYPE BOOLEAN

Pascal permits the definition and use of Boolean variables. The following program defines two Boolean variables, assigns values to them, and displays the results:

```
PROGRAM boole1 (OUTPUT);
VAR
```

```
    start,
    stop : BOOLEAN;
BEGIN
    start := TRUE;
    stop := FALSE;
    WRITELN (start);
    WRITELN (stop)
END.
```

The sample output follows:

```
TRUE
FALSE
```

RELATIONAL EXPRESSIONS

The value of a Boolean variable is often set with a relational comparison. The following program uses relational expressions to set the value of Boolean variables:

```
PROGRAM boole2 (OUTPUT);
VAR
    less,
    equal,
    greater : BOOLEAN;
    first,
    second : INTEGER;
BEGIN
    first := 19;
    second := 12;
    less := first < second;
    equal := first = second;
    greater := first > second;
    WRITELN ('Less than    ',less);
    WRITELN ('Equal to     ',equal);
    WRITELN ('Greater than ',greater)
END.
```

The sample output follows:

```
Less than    FALSE
Equal to     FALSE
Greater than TRUE
```

BOOLEAN OPERATORS

The Boolean operators are AND, OR, and NOT. The operation AND is true if both of the operands are true. The operation OR is true if at least one of the ex-

pressions is true. The operator NOT is true if the operand is false. The following program illustrates the use of the AND, OR, and NOT operators.

```
PROGRAM boole3 (OUTPUT);
VAR
    first,
    second,
    result1,
    result2,
    result3 : BOOLEAN;
BEGIN
    first := TRUE;
    second := FALSE;
    result1 := first OR second;
    result2 := first AND second;
    result3 := NOT first;
    WRITELN ('Result 1 ',result1);
    WRITELN ('Result 2 ',result2);
    WRITELN ('Result 3 ',result3)
END.
```

The sample output follows:

```
Result 1 TRUE
Result 2 FALSE
Result 3 FALSE
```

OPERATOR PRECEDENCE

Alcor Pascal uses the following precedence ordering of the arithmetic and logical operators:

Highest	( )	Parentheses
	+ -	When used as unary operators
	* / DIV MOD	
	+ -	
	= <> <> <= >=	
	NOT	
	AND	
Lowest	OR	

This precedence ordering differs from some Pascal implementations. They place NOT on the same order as unary operators, AND on the level of * and /, and OR on the same level as addition and subtraction.

RELATIONAL EXPRESSIONS AND BOOLEAN VARIABLES

The following program defines an enumeration variable containing the common three-letter abbreviations for the names of the days. It uses a FOR loop to test each day to determine whether it is a weekday or a weekend day.

```
PROGRAM boole4 (OUTPUT);
TYPE
    days = (SUN, MON, TUE, WED, THU, FRI, SAT);
VAR
    day : days;
    weekday,
    weekend : BOOLEAN;
BEGIN
    FOR day := SUN TO SAT DO
        BEGIN
            weekend := (day = SAT) OR (day = SUN);
            weekday := (day > SUN) AND (day < SAT);
            WRITE ('Week day  ',weekday);
            WRITELN ('      Weekend  ',weekend)
        END
END.
```

The sample output follows:

```
Weekday     FALSE     Weekend     TRUE
Weekday     TRUE      Weekend     FALSE
Weekday     TRUE      Weekend     FALSE
Weekday     TRUE      Weekend     FALSE
Weekday     TRUE      Weekend     FALSE
Weekday     TRUE      Weekend     FALSE
Weekday     FALSE     Weekend     TRUE
```

BOOLEAN VARIABLES FOR LOOPING CONSTRUCTS

The loop control variables for WHILE and REPEAT loops can be Boolean variables. The following program repeats the loop until the Boolean variable is TRUE:

```
PROGRAM boole5 (INPUT, OUTPUT);
VAR
    value,
    sum : REAL;
    stop : BOOLEAN;
BEGIN
    sum := 0.0;
    REPEAT
        WRITE ('Value ? ');
```

```
        READLN (value);
        stop := (value = -999.0);
        IF NOT stop THEN
            sum := sum + value;
    UNTIL stop;
    WRITELN ('Sum ',sum:9:2)
END.
```

The following is a sample test run:

```
Value ? 13.0
Value ? 15.5
Value ? -999.0
Sum     28.50
```

EXERCISES

1. Write a program that obtains data from the keyboard and determines both the largest and the smallest values. Use the trailer value -999.0 to terminate the data entry.
2. Write a program that obtains data from the keyboard and counts the number of runs of increasing or decreasing value. The data

   12 39 28 23 16 32 35 25 40

   has the five runs (12, 39), (39, 28, 23, 16), (16, 32, 35), (35, 25), and (25, 40).
3. Write a program that obtains a line of text from the keyboard and computes the average number of letters in each word. The comma, period, semicolon, and space may be used to separate words on the line.

6 Procedures and functions

OVERVIEW The subprogram brings order and discipline to the creation of large programs. The programmer divides large programs into modules, each of which performs one of the needed tasks. Subdividing a program into BEGIN/END blocks provides some help in organizing the program.

Pascal permits the programmer to define and use procedures and user-defined functions. Procedures bring order out of chaos. Large programs should be designed as an hierarchical tree with a main program calling procedures which call lower-level procedures, until the lowest-level procedures accomplish the processing steps.

The organization chart of a program looks just like an organization chart of a company. The span of control is just as important to a program as it is to a company. Rarely should one procedure directly call more than six or eight subordinate procedures.

Procedures within a large program should be independent of each other. The term loose coupling applies to this property of independence. Procedures communicate with each other through argument (parameter) lists. Another important property is cohesion. An internally cohesive procedure does only one task. Combining two or more disconnected tasks into one procedure is poor practice.

This chapter introduces the use of procedures and user-defined functions. It discusses the relative merits of global versus local variables. It compares the major types of procedure variable references.

6.1 Subprograms With Global Variables

Consider the following program which computes the area and circumference of a circle:

```
PROGRAM circle1 (INPUT, OUTPUT);
VAR
    radius,
    area,
```

```
    circum : REAL;
BEGIN
    WRITELN ('Compute the area');
    WRITELN ('and circumference');
    WRITELN ('of a circle.');
    WRITELN;
    WRITE ('Radius ? ');
    READLN (radius);
    area := 3.14159 * radius * radius;
    circum := 2.0 * radius * 3.14159;
    WRITELN;
    WRITELN ('Area              ',area:9:2);
    WRITELN ('Circumference     ',circum:9:2)
END.
```

The output from a sample run follows:

```
Compute the area
and circumference
of a circle.

Radius ? 2.5

Area              19.63
Circumference     15.71
```

This program displays an initial message identifying its purpose, requests the radius of the circle, and displays its area and circumference.

BEGIN/END BLOCKS AND DOCUMENTATION

Chapter 3 illustrates the use of BEGIN/END blocks and comments to organize the program into identifiable tasks. The following program uses this approach:

```
PROGRAM circle2 (INPUT, OUTPUT);
VAR
    radius,
    area,
    circum : REAL;
BEGIN
    { Compute the area and circumference
      of a circle }
    BEGIN
        { Initial message }
        WRITELN ('Compute the area');
        WRITELN ('and circumference');
        WRITELN ('of a circle.')
    END;
    BEGIN
```

```
        { Get data }
        WRITELN;
        WRITE ('Radius ? ');
        READLN (radius)
    END;
    BEGIN
        { Compute measures }
        area := 3.14159 * radius * radius;
        circum := 2.0 * radius * 3.14159
    END;
    BEGIN
        { Display result }
        WRITELN;
        WRITELN ('Area              ',area:9:2);
        WRITELN ('Circumference  ',circum:9:2)
    END
END.
```

The sample output is the same as for the previous program. This program more clearly identifies the three tasks of the program.

PROCEDURES

A procedure is a subprogram identified by name. The procedure heading

```
PROCEDURE name1
```

assigns the label name1 to the procedure consisting of the following statements including the BEGIN/END block. The following procedure displays the initial message:

```
PROCEDURE initial;
    { Initial message }
BEGIN
    WRITELN ('Compute the area');
    WRITELN ('and circumference');
    WRITELN ('of a circle.')
END;
```

The radius is requested interactively by the following procedure:

```
PROCEDURE getdata;
    { Get data }
BEGIN
    WRITELN;
    WRITE ('Radius ? ');
    READLN (radius)
END;
```

The following computes the area and circumference:

```
PROCEDURE compute;
    { Compute measures }
BEGIN
    area := 3.14159 * radius * radius;
    circum := 2.0 * radius * 3.14159
END;
```

The area and circumference is displayed by the following procedure:

```
PROCEDURE display;
    { Display results }
BEGIN
    WRITELN;
    WRITELN ('Area              ',area:9:2);
    WRITELN ('Circumference  ',circum:9:2)
END;
```

The following is the main program which uses the procedures:

```
PROGRAM circle3 (INPUT, OUTPUT);
    { Compute the area and circumference
      for a circle }
VAR
    radius,
    area,
    circum : REAL;
BEGIN
    initial;
    getdata;
    compute;
    display
END.
```

COMPLETE PASCAL PROGRAM USING PROCEDURES

Most higher-level languages require the compiler to make at least two passes through the source program while generating the object program. Pascal compilers need only one pass if the programmer defines everything before it is needed. Variables must be declared before they are used. Procedures must be defined before they are used. Procedure definitions follow VAR declarations and come before the BEGIN/END block of the main program.

The following complete Pascal program includes the procedure definitions and the main program as they should appear:

```
PROGRAM circle3 (INPUT, OUTPUT);
    { Compute the area and circumference
      of a circle }
VAR
    radius,
    area,
    circum : REAL;
PROCEDURE initial;
    { Initial message }
BEGIN
    WRITELN;
    WRITELN ('Compute the area');
    WRITELN ('and circumference');
    WRITELN ('of a circle.')
END;
PROCEDURE getdata;
    { Get data }
BEGIN
    WRITELN;
    WRITE ('Radius ? ');
    READLN (radius)
END;
PROCEDURE compute;
    { Compute measures }
BEGIN
    area := 3.14159 * radius * radius;
    circum := 2.0 * radius * 3.14159
END;
PROCEDURE display;
    { Display results }
BEGIN
    WRITELN;
    WRITELN ('Area          ',area:9:2);
    WRITELN ('Circumference',circum:9:2)
END;
BEGIN
    { Main program }
    initial;
    getdata;
    compute;
    display
END.
```

The output is the same as that of the previous programs.

EMPHASIZE PROGRAM LISTING DIVISIONS

Long Pascal programs will still be difficult to read unless the programmer uses strong measures to emphasize the divisions into procedures. Many programmers use one or more blank lines to separate procedure definitions from each other and from the body of the main program. Programmers familiar with Pascal's structure know to look for the body of the main program at the end of the listing with a period terminating the last END.

Other programmers place the name of the procedure in a box of asterisks as an initial identifying comment. The following program illustrates this practice:

```
PROGRAM circle4 (INPUT, OUTPUT);
    { Compute the area and circumference
      of a circle }
VAR
    radius,
    area,
    circum : REAL;
{ ****************************************
  *     initial                          *
  **************************************** }
PROCEDURE initial;
    { initial message }
BEGIN
    WRITELN;
    WRITELN ('Compute the area');
    WRITELN ('and circumference');
    WRITELN ('of a circle.')
END;
{ ****************************************
  *     getdata                          *
  **************************************** }
PROCEDURE getdata;
    { Get data }
BEGIN
    WRITELN;
    WRITE ('Radius ? ');
    READLN (radius)
END;
{ ****************************************
  *     compute measures                 *
  **************************************** }
PROCEDURE compute;
    { Compute measures }
```

```
BEGIN
    area := 3.14159 * radius * radius;
    circum := 2.0 * radius * 3.14159
END;
{ ****************************************
  *    display                           *
  **************************************** }
PROCEDURE display;
    { Display results }
BEGIN
    WRITELN;
    WRITELN ('Area          ',area:9:2);
    WRITELN ('Circumference ',circum:9:2)
END;
{ ****************************************
  *    circle4     main body             *
  **************************************** }
BEGIN
    initial;
    getdata;
    compute;
    display
END.
```

Sample output from the program follows:

```
Compute the area
and circumference
of a circle.

Radius ? 5.0

Area            78.54
Circumference   31.42
```

EXERCISES

1. Write a program to compute the value of ***n***! (***n*** factorial) for a positive integer ***n***. Limit ***n*** to the range 1, 2, . . . , 20. Compute ***n***! as the product of the first ***n*** integers, but use a real variable to contain the product. Use a modular program organization with one procedure displaying an initial message, the second calculating the product, and the third displaying the result. Test using an ***n*** of 6.
2. Write a program calculating the sum and sum of squares of the following values:

   83 91 75 69 82.

Use a modular organization with one procedure displaying an initial message, a second reading the data and accumulating the sum and the sum of squares, and the third displaying the result.

3. Write a program to compute the surface area, volume, and net weight of a cylinder 12 feet long by 3 feet in diameter. Assume that the cylinder is filled with water. Use modular organization with procedures.
4. Write a program to compute the regular pay, overtime pay, and gross pay for several individuals. Use modular organization with procedure calls. Test using the following: 36.5 hours at $8.25 per hour, 43.1 hours and $5.96 per hour, 32 hours at 7.45 per hour, and 47.5 hours at $9.25 per hour.

6.2 SUBPROGRAMS WITH LOCAL VARIABLES

Variables declared in the main program are global variables available to all subordinate procedures. After control returns to the main program, changes to global variables made in the subsidiary procedure remain. The variables A and B in the following program are global because they are defined in the main VAR declaration section:

```
PROGRAM global (OUTPUT);
{ Illustrate use of global variables }
VAR
    a,
    b : INTEGER;
{ Subordinate procedure }
PROCEDURE change;
BEGIN
    a := 21;
    b := 22;
    WRITELN (a,b)
END;
{ Body of main program }
BEGIN
    a := 11;
    b := 12;
    WRITELN (a,b);
    change;
    WRITELN (a,b)
END.
```

The program displays the following:

```
11    12
21    22
21    22
```

The main program sets the values of a to 11 and b to 12 and displays them. It then calls the subsidiary procedure which changes the variable values to 21 and 22 and displays them. After control returns to the main program, it displays the contents of a and b which reflect the changes made by the subsidiary procedure.

LOCAL VARIABLES

Variables declared in the subsidiary procedure are local to that procedure and are kept hidden from higher-level procedures, including the main program. Variables given similar names in both the main routine and subsidiary procedures will not interfere with each other. The following program illustrates this:

```
PROGRAM local (OUTPUT);
{ Illustrate use of local variables }
VAR
    a,
    b : INTEGER;
{ Subordinate procedure }
PROCEDURE change;
VAR
    a,
    b : INTEGER;
BEGIN
    a := 21;
    b := 22;
    WRITELN (a,b)
END;
{ Body of main program }
BEGIN
    a := 11;
    b := 12;
    WRITELN (a,b);
    change;
    WRITELN (a,b)
END.
```

The program displays the following output:

```
11    12
21    22
11    12
```

The main program initializes the variables a to 11 and b to 12 and displays their values. It calls the subsidiary procedure which declares and initializes its own local variables and displays their values. Upon returning to the main program, the last WRITELN statement displays the initial values of the variables a and b which were not changed by the subsidiary procedure.

INFORMATION HIDING AND LOOSE COUPLING

Information hiding is an important principle of computer science. Global variables declared at the level of the main program are available for all subsidiary procedures to inspect and change even though they are irrelevant. It is far better to define local variables for restricted use by the subsidiary procedures whenever possible. Keeping local variables hidden within procedures reduces the coupling between modules and enhances their mutual independence.

SENDING OUTPUT TO THE PRINTER

Future programs will be longer and will have more output. Redirecting video output to the printer is one way of providing a permanent record of the output. For many programs, however, the resulting output is not well-organized. A better practice is to use the keyboard and video screen for interactive data entry and to send the organized output to the printer for a permanent record.

All Pascal programs can use the standard files INPUT and OUTPUT. The INPUT file is usually the keyboard. The OUTPUT file is usually the video display. By using the physical device names :L (printer), :C (console), or :D (dummy device) the user can redirect I/O to an appropriate device. The printer does not make the best input device. Alternately, the I/O may be redirected to disk files by typing the desired TRSDOS file names for the run time prompt messages.

The programmer may declare and use files in addition to the normal INPUT and OUTPUT default files. The logical name of the file must be declared as type TEXT in the VAR section. It must also appear as the first name in the parameter list of the READ, READLN, WRITE, or WRITELN command. Input files must be RESET to initialize the pointer before the first READ or READLN. Output files require the REWRITE command to initialize their pointer before the first WRITE or WRITELN.

EXAMPLE GRADE ANALYSIS PROGRAM

The following program sends the printer some well-known lines attributed to Abraham Lincoln:

```
PROGRAM printer (out);
VAR
    out : TEXT;
BEGIN
    REWRITE (out);
    WRITELN (out,'You can fool all of the people');
    WRITELN (out,'some of the time, and some of the');
    WRITELN (out,'people all of the time, but you');
    WRITELN (out,'cannot fool all of the people');
    WRITELN (out,'all of the time.')
END.
```

Respond with the physical device name :L to the run time prompt for the file "out". The five lines will be sent to the printer.

EXAMPLE PROGRAM SENDING OUTPUT TO THE PRINTER

A class of six students have taken three exams with the following results:

ID	*Exam 1*	*Exam 2*	*Exam 3*
101	73	79	81
105	89	96	88
117	72	86	93
124	74	61	64
131	83	62	49
139	42	55	61

Compute the average exam score for each student and assign a letter grade of A, 90%; B, 80%; C, 70%; D, 60%; and F, below 60% for the average exam score. When running the program use the device parameter :L to send the normal output to the printer rather than to the video display.

The program follows:

```
PROGRAM grade1 (INPUT, OUTPUT, printer);
{ Compute exam averages and
  assign letter grade }
VAR
     printer : TEXT;
     answer,
     grade : CHAR;
     number : INTEGER;
     average : REAL;
{ ****************************************
  *     initial                          *
  **************************************** }
PROCEDURE initial;
BEGIN
     WRITELN ('Compute the average exam score');
     WRITELN ('for several students and assign');
     WRITELN ('appropriate letter grade.')
END;
{ ****************************************
  *     compute                          *
  **************************************** }
PROCEDURE compute;
VAR
     id,
     count : INTEGER;
```

```
    score,
    sum : REAL;
BEGIN
    sum := 0.0;
    WRITELN;
    WRITE ('ID number ? ');
    READLN (id);
    WRITELN 'Score for');
    FOR count := 1 TO number DO
        BEGIN
            WRITE ('Exam ',count:2);
            READLN (score);
            sum := sum + score
        END;
    average := sum / number;
END;
{ ****************************************
  *     assign                           *
  **************************************** }
PROCEDURE assign;
BEGIN
    IF average >= 90.0 THEN grade := 'A'
    ELSE IF average >= 80.0 THEN grade := 'B'
    ELSE IF average >= 70.0 THEN grade := 'C'
    ELSE IF average >= 60.0 THEN grade := 'D'
    ELSE grade := 'F'
END;
{ ****************************************
  *     body of main program             *
  **************************************** }
BEGIN
    REWRITE (printer);
    WRITELN (printer,'ID number   Average   Grade');
    WRITELN;
    WRITE ('Number of exams ? ');
    READLN (number);
    REPEAT
        compute;
        assign;
        WRITELN (printer,id:9,average:9:2,grade:5);
        WRITELN;
        WRITE ('Enter another student (Y/N)');
        READLN (answer)
    UNTIL answer = "N"
END.
```

EXERCISES

1. Debug and test the program calculating the grade point averages and letter grades. Route the file "printer" to the printer using the device name :L to the run time prompt. Press the enter key for the INPUT and OUTPUT file prompts to permit interactive data entry.
2. Write a program using subsidiary procedures and local variables to determine the elapsed time in hours and minutes for two given times using a 24-hour clock.
3. Write a program that will print a compound interest table. Use interactive data entry to obtain the beginning balance ($7,250), interest rate (12.5 percent), compounding periods per year (quarterly), and number of years (25). For each year, print one line giving the year, interest earned during the year, and the accumulated balance at the end of the year.

6.3 TYPES OF VARIABLE REFERENCES

Variables may be global or local. Variables declared by the Pascal program are global to all procedures defined within that program. Each subsidiary procedure defines its own local variables. A procedure that defines a local variable of the same name as a global variable cannot reference the global variable. The local declaration takes precedence.

HIERARCHICAL OR NESTED ORGANIZATION

Large Pascal programs are organized as hierarchical trees with the main program forming the root and the subsidiary subroutines tied together much as a company organization chart. The main routine calls procedures which, in turn, call lower-level procedures. The lower-level procedures are usually nested within their hosts. Variables declared in a procedure are global to those procedures nested within if they have not declared variables with the same name.

SUBPROGRAM PARAMETERS

A parameter is a value or variable used for communicating information between a procedure and its calling program. A value parameter is a variable local to a procedure whose value is passed to it as an argument of the procedure call. Pass-by-value communicates the actual value to the subprogram.

A variable parameter is a local identifier used as a synonym for the global variable. The local variable is used instead of the global variable name. This is called pass-by-reference.

The procedure call contains an argument list enclosed in parentheses. Commas separate entries in the list. The pass-by-value method may use constants and literals. The pass-by-reference method must use variables. The parameter list, or argument list, contains groups of parameters separated by semicolons. The

keyword VAR for a group of parameters identifies them as variable parameters. A group that does not begin with the keyword VAR designates value parameters.

PASS BY VALUE

For a rectangle of length *l* and width *w*, the expression *lw* gives the area, and $2(l+w)$ gives the length of the perimeter (hereafter referred to as the circumference). The following program uses value parameters to pass the length and width measurements to the procedure. It then calculates the area and circumference:

```
PROGRAM param1 (INPUT, OUTPUT);
VAR
    long,
    short : INTEGER;
PROCEDURE compute (length, width : INTEGER);
VAR
    area,
    circum : INTEGER;
BEGIN
    area := length * width;
    circum := 2 * (length + width);
    WRITELN;
    WRITELN ('Length          ',length);
    WRITELN ('Width           ',width);
    WRITELN ('Area            ',area);
    WRITELN ('Circumference ',circum)
END;
BEGIN
    compute (40, 30);
    long := 50;
    short := 25;
    compute (long, short)
END.
```

Output from the program follows:

```
Length          40
Width           30
Area          1200
Circumference  140

Length          50
Width           25
Area          1250
Circumference  150
```

LOOK BUT DON'T TOUCH

The pass-by-value method initializes the subprogram's local variable. The subprogram can use that value, but it cannot change the value of the variable in the calling program. Observe the result of running the following program:

```
PROGRAM look1 (OUTPUT);
VAR
    a,
    b : INTEGER;
PROCEDURE touch (a, b : INTEGER);
VAR
    product : INTEGER;
BEGIN
    product := a * b;
    WRITELN;
    WRITELN (a, b, product);
    a := 4;
    b := 5;
    product := a * b;
    WRITELN (a, b, product);
END;
BEGIN
    a := 3;
    b := 4;
    touch (a, b);
    WRITELN (a, b);
    touch (6, 8)
END.
```

Output from the program follows:

```
3     4     12
4     5     20
3     4
6     8     48
```

The subsidiary procedure cannot modify the values of the variables passed to it as value parameters.

PASS-BY-REFERENCE USING VARIABLE PARAMETERS

The calling program passes the address of the argument with the pass-by-reference method. This allows the subprogram to modify the contents of the variable within the calling program. This is stronger binding than the pass-by-value approach. Pass-by-value provides one-way communication from the calling program to the sub-

program. The subprogram can "look but not touch". Pass-by-reference through the use of variable parameters allows two-way communication. The subprogram can return values to the calling program.

The following program uses variable parameters and pass-by-reference, allowing complete two-way communication:

```
PROGRAM look2 (OUTPUT);
VAR
    product,
    a,
    b : INTEGER;
PROCEDURE touch (VAR x, y, p : INTEGER);
BEGIN
    a := x * y;
    WRITELN (x, y, p);
    x := 6;
    y := 8;
END;
BEGIN
    a := 3;
    b := 4;
    touch (a, b, product);
    WRITELN (a, b, product)
END.
```

The program output follows:

```
3    4    12
6    8    12
```

The main program placed the value 3 in a and the value 4 in b. It passed these as variable parameters to the procedure "touch" which computed the product and displayed the three values. It also changed the values of the variable parameters, referred to as x and y in the subprogram. (They referenced the variables a and b in the main program.) The parameter lists of the calling statement and the subprogram are linked together by position.

CHOICE OF METHOD

Many programming languages, including FORTRAN and COBOL, have subroutine capabilities with pass-by-reference methods. Few of the traditional languages implement pass-by-value parameters. Pascal programs can choose between the two methods.

Pass-by-value parameters provide the loosest coupling between subprogram and calling program and should be used where possible. Pass-by-reference using

variable parameters permit the subprogram to reach into the calling program's variable space to change their values. This is much stronger coupling and should be used only when needed.

MIXED PARAMETERS

The argument list may contain a mixture of value parameters and variable parameters. Value parameters are those which the subprogram may look at but not change. Variable parameters are the results returned to the calling program from the subprogram.

The following program computes the area and circumference of a rectangle sending the length and width as value parameters to the compute procedure which returns the area and circumference as variable parameters:

```
PROGRAM param2 (OUTPUT);
VAR
    length,
    width,
    area,
    circum : INTEGER;
PROCEDURE compute (a, b : INTEGER;
                   VAR p, s : INTEGER);
BEGIN
    p := a * b;
    s := 2 * (a + b)
END;
BEGIN
    length := 40;
    width := 30;
    compute (length, width, area, circum);
    WRITELN ('Length        ',length);
    WRITELN ('Width         ',width);
    WRITELN ('Area          ',area);
    WRITELN ('Circumference ',circum)
END.
```

The program output follows:

```
Length         40
Width          30
Area         1200
Circumference 140
```

EXERCISES

1. Write a program to compute the regular and overtime pay given the hours worked and the hourly pay rate. Use a procedure to accept the hours worked

and pay rate as value parameters, and return regular pay and overtime pay as variable parameters. Display the regular pay, overtime pay, and gross pay for a person who works 47.5 hours at the regular rate of $9.25 per hour using time-and-a-half over 40 hours.

2. An ID number is used for unique identification. A check digit is added to each unique code number to reduce the risk of making erroneous entries. A simple scheme forms the sum of the other digits and lets the check digit be sum modulo 10. The 5-digit ID number 43175 is valid as the sum of the first four digits is 15 for which 15 modulo 10 gives the check digit 5 which appears as the rightmost digit. Write an interactive program that validates ID numbers using a procedure that accepts an ID number and returns a Boolean variable that is TRUE if the number is valid and FALSE if it is not. Test using the numbers 11327, 51842, 31395, and 78533.
3. Write a program that accepts a line of characters one character at a time and computes the number of vowels in the line. Use a procedure which accepts a character as a value parameter and adds the value 1 to a variable parameter, giving the cumulative count of the number of vowels.

6.4 FUNCTIONS

Functions are subprograms that return exactly one value to the calling program. Pascal contains numerous built-in functions. The following built-in functions are available with Alcor Pascal:

ABS(X)	Absolute value
SQR(X)	Square
SQRT(X)	Square root
LN(X)	Natural logarithm
EXP(X)	Natural exponential
SIN(X)	Sine
COS(X)	Cosine
ARCTAN(X)	Arctangent
ODD(X)	Odd
EOLN(f)	End of line
EOF(f)	End of file
TRUNC(X)	Truncate
ROUND(X)	Round
SUCC(X)	Successor

PRED(X)	Predecessor
ORD(X)	Ordinal number of X
CHR(X)	Value having ordinal number
LOCATION(X)	Location of variable
SIZE(X)	Size of type in bytes

Most built-in functions require one argument and return one result. The statement

```
root := SQRT(value);
```

places the square root of the contents of the argument value into the variable named root. The argument may be an expression. The statement

```
diag := SQRT (a*a + b*b);
```

places the square root of the sum of the squares of a and b into the variable diag. The result of the function may be part of a larger expression. The statement

```
diag := SQRT(SQR(a) + SQR(b) );
```

uses the built-in SQR function to square each of the variables a and b in the process of calculating the square root of the sum of the squares.

USER-DEFINED FUNCTIONS

The heading of a function begins with the keyword FUNCTION rather than PROCEDURE. The function must represent a value and must have a type identifier. A colon separates the type identifier from the rest of the heading. The function name from the heading appears at least once in the body of the function. The result is assigned to that name as if it were a variable.

The argument list for a function rarely includes variable parameters. The arguments are usually value parameters. If more than one result is needed, use a procedure with appropriate variable parameters or define a separate function for each result.

Computer scientists suggest that each module should perform only one task. This keeps the internal cohesion of each module high. Those program segments that return values are best thought of in terms of functions. There may be several inputs, but only one defined output for each module.

LENGTH OF THE DIAGONAL

The length of the diagonal of a right triangle is the square root of the sum of the squares of the lengths of the two sides of the right angle. The following program

uses the function DIAGONAL to compute the square root of the sum of the squares of the two arguments:

```
PROGRAM distance (OUTPUT);
VAR
    side1,
    side2,
    length : REAL;
FUNCTION diagonal (a, b : REAL) : REAL;
BEGIN
    diagonal := SQRT(a * a + b * b)
END;
BEGIN
    side1 := 3.0;
    side2 := 4.0;
    length := diagonal (side1, side2);
    WRITELN ('First side     ',side1:9:2);
    WRITELN ('Second side    ',side2:9:2);
    WRITELN ('Diagonal       ',length:9:2)
END.
```

The output of the program follows:

```
First side      3.00
Second side     4.00
Diagonal        5.00
```

BOOLEAN FUNCTION

The runction type may be any of the simple variable types including REAL, INTEGER, CHAR, and BOOLEAN. The following program includes a function TEST which returns the value TRUE if the first argument is evenly divisible by the second argument and FALSE otherwise:

```
PROGRAM number1 (OUTPUT);
VAR
    value1,
    value2 : INTEGER;
    result : BOOLEAN;
FUNCTION test (a, b : INTEGER) : BOOLEAN;
VAR
    remain : INTEGER;
BEGIN
    remain := a MOD b;
    test := remain = 0
END;
```

```
BEGIN
     value1 := 24;
     value2 := 8;
     result := test (value1, value2);
     WRITELN (value1, value2, 'Result = ',result);
     WRITELN (30, 7, ' Result = ', test(30,7) );
     WRITELN (60, 5, ' Result = ', test(60,5) )
END.
```

The program output follows:

```
24     8  TRUE
30     7  FALSE
60     5  TRUE
```

EXERCISES

1. Write a function that returns the integer value 1 if the real argument is positive, 0 if the argument is 0, and -1 if the real argument is negative. Write a program to test the function using the values 12.75, 3.69, -.76 -75.95, and 325.00.
2. A machine costing $12,500 has a service life of five years and no salvage value. Write a program to calculate the amount of depreciation for a given year for each of the following depreciation methods. Define the function "straight" to give the straight-line depreciation. Define the function "double" to give the double declining balance depreciation for the specified year. Define the function "sum" to use the sum-of-the-years digits method.
3. Write a program to compute the value .998 raised to the 40th power. Define the function POWER requiring two arguments. The first argument is real and is the value to be raised to the power given by the second argument which is an integer.

7 Arrays

OVERVIEW Pascal provides the ability of maintaining lists of values in variables. The values may be integers, real numbers, characters, Boolean values, or enumerated data types. The values are referenced by relative position within the list using a subscript. A simple list is indexed in one dimension only. More complex lists are indexed to two or more dimensions. A table having rows and columns is a two-dimensional structure.

□

7.1 NUMERIC VECTORS

In the language of mathematics, the vector X contains an ordered set of values $x_1, x_2, \ldots, x_n$. The value x_3 is the third element of the vector. The numbers to the lower right are subscripts. The symbol x_i designates the ith element of the array.

In computer science, a vector is a one-dimensional array. A cell is an element of the array. Arrays are not limited to the numerical values. The array may consist of real, integer, character or Boolean values, but all must be of the same type. Subscripts are usually integers, but they may be of any ordinal type including user-defined enumerated types.

ARRAY DEFINITIONS

There are several methods for defining arrays within a program. The statements

```
VAR
    data : ARRAY [1 .. 25] OF REAL;
```

define the variable data to be an array of 25 real values.

The array type definition may be in the TYPE section. The statements

```
TYPE
    vector = ARRAY [1 .. 25] OF REAL;
VAR
    data : vector;
```

define the variable data to be of type vector which is an array of 25 real values.

The subscript range may be a type defined earlier in the TYPE section. All types must be defined before they are used if Pascal is to retain its one-pass compilation capability. The statements

```
TYPE
    subscripts = 1 . . 25;
    vector = ARRAY [subscripts] OF REAL;
VAR
    data : vector;
```

illustrate this cascading of type definitions.

PASSING ARRAYS TO PROCEDURES AND FUNCTIONS

Large arrays should be passed to procedures and functions as variable parameters. This pass-by-reference method gives the subprogram access to the actual array itself. Even though the Pascal translator may permit passing arrays as value parameters, this is rarely done.

Defining the array type in the VAR section is possible. The statement

```
VAR
    data : ARRAY [1 . . 25] OF REAL;
```

illustrates this. Pascal translators do not usually accept this form in the heading of the procedure or function.

The array type must be given a name in the TYPE section. The statements

```
TYPE
    vector = ARRAY [1 . . 25] OF REAL;
VAR
    data : vector;
```

define the type vector which is then used in the procedure or function heading.

COMPUTE AVERAGE

The following program computes the average of a set of values after reading them into an array:

```
PROGRAM average1 (INPUT, OUTPUT);
VAR
    data : ARRAY[1 . . 25] OF REAL;
    number,
    index : INTEGER;
    sum,
    average : REAL;
BEGIN
    WRITELN;
```

```
    WRITE ('Number of observations ? ');
    READLN (number);
    WRITELN;
    WRITELN ('Value for');
    FOR index := 1 TO number DO
        BEGIN
            WRITE ('Obs ',index:2,' ? ');
            READLN (data[index])
        END;
    SUM := 0.0;
    FOR index := 1 TO number DO
        sum := sum + data[index];
    average := sum / number;
    WRITELN;
    WRITELN ('Sum       ',sum:9:2);
    WRITELN ('Average   ',average:9:2)
END.
```

The following output results from a test run:

```
Number of observations ? 5

Value for
Obs 1 ? 12.0
Obs 2 ? 11.0
Obs 3 ? 19.0
Obs 4 ? 14.0
Obs 5 ? 15.0

Sum         71.00
Average     14.20
```

ARRAY WITH PROCEDURE CALLS

It is better to break the program into modules rather than have the program do everything in one long main BEGIN/END block. This may mean passing the array to the procedure as a variable parameter. The following program uses one procedure to read the values into the array and the second procedure to calculate the sum and the average:

```
PROGRAM average2 (INPUT, OUTPUT);
TYPE
    vector = ARRAY [1 .. 25] OF REAL;
VAR
    number : INTEGER;
    sum,
    average : REAL;
    data : vector;
```

```
PROCEDURE getdata
    (VAR number : INTEGER;
    VAR values : vector);
VAR
    index : INTEGER;
BEGIN
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    WRITELN;
    WRITELN ('Value for');
    FOR index := 1 TO number DO
        BEGIN
            WRITE ('Obs ',index:2,' ? ');
            READLN (values[index] )
        END
END;
PROCEDURE compute
    (VAR number : INTEGER;
    VAR data : vector;
    VAR sum,
        avg : REAL);
VAR
    index : INTEGER;
BEGIN
    sum := 0.0;
    FOR index := 1 TO number DO
        sum := sum + data[index] ;
    avg := sum / number
END;
BEGIN   { Main routine }
    getdata (number, data);
    compute (number, data, sum, average);
    WRITELN;
    WRITELN ('Sum       ',sum:9:2);
    WRITELN ('Average   ',average:9:2)
END.
```

The output is the same as for the previous program.

The procedures "getdata" and "compute" are kept as independent of the main program and of each other as possible. All communication is through the parameter list. This method of programming makes it possible to write very large programs. Also, procedures developed for one program are easily integrated into other programs.

CHARACTER ARRAYS

Although numeric arrays are common, they are not the only types of arrays. Character arrays are also important. The following program establishes an array containing the vowels a, e, i, o, and u and uses the array in counting the number of vowels in an input message:

```
PROGRAM count1 (INPUT, OUTPUT);
VAR
    vowels : ARRAY [1 . . 5] OF CHAR;
    index,
    count : INTEGER;
    symbol : CHAR;
BEGIN
    vowels[1] := 'a';
    vowels[2] := 'e';
    vowels[3] := 'i';
    vowels[4] := 'o';
    vowels[5] := 'u';
    WRITELN;
    WRITE ('Message ? ');
    READ (symbol);
    count := 0;
    WHILE NOT EOLN DO
        BEGIN
            FOR index := 1 TO 5 DO
                IF symbol = vowels[index]
                    THEN count := count + 1;
            READ (symbol)
        END;
    WRITELN;
    WRITELN ('Number of vowels = ',count)
END.
```

Sample output from this program follows:

```
Message ? Now is the time for all good men
Number of vowels =      10
```

ORDINAL SUBSCRIPTS

The subscripts must be one of the ordinal types. They can be integer, character or user-defined enumerated types. The following program uses the names of the days of the week to determine the price that an item costs on that day:

```
PROGRAM price (OUTPUT);
TYPE
```

```
    days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
VAR
    price : ARRAY [days] OF REAL;
    day : days;
BEGIN
    price[Sun]  := 19.95;
    price[Mon]  := 18.45;
    price[Tue]  := 17.95;
    price[Wed]  := 18.35;
    price[Thu]  := 19.20;
    price[Fri]  := 17.15;
    price[Sat]  := 18.75;
    FOR day := Sun TO Sat DO
        WRITELN (price[day]:9:2)
END.
```

The output of this program follows:

```
19.95
18.45
17.95
18.35
19.20
17.15
18.75
```

EXERCISES

1. Write a program to read a set of integer values into an array. Compute and display the average, minimum, and maximum values.
2. Write a program that will read a set of integer values into an array maintaining the array in ascending order. Inserting the *k*th value into its proper position may require moving some values down to make room. Display the sorted list when finished.
3. Write a program to compute the product of the first 100 integers (1) (2) (3) . . . (100). This is a very large number. Create an array of 200 integers, letting each integer contain one of the digits. Perform the repeated multiplications digit by digit to form the product.

7.2 NUMERIC MATRICES

A two-dimensional array has a base type array of one dimension. It is an array of arrays. The following table represents unit sales figures for an item that comes in three colors and four model numbers:

Color	*Model number*			
	1	2	3	4
Red	114	219	174	256
White	193	247	225	318
Blue	109	183	196	230

There is a vector of unit sales figures for each color.

The following type specification describes the table as an array of arrays:

```
TYPE
    color = (Red, White, Blue);
    model = 1 .. 4;
    sales = ARRAY [model] OF INTEGER;
    table = ARRAY [color] OF sales;
VAR
    unitsale : table;
```

The variable "unitsale" is a table of integers. Each element of the table consists of an array of type "sales" which is a vector of integers. The subscripts Red, White, and Blue designate the color dimension. The subscripts 1, 2, 3, and 4 designate the model dimension. The color dimension is the row of the table and the model dimension is the column.

The type specification can be combined into one statement as in the following:

```
TYPE
    color = (Red, White, Blue);
    model = 1 .. 4;
    table = ARRAY [color] OF ARRAY [model] OF INTEGER;
VAR
    unitsale : table;
```

Multidimensional arrays are so common that there is a shorthand method of defining them. The type specification

```
TYPE
    color = (Red, White, Blue);
    model = (1 .. 4);
    table = ARRAY [color, model] OF INTEGER;
VAR
    unitsale : table;
```

gives the primary (row) dimension as color and the secondary (column) dimension as model.

The last variable declaration reduces to the following:

```
TYPE
    color = (Red, White, Blue);
```

```
    model = (1 . . 4);
VAR
    table = ARRAY [color, model] OF INTEGER;
```

A comma separates the dimensions of the array definition. Pascal allows more than two dimensions for problems having more than two categorizations.

BUILD A MATRIX

The following simple Pascal program illustrates the definition of a three-row by two-column matrix of integers:

```
PROGRAM print (OUTPUT);
VAR
    matrix : ARRAY [1 . . 3, 1 . . 2] OF INTEGER;
    i,
    j : INTEGER;
BEGIN
    matrix[1,1] := 12;
    matrix[1,2] := 15;
    matrix[2,1] := 11;
    matrix[2,2] := 17;
    matrix[3,1] := 19;
    matrix[3,2] := 18;
    FOR i := 1 TO 3 DO
        BEGIN
            FOR j := 1 TO 2 DO
                WRITE (matrix[i,j] );
            WRITELN;
        END
END.
```

The program output follows:

```
12   15
11   17
19   18
```

TABLES WITH ENUMERATED TYPE SUBSCRIPTS

The subscripts must be ordinal. They can be integer, character, or enumerated types. The following table gives low and high values for each of the four seasons of the year:

Season	***Sales limits***	
	Low	High
Spring	124	263
Summer	217	403

Fall	254	388
Winter	176	293

The following program uses the subscripts Spring, Summer, Fall, and Winter to designate the season and the subscripts Low and High to designate the limits of the table:

```
PROGRAM demand1 (OUTPUT);
TYPE
    season = (Spring, Summer, Fall, Winter);
    level = (Low, High);
    table = ARRAY [season, level] OF INTEGER;
VAR
    demand : table;
    row : season;
    column : level;
BEGIN
    demand[Spring,Low] := 124;
    demand[Spring,High] := 263;
    demand[Summer,Low] := 217;
    demand[Summer,High] := 403;
    demand[Fall,Low] := 254;
    demand[Fall,High] := 388;
    demand[Winter,Low] := 176;
    demand[Winter,High] := 293;
    FOR row := Spring TO Winter DO
        BEGIN
            FOR column := Low TO High DO
                WRITE (demand[row,column] );
            WRITELN;
        END
END.
```

The program output follows:

```
124   263
217   403
254   388
176   293
```

COLUMN SUMS AND ROW SUMS

The following program uses interactive data entry for the values of the table and then computes the column sums and row sums:

```
PROGRAM table1 (INPUT, OUTPUT);
VAR
    table : ARRAY [1 .. 20, 1 .. 30] OF REAL;
```

```
    nrows,
    ncolumns,
    row,
    col : INTEGER;
    sum : REAL;
BEGIN
    { Get data }
    WRITELN;
    WRITE ('Number of rows          ? ');
    READLN (nrows);
    WRITE ('Number of columns     ? ');
    READLN (ncolumns);
    WRITELN;
    WRITELN ('Enter value for');
    FOR row := 1 TO nrows DO
        BEGIN
            WRITELN;
            WRITELN ('Row ',row:2);
            FOR col := 1 TO ncolumns DO
                BEGIN
                    WRITE ('   Col  ',col:2,' ? ');
                    READLN (table[row,col] )
                END
        END;
    { Row sums }
    WRITELN;
    WRITELN ('Row          Sum');
    FOR row := 1 TO nrows DO
        BEGIN
            sum := 0.0;
            FOR col := 1 TO ncolumns DO
                sum := sum + table[row,col];
            WRITELN (row:3, sum:12:2)
        END;
    { Column sums }
    WRITELN;
    WRITELN ('Column       Sum');
    FOR col := 1 TO ncolumns DO
        BEGIN
            sum := 0.0;
            FOR row := 1 TO nrows DO
                sum := sum + table[row,col];
            WRITELN (col:3, sum:12.2)
        END
END.
```

Sample output from this program follows:

```
Number of rows      ? 2
Number of columns   ? 3
Enter value for
Row  1
     Col    1 ? 2.4
     Col    2 ? 1.7
     Col    3 ? 3.5
Row  2
     Col    1 ? 2.2
     Col    2 ? 1.8
     Col    3 ? 3.3
Row          Sum
  1          7.10
  2          7.30
Column       Sum
  1          4.60
  2          3.50
  3          6.80
```

This program illustrates methods for creating and using tables of values.

EXERCISES

1. Write a program that will read a set of exam scores into a table. Make the students correspond to the rows and the exams correspond to the columns. Compute the average grade for each exam. Compute the average grade for each student assigning the letter grades using boundaries of 90% for A, 80% for B, etc.
2. Write a program that computes the area and circumference (length of the perimeter) of several rectangles. Define a table having each row represent one rectangle and the columns consist of an ID number, length, width, area, and circumference. Read in the ID number, length, and width. Compute the area and circumference and store them in the table. Print the results when finished.

7.3 NONNUMERIC VECTORS AND MATRICES

Arrays are not limited to numerical values. They may contain data of any defined Pascal type. The most common types are REAL, INTEGER, and CHAR. The most common nonnumeric arrays contain character data.

READING AND WRITING CHARACTER STRINGS

A character string is an array of character data. The data must be read in one character at a time and stored in the array. The following program uses the READ command to read one line of input from the keyboard one character at a time and then prints the result:

```
PROGRAM char1 (INPUT, OUTPUT);
VAR
    line : ARRAY [1 .. 80] OF CHAR;
    number,
    index : INTEGER;
    symbol : CHAR;
BEGIN
    WRITELN;
    WRITE ('Message ? ');
    index := 0;
    REPEAT
        READ (symbol);
        index := index + 1;
        line[index] := symbol;
    UNTIL EOLN;
    number := index;
    WRITELN;
    FOR index := 1 TO number DO
        WRITE (line[index] );
    WRITELN
END.
```

Sample output from the program follows:

```
Message ? Now is the time
Now is the time
```

The program uses the variable index to place each character as it is read in. The REPEAT UNTIL loop terminates when the end of the line is reached. The number of characters in the input line is stored in the variable called number.

PRINTABLE SYMBOLS

The built-in Pascal function CHR(X) gives the character corresponding to the integer value X. The code numbers run from 0 to 255 for a total of 256 possible symbols. Many of the common printing symbols fall between 33 and 122. The following program uses the CHR function to generate the symbols for the character array:

```
PROGRAM char2 (OUTPUT);
```

```
VAR
    index : INTEGER;
    vector : ARRAY [33 .. 122] OF CHAR;
BEGIN
    FOR index := 33 TO 122 DO
        vector[index] := CHR(index);
    FOR index := 33 TO 80 DO
        WRITE (vector[index]);
    WRITELN;
    FOR index := 81 TO 122 DO
        WRITE (vector[index]);
    WRITELN;
END.
```

The output of this program follows:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]^_@abcdefghijklmnopqrstuvwxyz
```

The integer variable index is used twice in the same line

```
vector[index] := CHR(index);
```

It uses CHR(index) to generate the symbol designated by the value of index and places that character into the array vector using index as its subscript.

TWO-DIMENSIONAL CHARACTER ARRAYS

By using two-dimensional character arrays it is possible to store several character strings in a form that is easy to examine. If all lines are approximately the same length, a two-dimensional table works well. The following program stores 5 names, up to 20 characters each, in a 5-row by 20-column array of character data:

```
PROGRAM name1 (INPUT, OUTPUT);
VAR
    names : ARRAY [1 .. 5, 1 .. 20] OF CHAR;
    person,
    position : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Type names of 5 people');
    FOR person := 1 TO 5 DO
        BEGIN
            WRITE ('Name ? ');
            READ (names[person,1]);
            FOR position := 2 TO 20 DO
                IF EOLN THEN
                    names[person,position] := ' '
```

```
                ELSE
                    READ (names[person,position]);
        END;
    WRITELN;
    FOR person := 1 TO 5 DO
        BEGIN
            FOR position := 1 TO 20 DO
                WRITE (names[person,position]);
            WRITELN
        END
END.
```

Sample output from the program follows:

```
Type the names of 5 people

Name ? John Smith
Name ? Mary Allen
Name ? Mark Miller
Name ? Sue Keller
Name ? Henry Taylor
```

The input loop reads the names of the 5 people, one at a time. It fetches the characters, one at a time, and fills out the row of the character array with spaces after sensing the end-of-line symbol.

EXERCISES

1. Write a program to read in several lines of input. Compute the frequency count for each letter of the alphabet. Include both the uppercase and lowercase forms of the same letter.
2. Write a program to compute the average exam grade and assign a letter grade to each of several students. Use interactive data entry to supply the students' names and exam scores. Send the names, exam averages, and letter grades to the printer for a permanent record.
3. Write a program that accepts several lines of text and then counts the number of occurrences of each of the words "a", "the", and "of". Assume that words do not span from one line (row) to the next. Spaces, commas, semicolons, and periods are the only symbols separating words.
4. Write a program that accepts several lines of text and counts the frequencies for words of length 1, 2, 3, . . . , 25.

8 Advanced features

OVERVIEW Pascal includes several advanced features which make it a powerful programming language. These include the definition of records, pointers, files, and recursion. Beginners should build a sound foundation with the elementary features of Pascal before attempting complex projects using these features.

The applications section starting with Chapter 9 builds on the first section. The significance of Pascal's advanced features will be seen in the applications.

□

8.1 RECORDS AND POINTERS

Arrays are useful structures for storing quantities of data. The indexing feature with subscripts provides ready access to individual elements of the array. Pascal allows other varieties of structured data types, including sets and records.

SETS

A set is a collection of elements. Order is not important. An element either belongs in the set or it does not. Order is important in the array structure. An individual element is referenced by location within the array.

In Pascal, a set is a collection of scalar values of the same type. Each elementary object is called an element. A set is represented by enclosing its list of elements in brackets. The scalar values must be ordinal; they cannot be real. The set may contain integers, characters, or enumerated data types.

The TYPE and VAR sections of the Pascal program define SET specifications and declare SET variables. The sections

```
TYPE
    grades = (A, B, C, D, F);
VAR
    result : SET OF grades;
```

define the set "result" to contain elements from among the five letter grades.

Pascal assignment statements establish the contents of set variables. The following statement places the letter grades "A", "C", and "D" into the set called result:

```
result := [A, C, D];
```

An empty set contains no elements. The statement

```
result := [ ]
```

places no elements in the set, resulting in an empty set.

The universe is the set of all elements defined for that set type. The universe consists of all five letter-grades. A subset is a collection of grades taken from the universe. The order is unimportant, and an individual element appears only once. The empty set, also called null set, is a subset of every other set. One set is a subset of a second set if every element of the first set is also contained in the second set.

SET OPERATIONS

Pascal set operations include the following:

+	Union
*	Intersection
-	Difference

The union of two sets gives a set that contains those elements contained in at least one of the two sets. The intersection of two sets gives a set that contains those elements that are common to both sets. The difference between two sets gives those elements contained in the first set but not the second.

RELATIONAL OPERATORS

Relational operators give the Boolean values TRUE or FALSE. The following are the Pascal relational operators for sets:

IN	True if element is contained in the set
=	True if both sets contain same elements
<>	True if sets do not contain same elements
<=	True if first set is subset of second set
>=	True if first set is superset of second set

For the first set to be a superset of the second, the second set must be a subset of the first.

DAYS OF THE WEEK

The following program defines several sets using the letter grades and illustrates the use of set operations and relational operators:

```
PROGRAM set1 (OUTPUT);
TYPE
    grades = (A, B, C, D, F);
VAR
    universe,
    superior,
    above,
    middle,
    below,
    passing,
    failure : SET OF grades;
BEGIN
    superior := [A];
    above := [A, B];
    below := [D, F];
    middle := [B, C, D];
    passing := above + middle;
    universe := passing + below;
    failure := universe - passing;
    IF D IN passing THEN
        WRITELN ('D is passing grade')
    ELSE
        WRITELN ('D is not passing grade');
    IF [ ] = (above * below) THEN
        WRITELN ('Mutually exclusive')
    ELSE
        WRITELN ('Not mutually exclusive');
    IF middle <= passing THEN
        WRITELN ('Middle is subset of passing')
    ELSE
        WRITELN ('Middle is not subset of passing')
END.
```

The output from the program follows:

```
D is passing grade
Mutually exclusive
Middle is subset of passing
```

RECORDS

An array or a set may be used to store related information only if that information consists of a single data type. The Pascal record data type permits the storing of related information together if the elementary elements are not of the same type.

The following list contains an ID number, name, unit price, and quantity on hand for several inventory items:

Code Number	*Name*	*Unit price*	*Quantity on hand*
103	Alpha	19.95	346
117	Sublime	47.99	91
142	Extra	29.95	185

The Pascal program is to handle up to 20 inventory items. Each variable can be defined using its own array.

The following TYPE and VAR sections define the variables code, name, price, and quantity as arrays containing the information for up to 20 items:

```
TYPE
    index = 1 . . 20;
VAR
    code : ARRAY [index] OF INTEGER;
    name : ARRAY [index, 1 . . 10] OF CHAR;
    price : ARRAY [index] OF REAL;
    quantity : ARRAY [index] OF INTEGER;
    item : INTEGER;
```

Defining the subscript range in the TYPE section makes it easier to change the program later for a larger number of inventory items.

Each component of a Pascal record is called a field and has a data type associated with it. The following TYPE and VAR sections define a record type and use that record type definition in the declaration of a variable:

```
TYPE
    iteminfo = RECORD
            code : INTEGER;
            name : ARRAY [1 . . 10] OF CHAR;
            price : REAL;
            quantity : INTEGER
        END;
VAR
    item : iteminfo;
```

Item is a record variable. The variable item code references the code number of the current record. The variable item.price references its price, and item.quantity references the number of units on hand. The variable item.name references a string of 10 characters, giving the name of the current item.

Some Pascal compilers provide two ways of storing character and Boolean data types. The normal mode is usually the fastest but makes inefficient use of internal memory. The PACKED mode makes most efficient use of memory but is slower. The line

```
name : PACKED ARRAY [1 .. 10] OF CHAR
```

instructs the compiler to make the most efficient use of memory.

DYNAMIC DATA STRUCTURES

Dynamic data structures change size and even shape to meet the current requirements of the program. These changes occur at run time without having to recompile the program. Records are a primary form of dynamic data structure. Pointer variables provide access to the records.

POINTERS

Pointers are used to reference records. The predefined function NEW allocates the amount of space from the heap for one record and assigns the address of that memory region to the pointer. The Pascal statement

```
NEW (pointer);
```

allocates storage space for the code number, name, price, and quantity information for one inventory item. It assigns the address to the pointer variable.

The symbols ^ and @ designate pointers. The variable declaration

```
pointer :^item;
```

declares the variable called pointer for the record called item. Within the body of the program the statement

```
printer^.price := 19.95;
```

places the value 19.95 into the price field of the current record item.

The Pascal predefined function DISPOSE returns memory to the heap and deletes the current value of the pointer. The Pascal keyword NIL can be assigned to the pointer to indicate that it does not point to a valid record.

An array of pointers can serve to access a set of dynamic records that are stored internally at the same time. Alternately, one or more of the fields within a record may point to other records.

INVENTORY LIST

The following program places the information for the three inventory items into three dynamic records:

```
PROGRAM inv1 (OUTPUT);
```

```
TYPE
    iteminfo = RECORD
          code : INTEGER;
          name : PACKED ARRAY [1 .. 10] OF CHAR;
          price : REAL;
          quantity : INTEGER
       END;
    itemptr = ^iteminfo;
    listptr = ARRAY [1 .. 20] OF itemptr;
VAR
    pointer : itemptr;
    ptrlist : listptr;
    position,
    index : INTEGER;
BEGIN
    NEW (pointer);
    ptrlist[1] := pointer;
    pointer^.code := 103;
    pointer^.name[1] := 'A';
    pointer^.name[2] := 'l';
    pointer^.name[3] := 'p';
    pointer^.name[4] := 'h';
    pointer^.name[5] := 'a';
    FOR position := 6 TO 10 DO
        pointer^.name[position] := ' ';
    pointer^.price := 19.95;
    pointer^.quantity := 346;
    NEW (pointer);
    ptrlist[2] := pointer;
    pointer^.code := 117;
    pointer^.name[1] := 'S';
    pointer^.name[2] := 'u';
    pointer^.name[3] := 'b';
    pointer^.name[4] := 'l';
    pointer^.name[5] := 'i';
    pointer^.name[6] := 'm';
    pointer^.name[7] := 'e';
    FOR position := 8 TO 10 DO
        pointer^.name[position] := ' ';
    pointer^.price := 47.99;
    pointer^.quantity := 91;
    NEW (pointer);
    ptrlist[3] := pointer;
    pointer^.code := 142;
    pointer^.name[1] := 'E';
    pointer^.name[2] := 'x';
```

```
        pointer^.name[3] := 't';
        pointer^.name[4] := 'r';
        pointer^.name[5] := 'a';
        FOR position := 6 TO 10 DO
            pointer^.name[position] := ' ';
        pointer^.price := 29.95;
        pointer^.quantity := 185;
        FOR index := 1 TO 3 DO
            BEGIN
                pointer := ptrlist[index];
                WRITE (pointer^.code,'  ');
                FOR position := 1 TO 10 DO
                    WRITE (pointer^.name[position] );
                WRITE ('  ',pointer^.price:7:2);
                WRITELN ('  ',pointer^.quantity)
            END
END.
```

The following output results:

```
103   Alpha       19.95     346
117   Sublime     47.99      91
142   Extra       29.95     185
```

EXERCISES

1. Write a program that will use interactive data entry to obtain information for the inventory items. List the contents of the records complete with column headings.
2. Write a program that will place the name, three exam scores, average score, and letter grade for each of several students in a dynamic record. Create an array of pointers to access the records.

8.2 FILES

Simple variables, arrays, sets, and records store data during the execution of a program. They do not retain values from one run to the next. The data must be stored in an external file for use at a later time.

An internal file is a dynamic data structure that exists only during the execution of the program. An external file is saved on an external storage device such as tape or disk and is available for use at a later time.

The two standard external files are named INPUT and OUTPUT. Normally, the keyboard provides data to be read by the READ and READLN commands and the video screen serves as the output device for the WRITE and WRITELN commands. These default files are of type TEXT.

Alcor Pascal allows file redirection at run time without recompiling the program. Typing the ENTER key uses the default device. Typing :L for the OUTPUT file sends the output to the printer. Typing the name of a disk file enables the program to obtain the data from the disk file or store the output data in the disk file.

USING TEXT EDITOR TO CREATE INPUT FILE

The Alcor Pascal program editor is called using the name ED. This text editor can create data files just as it can Pascal source programs. Use the editor to create a data file called DATA1/DAT containing the following data:

```
5
12
11
19
14
15
```

The first line contains the number 5, indicating that there are $n = 5$ values in the file.

Use the following program to form the sum of the five data values 12, 11, . . . , 15, remembering to type the disk file name DATA1/DAT for the INPUT = prompt:

```
PROGRAM sum1 (INPUT, OUTPUT);
VAR
    number,
    count,
    value,
    sum : INTEGER;
BEGIN
    READLN (number);
    sum := 0;
    FOR count := 1 TO number DO
        BEGIN
            READLN (value);
            sum := sum + value;
        END;
    WRITELN ('Sum = ',sum)
END.
```

The output from this program follows:

```
Sum =    71
```

This program merely redirects the source of the standard INPUT file from the keyboard to the disk file. This is a powerful feature that makes the use of interactive programs practical with large data files.

USER-DEFINED FILES

User defined files of type TEXT follow the same rules as the INPUT and OUTPUT files. Disk files generated by the Pascal program can later be printed. Disk files created by another Pascal program as TEXT files or by a text editor can be read in as if they came from the keyboard.

The following program defines an external file from which the program obtains its data:

```
PROGRAM sum2 (diskfile, OUTPUT);
VAR
    number,
    count,
    value,
    sum : INTEGER;
    diskfile : TEXT;
BEGIN
    RESET (diskfile);
    READLN (diskfile, number);
    sum := 0;
    FOR count := 1 TO number DO
        BEGIN
            READLN (diskfile, value);
            sum := sum + value
        END;
    WRITELN ('Sum = ',sum)
END.
```

The output from this program follows:

```
Sum =    71
```

The RESET procedure initializes the disk data file. Each READLN command references the Pascal file name as its first parameter.

USING PASCAL PROGRAMS TO CREATE DISK DATA FILE

Rather than use a text editor to creat the initial disk data file, it is possible to write an interactive Pascal program that obtains the values for the disk data file from the keyboard. The following program uses this approach:

```
PROGRAM create1 (INPUT, OUTPUT, diskfile);
VAR
    number,
    count,
    value : INTEGER;
    diskfile : TEXT;
```

```
BEGIN
    REWRITE (diskfile);
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    WRITELN (diskfile, number);
    WRITELN;
    WRITELN ('Value for');
    FOR count := 1 TO number DO
        BEGIN
            WRITE ('Obs ',count:3,' ? ');
            READLN (value);
            WRITELN (diskfile, value)
        END
    WRITELN;
    WRITELN ('End of data entry')
END.
```

The sample terminal session for this program follows:

```
Number of observations ? 5
Value for
Obs   1 ? 12
Obs   2 ? 11
Obs   3 ? 19
Obs   4 ? 14
Obs   5 ? 15
```

NONTEXT FILES

All nontext files are read and written in binary form. This is an internal form that may be more compact than text files and much faster since data conversions may not be needed between internal and external forms. All components of nontext files must be of the same type. The type may be simple or structured. The structured type may not be of type FILE or a structured type containing a component of type FILE.

Only the READ and WRITE commands are permitted. The READLN and WRITELN commands are available only for text files. Care should be taken not to read past the end of the file. This may involve using the EOF function or using a prior count of the number of components. If a structured record has been written to the external file with one WRITE command, then it can be read into a record of the same type using one READ command.

FILE OF REALS

The following programs write and then read a file of real values using the EOF function:

```
PROGRAM create2 (INPUT, OUTPUT, diskfile);
VAR
    value : REAL;
    diskfile : FILE of REAL;
BEGIN
    REWRITE (diskfile);
    WRITELN;
    WRITELN ('Create a file of real values');
    WRITELN ('using the value -999.00 to');
    WRITELN ('terminate data entry.');
    WRITELN;
    REPEAT
        WRITE ('Next value ? ');
        READLN (value);
        IF value <> -999.00 THEN
            WRITE (diskfile, value)
    UNTIL value = -999.00;
    WRITELN;
    WRITELN ('End of data entry')
END.
```

A sample terminal session for this program follows:

```
Create a file of real values
using the value -999.00 to
terminate the data entry.

Next value ? 2.3
Next value ? 3.1
Next value ? 2.8
Next value ? -999.00

End of data entry
```

The following program reads the data file until the end of file is reached and displays the sum of the values:

```
PROGRAM sum4 (OUTPUT, diskfile);
VAR
    value,
    sum : REAL;
    diskfile : FILE OF REAL;
```

```
BEGIN
    RESET (diskfile);
    WRITELN;
    WRITELN ('Compute the sum of the real');
    WRITELN ('values in a disk data file.');
    sum := 0.0;
    REPEAT
        READ (diskfile, value);
        sum := sum + value
    UNTIL EOF (diskfile);
    WRITELN;
    WRITELN ('Sum = ',sum:9:2)
END.
```

Sample output of the program follows:

```
Compute the sum of the real
values in a disk data file.
Sum =      8.20
```

FILE OF RECORDS

The following program uses interactive data entry to create a file of inventory item information:

```
PROGRAM inv2 (INPUT, OUTPUT, diskfile);
TYPE
    iteminfo = RECORD
            code : INTEGER;
            name : PACKED ARRAY [ 1 .. 10] OF CHAR;
            price : REAL;
            quantity : INTEGER
        END;
VAR
    item : inteminfo;
    diskfile : FILE OF iteminfo;
    number,
    position,
    count : INTEGER;
BEGIN
    REWRITE (diskfile);
    WRITELN;
    WRITELN ('Create file of inventory information.');
    WRITELN ('Use code number of 0 to terminate.');
    WRITELN;
    WRITE ('Number of inventory items ? ');
    READLN (number);
```

```
    FOR count := 1 TO number DO
        BEGIN
            WRITELN;
            WRITELN ('Item ',count:3);
            WRITE ('Code number ? ');
            READLN (item.code);
            WRITE ('Name        ? ');
            READ (item.name[1] );
            FOR position := 2 TO 10 DO
                IF NOT EOLN THEN
                    READ (item.name[position] )
                ELSE
                    item.name[position] := ' ';
            WRITE ('price       ? ');
            READLN (item.price);
            WRITE ('Quantity    ? ');
            READLN (item.quantity);
            WRITE (diskfile, item)
        END;
    WRITELN;
    WRITELN ('End of program')
END.
```

A sample terminal session for this program follows:

```
Create file of inventory information.
Use code number of 0 to terminate.

Number of inventory items ? 3

Item     1
Code number ? 103
Name        ? Alpha
Price       ? 19.95
Quantity    ? 346

Item     2
Code number ? 117
Name        ? Sublime
Price       ? 47.99
Quantity    ? 91

Item     3
Code number ? 142
Name        ? Extra
Price       ? 29.95
Quantity    ? 185

End of program
```

This program creates the inventory file.

The following program reads the inventory file and lists the contents to the normal output file:

```
PROGRAM inv3 (OUTPUT, diskfile);
TYPE
    iteminfo = RECORD
          code : INTEGER;
          name : ARRAY [1 . . 10] OF CHAR;
          price : REAL;
          quantity : INTEGER
       END;
VAR
    item : iteminfo;
    diskfile : FILE OF iteminfo;
    position : INTEGER;
BEGIN
    RESET (diskfile);
    WRITELN;
    WRITELN ('Code    Name    Price    Quantity');
    WRITELN ('number');
    REPEAT
        READ (diskfile, item);
        WRITE (item.code:6, ' ');
        FOR position := 1 TO 10 DO
            WRITE (item.name[position] );
        WRITE ('    ',item.price:10.2);
        WRITELN ('    ',item.quantity)
    UNTIL EOF (diskfile)
END.
```

Sample output from the program follows:

```
Code      Name       Price      Quantity
number
   103    Alpha      19.95       346
   117    Sublime    47.99        91
   142    Extra      29.95       185
```

EXERCISES

1. Write a program that will compute the average of a set of real values stored in a disk data file. Use a text editor to create the disk data file.
2. Write a program that will use one disk data file for input containing an ID number, length, and width of a rectangle. Have the program compute the area

and circumference. It should output a disk data file that contains the ID number, length, width, area, and circumference for each rectangle.

3. Write a program that reads the extended file created by the program of exercise 2 and lists the information to the normal output device.

8.3 RECURSION

Recursion is a capability of Pascal that sets it apart from many other programming languages. A recursive function or procedure is one that calls itself. This is usually a direct call. It may be an indirect call by way of another subroutine. Recursion is forbidden in many languages, including FORTRAN and BASIC.

PASSING ARGUMENTS

The two types of arguments for functions and procedures are value parameters and variable parameters. Variable parameters pass the addresses of the variables to the subroutine which can then modify those values in place. Value parameters pass the values to the subroutine which cannot change the actual values in the calling procedure.

PARAMETER STACK

When a procedure or function is called, the value parameters are saved in an area for use by that subroutine. This is done so that any changes made to the value parameters will not be carried back to the calling procedure. If the procedure calls itself, the new value parameters are saved in a new parameter save area. The parameter save areas form a push-down stack that follows a last-in-first-out policy similar to cafeteria trays.

NESTED RECURSIVE CALLS

Successive recursive calls of a procedure to itself result in a deeper and deeper nesting of calls. Unless there is a well-defined stopping point, the parameter save-area memory requirements will exceed available memory, and the system will crash. The recursive call must be part of an IF statement, and the processing steps of the procedure must eventually force a stop to the recursive calls.

Once the recursive calls have stopped, the program will extricate itself from the most nested level by working back to the top level. The Pascal system automatically manages the stack of value parameters. One of the main applications of recursion is stack management.

LOOPING AND RECURSION

Looping and recursion are similar in nature. Looping is done within the procedure itself. Recursion involves the procedure calling itself. Recursion automatically

maintains a stack of value parameters. Many simple programs can be handled either with looping or by recursion.

PRINTING A LINE OF ASTERISKS

Consider the problem of printing a line of asterisks. The following program uses a FOR loop with a loop control variable to generate a line of 25 asterisks:

```
PROGRAM aster1 (OUTPUT);
VAR
    count,
    number : INTEGER;
BEGIN
    number := 25;
    FOR count := 1 TO number DO
        WRITE ('*');
    WRITELN
END.
```

The output from the program follows:

```
*************************
```

The program using the looping mechanism is short and to the point.

The recursive version is a little harder to understand, but it does illustrate recursion. The following program uses recursive calls to the procedure NEXT:

```
PROGRAM aster2 (OUTPUT);
VAR
    number : INTEGER;
PROCEDURE next (number : INTEGER);
BEGIN
    number := number - 1;
    IF number > 0 THEN
       next (number);
    WRITE ('*')
END;
BEGIN
    number := 25;
    next (number);
    WRITELN
END.
```

The output from the program follows

```
*************************
```

The program works its way into deeper and deeper nested recursive calls, saving the current value of the parameter on the stack. The procedure subtracts

the value 1 from its argument and calls itself again if the result is greater than 0. The stack of argument values builds in the order 25, 24, 23, ... 1. As the program works from the nested recursive calls it prints an asterisk each time. As it pops from level to level out of the nested calls, the argument values pop from the stack in the order 1, 2, 3, . . . , 25.

For simple programs like this the looping approach is clearer. Recursion involves the procedure calling itself, and this is more difficult to understand. The payoff from recursion comes from its maintenance of the arguments as a push-down stack.

EXERCISES

1. Write a program using the looping approach to compute the value of 6 factorial.
2. Write a program using the recursive approach to compute the value of 6 factorial.
3. Write a recursive program that prints all of the Fibonacci numbers less than the value 1,000 in ascending order.
4. Write a recursive program that prints all of the Fibonacci numbers less than 1,000 in descending order.

II

Applications

9 Elementary mathematics

OVERVIEW One of the first applications for computers was to remove the drudgery from numerical calculation. Quantitative applications are still among the foremost uses of the computer.

Charles Babbage envisoned automatic calculating engines capable of generating and printing mathematical tables. The early electronic computers designed during World War II automatically calculated tables of numbers.

The current trend is to use computers to provide graphical output representing numerical quantities and their patterns. Increasing use of video display devices has fostered this application.

Solving equations and systems of equations plus maximization and minimization are important applications of mathematics. This chapter illustrates the use of the computer and the Pascal language in elementary mathematics.

□

9.1 TABULATING FUNCTIONS

Charles Babbage designed, but was unable to build, an ambitious machine he called an analytical engine that could generate and automatically print tables of numbers. The machine had a mill for doing the calculations and a store for storing values. It followed a program of instructions.

Generating mathematical tables is still an important application of computers. These are among the easiest applications to visualize and program.

TABLE OF SQUARES

The function

$$y = f(x) = x^2$$

computes the value y as the square of the value x. The value y is the result of the function. The value x is its argument. When $x = 0$, the result is $y = 0$. When $x = 2$, the result is $y = 4$. The following program evaluates the function for $x = -2, -1.5, \ldots, 2$:

```
PROGRAM tab1 (OUTPUT);
{ Tabulate squares }
VAR
    x,              { Argument for function }
    y,              { Result of function }
    a,              { Lower limit for argument }
    w               { Interval width for arguments }
        : REAL;
    number,         { Number of intervals }
    count           { Loop counter variable }
        : INTEGER;
BEGIN
{ Initial message }
WRITELN;
WRITELN ('Generate table for');
WRITELN ('function x squared');
{ Initialize parameters }
a := -2.0;
w := 0.5;
number := 9;
{ Generate table }
WRITELN;
WRITELN ('      Value      Squared');
x := a - w;
FOR count := 1 TO number DO
    BEGIN
        x := x + w;
        y := x * x;
        WRITELN (x:10:2, y:10:2)
    END
END.
```

The program output follows:

```
Value      Squared
-2.00         4.00
-1.50         2.25
-1.00         1.00
-0.50         0.25
 0.00         0.00
 0.50         0.25
 1.00         1.00
 1.50         2.25
 2.00         4.00
```

EVALUATE POLYNOMIAL

Polynomials are functions of one argument obtained by summing various powers of that argument. The function

$$y = f(x) = .25x^3 - x^2 + 3x + 4$$

is a polynomial of degree 3. The largest power of x is 3. The following program evaluates the polynomial for $x = 0, .5, \ldots, 2.5$:

```
PROGRAM tab2 (OUTPUT);
{ Tabulate polynomial }
VAR
    x,              { Argument for function }
    y,              { Result of function }
    a,              { Lower limit of argument }
    w               { Interval width for arguments }
        : REAL;
    number,         { Number of intervals }
    count           { Loop counter variable }
        : INTEGER;
    BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Generate table for');
    WRITELN ('polynomial');
    { Initialize parameters }
    a  := 0.0;
    w := 0.5;
    number := 6;
    { Generate table }
    WRITELN;
    WRITELN ('      Value      Polynomial result');
    x := a - w;
    FOR count :=1 TO number DO
        BEGIN
            x := x + w;
            y := 0.25 * x * x * x - x * x + 3.0 * x + 4.0;
            WRITELN (x:10:2, y:10:2)
        END
END.
```

Output from the program follows:

```
Value     Polynomial result
0.00            4.00
0.50            5.28
```

1.00	6.25
1.50	7.09
2.00	8.00
2.50	9.16

NATURAL AND BASE 10 LOGARITHMS

Every positive real number has a corresponding logarithm. Values between 0 and 1 have negative logarithms. Values greater than 1 have positive logarithms. The logarithm of 1 is 0. The natural logarithms use the mathematical constant $e = 2.71828\ldots$ as the base. Base 10 logarithms, also called common logarithms, use 10 as the base.

Many programming languages, including Alcor Pascal, provide only natural logarithms. Dividing the natural logarithm by the base logarithm of 10 gives the base 10 logarithm. The following program generates a table giving both the natural and common logarithms of a range of values defined at run time:

```
PROGRAM tab3 (INPUT, OUTPUT);
{ Tabulate natural and base 10 logarithms }
VAR
    arg,          { Argument for function }
    loge,         { Natural logarithm }
    log10,        { Base 10 logarithm }
    start,        { Starting value for table }
    width,        { interval width for arguments }
    loge10        { Natural logarithm of 10 }
      : REAL;
    number,       { Number of intervals }
    count         { Loop counter variable }
      : INTEGER;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Generate table for');
    WRITELN ('natural and base 10');
    WRITELN ('logarithms');
    { Initialize parameters }
    WRITELN;
    WRITE ('Starting value for argument ? ');
    READLN (start);
    WRITE ('Interval width for argument ? ');
    READLN (width);
    WRITE ('Number of computed values ? ');
    READLN (number);
    { Generate table }
```

```
    WRITELN;
    WRITELN ('Argument      Natural log      Base 10 log');
    arg := start - width;
    loge10 := LN(10.0);
    FOR count := 1 TO number DO
        BEGIN
            arg := arg + width;
            loge := LN(arg);
            log10 := loge / loge10;
            WRITELN (arg:10:2, loge:12:5, log10:12:5)
        END
END.
```

Sample output from this program follows:

```
Generate table for
natural and base 10
logarithms

Starting value for argument  ? 1.0
Interval width for argument  ? 1.0
Number of computed values ? 10

Argument      Natural log      Base 10 log
    1.00          0.00000          0.00000
    2.00          0.69315          0.30103
    3.00          1.09861          0.47712
    4.00          1.38629          0.60206
    5.00          1.60944          0.69897
    6.00          1.79176          0.77815
    7.00          1.94591          0.84510
    8.00          2.07944          0.90309
    9.00          2.19722          0.95424
   10.00          2.30259          1.00000
```

TABULATE SINE FUNCTION

Trigonometric functions play an important role in mathematics, physics, and engineering. The complete circle represents an angle of 360 degrees. The angle is also given as 2π radians. The constant $\pi = 3.14159\ldots$ is common in mathematics. Most built-in trigonometric functions measure degrees in radians. The following program computes $\sin(x)$ for $x = 0, .2, \ldots, 1.6$ radians:

```
PROGRAM tab4 (OUTPUT);
{ Tabulate sine function }
VAR
    start,          { Starting value for argument }
    width,          { Interval width for arguments }
```

```
    arg,            { Argument for sine function }
    result          { Result of sine function }
        : REAL;
    number,         { Number of iterations }
    count           { Counter for loop control }
        : INTEGER;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Generate table for');
    WRITELN ('sine function');
    { Problem parameters }
    start := 0.0;
    width := 0.2;
    number := 9;
    { Generate table }
    arg := start - width;
    WRITELN;
    WRITELN ('Argument      Sine');
    FOR count := 1 TO number DO
        BEGIN
            arg := arg + width;
            result := SIN(arg);
            WRITELN (arg:10:2, result:10:5)
        END
END.
```

Output from the program follows:

```
Argument      Sine
    0.00      0.00000
    0.20      0.19867
    0.40      0.38942
    0.60      0.56464
    0.80      0.71736
    1.00      0.84147
    1.20      0.93204
    1.40      0.98545
    1.60      0.99957
```

ACCELERATION, VELOCITY, AND DISTANCE

An object under constant acceleration ***a*** for time ***t*** attains a velocity

$v = at$

and a distance

$d = .5at^2$

at the end of the time. There are 3,600 seconds in an hour and 5,280 feet in a mile. The ratio 360/528 adjusts feet per second to miles per hour. The following program computes the velocity in feet per second and miles per hour and the distance in feet for each of the first few seconds:

```
PROGRAM tab5 (INPUT, OUTPUT);
{ Tabulate velocity and distance }
VAR
    time,       { Time under acceleration }
    maxtime,    { Upper limit for time }
    accel,      { acceleration in feet/sec/sec }
    velocity,   { Velocity in feet/sec }
    mph,        { Velocity in miles per hour }
    distance    { Distance in feet }
      : INTEGER;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Generate table giving velocity');
    WRITELN ('in feet per second, velocity in');
    WRITELN ('miles per hour, and distance in');
    WRITELN ('feet for object under constant');
    WRITELN ('acceleration.');
    { Problem parameters }
    WRITE ('Acceleration in feet/sec/sec ? ');
    READLN (accel);
    WRITE ('Maximum time for table    ? ');
    READLN (maxtime);
    { Generate table }
    WRITELN;
    WRITE ('Time      Velocity      ');
    WRITELN ('velocity      Distance');
    WRITE ('seconds   feet/sec      ');
    WRITELN ('miles/hour      feet');
    FOR time := 1 TO maxtime DO
        BEGIN
            velocity := accel * time;
            mph := (velocity * 360) DIV 528;
            WRITELN (time:5,velocity:10,mph:15,distance:12)
        END
END.
```

Sample output from this program follows:

```
Generate table giving velocity
in feet per second, velocity in
miles per hour, and distance in
feet for object under constant
acceleration.

Acceleration in feet/sec/sec ? 8
Maximum time for table    ? 10
```

Time seconds	Velocity feet/sec	Velocity miles/hour	Distance feet
1	8	5	4
2	16	10	16
3	24	16	36
4	32	21	64
5	40	27	100
6	48	32	144
7	56	38	196
8	64	43	256
9	72	49	324
10	80	54	400

EXERCISES

1. Write a program to generate e^x for $x = 0, 1, 2, \ldots, 10$.
2. Write a program to evaluate $\cos(x)$ for $x = 0, .1, \ldots, 1$ radians.
3. Write a program to evaluate the trigonometric function $\tan(x)$ computed as $\sin(x)/\cos(x)$ for $x = 0, .1, \ldots, 1$ radians.

9.2 PLOTTING FUNCTIONS

Increased use of video display terminals and microcomputers has resulted in an increased emphasis on visually enhanced output in the form of graphs. The most primitive graphs are called typewriter plots. These are generated one line at a time using normal printing symbols. The asterisk and plus sign are the most common plotting symbols since they are symmetric and display in the center of the character position.

PLOT OF FUNCTION X SQUARED

The easiest way to generate the plot of a function is to plot the location of $f(x)$ on the horizontal axis and generate the values of the argument x along the vertical axis. This is the reverse of the normal practice in elementary mathematics. It requires turning the printed graph on its sides for normal orientation.

The following program generates a plot of the function $f(x) = x^2$ for $x = -2, -1.5, \ldots, 2$:

```
PROGRAM plot1 (OUTPUT);
{ Plot function x squared }
VAR
    x,              { Argument for function }
    y,              { Result of function }
    a,              { Lower limit for argument }
    w               { Interval width for arguments }
        : REAL;
    number,         { Number of intervals }
    count,          { Loop counter variable }
    count2,         { Loop counter variable }
    position        { Position on line for plot }
        : INTEGER;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Generate plot for');
    WRITELN ('function X squared');
    { Initialize parameters }
    a  := -2.0;
    w := 0.5;
    number := 9;
    { Generate plot  }
    x  := a - w;
    FOR count := 1 TO number DO
        BEGIN
            x := x + w;
            y := x * x;
            position := round(1.0 + 10.0 * y);
            FOR count2 := 1 TO position DO
                WRITE (' ');
            WRITELN ('*')
        END
END.
```

The output from this program follows:

```
Generate plot for
function x squared
                                                  *
                                *
                    *
          *
*
          *
                    *
                                *
                                                  *
```

The minimum value of $f(x)$ is 0 which is the result of 0^2. The maximum value of $f(x)$ is 4 resulting from both $(-2)^2$ and 2^2. Multiplying $f(x)$ by 10 scales the result on the print line so that the plot fits within 40 print positions. Adding the value 1 to $40f(x)$ prints the asterisk in column 1 when $f(x) = 0$.

This program prints a number of spaces using the WRITE (' ') command. It finishes the line with the WRITELN ('*') command to place the plotting symbol where it belongs.

PLOT POLYNOMIAL USING LINE BUFFER

A second approach uses a line buffer to contain the plotting symbols in their desired printing positions. This method is necessary when plotting two or more functions on the same graph. The following program uses this approach to plot the values of the function

$$f(x) = .25x^3 - x^2 + 3x + 4$$

for $x = 0, .5, \ldots, 3$:

```
PROGRAM plot2 (OUTPUT);
{ Plot polynomial }
VAR
    x,              { Argument for function }
    y,              { Result of function }
    a,              { Lower limit for argument }
    w               { Interval width for arguments }
      : REAL;
    number,         { Number of intervals }
    count,          { Loop counter variable }
    index,          { Counter for plot line }
    position        { Position in plot line }
      : INTEGER;
    line            { Line for plot }
      : ARRAY [1 .. 60] OF CHAR;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Generate plot for');
    WRITELN ('polynomial');
    { Initialize parameters }
    a := 0.0;
    w := 0.5;
    number := 6;
    { Plot function }
    x := a - w;
    FOR count := 1 TO number DO
```

```
        BEGIN
            x := x + w;
            y := 0.25 * x * x * x - x * x + 3.0 * x + 4.0;
            position := ROUND(10.0 * y) - 39;
            FOR index := 1 TO 60 DO
                line[index] := ' ';
            line[position]   := '*';
            FOR index := 1 TO 60 DO
                WRITE (line[index]);
            WRITELN
        END
END.
```

The output from this program follows:

```
Generate plot for
polynomial
*
                *
                        *
                                *
                                        *
                                                *
```

PLOT OF NATURAL LOGARITHMS

Rather than locate one plotting symbol in its proper position on the line, some prefer plotting the symbols from the lower limit of the graph to the position representing the value of the function. This more clearly delineates the relative values pictured. The following program uses this method to plot the natural logarithms of the values 1, 2, . . . , 10:

```
PROGRAM plot3 (INPUT, OUTPUT);
{ Plot natural logarithms }
VAR
    arg,          { Argument for function }
    loge,         { Natural logarithm }
    start,        { Starting value for table }
    width         { Interval width for arguments }
      : REAL;
    number,       { Number of intervals }
    count2,       { Loop counter variable for line }
    length,       { Length of line for plot }
    count         { Loop counter variable for function }
      : INTEGER;
BEGIN
    { Initial message }
```

```
    WRITELN;
    WRITELN ('Generate plot for');
    WRITELN ('natural logarithms');
    { Initialize parameters }
    WRITELN;
    WRITE ('Starting value for argument  ? ');
    READLN (start);
    WRITE ('Interval width for argument  ? ');
    READLN (width);
    WRITE ('Number of computed values ? ');
    READLN (number);
    { Generate plot  }
    arg := start - width;
    FOR count := 1 TO number DO
        BEGIN
            arg := arg + width;
            loge := LN(arg);
            length := 1 + ROUND(15.0 * loge);
            FOR count2 := 1 TO length DO
                WRITE ('*');
            WRITELN
        END
END.
```

The output from the program follows:

```
Generate plot for
natural logarithms

Starting value for argument  ? 1.0
Interval width for argument  ? 1.0
Number of computed values ? 10
*
**********
******************
**************************
******************************
**********************************
**************************************
******************************************
**********************************************
**************************************************
```

PLOT SINE FUNCTION USING PAGE BUFFER

It is also possible to define a screen buffer in the same manner used for defining one print line. The screen buffer is a two-dimensional table containing as many

rows as desired for the display and as many columns as needed. The TRS-80 Model III has a video screen containing 16 lines and 64 columns. The TRS-80 Model IV uses the industry standard video screen size of 24 lines by 80 columns.

Using a two-dimensional character array is the most flexible. There is complete freedom to locate printing symbols anywhere in the table. The table is printed character by character.

The following program uses this method to plot a portion of a sine function for $x = -2, -1.7, \ldots$:

```
PROGRAM plot4 (OUTPUT);
{ Plot sine function }
VAR
    start,          { Starting value for argument }
    width,          { Interval width for arguments }
    arg,            { Argument for sine function }
    result          { Result of sine function }
      : REAL;
    number,         { Number of iterations }
    count,          { Counter for loop control }
    row,            { Row index }
    col             { Column index }
      : INTEGER;
    page            { Two dimensional page for plot }
      : ARRAY [1 .. 15, 1 .. 62] OF CHAR;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Generate plot for');
    WRITELN ('sine function');
    { Problem parameters }
    start := -2.0;
    width := 0.3;
    number := 15;
    { Generate page }
    arg := start - width;
    FOR row := 1 TO 15 DO
        FOR col := 1 TO 62 DO
            page[row,col] := ' ';
    FOR count := 1 TO number DO
        BEGIN
            arg := arg + width;
            result := SIN (arg);
            row := count;
            col := ROUND(30.0 * (1.0 + result) ) + 1;
            page[row,col] := '*'
        END;
```

```
    FOR row := 1 TO 15 DO
        BEGIN
            FOR col := 1 TO 62 DO
                WRITE (page[row,col]);
            WRITELN;
        END;
    WRITE ('TYPE THE VALUE 1 TO CONTINUE ');
    READLN (NUMBER);
END.
```

The output from this program follows:

```
Generate plot for
sine function
     *
*
*
     *
          *
               *
                    *
                         *
                              *
                                   *
                                        *
                                         *
                                          *
                                        *
```

```
Type the value 1 to continue 1
```

EXERCISES

1. Write a program to plot the function e^x for $x = 0, 1, 2, \ldots, 10$.
2. Write a program to plot the function $\cos(x)$ for $x = 0, .1, \ldots, 1$ radians.
3. Write a program to plot the trigonometric function $\tan(x)$ computed as $\sin(x)/\cos(x)$ for $x = .1, .15, \ldots, .6$.
4. Write a program that superimposes plots for both $\sin(x)$ and $\cos(x)$ using '*' for $\sin(x)$ and '+' for $\cos(x)$ for $x = -2, -1.8, \ldots, 1$.

9.3 SOLVING EQUATIONS

Another important application of mathematics is solving equations. For what values of x is the equation

$$4x^2 = 100$$

valid? In this case the solutions are $x = 5$ and $x = -5$. The equation

$$4x^2 - 100 = 0$$

is equivalent since it results from subtracting the value 100 from both sides of the equal sign. The function

$f(x) = 4x^2 - 100$

is the function of interest. Solving for zeros of that function involves solving for those values of x for which the equation

$f(x) = 0$

is valid. The solution values of x are called the roots of the equation.

The first example of this section sets the stage by solving a simple linear equation $ax = b$ for its unknown value of x. The second example solves the quadratic equation $ax^2 + bx + c = 0$ for its real roots, if they exist. The third, fourth, and fifth examples illustrate methods of searching for roots of equations one at a time.

SIMPLE LINEAR EQUATION

The equation

$ax = b$

is a simple linear equation. The equation

$ax - b = 0$

is equivalent. The function

$f(x) = ax - b$

describes the function as a function of x. Solving the equation is equivalent to solving for the root of the function $f(x)$. This means finding the value of x for which the equation

$f(x) = 0$

is valid.

The following program solves simple linear equations:

```
PROGRAM solve1 (INPUT, OUTPUT);
{ Solve simple linear equation }
VAR
    a,          { Coefficient a }
    b,          { Constant b }
    x           { Unknown x }
      : REAL;
BEGIN
    { Initial message }
    WRITELN;
```

```
    WRITELN ('Solve simple linear equation');
    { Problem parameters }
    WRITELN;
    WRITE ('Coefficient a ? ');
    READLN (a);
    WRITE ('Constant b   ? ');
    READLN (b);
    { Solve for x }
    IF a <> 0.0 THEN
        x := -(b / a);
    { Display result }
    IF a = 0.0 THEN
        WRITELN ('There is no solution if a = 0.0')
    ELSE
        WRITELN ('Value of x = ',x:9:4)
END.
```

The following test run solves the equation $2.5x - 30 = 0$:

```
Solve simple linear equation

Coefficient for a ? 2.5
Constant b        ? -30.0
Value of x =       12.0000
```

The root is at $x = 12$. The equation $2.5x - 30 = 0$ is true when $x = 12$.

QUADRATIC EQUATIONS

The function

$$f(x) = ax^2 + bx + c$$

is quadratic since it has 2 as its largest power of x. The equation

$$f(x) = ax^2 + bx + c = 0$$

usually has two roots. The equation

$$x = (-b \pm \sqrt{(b^2 - 4ac)})/a$$

gives the roots of the equation. The roots are real if the expression

$$b^2 - 4ac$$

is positive. If the expression is negative, the roots are complex.

The following program solves for the real roots of quadratic equations:

```
PROGRAM solve2 (INPUT, OUTPUT);
{ Solve quadratic equation }
VAR
```

```
      a,          { Coefficient of square term }
      b,          { Coefficient of x term }
      c,          { Constant term }
      x1,         { First root }
      x2,         { Second root }
      d,          { Computed argument for square root }
      e           { Result of square root }
         : REAL;
BEGIN
      { Initial message }
      WRITELN;
      WRITELN ('Solve quadratic equation for');
      WRITELN ('its real roots, if they exist.');
      { Problem parameters }
      WRITELN;
      WRITE ('Coefficient of square term ? ');
      READLN (a);
      WRITE ('Coefficient of x term      ? ');
      READLN (b);
      WRITE ('Constant                   ? ');
      READLN (c);
      { Solve for x }
      d := b * b - 4.0 * a * c;
      IF d >= 0.0 AND a <> 0.0 THEN
          BEGIN
              e := SQRT(d);
              x1 := 0.5 * (-b - e) / a;
              x2 := 0.5 * (-b + e) / a
          END;
      { Display result }
      WRITELN;
      IF d >= 0.0 AND a <> 0.0 THEN
           BEGIN
               WRITELN ('First root    = ', x1:12:4);
               WRITELN ('Second root = ', x2:12:4)
          END
      ELSE
          WRITELN ('There are no real roots')
END.
```

The following illustrates the use of the program:

```
Solve quadratic equation for
its real roots, if they exist.

Coefficient of square term ? 1.0
Coefficient of x term      ? 1.0
```

```
Constant                ? -6.0
First root     =       -3.0000
Second root    =        2.0000
```

Solving the equation

$$x^2 + x - 6 = 0$$

results in the roots $x = -3$ and $x = 2$.

The function $f(x)$ is

$$(-3)^2 + (-3) - 6 = 9 - 3 - 6 = 0$$

when $x = -3$. The function is

$$2^2 + 2 - 6 = 4 + 2 - 6 = 0$$

when $x = 2$.

BRUTE FORCE ENUMERATION

Brute force enumeration consists of evaluating the function of a sequence of values of x. These values may be listed in table form or plotted to give a picture of the behavior of the function. Tabulating a function at ten points (values of x) may give a rough picture of the shape of the function. Tabulating the function at one thousand points gives much better resolution, but the printed output becomes overwhelming.

Brute force enumeration is one of the techniques the analyst uses in solving for the roots of the equation. Initially, a rough grid of points gives an overall picture of the function within the range of x that is of interest. These points may be tabulated or plotted to determine the neighborhoods (subranges of x) which contain the roots.

Once the neighborhood of a root is known, the analyst may use another tool or continue using brute force enumeration to obtain a refined estimate of the actual value of the root. Enumeration is brute force in comparison to bisection or Newton-Raphson methods, but it is an effective tool in the age of high-speed computers.

The function

$$x^3 - 6.9x - 5.83$$

has three roots. One root is near -2, the second root is near -1, and the third root is near 3. The example of this section illustrates the use of brute force enumeration. The first step involves tabulating the function for $x = -4, -3, \ldots, 5$ and displaying the results. The second stage uses enumeration to inspect the region between -2.5 and -.50 which appears to contain two roots.

Once it is known that there is only one root in a specified subrange of ***x***, it is no longer necessary to display all of the values. Evaluating a large number of points of ***x*** is the computer's forte. The next stages use finer and finer resolutions to make more refined estimates of the root within its subrange.

The following program evaluates the function ***fnc*** at a sequence of points of ***x*** with optional display of those values:

```
PROGRAM solve3 (INPUT, OUTPUT);
{ Use brute force method to estimate
  roots of a function }
VAR
    lower,          { Lower limit for argument }
    upper,          { Upper limit for argument }
    width,          { Interval width for argument }
    arg,            { Argument for function }
    result,         { Calculated result }
    bestarg,        { Argument having result closest to 0.0 }
    bestval         { Value of function at bestarg }
      : REAL;
    print,          { User response to print question }
    response        { User response to continue question }
      : CHAR;
FUNCTION fnc (x : REAL) : REAL;
{ Function to solve for root }
BEGIN
    fnc := x * x * x - 6.9 * x - 5.83
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Use brute force enumeration');
    WRITELN ('to estimate the roots of an');
    WRITELN ('equation. The results may be');
    WRITELN ('listed if desired.');
    { process }
    print := 'Y';
    REPEAT
        { Argument parameters }
        WRITELN;
        WRITE ('Lower limit for argument ? ');
        READLN (lower);
        WRITE ('Upper limit for argument ? ');
        READLN (upper);
        WRITE ('Interval width           ? ');
        READLN (width);
```

```
        WRITE ('Display values (Y/N)      ? ');
        READLN (print);
        IF print = 'Y' THEN
            BEGIN
                WRITELN;
                WRITELN ('Argument      Value')
            END;
        { Enumerate }
        arg := lower;
        bestarg := lower;
        bestval := fnc (lower);
        WHILE arg <= upper DO
            BEGIN
                result := fnc (arg);
                IF ABS(result) < ABS(bestval) THEN
                    BEGIN
                        bestval := result;
                        bestarg := arg
                    END;
                IF print = 'Y' THEN
                    WRITELN (arg:15:5, result:15:5);
                arg := arg + width;
            END;
        { Display best estimate }
        WRITELN;
        WRITELN ('Approx root = ', bestarg:15:5);
        WRITELN ('f(x) =        ', bestval:15:5);
        WRITELN;
        WRITE ('Try another set of limits (Y/N) ? ');
        READLN (response)
    UNTIL response = 'N'
END.
```

The test run uses several stages in its analysis. It inspects the function on a broad level first and then zeros in on finding more refined estimates of a particular root.

```
Use brute force enumeration
to estimate the roots of an
equation. The results may be
listed if desired.

Lower limit for argument ? -4.0
Upper limit for argument ? 4.0
Interval width           ? 1.0
Display values (Y/N)     ? Y
```

```
Argument        Value
      -4.00000      -42.23000
      -3.00000      -12.13000
      -2.00000       -0.03000
      -1.00000        0.07000
       0.00000       -5.83000
       1.00000      -11.73000
       2.00000      -11.63000
       3.00000        0.47000
       4.00000       30.57000
       5.00000       84.67000

Approx root =        -2.00000
f(x) =               -0.03000

Try another set of limits (Y/N) ? Y

Lower limit for argument ? -2.5
Upper limit for argument ? -0.5
Interval width           ? 0.25
Display values (Y/N)     ? Y

Argument        Value
      -2.50000      -4.20500
      -2.25000      -1.69562
      -2.00000      -0.03000
      -1.75000       0.88562
      -1.50000       1.14500
      -1.25000       0.84188
      -1.00000       0.07000
      -0.75000      -1.07687
      -0.50000      -2.50500

Aprox root =         -2.00000
f(x) =               -0.03000

Try another set of limits (Y/N) ? Y

Lower limit for argument ? -2.2
Upper limit for argument ? -1.8
Interval width           ? 0.01
Display values (Y/N)     ? N

Approx root =        -1.99000
f(x) =                0.02040

Try another set of limits (Y/N) ? Y

Lower limit for argument ? -2.0
Upper limit for argument ? -1.98
Interval width           ? .001
Display values (Y/N)     ? N
```

```
Approx root =        -1.99400
f(x) =                0.00039
Try another set of limits (Y/N) ? N
```

The refined estimate of the root in the neighborhood of -2 is -1.994. This is accurate to the nearest .001 for x. The value of the function at $x = -1.994$ is 0.00039.

BISECTION

A young man walks toward his girlfriend in an unusual way, cutting the distance between them in half with each step. They start 100 feet apart. The first step cuts the distance to 50 feet. The next step to 25 feet. The third step to 12.5 feet. The fourth step to 6.25 feet. In how many steps will they meet? The mathematician says that they will never meet. The boy and girl say that they can get close enough.

After 10 steps the distance will be one-thousandth what it was at the beginning. After 20 steps it will be one-millionth, and after 30 steps one-billionth what it was originally. If two points (values of x) bracket a root, the value of the function $f(x)$ will be positive for the one limit and negative for the other. If the function is continuous in the range defined by the two limits, there will be at least one root between the two limits.

The term bisection comes from taking the midpoint between the two limits. If the function $f(x)$ has the same sign as the left-hand boundary, it must have the opposite sign of the right-hand boundary and vice versa. The midpoint becomes a new boundary and the bisection process continues.

The more bisections performed, the more accurate the estimate of the root. Usually, 15 or 20 bisections are sufficient. This method is much more computationally efficient than enumeration. The following program uses bisection to estimate one of the roots of the function

$$f(x) = x^3 - 6.9x + 5.83 = 0$$

```
PROGRAM solve4 (INPUT, OUTPUT);
{ Use bisection to estimate
  roots of a function }
VAR
    lower,          { Lower limit for argument }
    upper,          { Upper limit for argument }
    arg,            { Argument for function }
    result,         { Calculated result }
    result1,        { Result for lower limit }
    resultu         { Result for upper limit }
      : REAL;
    number,         { Number of bisections }
    counter         { Counter for loop control }
      : INTEGER;
```

```
FUNCTION fnc (x : REAL) : REAL;
{ Function to solve for root }
BEGIN
    fnc := x * x * x - 6.9 * x - 5.83
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Use bisection (binary search)');
    WRITELN ('to estimate the root of an');
    WRITELN ('equation.');
    { Argument parameters }
    WRITELN;
    WRITE ('Lower limit for argument ? ');
    READLN (lower);
    WRITE ('Upper limit for argument ? ');
    READLN (upper);
    WRITE ('Number of bisections     ? ');
    READLN (number);
    { Evaluate initial boundaries }
    result1 := fnc(lower);
    resultu := fnc(upper);
    IF result1 * resultu >= 0 THEN
        BEGIN
            WRITELN;
            WRITELN ('Initial lower and upper limits');
            WRITELN ('do not bracket root.')
        END;
    { Perform bisections }
    FOR counter := 1 TO number DO
        BEGIN
            arg := 0.5 * (lower + upper);
            result := fnc (arg);
            IF result1 * result < 0 THEN
                BEGIN
                    resultu := result;
                    upper := arg
                END
            ELSE
                BEGIN
                    result1 := result;
                    lower := arg
                END
        END;
        { Display best estimate }
    WRITELN;
```

```
    WRITELN ('Approx root = ', lower:15:5);
    WRITELN ('f(x) =        ', result:15:5);
END.
```

The following is a sample run using this program

```
Use bisection (binary search)
to estimate the roots of an
equation.

Lower limit for argument ? -2.5
Upper limit for argument ? -1.5
Number of bisections     ? 15
Approx root =          -1.99408
f(x) =                  0.00014
```

A program using enumeration gives a broad picture of the shape of the function and helps to locate regions which are likely to contain roots. The more efficient bisection program can quickly converge on a specific root when its approximate location is determined.

NEWTON-RAPHSON METHOD

The Newton-Raphson method also converges on a root very quickly. Often, it requires fewer iterations than the bisection approach. This method requires both the function $f(x)$ and its derivative $f'(x)$. Given a root estimate of x, the value $f(x)$ and the derivative $f'(x)$ are calculated. The tangent line defined by the slope of the function at that point is projected to the x-axis for which $f(x) = 0$ is predicted. The point

$$x_1 = x - f(x)/f'(x)$$

becomes the new estimate of the root.

The value $f(x)$ and slope $f'(x)$ is recalculated for the new estimate of the root to obtain an even better estimate. The convergence can be very rapid. The following program uses the Newton-Raphson method to estimate a root, given a starting guess:

```
PROGRAM solve5 (INPUT, OUTPUT);
{ Use Newton-Raphson method to
  estimate root of equation }
VAR
    arg,          { Argument for function }
    result,       { Calculated result of function }
    dresult       { Result for derivative }
      : REAL;
    number,       { Number of iterations }
    counter       { Counter for loop control }
```

```
        : INTEGER;
FUNCTION fnc (x : REAL) : REAL;
{ Function to solve for root }
BEGIN
    fnc := x * x * x - 6.9 * x - 5.83
END;
FUNCTION dfnc (x : REAL) : REAL;
{ Derivative of function }
BEGIN
    dfnc := 3.0 * x * x - 6.9;
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Use Newton-Raphson method');
    WRITELN ('to estimate the root of an');
    WRITELN ('equation. This uses both the');
    WRITELN ('function and its derivative.');
    { Argument parameters }
    WRITELN;
    WRITE ('Initial estimate of root ? ');
    READLN (arg);
    WRITE ('Number of iterations   ? ');
    READLN (number);
    { Perform iterations }
    FOR counter := 1 TO number DO
        BEGIN
            result := fnc (arg);
            dresult := dfnc (arg);
            arg := arg - result / dresult
        END
    result := fnc(arg);
    { Display best estimate }
    WRITELN;
    WRITELN ('Approx root = ', arg:15:8);
    WRITELN ('f(x) =        ', result:15:8);
END.
```

The following output illustrates use of the program:

```
Use Newton-Raphson method
to estimate the roots of an
equation. This uses both the
function and its derivative.

Initial estimate of root ? -2.5
Number of iterations   ? 5
```

```
Approx root =     -1.99408000
f(x) =             0.00000095
```

The Newton-Raphson method gave a much closer approximation after only five iterations than the bisection approach did in 15. Not all roots lend themselves to easy solutions using either of the more efficient methods.

PITFALLS AND PRATFALLS

At first glance the efficient search strategies seem to be made to order for finding roots. There are pitfalls to catch the unwary. The Newton-Raphson method uses tangent lines to quickly converge. Under certain conditions the sequence of estimates may oscillate back and forth around the root but not actually converge.

Roots that are closely spaced are difficult to find. A method may find one of them and give no indication that there is another root close by. All search strategies are prone to overlook roots that are close together.

Some roots may result from the function having a local minimum or local maximum that just touches the x-axis. Enumeration may be the only reliable way find to such roots.

EXERCISES

1. Use the program SOLVE3 to find the other roots of the example function of the text.
2. Use the program SOLVE4 to find the other roots of the example function of the text.
3. Use the program SOLVE5 to find the other roots of the example function of the text.
4. Use the programs of this section to find the positive roots of the function

   $f(x) = .1x + \cos(x)$

9.4 SEARCHING FOR MAXIMUM AND MINIMUM

Searching for maximum and minimum values of a function is similar to searching for its roots. As with solving for roots, brute force enumeration is practical in the age of high-speed computers. Efficient search strategies for optimization are more likely to rely on the properties of the derivative function.

OPTIMIZATION

Optimization is one of the important subject areas of mathematics. This involves finding the value of x and the value of the function $f(x)$ that is either the minimum $f(x)$ or the maximum $f(x)$ within a specified range of x. In some cases the search is for the global optimum. In other cases it is for all local optima.

ENUMERATION

Brute force enumeration is one of the tools for optimization problems. The program may list the points x and the values $f(x)$ for a sequence of points. This list is useful for finding rough approximations. The program may compare a large number of points in an interval and display the best of the values compared.

The function

$$f(x) = x^3 - 6.9x + 5.83$$

is the same one used in the last section for estimating roots for which $f(x) = 0$. Now the focus is on finding local minimum and maximum values of $f(x)$. The following program uses enumeration to search for maximum and minimum values:

```
PROGRAM opt1 (INPUT, OUTPUT);
{ Use brute force method to estimate
  minimum and maximum values of a function }
VAR
    lower,      { Lower limit for argument }
    upper,      { Upper limit for argument }
    width,      { Interval width for argument }
    arg,        { Argument for function }
    result,     { Calculated result }
    minarg,     { Argument having minimum value }
    minval,     { Minimum value of function }
    maxarg,     { Argument having maximum value }
    maxval      { Maximum value of function }
      : REAL;
    print,      { User response to print question }
    response    { User response to continue question }
      : CHAR;
FUNCTION fnc (x : REAL) : REAL;
{ Function optimize }
BEGIN
    fnc := x * x * x - 6.9 * x - 5.83
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Use brute force enumeration');
    WRITELN ('to estimate minimum and maximum');
    WRITELN ('values of a function with');
    WRITELN ('optional listing of values');
    { process }
    print := 'Y';
    REPEAT
        { Argument parameters }
```

```
WRITELN;
WRITE ('Lower limit for argument ? ');
READLN (lower);
WRITE ('Upper limit for argument ? ');
READLN (upper);
WRITE ('Interval width              ? ');
READLN (width);
WRITE ('Display values (Y/N)      ? ');
READLN (print);
IF print = 'Y' THEN
    BEGIN
        WRITELN;
        WRITELN ('      Argument          Value');
    END;
{ Enumerate }
arg := lower;
minarg := lower;
minval := fnc (lower);
maxarg := lower;
maxval := minval;
WHILE arg <= upper DO
    BEGIN
        result := fnc (arg);
        IF print = 'Y' THEN
            WRITELN (arg:15:5, result:15:5);
        IF result < minval THEN
            BEGIN
                minarg := arg;
                minval := result
            END;
        IF result > maxval THEN
            BEGIN
                maxarg := arg;
                maxval := result
            END;
        arg := arg + width;
    END;
{ Display best estimate }
WRITELN;
WRITELN ('Minimum');
WRITELN ('Argument = ', minarg:15:5);
WRITELN ('Value =      ', minval:15:5);
WRITELN;
WRITELN ('Maximum');
WRITELN ('Argument = ', maxarg:15:5);
WRITELN ('Value =      ', maxval:15:5);
```

```
        WRITELN;
        WRITE ('Try another set of limits (Y/N) ? ');
        READLN (response)
    UNTIL response = 'N'
END.
```

The following test run explores the function in stages. The first enumeration computes the value of $f(x)$ for $x = -4, -3, \ldots, 5$. Based on the results of the first rough picture, the further stages explore the neighborhood of the local maximum near -1. The test run follows:

```
Use brute force enumeration
to estimate minimum and maximum
values of a function with
optional listing of values

Lower limit for argument ? -4.0
Upper limit for argument ? 5.0
Interval width           ? 1.0
Display values (Y/N)     ? Y

    Argument          Value
    -4.00000       -42.23000
    -3.00000       -12.13000
    -2.00000        -0.03000
    -1.00000         0.07000
     0.00000        -5.83000
     1.00000       -11.73000
     2.00000       -11.63000
     3.00000         0.47000
     4.00000        30.57000
     5.00000        84.67000

Minimum
Argument =      -4.00000
Value =        -42.23000

Maximum
Argument =       5.00000
Value =         84.00000

Try another set of limits (Y/N) ? Y

Lower limit for argument ? -2.5
Upper limit for argument ? -0.5
Interval width           ? 0.25
Display values (Y/N)     ? Y

    Argument          Value
    -2.50000        -4.20500
    -2.25000        -1.69562
```

```
-2.00000        -0.03000
-1.75000         0.88562
-1.50000         1.14500
-1.25000         0.84188
-1.00000         0.07000
-0.75000        -1.07687
-0.50000        -2.50500

Minimum
Argument =      -2.50000
Value =         -4.20500

Maximum
Argument =      -1.50000
Value =          1.14500

Try another set of limits (Y/N) ? Y

Lower limit for argument ? -2.0
Upper limit for argument ? -1.0
Interval width           ? 0.01
Display values (Y/N)     ? N

Minimum
Argument =      -2.00000
Value =         -0.03000

Maximum
Argument =      -1.52000
Value =          1.14619

Try another set of limits (Y/N) ? Y

Lower limit for argument ? -1.54
Upper limit for argument ? -1.50
Interval width           ? .0001
Display values (Y/N)     ? N

Minimum
Argument =      -1.54000
Value =          1.14374

Maximum
Argument =      -1.51860
Value =          1.14625

Try another set of limits (Y/N) ? N
```

INCREMENTAL ANALYSIS WITH ENUMERATION

Listing all points of an evaluated function is not practical when there are many enumerated points. Let the computer compare the last three values of $f(x)$ and list those values only if the middle value is either the minimum or the maximum

of the three. If the function is either increasing or decreasing within a range of the values of *x*, listing those values is pointless.

A local minimum occurs when a function that has been increasing begins to decrease. A local minimum occurs when a function that has been decreasing starts to increase. The endpoints are of interest for possible global maximum and minimum points.

The following program lists the endpoints and the three values that signal each local optimum value:

```
PROGRAM opt2 (INPUT, OUTPUT);
{ Use incremental analysis with enumeration
  to estimate the minimum and maximum values
  of a function }
VAR
    lower,          {Lower limit for argument }
    upper,          {Upper limit for argument }
    width,          {Interval width for argument }
    arg,            {Argument for function }
    arga,           {Argument for function }
    argb,           {Argument for function }
    y,              {Calculated result for current argument }
    ya,             {First of three results }
    yb              {Second of three results }
      : REAL;
    response        {User response to continue question }
      : CHAR;
FUNCTION fnc (x : REAL) : REAL;
{ Function optimize }
BEGIN
    fnc := x * x * x - 6.9 * x - 5.83
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Use incremental analysis to');
    WRITELN ('estimate minimum and maximum');
    WRITELN ('values of a function.');
    { process }
    REPEAT
        { Argument parameters }
        WRITELN;
        WRITE ('Lower limit for argument ? ');
        READLN (lower);
        WRITE ('Upper limit for argument ? ');
        READLN (upper);
```

```
        WRITE ('Interval width              ? ');
        READLN (width);
        WRITELN;
        WRITELN ('      Argument          Value');
        { Enumerate }
        arg := lower;
        ya := fnc(arg);
        WRITELN (arg:15:5, ya:15:5);
        arg := arg + width;
        yb := fnc(arg);
        arg := arg + width;
        WHILE arg <= upper DO
            BEGIN
                y := fnc (arg);
                IF (y < yb AND ya < yb)
                    OR (y > yb AND ya > yb) THEN
                    BEGIN
                        arga := arg - 2.0 * width;
                        argb := arg - width;
                        WRITELN (arga:15:5, ya:15:5);
                        WRITELN (argb:15:5, yb:15:5);
                        WRITELN (arg:15:5, y:15:5)
                    END;
                ya := yb;
                yb := y;
                arg := arg + width;
            END;
        { Display best estimate }
        arg:= arg - width;
        WRITELN (arg:15:5, yb:15:5);
        WRITE ('Try another set of limits (Y/N) ? ');
        READLN (response)
    UNTIL response = 'N'
END.
```

Output from this program follows:

```
Use incremental analysis to
estimate minimum and maximum
values of a function.
Lower limit for argument ? -4.0
Upper limit for argument  ? 5.0
Interval width              ? 0.1

      Argument          Value
      -4.00000       -42.23000
      -1.60000         1.11400
```

```
    -1.50000          1.14500
    -1.40000          1.08600
     1.40000        -12.74600
     1.50000        -12.80500
     1.60000        -12.77400
     5.00000         84.66980
Try another set of limits (Y/N) ? Y

Lower limit for argument ? -1.6
Upper limit for argument ? -1.4
interval width           ? .001

    Argument          Value
    -1.60000          1.11400
    -1.51800          1.14624
    -1.51700          1.14625
    -1.51600          1.14624
    -1.40000          1.08701
Try another set of limits (Y/N) ? N
```

BISECTION USING SIGN OF DERIVATIVE

When there is a local minimum or maximum, the slope of $f(x)$ changes for values of x on opposite sides of the optimum point. Bisection is a useful method for converging on the optimum point if the derivative function $f'(x)$ is known. The following program uses bisection with the derivative function to converge to the local optimum point:

```
PROGRAM opt3 (INPUT, OUTPUT);
{ Use bisection and derivative
  function to estimate a local
  minimum or maximum }
VAR
    lower,      { Lower limit for argument ,
    upper,      { Upper limit for argument }
    arg,        { Argument for function }
    result,     { Calculated result of function }
    slope,      { Derivative (slope) of function }
    slope1,     { Result of derivative for left }
    slope2,     { Result of derivative for right }
      : REAL;
    number,     { Number of iterations }
    counter     { Counter for loop control }
      : INTEGER;
FUNCTION fnc (x : REAL) : REAL;
{ Function to solve for root }
```

```
BEGIN
    fnc := x * x * x - 6.9 * x - 5.83
END;
FUNCTION dfnc (x : REAL) : REAL;
{ Derivative of function }
BEGIN
    dfnc := 3.0 * x * x - 6.9;
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Use bisection and derivative');
    WRITELN ('function to estimate the local');
    WRITELN ('minimum or maximum of a function.');
    { Argument parameters }
    WRITELN;
    WRITE ('Lower limit for argument ? ');
    READLN (lower);
    WRITE ('Upper limit for argument ? ');
    READLN (upper);
    WRITE ('Number of iterations      ? ');
    READLN (number);
    slope1 := dfnc(lower);
    slope2 := dfnc(upper);
    IF slope1 * slope2 > 0.0 THEN
        BEGIN
            WRITELN;
            WRITELN ('The function at the argument');
            WRITELN ('limits has the same slope')
        END;
    { Perform iterations }
    counter := 1;
    WHILE slope1 * slope2 < 0.0 AND counter <= number DO
        BEGIN
            arg := 0.5 * (lower + upper);
            slope := dfnc(arg);
            IF slope * slope1 < 0.0 THEN
                BEGIN
                    upper := arg;
                    slope2 := slope
                END
            ELSE
                BEGIN
                    lower := arg;
                    slope1 := slope
                END;
```

```
            counter := counter + 1
        END;
    result := fnc(arg);
    { Display best estimate }
    WRITELN;
    IF slope1 > 0.0 THEN
        WRITELN ('Maximum      ', arg:15:5, result:15:5)
    ELSE
        WRITELN ('Minimum      ', arg:15:5, result:15:5)
END.
```

The output from this program follows:

```
Use bisection and derivative
function to estimate the local
minimum or maximum of a function.

Lower limit for argument ? -2.0
Upper limit for argument ? -1.0
Number of iterations     ? 15

Maximum        -1.51657      1.14625
```

EXERCISES

1. Use the program OPT1 to explore the function of this section in the neighborhood of 1.5.
2. Use the program OPT2 to find the local minimum value of $f(x)$ in the neighborhood of 1.5.
3. Use the program OPT3 to find the local minimum value of $f(x)$ in the neighborhood of 1.5.
4. Use the programs of this section to find the local minimum and maximum values in the range $0 \leqslant x \leqslant 9$ for the function

   $f(x) = .1x + \cos(x)$.

   Determine the global maximum and minimum values in the specified range.

10 Matrix applications

OVERVIEW A matrix is a two-dimensional table of numbers. A vector is an ordered set of numbers. Matrices and vectors are important because of their application to the solution of simultaneous linear equations.

The application of matrix algebra in constrained optimization (linear programming) is not as well-known. Matrices are also useful in Markov analysis, payoff table analysis, and the generation of revised probability distributions for Bayesian analysis.

□

10.1 BUSINESS APPLICATIONS

Quantitative applications are becoming more important in business. These include the statistical analysis of data and the building of models to aid in the decision process. This section explores several models and gives example programs.

PAYOFF TABLES

A decision-maker must choose from among several courses of action. He may face uncertainty in demand for a product, action of competitors, prices of raw materials, and the weather. These uncertainties are called events or states of nature.

A payoff table gives the outcome for each combination of course of action and state of nature. The decision-maker objectively or subjectively assigns probabilities to the states of nature.

The following payoff table has two courses of action and three states of nature:

Payoff table

State of nature	*Probability*	*Course of action* Act 1	Act 2
1	.2	-200	400
2	.5	100	0
3	.3	300	-200

The expected value of a course of action is the sum of the products of the outcomes and the probabilities. The expected value for Act 1 is

$$E(\text{Act } 1) = -200x.2 + 100x.5 + 300x.3 = 100.$$

The expected value for Act 2 is

$$E(\text{Act } 2) = 400x.2 + 0x.5 - 200x.3 = 20.$$

If the outcomes represent profits, then Act 1 has the largest expected profit. The negative outcomes represent losses (negative profits).

The following program computes the expected values for the courses of action of a payoff table:

```
PROGRAM bus1 (INPUT, OUTPUT);
{*****************************************
 *     Payoff table analysis                *
 *****************************************}
VAR
    nstates,          { Number of states of nature }
    nacts,            { Number of courses of action }
    i,                { Subscript }
    j                 { Subscript }
       : INTEGER;
    sum               { Sum of values or sum of products }
       : REAL;
    prob              { Prodability vector }
       :ARRAY [1 .. 10] OF REAL;
    payoff            { Payoff table of outcomes }
       : ARRAY [1 .. 10, 1 .. 10] OF REAL;
    value             { Expected values }
       : ARRAY [1 .. 10] OF REAL;
BEGIN
    {*****************************************
     *     Initial message                    *
     *****************************************}
    WRITELN;
    WRITELN ('Program BUS1');
    WRITELN;
    WRITELN ('Payoff table analysis computing');
    WRITELN ('the expected value for each');
    WRITELN ('course of action using the');
    WRITELN ('vector of probabilities.');
    {*****************************************
     *     Problem parameters                 *
     *****************************************}
```

```
WRITELN;
WRITE ('Number of courses of action ? ');
READLN (nacts);
WRITE ('Number of states of nature  ? ');
READLN (nstates);
REPEAT
    WRITELN;
    WRITELN ('Probability for');
    sum := 0.0;
    FOR i := 1 TO nstates DO
        BEGIN
            WRITE ('State ', i:2, ' ? ');
            READLN (prob[i]);
            sum := sum + prob[i]
        END;
        IF ABS(sum - 1.0) >= 0.00001 THEN
            BEGIN
                WRITELN;
                WRITELN ('Probabilities must sum to 1.0')
            END
UNTIL ABS(sum - 1.0) < 0.00001;
WRITELN;
WRITELN ('Payoff for');
FOR j := 1 TO nacts DO
    BEGIN
        WRITELN;
        WRITELN ('Act ', j:2);
        FOR i := 1 TO nstates DO
            BEGIN
                WRITE ('  State ', i:2, ' ? ');
                READLN (payoff[i,j]);
            END
    END;
{*****************************************
 *     Compute expected values           *
 *****************************************}
FOR j := 1 TO nacts DO
    BEGIN
        sum := 0.0;
        FOR i := 1 TO nstates DO
            sum := sum + prob[i] * payoff[i, j];
        value[j] := sum;
    END;
{*****************************************
 *     Display results                   *
 *****************************************}
```

```
    WRITELN;
    WRITELN ('Course of        Expected');
    WRITELN ('action           value');
    FOR j := 1 TO nacts DO
        WRITELN ('      ', j:3, '        ', value[j]:15:5);
    WRITELN;
    WRITELN ('End of program')
END.
```

Sample output from this program follows:

```
Program BUS1

Payoff table analysis computing
the expected value for each
course of action using the
vector of probabilities.

Number of courses of action ? 2
Number of states of nature  ? 3

Probablity for
State 1 ? 0.2
State 2 ? 0.5
State 3 ? 0.3

Payoff for

Act 1
    State  1 ? -200.0
    State  2 ? 100.0
    State  3 ? 300.0

Act 2
    State  1 ? 400.0
    State  2 ? 0.0
    State  3 ? -200.0

Course of       Expected
action          value
      1          100.00000
      2           20.00000

End of program
```

BAYES' THEOREM

Bayes' theorem is useful for combining prior probability information with sample evidence to derive revised probabilities. The sample information is in the form of conditional probabilities of observing the sample outcome given each possible state of nature. The revised probabilities are conditional probabilities for the states of nature given the sample evidence.

Let S_i represent the ith state of nature. The symbol $P(S_i)$ is the prior probability of the ith state of nature. The probability $P(X{:}S_i)$ is the conditional probability (likelihood) of observing the sample outcome X given the ith state of nature. The revised probability $P(S_i{:}X)$ is the conditional probability of being in the ith state given the experimental evidence X.

The probability $P(X$ and $S_i)$ is the joint probability of being in the ith state and observing the sample outcome X. The product $P(S_i) \times P(X{:}S_i)$ gives the joint probability. The sum of the joint probabilities gives the unconditional probability of observing the sample result and is denoted $P(X)$.

The prior probability distribution gives the probabilities $P(S_i)$ for all possible states of nature. The revised probability distribution gives the corresponding probabilities of being in the states of nature given the sample evidence. Dividing $P(X)$ into the joint probability $P(X$ and $S_i)$ gives the revised probability $P(S_i{:}X)$ for the ith state.

A sample problem has three states of nature—A, B, and C—with prior probabilities .7, .2, and .1. An experiment is performed resulting in an outcome having conditional probabilities .2, .5, and .8 for the three states. Determine the revised probability distribution for the three states of nature. The following program computes revised probabilities given the prior and likelihood probabilities:

```
PROGRAM bus2 (INPUT, OUTPUT);
{**************************************
 *    Bayes' theorem                  *
 **************************************}
TYPE
    vector = ARRAY [1 .. 10] OF REAL;
VAR
    index,          { Subscript }
    nstates         { Number of states of nature }
       : INTEGER;
    sum             { Sum of values }
       : REAL;
    prior,          { Prior probabilities }
    like,           { Likelihoods }
    joint,          { Joint probabilities }
    revised         { Revised probabilities }
       : vector;
BEGIN
    {**************************************
     *Initial message                     *
     **************************************}
    WRITELN;
    WRITELN ('Program BUS2');
    WRITELN;
```

```
WRITELN ('Use Bayes'' theorem to compute');
WRITELN ('revised probability distributions');
WRITELN ('given a prior distribution and');
WRITELN ('likelihoods.');
{*****************************************
 *     Problem parameters                *
 *****************************************}
WRITELN;
WRITE ('Number of states of nature ? ');
READLN (nstates);
REPEAT
     WRITELN;
     sum := 0.0;
     WRITELN ('Prior prob, likelihood for');
     FOR index := 1 TO nstates DO
          BEGIN
               WRITE ('state ', index:2, ' ? ');
               READLN (prior[index], like[index]);
               sum := sum + prior[index]
          END;
     IF ABS(sum - 1.0) >= 0.00001 THEN
          BEGIN
               WRITELN;
               WRITELN ('Prior probs must sum to 1.0')
          END
UNTIL ABS(sum - 1.0) < 0.00001;
{*****************************************
 *     Compute revised probabilities     *
 *****************************************}
sum := 0.0;
FOR index := 1 TO nstates DO
     BEGIN
          joint[index] := prior[index] * like[index] ;
          sum := sum + joint[index]
     END;
FOR index := 1 TO nstates DO
     revised[index] := joint[index] / sum;
{*****************************************
 *     Display results                   *
 *****************************************}
WRITELN;
WRITE ('  Prior        Likelihood        ');
WRITE ('Joint        Revised');
FOR index := 1 TO nstates DO
```

```
        BEGIN
            WRITE (prior[index]:10:5, like[index]:15:5);
            WRITELN (joint[index]:15:5, revised[index]:15:5)
        END;
    WRITELN;
    WRITELN ('Probability of sample outcomes = ', sum:9:5);
    WRITELN;
    WRITELN ('End of program')
END.
```

The sample output follows:

```
Program BUS2

Use Bayes' theorem to compute
revised probability distributions
given a prior distribution and
likelihoods.

Number of states of nature ? 3

Prior prob, likelihood for
state  1 ? .7 .2
state  2 ? .2 .5
state  3 ? .1 .8
```

Prior	Likelihood	Joint	Revised
0.70000	0.20000	0.14000	0.43750
0.20000	0.50000	0.10000	0.31250
0.10000	0.80000	0.08000	0.25000

```
Probability of sample outcome =    0.32000

End of program
```

BINOMIAL PROBABILITY DISTRIBUTION

The binomial distribution gives probabilities for $r = 0, 1, \ldots, n$ successes in n trials where p gives the probability of a success on any one trial and $q = 1 - p$ gives the probability of failure on any one trial. The formula

$$\frac{n!}{r!(n-r)!} \cdot p^r q^{n-r}$$

gives the value of the binomial probability for r successes. The symbol "!" used in $r!$, $n!$, and $(n-r)!$ means factorial. The value of $r!$ is found by taking the product of the first r integers for $r \geq 1$. The value of $r!$ is 1 for $r = 0$.

The binomial probability of $r = 0$ successes in n trials is

$$P(r=0) = \frac{n!}{0!(n-0!)} \cdot p^0 q^n = q^n$$

The binomial probability of $r = 1$ successes is

$$P(r = 1) = npq^{(n-1)} = P(r = 0)np/q$$

The binomial probability of $r = 2$ successes is

$$P(r = 2) = \frac{n(n-1)}{2} p^2 q^{n-2}$$

In general, the recursive expression

$$P(r) = P(r-1)(n-r+1)p/(rq)$$

gives the binomial probability for r successes as a function of the probability for $r - 1$ successes.

The first step is to generate the probability of $r = 0$ successes as the probability q of a failure on any one trial raised to the nth power. The recursive expression then generates the probabilities for $r = 1, 2, \ldots, n$ successes. The following program generates probabilities for the binomial distribution:

```
PROGRAM bus3 (INPUT, OUTPUT);
{*****************************************
 *    Binomial distribution                *
 *****************************************}
TYPE
    vector = ARRAY [0 .. 100] OF REAL;
VAR
    number,          { Number of trials }
    index            { Number of successes }
       : INTEGER;
    sum,             { Cumulative probability }
    prob             { Probability of success }
       : REAL;
    probdist         { Vector of probabilities }
       : vector;
{*****************************************
 *    Generate binomial probabilities      *
 *****************************************}
PROCEDURE binomial
    (n : INTEGER;          { Number of trials }
     p : REAL;             { P(success) }
     VAR pdist : vector);  { Binomial probabilities }
VAR
    i                { Index for distribution }
       : INTEGER;
    b,               { Current binomial probability }
```

```
    q                { P(failure)  }
       : REAL;
BEGIN
    q := 1.0 - p;
    b := 1.0;
    FOR i := 1 TO n DO
        b := b * q;
    pdist[0] := b;
    FOR i := 1 TO n DO
        BEGIN
            b := b * (n - i + 1.0) * p / (i * q);
            pdist[i] := b
        END
END;
{*****************************************
 *    Body of main routine                *
 *****************************************}
BEGIN
   {*****************************************
    *    Initial message                  *
    *****************************************}
    WRITELN;
    WRITELN ('Program BUS3');
    WRITELN;
    WRITELN ('Generate binomial probability');
    WRITELN ('distribution given the number');
    WRITELN ('of trials and the probability');
    WRITELN ('of a success on any one trial.');
    {*****************************************
     *    Problem parameters               *
     *****************************************}
    WRITELN;
    WRITE ('Probability of success on one trial ? ');
    READLN (prob);
    WRITE ('Number of trials for distribution  ? ');
    READLN (number);
    {*****************************************
     *    Generate and display             *
     *****************************************}
    binomial (number, prob, probdist);
    WRITELN;
    WRITELN ('Successes      Probability      Cumulative');
    sum := 0.0;
    FOR index := 0 TO number DO
```

```
        BEGIN
            sum := sum + probdist[index] ;
            WRITE (index:10, probdist[index]:15:5);
            WRITELN (sum:15:5);
        END;
    WRITELN;
    WRITELN ('End of program')
END.
```

A test run follows:

```
Program BUS3

Generate binomial probability
distribution given the number
of trials and the probability
of a success on any one trial.

Probability of success on one trial ? 0.2
Number of trials for distribution  ? 5
```

Successes	Probability	Cumulative
0	0.32768	0.32768
1	0.40960	0.73728
2	0.20480	0.94208
3	0.05120	0.99328
4	0.00640	0.99968

```
End of program
```

REVISED PROBABILITIES WITH BINOMIAL PROCESS

A machine is in one of three states. When in state 1 it produces 10 percent defectives. In state 2 it produces 40 percent defectives. In state 3 it produces 80 percent defectives. The decision-maker's prior probabilities for the machine's current state are .7 for state 1, .2 for state 2, and .1 for state 3.

A statistical quality control plan is to be implemented. A small sample is taken and the number of defectives counted. On the basis of the sample, the quality control inspector will draw conclusions about the current state of the machine.

The binomial distribution gives the likelihood for the observed number of defectives in the sample given the state of the machine. For a machine in state 1 the probability of a defective on any one trial is .1. In state 2 the probability of a defective on any one trial is .4.

This is a binomial process. The probability of a success on any one trial is the binomial process parameter. The actual current state of the machine is uncertain, and the current fraction of defectives is unknown. Bayes' theorem combines

the prior probability distribution for the states of the machine with experimental evidence in the form of binomial likelihoods to compute a revised probability distribution of the states.

There is a separate revised probability distribution for each sample result. A sample of size n contains $r = 0, 1, \ldots, n$ possible successes. The following program generates a table of revised distributions for the possible sample results:

```
PROGRAM bus4 (INPUT, OUTPUT);
{*****************************************
 *    Bayesian analysis                  *
 *****************************************}
TYPE
    vector = ARRAY [0 .. 50] OF REAL;
VAR
    fraction,       { Process fraction defective }
    prior,          { Prior distribution }
    like,           { Likelihoods for current outcome }
    joint,          { Joint probabilities }
    revised         { Revised probabilities }
       : vector;
    response        { User response }
       : CHAR;
    nstates,        { Number of states of nature }
    number,         { Number of trials = sample size }
    outcome,        { Number of successes observed }
    jndex,          { Index }
    index           { Subscript }
       : INTEGER;
    bin,            { Binomial probability }
    sum,            { Sum of joint probabilities }
    pfailure        { Probability of a failure }
       : REAL;
{*****************************************
 *    Initial message                    *
 *****************************************}
PROCEDURE initial;
BEGIN
    WRITELN;
    WRITELN ('Program BUS4');
    WRITELN;
    WRITELN ('Generate revised probability');
    WRITELN ('distribution for each sample');
    WRITELN ('outcome of a binomial distribution.')
END;
```

```
{*****************************************
 *    Problem parameters                 *
 *****************************************}
PROCEDURE parameters;
BEGIN
    WRITELN;
    WRITE ('Number of states of nature ? ');
    READLN (nstates);
    WRITELN;
    WRITELN ('Fraction successes, Prior prob for');
    FOR index := 1 TO nstates DO
        BEGIN
            WRITE ('state ', index:2, ' ? ');
            READLN (fraction[index], prior[index])
        END
END;
{*****************************************
 *    Likelihoods                        *
 *****************************************}
PROCEDURE likelihoods;
BEGIN
    FOR index := 1 to nstates DO
        IF outcome = 0 THEN
            BEGIN
                bin := 1.0;
                pfailure := 1.0 - fraction[index];
                FOR jndex := 1 TO number DO
                    bin := bin * pfailure;
                like[index] := bin
            END
        ELSE
            like[index] := like[index]
                * (number - outcome + 1.0)
                * fraction[index]
                / (outcome * (1.0 - fraction[index]))
END;
{*****************************************
 *    Revise                             *
 *****************************************}
PROCEDURE revise;
BEGIN
    sum := 0.0
    FOR index := 1 TO nstates DO
        BEGIN
            joint[index] := prior[index] * like[index];
```

```
                sum := sum + joint[index]
            END;
        FOR index := 1 TO nstates DO
            revised[index] := joint[index] / sum
END;
{*****************************************
 *     Display results                   *
 *****************************************}
PROCEDURE display;
BEGIN
    WRITE (outcome:2, sum:10:6);
    FOR index := 1 TO nstates DO
        WRITE (revised[index]:10:6);
    WRITELN;
END;
{*****************************************
 *     Body of main program              *
 *****************************************}
BEGIN
    initial;
    parameters;
    REPEAT
        WRITELN;
        WRITE ('Sample size ? ');
        READLN (number);
        WRITELN;
        WRITELN ('r     P(r)     Revised probabilities');
        FOR outcome := 0 TO number DO
            BEGIN
                likelihoods;
                revise;
                display
            END;
        WRITELN;
        WRITE ('Try another sample size (Y/N) ? ');
        READLN (response)
    UNTIL response = 'N';
    WRITELN;
    WRITELN ('End of program')
END.
```

A test run follows:

```
Program BUS4
Generate revised probability
```

```
distribution for each sample
outcome of a binomial distribution.

Number of states of nature ? 3

Fraction successes, Prior prob for
state  1 ? 0.1  0.7
state  2 ? 0.4  0.2
state  3 ? 0.8  0.1

Sample size ? 2

r    P(r)          Revised probabilities
 0   0.643000      0.881804    0.111975    0.006221
 1   0.254000      0.496063    0.377953    0.125984
 2   0.103000      0.067961    0.310680    0.621359

Try another sample size (Y/N) ? Y

Sample size ? 5

r    P(r)          Revised probabilities
 0   0.428927      0.963668    0.036258    0.000075
 1   0.282115      0.813977    0.183755    0.002269
 2   0.125270      0.407360    0.551768    0.040872
 3   0.072230      0.078499    0.637962    0.283539
 4   0.056635      0.005562    0.271210    0.723228
 5   0.034823      0.000201    0.058812    0.940987

Try another sample size (Y/N) ? N

End of program
```

EXERCISES

1. Write a program to compute the expected payoffs for the following payoff table:

State	*Prob*	*Act 1*	*Act 2*	*Act 3*
1	.6	1200	450	900
2	.4	200	750	300

2. An estimated 5 percent of the inhabitants of a remote village have active TB (tuberculosis). A public health team gives a patch test to all of the inhabitants. The patch test is 98 percent reliable at detecting active TB, i.e., 98 percent of those who have active TB have positive reactions. The test is 90 percent reliable for those who do not have active TB, i.e., 10 percent of those without active TB have positive reactions. Of those who have active TB, determine the revised probability distribution for the states "active TB" and "do not have active TB."

3. Write a program to generate probabilities for the binomial distribution $n = 20$ and $p = 4$.

4. Generate the revised probability distributions for a sample of size $n = 10$ and the following binomial process states:

Fraction defective	*Prior probability*
.1	.2
.2	.4
.3	.3
.4	.1

10.2 MARKOV CHAINS

A system can be in any one of several states and changes from state to state each period. A transition probability matrix gives the probabilities of switching from one state to another during a given period.

The transition probability matrix—

Transition probability matrix

From	*To*	
	State 1	*State 2*
State 1	.8	.2
State 2	.1	.9

—has two states. The system moves from state to state among the set of states according to the probabilities of the table. Each row is a conditional probability distribution. If the system is now in state 1, the probability is .8 that it will remain in state 1 and .2 that it will switch to state 2.

The initial state for the system may be probabilistically determined. Let .6 be the probability that the initial state is 1 and .4 be the probability that the initial state is 2. The matrix product

$$(.6 \quad .4) \begin{matrix} .8 & .2 \\ .1 & .9 \end{matrix}$$

gives the probability distribution for being in state 1 and state 2 for the second stage. The expression

$$.6x.8 + .4x.1 = .52$$

gives the probability of state 1 for the second stage. The expression

$$.6x.2 + .4x.9 = .48$$

gives the probability of state 2 for the second stage.

The following program generates the probability distributions for the states for the first few stages:

```
PROGRAM markov1 (INPUT, OUTPUT);
```

```
{*****************************************
 *     Finite stage Markov chain         *
 *****************************************}
VAR
    tran            { Transition probability matrix }
       : ARRAY [1 .. 10, 1 .. 10] OF REAL;
    prob,           { Probability vector }
    result          { Resulting probability vector }
       : ARRAY [1 .. 10] OF REAL;
    nstages,        { Number of stages }
    stage,          { Current stage }
    size,           { Size = rows = columns }
    row,            { Row subscript }
    col             { Column subscript }
       : INTEGER;
    sum               Sum of products
       : REAL;
BEGIN
{*********************************************
 *     Initial message                       *
 *********************************************}
    WRITELN;
    WRITELN ('Program MARKOV1');
    WRITELN;
    WRITELN ('Determine probabilities for');
    WRITELN ('states of nature for the first');
    WRITELN ('stages of a Markov process.');
    {*****************************************
     *     Problem parameters                *
     *****************************************}
    WRITELN;
    WRITE ('Number of states of nature ? ');
    READLN (size);
    WRITELN;
    WRITELN ('Initial probability vector');
    WRITELN;
    WRITELN ('Probability for');
    FOR col := 1 TO size DO
        BEGIN
            WRITE ('State ', col:2, ' ? ');
            READLN (prob[col])
        END;
    WRITELN;
    WRITELN ('Transition probability matrix');
    FOR row := 1 TO size DO
```

```
        BEGIN
            WRITELN;
            WRITELN ('From row ', row:2);
            FOR col := 1 TO size DO
                BEGIN
                    WRITE ('     to col  ', col:2, ' ? ');
                    READLN (tran[row,col])
                END
        END;
    WRITELN;
    WRITE ('Number of stages to generate ? ');
    READLN (nstages);
    {*****************************************
     *     Process                           *
     *****************************************}
    WRITELN;
    WRITELN ('Stage     Probabilities for states');
    FOR stage := 1 TO nstages DO
        BEGIN
            WRITE (stage:5);
            FOR col := 1 TO size DO
                WRITE (prob[col]:10:6);
            WRITELN;
            FOR col := 1 TO size DO
                BEGIN
                    sum := 0.0;
                    FOR row := 1 TO size DO
                        sum := sum + prob[row] * tran[row,col];
                    result[col] := sum
                END;
            FOR col := 1 TO size DO
                prob[col] := result[col]
        END;
    WRITELN;
    WRITELN ('End of program');
END.
```

The test run for this program follows:

```
Program MARKOV1

Determine probabilities for
states of nature for the first
stages of a Markov process.

Number of states of nature? 2

Initial probability vector
```

```
Probability for
State  1 ? 0.6
State  2 ? 0.4
Transition probability matrix
From row 1
    To col  1 ? 0.8
    To col  2 ? 0.2
From row 2
    To col  1 ? 0.1
    To col  2 ? 0.9
Number of stages to generate ? 5
Stage     Probabilities for states
    1   0.600000   0.400000.
    2   0.520000   0.480000
    3   0.464000   0.536000
    4   0.425800   0.575200
    5   0.397360   0.602640
End of program
```

MARKOV CHAIN STEADY-STATE PROBABILITIES

After a large number of stages, the state probabilities of a Markov chain converge to equilibrium values. One way to observe this is to multiply the transition probability by itself many times. The transition probability matrix

```
.8   .2
.1   .9
```

is a square matrix having two rows and two columns. Each row and each column represents one of two states.

Matrix multiplication involves summing the products of rows of the first matrix with columns of the second matrix. The matrix product

```
.8   .2   x   .8   .2
.1   .9       .1   .9
```

multiplies the transition probability matrix by itself. The matrix

```
.66   .34
.17   .83
```

results.

If the system is now in state 1, the probability is .8 of being in state 1 in the next stage and .66 of being in state 1 during the following stage. The sum of

the products of the elements in row 1 of the first matrix and column 1 of the second matrix gives the .66. The expression

$.8x.8+.2x.1=.66$

shows the calculations.

Multiplying the transition probability matrix by itself is called squaring the transition probability matrix. Multiplying the square by itself is equivalent to raising the original transition probability matrix to the 4th power. Squaring that matrix results in a matrix equivalent to raising the original matrix to the 8th power. Continuing this repetitive squaring for 10 iterations is equivalent to raising the original matrix to the 1,024th power.

By successive squarings the rows of the resulting matrix should be equal or nearly equal. This represents the fact that the state after that many stages is independent of the starting state. If there is a steady state solution, it should be evident at this point. The following program estimates the steady-state probabilities by raising the original transition probability matrix to the 1,024th power using 10 successive squarings.

```
PROGRAM markov2 (INPUT, OUTPUT);
{*****************************************
 *     Infinite stage Markov chain       *
 *****************************************}
TYPE
    matrix = ARRAY [1 .. 10, 1 .. 10] OF REAL;
VAR
    tran1,          { First matrix }
    tran2           { Second matrix }
       : matrix;
    nstates,        { Number of states of nature }
    stage           { Stage for squaring matrix }
       : INTEGER;
{*****************************************
 *     INITIAL MESSAGE                   *
 *****************************************}
PROCEDURE initial;
BEGIN
    WRITELN;
    WRITELN ('Program MARKOV2')
    WRITELN;
    WRITELN ('Estimate steady-state');
    WRITELN ('probabilities by raising');
    WRITELN ('the transition probability');
    WRITELN ('matrix to the 1,024th power.');
END;
```

```
{*****************************************
 *    parameters                           *
 *****************************************}
PROCEDURE parameters
    (VAR size : INTEGER;
     VAR table : matrix);
VAR
    row,          { Row subscript }
    col           { Column subscript }
       : INTEGER;
BEGIN
    WRITELN;
    WRITE ('Number of states of nature ? ');
    READLN (size);
    WRITELN;
    WRITELN ('Transition probability matrix');
    FOR row := 1 TO size DO
        BEGIN
            WRITELN;
            WRITELN ('From row ', row:2);
            FOR col := 1 TO size DO
                BEGIN
                    WRITE ('    To col ', col:2, ' ? ');
                    READLN (table[row,col])
                END
        END
END;
{*****************************************
 *    square                               *
 *****************************************}
PROCEDURE SQUARE
    (VAR size             { Number of rows, columns }
          : INTEGER;
     VAR first,           { First matrix }
         result           { Square of first matrix }
          : matrix);
VAR
    i,                    { Subscript }
    j,                    { Subscript }
    k                     { Subscript }
       : INTEGER;
    sum                   { Sum of products }
       : REAL;
BEGIN
    FOR i := 1 TO size DO
```

```
        FOR j := 1 TO size DO
            BEGIN
                sum := 0.0;
                FOR k := 1 TO size DO
                    sum := sum
                        + first[i,k] * first[k,j];
                result[i,j] := sum
            END
END;
{*****************************************
 *    display                            *
 *****************************************}
PROCEDURE display
    (VAR size           { Number of stages }
          : INTEGER;
     VAR table          { Table of probabilities }
          : matrix);
VAR
    row,                { Row subscript  }
    col                 { Column subscript  }
       : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Transition matrix raised');
    WRITELN ('to 1,204th power');
    FOR row := 1 TO size DO
        BEGIN
            WRITELN;
            FOR col := 1 TO size DO
                WRITE (table[row,col]:10:6);
            WRITELN
        END;
END;
{*****************************************
 *    Body of main routine               *
 *****************************************}
BEGIN
    initial;
    parameters (nstates, tran1);
    FOR stage := 1 TO 5 DO
        BEGIN
            square (nstates, tran1, tran2);
            square (nstates, tran2, tran1)
        END;
```

```
    display (nstates, tran1);
    WRITELN;
    WRITELN ('End of program')
END.
```

A test run for this program follows:

```
Program MARKOV2

Estimate steady-state
probabilities by raising
the transition probability
matrix to the 1,204th power.

Number of states of nature ? 2

Transition probability matrix

From row 1
    To col  1 ? 0.8
    To col  2 ? 0.2

From row 2
    To col  1 ? 0.1
    To col  2 ? 0.9

Transition matrix raised
to the 1,024th power

  0.3333329   0.666658
  0.3333329   0.666658

End of program
```

EXERCISES

1. Use program MARKOV1 to generate the state probabilities for twenty stages.
2. Rewrite program MARKOV2 to print the contents of the power matrix resulting from the successive squarings for each iteration.

10.3 SIMULTANEOUS EQUATIONS

The following is an example of a simple linear equation:

$2x = 50.$

The solution to this equation is $x = 25$. The following is a set of two simultaneous linear equations in two unknowns:

$4x + 2y = 60$
$2x + 4y = 48.$

The solution is the value for x and the value for y which jointly satisfy both equations. If the solution exists, it is unique. There will be no more than one set of values.

Elementary linear algebra presents a step-by-step procedure for solving sets of linear simultaneous equations. The first step is to place the coefficients into a matrix exactly as they appear in the equations. The matrix

```
4    2    60
2    4    48
```

contains the coefficients.

An identity matrix has the values 1 down the diagonal and zeros off the diagonal. The matrix

```
1    0
0    1
```

is an identity matrix. Multiplying any row by a constant results in an equivalent matrix having the same ultimate solution values. Subtracting a constant multiple of one row from another row also results in a matrix having the same solution values. Interchanging any two rows does not change the solution values, either.

The step-by-step procedure uses these three row operations to generate an identity matrix in the first n columns. When finished, the right-hand column will contain the solution values for the unknowns.

Divide the first row by 4 to place the value 1 in the upper left-hand corner. The matrix

```
1    .5   15
2    4    48
```

results. Subtract 2 times the first row from the second row to force a 0 in the lower left-hand corner. The matrix

```
1    .5   15
0    3    18
```

results. The first column is the desired first column of the identity matrix.

Divide the second row by 3 to place the 1 in its appropriate position in the identity matrix. The matrix

```
1    .5   15
0    1     6
```

results. Now subtract 2 times the second row from the first row to place a 0 in its appropriate position in the second column. The matrix

```
1    0    12
0    1     6
```

results. The identity matrix is complete; therefore, the first solution value is $x = 12$ and the second is $y = 6$.

Substituting these values into the original equations gives

$$4x + 2y = 4x12 + 2x6 = 60$$

and

$$2x + 4y = 2x12 + 4x6 = 48.$$

The solution values are $x = 12$ and $y = 6$.

During the course of developing the identity matrix there may be a 0 where a 1 belongs. Exchanging that row with another row further down in the matrix is required if a row can be found which has a nonzero value in the needed column. If no such row is available, the system of equations has no solution.

During the course of the calculations the pivot value is that value used as a divisor to force the value 1 where it belongs. The determinant of the matrix is the product of the pivot values. If a nonzero pivot is not available, the determinant is 0. This results when the system of simultaneous linear equations has no solution. A determinant nearly equal to zero may indicate roundoff error problems.

The following program uses the step-by-step procedure to solve systems of simultaneous linear equations:

```
PROGRAM matrix1 (INPUT, OUTPUT);
{ ****************************************
  *     Simultaneous equations           *
  **************************************** }
TYPE
    matrix = ARRAY [1 .. 10, 1 .. 11] OF REAL;
VAR
    table           { Augmented matrix }
        : matrix;
    size            { Number of equations, unknowns }
        : INTEGER;
    singular        { Flag indicating singularity }
        : CHAR;
{ ****************************************
  *     initial message                  *
  **************************************** }
PROCEDURE initial;
BEGIN
    WRITELN;
    WRITELN ('Program MATRIX1');
    WRITELN;
    WRITELN ('Solve a set of simultaneous');
```

```
      WRITELN ('linear equations using');
      WRITELN ('elimination method.')
END;
{ ****************************************
  *     PARAMETERS                       *
  **************************************** }
PROCEDURE PARAMETERS
      (VAR size              { Number of unknowns }
            : INTEGER;
       VAR table             { Augmented matrix }
            : matrix);
VAR
      row,           { Row subscript }
      col            { Column subscript }
         : INTEGER;
BEGIN
      WRITELN;
      WRITE ('Number of equations (unknowns) ? ');
      READLN (size);
      WRITELN;
      WRITELN ('Coefficients for');
      FOR row := 1 TO size DO
            BEGIN
                  WRITELN;
                  WRITELN ('Equation ',row:2);
                  WRITELN;
                  FOR col := 1 TO size DO
                        BEGIN
                              WRITE ('Variable ', col:2, ' ? ');
                              READLN(table[row,col])
                        END;
                  WRITELN;
                  WRITE ('Constant term ? ');
                  col := size + 1;
                  READLN (table[row,col])
            END
END;
{ ****************************************
  *     solve                            *
  **************************************** }
PROCEDURE solve
     (size : INTEGER;
      VAR table : matrix;
      VAR singular : CHAR);
```

```
VAR
    i,              { Index }
    j,              { Index }
    k,              { Index }
    loc             { Location of largest magnitude }
      : INTEGER;
    temp,           { Temporary value for exchange }
    determ,         { Determinant }
    pivot,          { Pivot element value }
    mult            { Multiplier for transformation }
      : REAL;
BEGIN
    determ := 1.0;
    singular := 'N';
    k := 0;
    WHILE k < size AND singular = 'N' DO
        BEGIN
            k := k + 1;
            { Search for largest magnitude }
            loc := k;
            FOR i := k TO size DO
                IF ABS(table[i,k]) > ABS(table[loc,k]) THEN
                    loc := i;
            { Interchange rows }
            FOR j := k TO size + 1 DO
                BEGIN
                    temp := table[k,j];
                    table[k,j] := table[loc,j];
                    table[loc,j] := temp
                END;
            pivot := table[k,k];
            determ := determ * pivot;
            IF determ = 0.0 THEN
                singular := 'Y';
            { Transform augmented matrix }
            IF singular = 'N' THEN
                BEGIN
                    FOR j := k TO size + 1 DO
                        table[k,j] := table[k,j] / pivot;
                    FOR i := 1 TO size DO
                        IF i <> k THEN
                            BEGIN
                                mult := table[i,k];
                                FOR j := k TO size + 1 DO
                                    table[i,j] := table[i,j]
```

```
                                                - mult * table[k,j]
                                    END
                            END
                END;
                WRITELN;
                WRITELN ('Determinant = ',determ)
END;
{ ****************************************
  *     display results                    *
  **************************************** }
PROCEDURE display
     (size : INTEGER;
      VAR table : matrix);
VAR
     row,              { Row subscript }
     col               { Column subscript }
        : INTEGER;
BEGIN
     WRITELN;
     WRITELN ('Variable     Value');
     col := size + 1;
     FOR row := 1 TO size DO
          WRITELN (row:6, table[row,col]:15:5);
END;
{ ****************************************
  *     Body of main program               *
  **************************************** }
BEGIN
     initial;
     parameters (size, table);
     solve (size, table, singular);
     IF singular = 'N' THEN
          display (size, table);
     WRITELN;
     WRITELN ('End of program')
END.
```

The test run using the problem of this section follows:

```
Program MATRIX1

Solve a set of simultaneous
linear equations using
elimination method.

Number of equation (unknowns) ? 2
```

```
Coefficients for
Equation 1
Variable  1 ? 4.0
Variable  2 ? 2.0
Constant term ? 60.0
Equation 2
Variable  1 ? 2.0
Variable  2 ? 4.0
Constant term ? 48.0
Determinant = 1.20000E+01
Variable   Value
    1       12.00000
    2        6.00000
End of program
```

EXERCISES

1. Use the program of this section to solve the following system of equations:

$$\begin{aligned} 6x + 5y - 4z &= 124 \\ x - 4y + 2z &= 200 \\ 3x + y + 3z &= 125 \end{aligned}$$

2. Use the program of this section to solve the following system of equations:

$$\begin{aligned} 12x + 19y &= 2{,}520 \\ 24x + 11y &= 3{,}186 \end{aligned}$$

10.4 LINEAR PROGRAMMING

The purpose of linear programming is to optimize a linear objective function subject to a set of linear constraints. The objective may be to maximize profits or to minimize costs. The problem statement

$$\begin{aligned} \text{Max} \quad & 80X1 + 60X2 \\ \text{S.T.} \quad & 4X1 + 2X2 \leqslant 600 \\ & 2X1 + 4X2 \leqslant 480 \\ & X1, \quad X2 \geqslant 0 \end{aligned}$$

is typical.

To put the numbers in perspective, consider a furniture factory that makes two items, desks and chairs. The factory makes a profit of $80 for each desk and a profit of $60 for each chair. Each desk requires four hours of labor in the mill room and two hours of labor in the finishing department. Each chair requires

two hours in the mill room and four hours in the finishing department. There are 600 manhours of mill room labor available and 480 manhours of finishing department labor available.

What is the optimum production plan? What combination of desks and chairs will maximize total profit and still remain within the labor hour constraints in the two departments? The solution to the problem consists of the number of units of $X1$ (desks), the number of units of $X2$ (chairs), and the resulting total profit.

SLACK VARIABLES

Each constraint has an associated slack variable measured in the same units as the constraint. The value of the slack variable is the unused units of the constraint. It is the value that converts the costraint inequality into an equality. The profit maximization problem becomes

$$\begin{aligned} \text{Max} \quad & 80X1 + 60X2 \\ \text{S.T.} \quad & 4X1 + 2X2 + S1 = 600 \\ & 2X1 + 4X2 + S2 = 480 \\ & X1, \quad X2 \quad S1 \quad S2 \geqslant 0. \end{aligned}$$

The variable $S1$ is the slack variable for constraint 1. The variable $S2$ is the slack variable for constraint 2.

SIMPLEX TABLEAU

The simplex tableau is an extended matrix that looks much like the matrix for solving simultaneous equations. The simplex method for solving linear programming is related to the method for solving equations, including the same column-by-column transformations. The major difference is in the order of selecting columns and in the choice of the pivot row to place the coefficient 1.

The initial tableau comes directly from the initial problem statement including slack variables. The first tableau follows:

80	60	0	0	0
4	2	1	0	600
2	4	0	1	480

The interpretation of the coefficients will always be much as they are for the initial tableau. The top row gives the per-unit contribution of each column variable to the objective function. The right-hand column gives the number of units of the variable represented by the row. All other variables have the value 0. The body of the tableau gives the number of units of the row variable required for each unit of the column variable brought into the solution. The value to the far upper right is the negative of the objective function.

The best variable (column) to bring into the solution is the one having the most desirable per-unit change for the objective function. If no coefficient changes the objective in the desired direction, then the current solution is the optimum solution. The question of how much to bring in depends on the ratios resulting from dividing the right-hand column by the column coefficients for the second and following rows. Ignore those ratios for which the divisors are 0 or negative.

SIMPLEX METHOD ILLUSTRATED

The top row of the initial simplex tableau contains the values 80, 60, 0, 0, and 0. Since the objective is to maximize profit, the best column (variable) to bring in is the one with the largest positive coeffficient. Column 1 (desks) gives the largest per-unit profit contribution.

Each table uses 4 of the 600 available hours in the mill room and 2 of the 480 hours in the finishing department. The mill room constraint limits desk production to 150 units obtained by dividing the column mill room coefficient 4 into the available 600 hours. Column 1 is the pivot column. The second row of the simplex tableau is the first constraint row which is the pivot row.

The simplex procedure places the value 1 in the intersection of the pivot column and pivot row by dividing the pivot row by the pivot element originally located at the intersection of the pivot row and pivot column. The simplex tableau

80	60	0	0	0
1	.5	.25	0	150
2	4	0	1	480

results.

The next step is to transform the tableau so that all other rows including the top row have zeros in the pivot column. These row transformations are the same as those done in solving a system of linear simultaneous equations. Subtract from the row in question some constant multiple of the pivot row. Subtracting from the top row, 80 times the pivot row, places a zero in the pivot column position. Subtracting from the bottom row, 2 times the pivot row, places a zero in its pivot column position. The simplex tableau

0	20	-20	0	-12000
1	.5	.25	0	150
0	3	-.5	1	180

is the result.

The value in the upper right-hand corner is the negative of the value of the objective function. The current solution is to make 150 desks at $80 each for a total profit of $12,000. In the process, there are 180 unused hours of labor in the finishing department.

Column 2 (chairs) now has the coefficient 20. This means that each chair brought into the solution nets $20 profit. How can this be? The binding constraint is the mill room hour constraint. Each chair uses 2 hours of this constraint while each desk uses 4 hours of this constraint. The tradeoff is 2 chairs for each desk or one-half of one desk for each chair. Each chair brings in $60. One-half of the $80 profit per desk gives the tradeoff value of $40 for the loss of one-half of a desk. The net increase in total profit is $20 for each chair brought into the solution.

How many chairs should the production manager trade for desks? The answer is, as many as possible at the mill room constraint substitution rates. Thus begins another round of computing ratios to determine the new pivot row to correspond to the pivot column (chairs). This process continues iteratively until no coefficients in the top row are positive.

The following program uses the simplex method to solve linear programming problems.

```
PROGRAM 1prog1 (INPUT, OUTPUT);
{ ****************************************
  *     Linear Programming                 *
  **************************************** }
VAR
     tableau        { Augmented tableau }
        : ARRAY [1 .. 20, 1 .. 31] OF REAL;
     obj            { Objective function coefficients }
        : ARRAY [1 .. 31] OF REAL;
     basis          { Basis vector subscripts }
        : ARRAY [1 .. 21] OF INTEGER;
     nrows,         { Number of rows in tableau }
     ncols,         { Number of columns in tableau }
     row,           { Row subscript }
     col,           { Column subscript }
     inp,           { Variable to bring in }
     out,           { Variable to drive out }
     optcode        { Optimization code 1=max, -1=min }
        : INTEGER;
     quant,         { Quantity for new variable }
     poss,          { Possible quantity }
     pivot,         { Pivot value for transformation }
     mult           { Constant multiplier for transformation }
        : REAL;
{ ****************************************
  *     Initial message                    *
  **************************************** }
PROCEDURE initial;
```

```
BEGIN
    WRITELN;
    WRITELN ('Program LPROG1');
    WRITELN;
    WRITELN ('Linear programming analysis');
    WRITELN ('maximizing or minimizing a');
    WRITELN ('linear objective function');
    WRITELN ('subject to linear constraints.');
END;
{ ****************************************
  *     Get data                         *
  **************************************** }
PROCEDURE get data;
BEGIN
    { Parameters }
    WRITELN;
    WRITELN ('Type of objective function');
    WRITE ('    1 = Max    -1 = min    ? ');
    READLN (optcode);
    WRITELN;
    WRITE ('Number of constraint equations ? ');
    READLN (nrows);
    WRITELN;
    WRITE ('Number of variables including');
    WRITE ('slack and artificial          ? ');
    READLN (ncols);
    { Objective function }
    WRITELN;
    WRITELN ('Coefficients for objective function');
    WRITELN;
    WRITELN ('Coefficient for');
    FOR col := 1 TO ncols DO
        BEGIN
            WRITE ('Variable ',col:2, ' ? ');
            READLN (obj[col]);
            tableau[1,col] := obj[col]
        END;
    tableau[1, ncols + 1] := 0.0;
    { Constraint equations }
    WRITELN;
    WRITELN ('Constraint equations');
    FOR row := 1 TO nrows DO
        BEGIN
            WRITELN;
            WRITELN ('Constraint ', row:2);
```

```
                WRITELN;
                WRITELN ('Coefficient for');
                FOR col := 1 TO ncols DO
                    BEGIN
                        WRITE ('Variable ', col:2, ' ? ');
                        READLN (tableau[row + 1, col])
                    END;
                WRITELN;
                WRITE ('Constraint capacity ? ');
                READLN (tableau[row + 1, ncols + 1])
            END;
        { Initial basis vector subscripts }
        WRITELN;
        WRITELN ('Subscripts for variables forming');
        WRITELN ('Initial identity matrix');
        WRITELN;
        WRITELN ('Basis variable subscript for');
        FOR row := 1 TO nrows DO
            BEGIN
                WRITE ('Row ', row:2, ' ? ');
                READLN (basis[row])
            END;
        { Adjust for artificial variables }
        FOR row := 1 TO nrows DO
            BEGIN
                inp := basis[row];
                mult := tableau[1,inp];
                IF mult <> 0.0 THEN
                    FOR col := 1 TO ncols + 1 DO
                        tableau[1,col] := tableau[1,col]
                            - mult * tableau[row + 1,col]
            END
END;

{ ****************************************
  *     Bring in                         *
  **************************************** }

PROCEDURE bringin;
BEGIN
    inp := 1;
    FOR col := 2 TO ncols DO
        IF tableau[1,col] * optcode
            > tableau[1,inp] * optcode
            THEN inp := col
END;
```

```
{*****************************************
 *    Drive out                          *
 *****************************************}
PROCEDURE driveout;
BEGIN
    quant := 1E30;
    out := 0;
    FOR row := 2 TO nrows + 1 DO
        IF tableau[row,inp] > 0.0 THEN
            BEGIN
                poss := tableau[row, ncols + 1]
                        /tableau[row,inp];
                IF poss < quant THEN
                    BEGIN
                        quant := poss;
                        out := row
                    END
            END;
    IF out = 0 THEN
        BEGIN
            WRITELN;
            WRITELN ('No feasible solution');
        END
END;
{*****************************************
 *    Transform tableau                  *
 *****************************************}
PROCEDURE transform;
BEGIN
    pivot := tableau[out, inp];
    FOR col := 1 TO ncols + 1 DO
        tableau[out,col] := tableau[out,col] / pivot;
    FOR row := 1 TO nrows + 1 DO
        IF row <> out THEN
            BEGIN
                mult := tableau[row,inp];
                FOR col := 1 TO ncols + 1 DO
                    tableau[row,col] := tableau[row,col]
                        - mult * tableau[out,col]
            END;
    basis[out - 1] := inp
END;
{*****************************************
 *    Display results                    *
 *****************************************}
```

```
PROCEDURE display;
BEGIN
    WRITELN;
    WRITELN ('Variable     Value');
    FOR row := 1 TO nrows DO
        BEGIN
            col := basis[row];
            WRITELN (col:5, tableau[row + 1, ncols + 1]:15:5);
        END;
    WRITELN;
    WRITE ('Value of obj function ');
    WRITELN (-tableau[1, ncols + 1]:15:5)
END;
{ ****************************************
  *     Body of main program             *
  **************************************** }
BEGIN
    initial;
    getdata;
    REPEAT
        bringin;
        IF tableau[1,inp] * optcode <= 0 THEN
            inp := 0
        ELSE
            BEGIN
                driveout;
                IF out > 0 THEN
                    transform
            END
    UNTIL inp = 0 OR out = 0;
    display;
    WRITELN;
    WRITELN ('End of program');
END.
```

The test run follows:

```
Program LPROG1

Linear programming analysis
maximizing or minimizing a
linear objective function
subject to linear constraints.

Type of objective function
  1 = Max   -1 = min      ? 1

Number of constraint equations ? 2
```

```
Number of variables including
slack and artificial              ? 4

Coefficient for objective function

Coefficient for
Variable  1 ? 80.0
Variable  2 ? 60.0
Variable  3 ? 0.0
Variable  4 ? 0.0

Constraint equation

Constraint 1

Coefficient for
Variable  1 ? 4.0
Variable  2 ? 2.0
Variable  3 ? 1.0
Variable  4 ? 0.0

Constraint capacity ? 600.0

Constraint 2

Coefficient for
Variable  1 ? 2.0
Variable  2 ? 4.0
Variable  3 ? 0.0
Variable  4 ? 1.0

Constraint capacity ? 480.0

Subscript for variables forming
initial identity matrix

Basis variable subscript for
Row  1 ? 3
Row  2 ? 4

Variable    Value
    1     120.00000
    2      60.00000

Value of objective function          13200.00000

End of program
```

MINIMIZATION AND ARTIFICIAL VARIABLE

Minimization problems require that at least some constraints be of the $\geqslant$ type. Constraints of this type result in -1 coefficients for the slack variables. The approach used by most introductory quantitative methods textbooks is to define artificial variables with very undesirable objective function coefficients to give a starting identity matrix.

The program checks for these nonzero objective function coefficients first. It then does row transformations on the first row to adjust it to reflect this first starting solution. If the set of constraints define a feasible space (one that is non-empty), the undesirable coefficients for the artificial variables assure that they will be driven out of the optimum solution.

Consider the problem statement

$$\begin{array}{lll} \text{Min} & 80X1 + 60X2 & \\ \text{S.T.} & 4X1 + 2X2 & \geqslant 600 \\ & 2X1 + 4X2 & \geqslant 480 \\ & X1, \quad X2 & \geqslant 0 \end{array}$$

which has $\geqslant$ constraints and requires minimization. The augmented problem statement

$$\begin{array}{lll} \text{Min} & 80X1 + 60X2 & \\ \text{S.T.} & 4X1 + 2X2 - S1 & = 600 \\ & 2X1 + 4X2 - S2 & = 480 \\ & X1, \quad X2, \; S1, \; S2 & \geqslant 0 \end{array}$$

adds the slack variables but does not have a starting identity matrix.

Artificial variables provide a starting solution for the simplex procedure. Since the problem is to minimize the objective function, large objective function coefficients for the artificial variables assure that they will not appear in the optimum solution. The problem statement

$$\begin{array}{lll} \text{Min} & 80X1 + 60X2 + 10000A1 + 10000A2 & \\ \text{S.T.} & 4X1 + 2X2 - S1 + A1 & = 600 \\ & 2X1 + 4X2 - S2 + A2 & = 480 \\ & X1, \quad X2, \; S1, \; S2, \; A1, \; A2 & \geqslant 0 \end{array}$$

includes both slack and artificial variables.

The following test run uses the program of this section to solve this minimization problem:

```
Program LPROG1

Linear programming analysis
maximizing or minimzing a
linear objective function
subject to linear constraints.

Type of objective function
  1 = Max   -1 = Min   ? -1

Number of constraint equations ? 2

Number of variables including
slack and artificial          ? 6
```

```
Coefficient for
Variable  1 ? 80.0
Variable  2 ? 60.0
Variable  3 ? 0.0
Variable  4 ? 0.0
Variable  5 ? 10000
Variable  6 ? 10000

Constraint equations

Constraint 1

Coefficient for
Variable  1 ? 4.0
Variable  2 ? 2.0
Variable  3 ? -1.0
Variable  4 ? 0.0
Variable  5 ? 1.0
Variable  6 ? 0.0

Constraint capacity ? 600.0

Constraint 2

Coefficient for
Variable  1 ? 2.0
Variable  2 ? 4.0
Variable  3 ? 0.0
Variable  4 ? -1.0
Variable  5 ? 0.0
Variable  6 ? 1.0

Constraint capacity ? 480

Subscripts for variables forming
initial identity matrix

Basis variable subscript for
Row  1 ? 5
Row  2 ? 6

Variable    Value
    1     120.00000
    2      60.00000

Value of objective function        13200.00000

End of program
```

EXERCISES

1. Use the program of this section to solve the following linear programming problem:

$$\begin{array}{ll} \text{Max} & 12X1 + 25X2 + 15X3 \\ \text{S.T.} & 2X1 + X2 + 4X3 \leqslant 400 \\ & 5X1 + 3X2 + 2X3 \leqslant 600 \end{array}$$

2. Use the program of this section to solve the following linear programming problem:

$$\begin{array}{ll} \text{Min} & 6X1 + 5X2 \\ \text{S.T.} & X1 \geqslant 50 \\ & X2 \geqslant 20 \\ & 3X1 + 4X2 \leqslant 200 \end{array}$$

11 Calculus

OVERVIEW Calculus is the mathematical study of movement and change with applications to virtually all areas of science and economics. The origins date back to Newton and Leibnitz. The first applications were in physics, astronomy, and geometry. Immediate applications are to find the slope of the tangent line of a curve and finding the area bounded by a curve.

The basic calculus operations are differentiation and integration. Differential calculus involves finding the rate of change of a function. Integral calculus concerns the inverse problem of finding the function that has a given rate of change.

Sequences and series are important topics in calculus. A sequence is an ordered set of terms calculated by using a specified rule. A series is the sum of a sequence of terms. Calculus approximations abound using sequences, series, and limits. A limit is a troublesome concept from the perspectives of both theory and practice.

□

11.1 SEQUENCES

A sequence is an ordered set of values derived using a specified rule. The sequence of values

1 2 3 4 5 ...

consists of the set of integers. The rule

$x_i = i$ for $i = 1, 2, \ldots$

describes the sequence. The rule

$x_i = 1/i^2$ for $i = 1, 2, \ldots$

describes another sequence. The values

1 1/4 1/9 1/16 ...

are the terms of this sequence. Both of these are infinite sequences.

A sequence may have a limit, but many do not have finite limits. A sequence has a limit if its larger numbered terms converge to a finite value. The first sequence does not converge. The second sequence converges to the value 0. In the limit, as i approaches infinity, the term $1/i^2$ approaches 0.

SEQUENCE ESTIMATE OF EXP(*X*)

The limit of the sequence defined by $(1+x/n)^n$ is e^x as n approaches infinity. The following program computes the nth term of the sequence for a given value of x:

```
PROGRAM seq1 (INPUT, OUTPUT);
{ Compute nth term of sequence (1 + x/n)^n }
VAR
    x,          { value of x }
    y           { Value of nth term of sequence }
      : REAL;
    n           { Number of term }
      : INTEGER;
FUNCTION power (a : REAL; n : INTEGER) : REAL;
{ Raise a to the nth power }
BEGIN
    power := EXP(n * LN(a))
END;
BEGIN
    WRITELN;
    WRITELN ('Compute the value (1 + x/n)');
    WRITELN ('raised to the nth power as');
    WRITELN ('estimate of exp(x).');
    WRITELN;
    WRITE ('Value for x, value for n ? ');
    READLN (x, n);
    REPEAT
        y := power(1.0 + x / n, n);
        WRITELN ('Result = ',y:15:5);
        WRITELN;
        WRITE ('Value for x, value for n ? ');
        READLN (x, n)
    UNTIL n = 0
END.
```

The test run computes the terms of the sequence for $x = 1$ and several values of n:

```
Compute the value (1 + x/n)
raised to the nth power as
estimate of exp(x).
```

```
Value for x, value for n ? 1.0 1
Result =        2.00000
Value for x, value for n ? 1.0 2
Result =        2.25000
Value for x, value for n ? 1.0 3
Result =        2.37037
Value for x, value for n ? 1.0 4
Result =        2.44141
Value for x, value for n ? 1.0 10
Result =        2.59374
Value for x, value for n ? 1.0 100
Result =        2.70478
Value for x, value for n ? 1.0 1000
Result =        2.71689
Value for x, value for n ? 0.0 0
```

The value n = 0 terminates the program.

INTEREST RATE COMPOUNDING

A savings institution compounding interest quarterly has a higher effective interest rate than another institution offering the same rate but compounding annually. An institution offering monthly compounding has an even better effective rate. Some savings institutions offer daily compounding. How much better is daily compounding than annual compounding? What about continuous compounding?

The expression

$$p(1+r/n)^n$$

gives the value at the end of the year of the present amount p earning interest at $100r$ percent interest compounded n times per year. The term

$$(1+r/n)^n$$

is the multiplier of p giving the value at the end of the year. Subtracting this multiplier from 1 gives the effective interest rate as a fraction.

The multiplier is the sequence $(1+x/n)^n$ which has $\exp(x)$ as its limit. Continuous compounding gives the multiplier $\exp(r)$ for the interest rate r given as a fraction. The following program asks for an interest rate and computes the annual effective interest rates for annual, quarterly, monthly, daily, and continuous compounding:

```
PROGRAM seq2 (INPUT, OUTPUT);
{ Compare interest compounding frequencies }
```

```
VAR
    interest,       { Nominal interest rate }
    effective       { Effective interest rate }
      : REAL;
FUNCTION compound
    (interest       { Nominal interest rate }
      : REAL;
    number          { Compounding periods per year }
      : INTEGER) : REAL;
BEGIN
    compound := 100.0 * (-1.0
      + EXP(number * LN(1.0 + interest / number)))
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Compare effective interest');
    WRITELN ('rates for several different');
    WRITELN ('compounding frequencies');
    { Get data }
    WRITELN;
    WRITE ('Interest rate (per cent) ? ');
    READLN (interest);
    interest := interest / 100.0;
    { Compare compounding frequencies }
    WRITELN;
    WRITELN ('Compounding      Effective');
    WRITELN ('periods          rate');
    effective := 100.0 * interest;
    WRITELN ('    1        ', effective:10:3);
    effective := compound (interest, 4);
    WRITELN ('    4        ', effective:10:3);
    effective := compound (interest, 12);
    WRITELN ('   12        ', effective:10:3);
    effective := compound (interest, 365);
    WRITELN ('  365        ', effective:10:3);
    effective := 100.0 * (EXP(interest) - 1.0);
    WRITELN ('continuous   ', effective:10:3);
    WRITELN;
    WRITELN ('End of program')
END.
```

A sample test run follows:

```
Compare effective interest
rates for several different
compounding frequencies
```

```
Interest rate (per cent) ? 12.5
Compounding     Effective
periods         rate
     1          12.500
     4          13.098
    12          13.241
   365          13.304
continuous      13.315
End of program
```

LIMITS AT POINT HAVING UNDEFINED VALUE

The expression $(x^2 - 4)/(x - 2)$ is undefined at the point $x = 2$ since the denominator $x - 2$ becomes 0. The limit as x approaches 2 is defined. The following program estimates this limit by trying values that are closer and closer to the value 2.0:

```
PROGRAM seq3 (OUTPUT);
{ Limit of (x*x - 4)/(x - 2) for x approaching 2 }
VAR
    width,          { Width of limit }
    left,           { Evaluation of left limit }
    right           { Evaluation of right limit }
        : REAL;
    counter         { Loop control variable for iterations }
        : INTEGER;
FUNCTION fnc (arg : REAL) : REAL;
{ Function to evaluate }
BEGIN
    fnc := (arg * arg - 4.0) / (arg - 2.0)
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Evaluate function');
    WRITELN ('(x*x - 4) / (x - 2)');
    WRITELN ('at the limit x = 2');
    { Generate table }
    WRITELN;
    WRITELN ('Error      Left limit      Right limit');
    width := 1.0;
    FOR counter := 1 TO 7 DO
        BEGIN
            width := width / 10.0;
            left := fnc(2.0 - width);
            right := fnc (2.0 + width);
```

```
            WRITELN (width, left:15:8, right:15:8)
        END;
END.
```

The test run follows:

```
Evaluate function
(x*x - 4) / (x - 2)
at the limit x = 2
Error          Left limit          Right limit
 1.00000E-01    3.90000000          4.10000000
 1.00000E-02    3.99001000          4.01001000
 1.00000E-03    3.99905000          4.00095000
 1.00000E-04    4.00000000          4.00000000
 1.00000E-05    4.00000000          4.00000000
 1.00000E-06    4.00000000          4.00000000
RUNTIME ERROR 05 DIVIDE BY 0
```

The error message indicates an attempt to divide by 0. The single precision real number representation cannot distinguish between the value 2.0 and the value 1.9999999. Subtracting .0000001 from 2.0 results in 2.0. Similarly, adding the value .0000001 to 2.0 gives 2.0. The denominator eventually goes to 0, causing the divide by 0 error. Single precision real numbers have only six significant digits of precision.

LIMIT OF AN INFINITE SEQUENCE

Determine the limit of

$(x + 1/x)/(2x - 1/x)$

as x approaches infinity. The following program evaluates this limit for $x = 1$, 10, 100, etc.:

```
PROGRAM seq4 (OUTPUT);
{ Limit of (x+1/x)/(2x-1/x) as x goes to infinity }
VAR
    x,          { Value of x }
    y           { Value of f(x) }
       : REAL;
    i           { Loop control variable }
       : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Estimate the limit');
    WRITELN ('of the function');
```

```
    WRITELN ('(x+1/x)/(2x-1/x)');
    WRITELN ('as x goes to infinity.');
    WRITELIN;
    WRITELN ('Value of x     Value of function');
    x := 1.0;
    FOR i := 1 TO 8 DO
        BEGIN
            y := (x + 1.0 / x) / (2.0 * x - 1.0 / x);
            WRITELN (x:15:2, y:15:8);
            x := x * 10.0
        END
END.
```

The test run follows:

```
Estimate the limit
of the function
(x+1/x)/(2x-1/x)
as x goes to infinity.
Value of x      Value of function
          1.00      2.00000000
         10.00      0.50753800
        100.00      0.50007500
       1000.00      0.50000100
      10000.00      0.50000000
     100000.00      0.50000000
    1000000.00      0.50000000
   10000000.00      0.50000000
```

EXERCISES

1. Write a program that computes the first 10 terms of the sequence $1/2^n$ for $n = 1, 2, \ldots, 10$.
2. Write a program to estimate the limit of $1/2^n$ as n approaches infinity.
3. Write a program to estimate the limit of the term

   $(3x^2 - 5x + 4)/(x^2 + 2)$

   as x approaches infinity.
4. Write a program to estimate the limit of the term

   $(x^2 - 16)/(x - 4)^2$

   as x approaches 4.

11.2 SERIES

A sequence is a succession of terms formed according to a fixed rule. A series is the sum of the terms of a sequence. The sum of the sequence 1, 2, . . . , 5 is the series

1 + 2 + 3 + 4 + 5 = 15.

The sum of the infinite series

1 + 1/2 + 1/4 + 1/8 + . . .

is 2.0. In this case the infinite series converges to a finite value. Not all infinite series have a finite limit. The infinite series

1 + 2 + 3 + 4 + . . .

does not have a finite limit.

FINITE SERIES

The simple series giving the sum of the successive integers has the value 15 for its first five terms. The following program evaluates this finite series:

```
PROGRAM series1 (INPUT, OUTPUT);
{ Finite series, sum of the first few integers }
VAR
    value,      { Current value }
    number,     { Upper limit }
    sum         { Sum of the values }
        : INTEGER;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SERIES1');
    WRITELN;
    WRITELN ('Compute the sum of the');
    WRITELN ('first few integers');
    { Problem parameter }
    WRITELN;
    WRITE ('Number of terms ? ');
    READLN (number);
    { Process }
    WRITELN;
    WRITELN ('Term      Sum');
    sum := 0;
    FOR value := 1 TO number DO
        BEGIN
            sum := sum + value;
```

```
            WRITELN (value:4, sum:10)
        END;
    { Final message }
    WRITELN;
    WRITELN ('End of program')
END.
```

The test run follows:

```
Program SERIES1

Compute the sum of the
first few integers

Number of terms ? 5

Term      Sum
   1        1
   2        3
   3        6
   4       10
   5       15

End of program
```

FUTURE VALUE OF REGULAR DEPOSITS

Many pension plans consist of regular deposits by employers or employees into a savings account held in trust until retirement. One question concerns the future value at the end of a specified number of years. What will the future accumulated value be at the end of 20 years for a plan involving $50.00 monthly deposits? Assume an interest rate of 12.5 percent compounded monthly at the time of deposit.

The following program evaluates plans consisting of regular deposits:

```
PROGRAM series2 (INPUT, OUTPUT);
{ Finite series, future value of an annuity }
VAR
    interest,       { Annual interest rate (per cent }
    rate,           { Monthly interest multiplier }
    deposit,        { Monthly deposit }
    future          { Future value of the annuity }
        : REAL;
    number,         { Number of monthly deposits }
    current         { Current month }
        : INTEGER;
BEGIN
    { Initial message }
    WRITELN;
```

```
    WRITELN ('Program SERIES2');
    WRITELN;
    WRITELN ('Compute the future value');
    WRITELN ('or regular deposits into');
    WRITELN ('a savings account');
    { problem parameters }
    WRITELN;
    WRITE ('Size of regular monthly deposit ? ');
    READLN (deposit);
    WRITE ('Number of monthly deposits    ? ');
    READLN (number);
    WRITE ('Interest rate (per cent)      ? ');
    READLN (rate);
    { Process }
    rate := 1.0 + rate / 1200.0;
    future := 0.0;
    FOR current := 1 TO number DO
        future := (future + deposit) * rate;
    WRITELN;
    WRITELN ('Future value ', future:12:2);
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program SERIES2

Compute the future value
of regular deposits into
a savings account.

Size of regular monthly deposit ? 50.00
Number of monthly deposits    ? 240
Interest rate (per cent)      ? 12.5

Future value      53475.60

End of program
```

INFINITE SERIES

Consider the infinite series

$1 + 1/2 + 1/4 + 1/8 \ldots$

which has $1/2^n$ as its nth term (assuming 1 is the 0th term). Does this infinite series have a finite value? If so, what is its value? The following program permits the interactive trial of several upper limits to estimate the limit:

```
PROGRAM series3 (INPUT, OUTPUT);
{ Infinite series, 1 + 1/2 + 1/4 + 1/8 . . . }
VAR
    number,         { Number of terms }
    counter         { Loop control variable }
      : INTEGER;
    sum,            { Sum of the terms }
    value           { Value of the current term }
      : REAL;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SERIES3');
    WRITELN;
    WRITELN ('Estimate value of the infinite');
    WRITELN ('series 1 + 1/2 + 1/4 + 1/8 . . . ');
    WRITELN ('by taking a specified number of terms.');
    { Process }
    WRITELN;
    WRITE ('Number of terms ? ');
    READLN (number);
    REPEAT
        sum := 0.0;
        value := 2.0;
        FOR counter := 1 TO number DO
            BEGIN
                value := value / 2.0;
                sum := sum + value
            END;
        WRITELN ('Sum of terms = ', sum:15:8);
        WRITELN;
        WRITE ('Number of terms (use 0 to terminate)? ');
        READLN (number)
    UNTIL number = 0;
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program SERIES3
Estimate the value of the infinite
series 1 + 1/2 + 1/4 + 1/8 . . .
by taking a specified number of terms.
```

```
Number of terms ? 2
Sum of terms =        1.50000000

Number of terms (use 0 to terminate) ? 5
Sum of terms =        1.93750000

Number of terms (use 0 to terminate) ? 10
Sum of terms =        1.99805000

Number of terms (use 0 to terminate) ? 15
Sum of terms =        1.99994000

Number of terms (use 0 to terminate) ? 0

End of program
```

The limit of the infinite series appears to converge to 2.0.

POWER SERIES ESTIMATE OF EXP(*X*)

The series $1 + 1/2 + 1/4 + 1/8 + \ldots$ is an example of a power series. The nth term $1/2^n$ contains a power of 2. The series

$$1 + x + x^2/2 + x^3/3! + x^4/4! + \ldots$$

converges to the value e^x. The nth term $x^n/n!$ is the ratio x to the nth power divided by $n!$ (n factorial). The value of $n!$ is the product of the first n integers.

Recursive formulas simplify the calculation of the successive terms of the series. Multiplying the previous term by the ratio x/n gives the new term. The value of the series for the first n terms is the sum of those terms. The following program uses this recursive method to calculate the individual terms of the series:

```
PROGRAM series4 (INPUT, OUTPUT);
{ Power series estimate of exp(x) }
VAR
    number,         { Number of terms }
    counter         { Loop control variable }
      : INTEGER;
    arg,            { Value of argument }
    term,           { Current term in the series }
    sum             { Sum of the terms in the series }
      : REAL;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SERIES4');
    WRITELN;
    WRITELN ('Estimate exp(x) using');
    WRITELN ('power series expansion.');
```

```
    { Problem parametes }
    WRITELN;
    WRITE ('Value of x        ? ');
    READLN (arg);
    WRITE ('Number of terms ? ');
    READLN (number);
    { Process }
    REPEAT
        sum := 1.0;
        term := 1.0;
        FOR counter := 1 TO number DO
            BEGIN
                term := term * arg / counter;
                sum := sum + term
            END;
        WRITELN ('Estimate of exp(x) = ', sum:15:8);
        WRITELN;
        WRITE ('Number of terms (use 0 to terminate)? ');
        READLN (number)
    UNTIL number = 0;
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

The following test run estimates e^2:

```
Program SERIES4

Estimate exp(x) using
power series expansion.

Value of x        ? 2.0
Number of terms ? 1
Estimate of exp(x)          3.00000000

Number of terms (use 0 to terminate) ? 2
Estimate of exp(x)          5.00000000

Number of terms (use 0 to terminate) ? 3
Estimate of exp(x)          6.33333000

Number of terms (use 0 to terminate) ? 5
Estimate of exp(x)          7.26667000

Number of terms (use 0 to terminate) ? 10
Estimate of exp(x)          7.38906000

Number of terms (use 0 to terminate) ? 20
Estimate of exp(x)          7.38906000
```

Number of terms (use 0 to terminate) ? 0

End of program

The series converges to the value 7.38906 by the 15th term.

EXERCISES

1. Write a program to determine the limit, if it exists, of the infinite series

   $1/2 - 3/4 + 5/8 - 7/16 + - \ldots .$

2. Write a program to estimate the limit, if it exists, of the infinite series

   $x + x/1.15 + x/1.15^2 + x/1.15^3 + \ldots$

   and test using $x = 100.00$.

3. Write a program to estimate the limit, if it exists, of the infinite series

   $1/2 - 4/4 + 9/8 - 16/16 + 25/32 - 36/64 + \ldots$

   The signs alternate and the expression $n^2/2^n$ gives the magnitude of the nth term.

11.3 ESTIMATING THE SLOPE

Differential calculus involves estimating the slope of the tangent line to a function $f(x)$. Calculus courses present methods for analytically determining the derivative function $f'(x)$ giving the slope of $f(x)$ for every defined value of x.

The computer is a powerful tool for estimating the slope of a function at a point. Consider estimating the slope of $f(x)$ at the point $x = a$. Select two points $x1$ and $x2$ close together and evaluate $y1 = f(x1)$ and $y2 = f(x2)$. The equation

$$b = (y1 - y2)/(x1 - x2)$$

is approximately equal to the slope.

Let w be a small incremental value. The equation

$$b = (f(a + w) - f(a))/w$$

is an equivalent approximation of the slope at the point $x = a$. The limit as w approaches 0 is the derivative (slope of tangent line) at that point. This is the right-hand limit since the limit $x + a$ approaches x from the right. The equation

$$b = (f(a - w) - f(a))/(-w)$$

gives the left-hand limit as w approaches 0 since it approaches $x = a$ from the left.

Many pathologies result in an undefined derivative. The function must be defined and continuous at the point in question. The left-hand and right-hand limits for the derivative must be equal and finite.

The natural logarithm $ln(x)$ has a derivative $1/x$. The derivative of $ln(x)$ at the point $x = 3$ is 1/3. The programs of this section tackle the computational task of estimating this slope. Estimating the slope of a function is a difficult computing task because it involves the subtraction of values that are almost equal.

Subtract the value 3.754689 from the value 3.754713. The difference is .000024. The real data type only allows six or seven significant-digit precision. Normalization for additions and subtractions requires aligning the decimal points. In this simple example, each term in the subtraction has seven significant digits. The result has only two significant digits.

Carefully select the interval width for use in estimating the slope of the function. Too broad an interval does not give an accurate estimate of the slope. Too narrow an interval will result in roundoff errors caused by the subtraction of nearly equal quantities.

DEFINITIONAL FORMULA $(Y1 - Y2)/(X1 - X2)$

The following program estimates the derivative of $ln(x)$ using the definitional formula

$$(f(x+w) - f(x))/((x+w) - x)$$

```
PROGRAM slope1 (INPUT, OUTPUT);
{ Estimate slope b = (f(x+w) - f(x))/((x+w) - x) }
VAR
    arg,        { Argument x for slope estimate }
    width,      { Width of interval for estimate }
    value1,     { Value of f(x) at x }
    value2,     { Value of f(x) at x + width }
    slope       { Estimated slope of f(x) at x }
      : REAL;
    number,     { Number of interval widths to try }
    counter     { Loop control variable }
      : INTEGER;
FUNCTION fnc (x : REAL) : REAL;
BEGIN
    fnc := LN(x)
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SLOPE1');
    WRITELN;
    WRITELN ('Estimate the slope of f(x)');
    WRITELN ('at a point x using the');
    WRITELN ('following formula');
```

```
    WRITELN ('(f(x+w)-f(x))/((x+w)-x)');
    { Problem parameters }
    WRITELN;
    WRITE ('Estimate slope at point x = ? ');
    READLN (arg);
    WRITE ('Number of interval widths ? ');
    READLN (number);
    { Process }
    WRITELN;
    WRITELN ('Interval      Estimated');
    WRITELN ('width         slope');
    width := 1.0;
    FOR counter := 1 TO number DO
        BEGIN
            width := width / 10.0;
            value1 := fnc(arg);
            value2 := fnc(arg + width);
            slope := (value2 - value1)
                     / ((arg + width) - arg);
            WRITELN (width, slope)
        END;
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program SLOPE1

Estimate the slope of f(x)
at a point x using the
following formula
(f(x+w)-f(x))/((x+w)-x)

Estimate slope at point x = ? 3.0
Number of interval widths ? 10

Interval              Estimated
width                 slope
 1.000000E-01  3.27899E-01
 1.000000E-02  3.32773E-01
 1.000000E-03  3.33333E-01
 1.000000E-04  3.31742E-01
 1.000000E-05  3.33333E-01
 1.000000E-06  3.75000E-01
```

RUN TIME ERROR 05 DIVIDE BY 0

The value of w eventually approaches the value 3.0 so closely that 3.0+w gives 3.0 exactly. When that happens the denominator $(x+w)-x$ is 0, resulting in the divide by 0 message.

ALTERNATE DEFINITIONAL FORMULA

The alternate definitional formula

$$(f(x+w)-f(x))/w$$

avoids dividing by zero, but $x+w$ and x will eventually be equal in a machine that has a finite level of precision. The following program uses this formula:

```
PROGRAM slope2 (INPUT, OUTPUT);
{ Estimate slope b = (f(x+w) - f(x))/w }
VAR
    arg,          { Argument x for slope estimate }
    width,        { Width of interval for estimate }
    value1,       { Value of f(x) at x }
    value2,       { Value of f(x) at x + width }
    slope         { Estimated slope of f(x) at x }
      : REAL;
    number,       { Number of interval widths to try }
    counter       { Loop control variable }
      : INTEGER;
FUNCTION fnc (x : REAL) : REAL;
BEGIN
    fnc := LN(x)
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SLOPE2');
    WRITELN;
    WRITELN ('Estimate the slope of f(x)');
    WRITELN ('at a point x using the');
    WRITELN ('following formula');
    WRITELN ('(f(x+w)-f(x))/w');
    { Problem parameters }
    WRITELN;
    WRITE ('Estimate slope at point x = ? ');
    READLN (arg);
    WRITE ('Number of interval widths ? ');
    READLN (number);
    { Process }
```

```
    WRITELN;
    WRITELN ('Interval      Estimated');
    WRITELN ('width         slope');
    width := 1.0;
    FOR counter := 1 TO number DO
        BEGIN
            width := width / 10.0;
            value1 := fnc(arg);
            value2 := fnc(arg + width);
            slope := (value2 - value1) / width;
            WRITELN (width, slope)
        END;
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program SLOPE2

Estimate the slope of f(x)
at a point x using the
following formula
(f(x+w)-f(x))/w

Estimate the slope at the point x = ? 3.0
Number of interval widths          ? 10

Interval          Estimated
width             slope
 1.00000E-01  3.27898E-01
 1.00000E-02  3.32773E-01
 1.00000E-03  3.33309E-01
 1.00000E-04  3.31402E-01
 1.00000E-05  3.33786E-01
 1.00000E-06  3.57628E-01
 1.00000E-07  0.00000E+00
 1.00000E-08  0.00000E+00
 1.00000E-09  0.00000E+00
 1.00000E-10  0.00000E+00

End of program
```

If the program doesn't abort, the interval width gets successively smaller, and the estimate of the slope goes to zero. Alcor Pascal single precision real values have about six significant digits of accuracy. A double precision mode offering 16 digits of accuracy is available as a compiler option. However, the built-in functions do not provide double precision accuracy.

SINGLE PRECISION ESTIMATE OF SLOPE

The function $f(x) = x^2$ has the derivative function $f'(x) = 2x$. The derivative of $f(x)$ at the point $x = 2$ is $f'(x = 2) = 4$. The following program uses single precision real numbers with six significant digits to estimate the slope:

```
PROGRAM slope3 (INPUT, OUTPUT);
{ Estimate slope of f(x) using single precision }
VAR
    arg,        { Argument x for slope estimate }
    width,      { Width of interval for estimate }
    value1,     { Value of f(x) at x }
    value2,     { Value of f(x) at x + width }
    slope       { Estimated slope of f(x) at x }
      : REAL;
    number,     { Number of interval widths to try }
    counter     { Loop control variable }
      : INTEGER;
FUNCTION fnc (x : REAL) : REAL;
BEGIN
    fnc := x * x
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SLOPE3');
    WRITELN;
    WRITELN ('Estimate the slope of');
    WRITELN ('f(x) at a point x using');
    WRITELN ('single precision.');
    { Problem parameters }
    WRITELN;
    WRITE ('Estimate slope at point x = ? ');
    READLN (arg);
    WRITE ('Number of interval widths ? ');
    READLN (number);
    { Process }
    WRITELN;
    WRITELN ('Interval      Estimated');
    WRITELN ('width         slope');
    width := 1.0;
    FOR counter := 1 TO number DO
        BEGIN
            width := width / 10.0;
            value1 := fnc(arg);
            value2 := fnc(arg + width);
```

```
            slope := (value2 - value1) / width;
            WRITELN (width, slope)
        END;
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

The test run follows:

```
Program SLOPE3

Estimate the slope of
f(x) at a point x using
single precision.

Estimate slope at point x = ? 2.0
Number of interval widths  ? 10

Interval          Estimated
width             slope
 1.00000E-01  4.09999E+00
 1.00000E-02  4.01001E+00
 1.00000E-03  4.00066E+00
 1.00000E-04  3.99590E+00
 1.00000E-05  4.00543E+00
 1.00000E-06  3.81470E+00
 1.00000E-07  0.00000E+00
 1.00000E-08  0.00000E+00
 1.00000E-09  0.00000E+00
 1.00000E-10  0.00000E+00

End of program
```

The best estimate of the slope results from using $w = .001$.

DOUBLE PRECISION

Alcor Pascal provides several compiler options including the specification that real variables use a 16-digit double precision format. Double precision variables require more memory and the arithmetic takes longer, but they offer much greater precision.

Compiler options appear as comments within the program listing with the dollar sign as the first symbol. The double precision comment

```
{ $DOUBLE }
```

must appear before the PROGRAM header line of the program. The default output formats display 15 significant digits. The following program uses double precision in the estimate of the slope:

```
{ $DOUBLE }
PROGRAM slope4 (INPUT  OUTPUT);
{ Estimate slope of f(x) using double precision }
VAR
     arg,          { Argument x for slope estimate }
     width,        { Width of interval for estimate }
     value1,       { Value of f(x) at x }
     value2,       { Value of f(x) at x + width }
     slope         { Estimated slope of f(x) at x }
        : REAL;
     number,       { Number of interval widths to try }
     counter       { Loop control variable }
        : INTEGER;
FUNCTION fnc (x : REAL) : REAL;
BEGIN
     fnc := x * x
END;
BEGIN
     { Initial message }
     WRITELN;
     WRITELN ('Estimate the slope of');
     WRITELN ('f(x) at a point x using');
     WRITELN ('double precision.');
     { Problem parameters }
     WRITELN;
     WRITE ('Estimate slope at point x = ? ');
     READLN (arg);
     WRITE ('Number of interval widths  ? ');
     READLN (number);
     { Process }
     WRITELN;
     WRITELN ('Interval                Estimated');
     WRITELN ('width                   slope');
     width := 1.0;
     FOR counter := 1 TO number DO
          BEGIN
               width := width / 100.0;
               value1 := fnc(arg);
               value2 := fnc(arg + width);
               slope := (value2 - value1) / width;
               WRITELN (width, slope)
          END;
     { Final message }
     WRITELN;
     WRITELN ('End of program');
END.
```

Sample output from this program follows:

```
Program SLOPE4

Estimate the slope of
f(x) at a point x using
double precision.

Estimate slope at point x = ? 2.0
Number of interval widths  ? 10

Interval                    Estimated
width                       slope
 1.00000000000000D-02 4.0100000000000D+00
 1.00000000000000D-04 4.0001000000001D+00
 1.00000000000000D-06 4.0000009998709D+00
 1.00000000000000D-08 4.0000000089968D+00
 1.00000000000000D-10 4.0000003309615D+00
 1.00000000000000D-12 3.9999115131195D+00
 1.00000000000000D-14 3.9968028886506D+00
 1.00000000000000D-16 4.4408920985006D+00
 1.00000000000000D-18 1.4693679385279D-21
 1.00000000000000D-20 1.4693679385279D-19

End of program
```

The best estimated slope for this example is accurate to eight significant digits for an interval width of 10^{-8}.

EXERCISES

1. Write a program to estimate the derivative of the following function at the point $x = 2$:

   $3/(1 + 2x^2)$

2. Write a program to estimate the derivative of the following function at the point $x = 1.5$:

   $26 - 2x - 4x^2$

11.4 AREA UNDER A CURVE

Integration and differentiation are inverse operations in a manner similar to multiplication and division. The indefinite integral of the function $f(x)$ is that function which has $f(x)$ as its derivative. There may be a family of functions to choose from. For example, the functions

$3x^2 - 2x + 4$

and

$3x^2 - 2x + 12$

both have

$6x - 2$

as their derivative function. The constant term is always involved since the derivative of that constant drops out.

A definite integral corresponds to the area under a curve if the function value $f(x)$ is positive within the region of interest. Numerical integration requires dividing the region into pieces and estimating the integral as the sum of the areas of the pieces. Two approaches commonly introduced in elementary calculus are the trapezoid method and Simpson's method.

The function

$f(x) = x^2 - 3x + 7$

has

$(1/3)x^3 - (3/2)^2 + 7X + C$

as its integral. The definite integral between the limits 1 and 2 is 4.83333333. The two programs of this section use numerical methods to estimate this definite integral.

TRAPEZOID METHOD

A trapezoid has two sides of unequal length. The area of the trapezoid is the width times the average length. Dividing the region of interest into n subintervals results in n trapezoids. There are a total of $n+1$ boundaries. The function $f(x)$ gives the lengths of the sides of the trapezoids. Each trapezoid has the same width.

If $f1$ is the length of the left-hand side, $f2$ is the length of the right-hand side, and w is the subinterval width, the formula $w(f1+f2)/2$ gives the area of the trapezoid. The right-hand boundary of one trapezoid is also the left-hand boundary of the following trapezoid. Except for the left-hand boundary of the leftmost trapezoid and the right-hand boundary of the rightmost trapezoid, the boundaries appear twice.

It is better to multiply by the width after forming the sums of the sides of the trapezoids. The following program uses this more efficient computational procedure:

```
PROGRAM area1 (INPUT, OUTPUT);
{ Trapezoid rule for numerical integration }
VAR
    lower,      { Left boundary of x for area }
```

```
    upper,      { Right boundary of x for area }
    width,      { Subinterval width }
    arg,        { Argument for f(x) }
    sum,        { Sum of the ordinates f(x) }
    area        { Estimated area of f(x) }
        : REAL;
    number,     { Number of subintervals }
    counter     { Loop control variable }
        : INTEGER;
FUNCTION fnc (x : REAL) : REAL;
BEGIN
    fnc := x * x - 3.0 * x + 7.0
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program AREA1');
    WRITELN;
    WRITELN ('Estimate area under f(x)');
    WRITELN ('using tapezoidal method.');
    { Problem parameters }
    WRITELN;
    WRITE ('Lower limit for x        ? ');
    READLN (lower);
    WRITE ('Upper limit for x        ? ');
    READLN (upper);
    WRITE ('Number of subintervals ? ');
    READLN (number);
    { Process }
    width := (upper - lower) / number;
    sum := 0.0;
    arg := lower;
    FOR counter := 1 TO number - 1 DO
        BEGIN
            arg := arg + width;
            sum := sum + fnc(arg)
        END;
    area := sum + 0.5 * (fnc(lower) + fnc(upper));
    area := area * width;
    WRITELN;
    WRITELN ('Estimated area = ', area:15:8);
    { Final message }
    WRITELN;
    WRITELN ('End of program')
END.
```

The test run follows:

```
Program AREA1

Estimate area under f(x)
using trapezoidal method.

Lower limit for x          ? 1.0
Upper limit for x          ? 2.0
Number of subintervals ? 100

Estimated area =        4.83335000

End of program
```

The integral of 4.83335 agrees closely with the known solution.

SIMPSON'S METHOD

Simpson's method provides a more refined method of numerically estimating the definite integral. The method involves fitting a parabolic curve to the three points defined by the midpoint and the two boundaries. Let w be the width between the midpoint and either of the two boundaries. The formula

$$(f(L)+4f(M)+f(U))w/3$$

gives the area of one of the trapezoids. L is the first side, U is the upper side, and M is the midpoint.

The following program uses Simpson's method for numerical integration:

```
PROGRAM area2 (INPUT, OUTPUT);
{ Simpson's method for numerical integration }
VAR
    lower,      { Left boundary of x for area }
    upper,      { Right boundary of x for area }
    width,      { Subinterval width }
    arg,        { Argument for f(x) }
    sum,        { Sum of f(x1) + 4f(xm) + f(xr) }
    area        { Estimated area of f(x) }
      : REAL;
    number,     { Number of subintervals }
    counter     { Loop control variable }
      : INTEGER;
FUNCTION fnc (x : REAL) : REAL;
BEGIN
    fnc := x * x - 3.0 * x + 7.0
END;
BEGIN
    { Initial message }
```

```
    WRITELN;
    WRITELN ('Program AREA2');
    WRITELN;
    WRITELN ('Estimate area under f(x)');
    WRITELN ('using Simpson''s method.');
    { Problem parameters }
    WRITELN;
    WRITE ('Lower limit for x        ? ');
    READLN (lower);
    WRITE ('Upper limit for x        ? ');
    READLN (upper);
    WRITE ('Number of subintervals ? ');
    READLN (number);
    { Process }
    width := 0.5 * (upper - lower) / number;
    sum := 0.0;
    arg := lower - width;
    FOR counter := 1 TO number DO
        BEGIN
            arg := arg + width + width;
            sum := sum + fnc(arg - width)
                       + 4.0 * fnc(arg)
                       + fnc(arg + width)
        END;
    area := sum * width / 3.0;
    WRITELN;
    WRITELN ('Estimated area = ', area:15:8);
    { Final message }
    WRITELN;
    WRITELN ('End of program')
END.
```

The test run follows:

```
Program AREA2

Estimate area under f(x)
using Simpson's method.

Lower limit for x        ? 1.0
Upper limit for x        ? 2.0
Number of subintervals ? 10

Estimated area =       4.83333000

End of program
```

EXERCISES

1. Write a program using the trapezoid method to integrate the following function in the interval $-2 \leqslant x \leqslant 2$:

   $x(2x + x^2)$

2. Write a program using Simpson's method to integrate the function of exercise 1.

12 Simulation

OVERVIEW Simulation involves modeling, and high-speed digital computers make large-scale simulation practical. Some problems do not have neat, easy, analytical solutions. Simulation is a tool for exploring the unknown.

Random number generators are indispensible tools for simulations of random phenomena. Simple, yet entertaining, random walk problems provide examples using the tools of simulation. More esoteric applications include Monte Carlo simulation to estimate definite integrals and Monte Carlo optimization.

□

12.1 RANDOM NUMBERS

Random events usually involve factors beyond the control of the decision-maker. The random component represents the degree of uncertainty. External events happen which are not under the direct control of the model. The computer generates the random values as needed, but this scheme is transparent to the model under investigation.

If the model is realistic, it will react to the random events as if it were the real system. Computer output will include summary measures that aid in analyzing the performance of the model.

UNIFORM RANDOM NUMBERS

Uniform random numbers form the backbone of probabilistic modeling. Each value is equally possible. Each toss of a coin has two possible outcomes, each with a probability of .5. A toss of a six-sided die has six possible outcomes, each having a probability of .1666667. Not only must the outcome be equally likely, there must be no discernible pattern in those outcomes. For the numbers to be random, not just uniform, their outcomes must be independent.

Uniform random numbers fall in the interval 0-1 because they represent probabilistic outcomes. A bent coin has a probability of .2 of a head and the probability of .8 of a tail. Divide the range 0-1 into two segments, 0-.2 and .2-1. Let a number in the interval 0-.2 represent a head and a number in the interval

.2-1 represent a tail. Twenty percent of the entire range 0-1 falls in the interval 0-.2. Eighty percent falls in the interval .2-1.

UNIFORM RANDOM NUMBER GENERATOR

A random number generator should generate values that have no discernible pattern. True random number generators are not very practical. Common practice in computer simulation is to use pseudo-random number generators. Pseudo-random numbers result from a programmed process. The computer can even generate the same sequence of numbers on demand. By definition, pseudo-random numbers do have a pattern. All that is required is that the pattern appear random to the model.

Some Pascal compilers, such as Alcor Pascal, include a random number generator as a built-in function. The simple pseudo-random number generator of this chapter is adequate for beginners. It illustrates the most popular method for generating pseudo-random numbers, the congruential method.

The congruential method generates the next random number as a function of the last number, using multiplication as the primary operation. The purpose of the other operations is to adjust the result of the primary operation to the desired range. If 17.1139 is the multiplier, the next random number is the fractional part of the product of the last number and the multiplier.

There must be a starting value for generating the first random number. This starting value is the "seed"; the user has some control over the sequence by specifying the starting seed. By specifying the same seed for several different runs, the user can rerun history, so to speak. By adjusting the model and rerunning it with the same sequence of pseudo-random numbers, the analyst can better compare model behavior.

What happens if a random number exactly equals 0.0? The product of 0.0 and the multiplier 17.1139 is 0.0. All following pseudo-random numbers will also be 0.0. This possibility is rare. Adding a constant term, such as .513693, reduces the risk of this happening.

The following program uses the simple congruential random number generator to generate a set of uniform random numbers:

```
PROGRAM random1 (INPUT, OUTPUT);
{ Generate a set of random numbers }
VAR
    number,     { Number needed }
    counter     { Loop control variable }
       : INTEGER;
    value
       : REAL;
FUNCTION rnd (value : REAL) : REAL;
{ Generate pseudo random number }
```

```
VAR
    temp        { Temporary value }
      : REAL;
BEGIN
    temp := 17.1139 * value;
    rnd := temp - TRUNC(temp)
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program RANDOM1');
    WRITELN;
    WRITELN ('Generate a set of');
    WRITELN ('pseudorandom numbers.');
    { Problem parameters }
    WRITELN;
    WRITE ('Number to generate ? ');
    READLN (number);
    WRITE ('Starting seed          ? ');
    READLN (value);
    { Generate }
    WRITELN;
    FOR counter := 1 TO number DO
        BEGIN
            value := rnd(value);
            WRITELN (value:9:6);
        END;
    WRITELN;
    WRITELN ('End of program')
END.
```

A sample run of this program follows:

```
Program RANDOM1

Generate a set of
pseudorandom numbers.

Number to generate ? 5
Starting seed          ? 0.214583

 0.672352
 0.506571
 0.669403
 0.456098
 0.805609

End of program
```

The starting seed should be a fraction in the interval 0-1. Most computers have an internal clock giving the time in hours, minutes, seconds, and milliseconds. Including a fraction to read the clock and using the seconds portion to generate the starting seed bypasses the need for the user's starting seed. However, reading the clock to obtain the starting seed eliminates the opportunity of rerunning history with the same sequence of numbers.

NORMAL RANDOM NUMBERS

The normal distribution is the most common probability distribution. It is the well-known bell-shaped curve. Most of the values fall in the center. Values are less frequent farther away from the midpoint. The distribution is symmetric about its midpoint.

Two parameters specify the normal distribution. The mean gives its central tendency, and most values cluster around it. The standard deviation gives the variability of the data around the mean. About 68 percent of the values fall within one standard deviation of the mean. About 95 percent fall within two standard deviations. Almost all (99.7 percent) fall within three standard deviations of the mean.

The central limit theorem states that the sum of a set of random values converges to the normal distribution as the number increases. This is true regardless of the population distributions for individual values.

One method of generating normal random numbers is to depend on the central limit theorem for normality. The sum of a set of 12 uniform random numbers is about normally distributed with an expected value of 6.0 and a standard deviation of 1.0. Subtracting the value 6.0 from the sum gives a value from the population which is normally distributed with a mean of 0.0 and a standard deviation of 1.0.

The standard normal distribution of probability theory has a mean of 0.0 and a standard deviation of 1.0. Published normal distribution tables assume this standard normal distribution. Multiplying the standard normal score by the desired population standard deviation and adding the population mean gives the resulting normal value in the scale of interest.

The following program generates a set of normal random numbers.

```
PROGRAM random2 (INPUT, OUTPUT);
{ Generate normal random numbers }
VAR
    number,     { Number needed }
    counter     { Loop control variable }
      : INTEGER;
    mean,       { Mean for normal distribution }
    std,        { Standard deviation }
    seed,       { Seed for random number generator }
```

```
    zscore,        { Standard score for normal random number }
    value          { Normal random number }
      : REAL;
FUNCTION rnd (VAR seed : REAL) : REAL;
{ Generate uniform random number }
BEGIN
    seed := 17.1139 * seed + 0.513693;
    seed := seed - TRUNC(seed);
    rnd := seed
END;
FUNCTION normal (VAR seed : REAL) : REAL;
{ Generate standard normal variate }
VAR
    index          { Loop control variable }
      : INTEGER;
    zscore         { Value of standard score }
      : REAL;
BEGIN
    zscore := -6.0;
    FOR index := 1 TO 12 DO
        zscore := zscore + rnd(seed);
    normal := zscore
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program RANDOM2');
    WRITELN;
    WRITELN ('Generate a set of');
    WRITELN ('normally distributed');
    WRITELN ('pseudo-random numbers.');
    { Problem parameters }
    WRITELN;
    WRITELN ('Parameters of normal distribution');
    WRITE ('Mean    ? ');
    READLN (mean);
    WRITE ('Std dev ? ');
    READLN (std);
    WRITELN;
    WRITE ('Number of values to generate ? ');
    READLN (number);
    WRITE ('Starting seed                ? ');
    READLN (seed);
    { Generate }
    WRITELN;
    FOR counter := 1 TO number DO
```

```
        BEGIN
            zscore := normal(seed);
            value := mean + zscore * std;
            WRITELN (value:12:4)
        END;
    WRITELN;
    WRITELN ('End of program')
END.
```

A sample test run with this program follows:

```
Program RANDOM2

Generate a set of
normally distributed
pseudo-random numbers.

Parameters of normal distribution
Mean    ? 100.0
Std dev ? 20.0

Number of values to generate ? 5
Starting seed                ? 0.673245

    80.0972
    76.5785
   105.7040
    88.0190
    98.7245

End of program
```

NEGATIVE EXPONENTIAL DISTRIBUTION

The negative exponential distribution is another distribution of special interest to simulation. Random phenomena often exhibit properties of this distribution. Switchboard telephone calls, customer arrivals, traffic accidents, and machine breakdowns are common examples.

If the occurrences are randomly distributed throughout the time interval, the time between occurrences is negatively exponentially distributed. The mean, or expected value, is the only parameter of the distribution.

Generating random values that fit the negative exponential distribution is simple. Generate a sequence of uniform random numbers falling in the interval 0-1. The natural logarithms of these uniform random numbers will be negative since the logarithm of a fractional value falling in the interval 0-1 is negative. Multiplying the natural logarithms of the random numbers by the desired mean and converting them to positive values gives values which fit the negative exponential form.

The following program generates a set of values that fit the negative exponential distribution:

```
PROGRAM random3 (INPUT, OUTPUT);
{ Generate negative exponential values }
VAR
    number,     { Number needed }
    counter     { Loop control variable }
        : INTEGER;
    mean,       { Mean for distribution }
    seed,       { Seed for random number generator }
    value       { Negative exponential value }
        : REAL;
FUNCTION rnd (VAR seed : REAL) : REAL;
{ Generate pseudo random number }
BEGIN
    seed := 17.1139 * value + 0.513693;
    seed := seed - TRUNC(seed);
    rnd := seed
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program RANDOM3');
    WRITELN;
    WRITELN ('Generate a set of');
    WRITELN ('negative exponentially');
    WRITELN ('distributed values.');
    { Problem parameters }
    WRITELN;
    WRITE ('Mean for distribution ? ');
    READLN (mean);
    WRITE ('Number to generate ? ');
    READLN (number);
    WRITE ('Starting seed        ? ');
    READLN (value);
    { Generate }
    WRITELN;
    FOR counter := 1 TO number DO
        BEGIN
            value := -mean * LN(rnd(seed));
            WRITELN (value:12:6)
        END;
    WRITELN;
    WRITELN ('End of program')
END.
```

A sample test run follows:

```
Program RANDOM3

Generate a set of
negative exponentially
distributed values.

Mean for distribution ? 5.0
Number to generate ? 5
Starting seed        ? 0.576327

    4.878920
   22.541600
    6.202920
    2.003200
    1.139330

End of program
```

Assume that the average time between incoming telephone calls at a mailorder department is 5 minutes. The set of random numbers represents the times between five successive calls.

DISCRETE PROBABILITY DISTRIBUTION

A discrete probability distribution is of the form:

Value	*Probability*
10	.3
20	.5
30	.2

The value 10 occurs 30 percent of the time, 20 occurs 50 percent of the time, and 30 occurs 20 percent of the time.

Form the cumulative probabilities as in the following table:

Value	*Probability*	*Cumulative*
10	.3	.3
20	.5	.8
30	.2	1.0

The cumulative probabilities divide the region 0-1 of the uniform random numbers appropriately. A random number in the region 0-.3 represents the value 10. A number in the region .3-.8 represents the value 20. A random number in the region .8-1 represents the value 30.

The following program generates random values from a discrete probability distribution:

```
PROGRAM random4 (INPUT, OUTPUT);
```

```
{ Generate random values from discrete distribution }
TYPE
    vector = ARRAY [1 .. 20] OF REAL;
VAR
    values,         { Prob dist values }
    probs           { Probabilities }
        : vector;
    seed,           { Seed for random number generator }
    random,         { Pseudo random number }
    outcome         { Outcome from random selection }
        : REAL;
    number,         { Number of values needed }
    counter,        { Loop control variable }
    nstates         { Number of states }
        : INTEGER;
FUNCTION rnd (VAR seed : REAL) : REAL;
{ Generate pseudo random number }
BEGIN
    seed := 17.1139 * seed + 0.513693;
    seed := seed - TRUNC(seed);
    rnd := seed
END;
FUNCTION sample
    (VAR seed : REAL;
         random : REAL;
         nstates : INTEGER;
         values, probs : vector) : REAL;
VAR
    index           { Index for distribution }
        : INTEGER;
    sum             { Cumulative probability }
        : REAL;
BEGIN
    sum := 0.0;
    index := 0;
    WHILE index < nstates AND random > sum DO
        BEGIN
            index := index + 1;
            sum := probs[index] + sum
        END;
    sample := values[index]
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program RANDOM4');
```

```
    WRITELN;
    WRITELN ('Generate a set of random');
    WRITELN ('values from a discrete');
    WRITELN ('probability distribution.');
    { Problem parameters }
    WRITELN;
    WRITELN ('Probability distribution');
    WRITELN;
    WRITE ('Number of states ? ');
    READLN (nstates);
    WRITELN;
    WRITELN ('Value, probability for');
    FOR counter := 1 TO nstates DO
        BEGIN
            WRITE ('state ', counter:2, ' ? ');
            READLN (values[counter], probs[counter])
        END;
    WRITELN;
    WRITE ('Number of values to generate ? ');
    READLN (number);
    WRITE ('Starting seed                ? ');
    READLN (seed);
    { Generate }
    WRITELN;
    FOR counter := 1 TO number DO
        BEGIN
            random := rnd(seed);
            outcome := sample(seed, random,
                nstates, values, probs);
            WRITELN (outcome:12:2)
        END;
    WRITELN;
    WRITELN ('End of program')
END.
```

A sample test run follows:

```
Program RANDOM4

Generate a set of random
values from a discrete
probability distribution.

Number of states ? 3

Value, probability for
state 1 ? 10.0, 0.3
```

```
state 2 ? 20.0, 0.5
state 3 ? 30.0, 0.2
Number of values to generate ? 5
Starting seed                ? 0.678523
      10.0
      30.0
      30.0
      30.0
      20.0
End of program
```

EXERCISES

1. Use the program RANDOM1 to generate a set of 25 uniform random numbers.
2. Use the program RANDOM2 to generate a set of 10 normal random numbers.
3. Use the program RANDOM3 to generate a set of 15 negative exponentially distributed values.
4. Use the program RANDOM4 to generate a set of 10 values from the following distribution:

Value	*Probability*
0	.1
1	.2
2	.3
3	.2
4	.1
5	.1

12.2 RANDOM WALK

A newcomer stands on a street corner not knowing which way to turn. Too proud to ask directions, the intrepid adventurer sets out to explore the town. At every intersection, the choice of direction depends on the flip of a coin. After walking 10 blocks, how many blocks from the starting point will the newcomer expect to be?

TOSSING A COIN

The first step is to simulate tossing the coin. The outcome of a single toss is either a head or a tail. If the coin is unbiased, the probability of a head is .5 and the probability of a tail is also .5. A uniform random number in the region 0-.5 represents a head and a number in the region .5-1 represents a tail.

The following program represents these outcomes by displaying appropriate messages:

```
PROGRAM walk1 (INPUT, OUTPUT);
{ Simulate tossing a coin }
VAR
    number,     { Number of tosses }
    counter     { Loop control variable }
        : INTEGER;
    random      { Random number }
        : REAL;
FUNCTION rnd (seed : REAL) : REAL;
{ Generate pseudo random number }
BEGIN
    seed := 17.1139 * seed + 0.513693;
    seed := seed - TRUNC(seed);
    rnd := seed
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program WALK1');
    WRITELN;
    WRITELN ('Simulate tossing a coin');
    WRITELN ('indicating the outcomes.');
    { Problem parameters }
    WRITELN;
    WRITE ('Number of tosses ? ');
    READLN (number);
    WRITE ('Starting seed      ? ');
    READLN (random);
    { Generate }
    WRITELN;
    FOR counter := 1 TO number DO
        BEGIN
            random := rnd(random);
            IF random < 0.5 THEN
                WRITELN ('Head')
            ELSE
                WRITELN ('Tail')
        END;
    WRITELN;
    WRITELN ('End of program')
END.
```

A sample run follows:

```
Program WALK1

Simulate tossing a coin
indicating the outcomes.

Number of tosses ? 5
Starting seed      ? 0.325683

Head
Head
Tail
Head
Tail

End of program
```

COUNTING THE NUMBER OF HEADS

Displaying every outcome for 1,000 simulated tosses of a coin isn't practical. It is better to use the computer to summarize the results. Of primary interest is the number of heads out of the 1,000 tosses.

The following program counts the number of heads and the percentage of heads in the set of simulated tosses:

```
PROGRAM walk2 (INPUT, OUTPUT);
{ Count the number of heads in simulated tosses }
VAR
    number,     { Number of tosses }
    sum,        { Number of heads }
    counter     { Loop control variable }
      : INTEGER;
    random,     { Random number }
    percent     { Percentage heads }
      : REAL;
FUNCTION rnd (seed : REAL) : REAL;
{ Generate pseudo random number }
BEGIN
    seed := 17.1139 * seed + 0.513693;
    seed := seed - TRUNC(seed);
    rnd := seed
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program WALK2');
```

```
    WRITELN;
    WRITELN ('Simulate tossing a coin');
    WRITELN ('counting the number of heads.');
    { Problem parameters }
    WRITELN;
    WRITE ('Number of tosses ? ');
    READLN (number);
    WRITE ('Starting seed      ? ');
    READLN (random);
    { Generate }
    sum := 0;
    FOR counter := 1 TO number DO
        BEGIN
            random := rnd(random);
            IF random < 0.5 THEN
                sum := sum + 1
        END;
    percent := 100.0 * sum / number;
    WRITELN;
    WRITELN ('Number of heads ', sum:5);
    WRITELN ('Percentage      ', percent:10:5);
    WRITELN;
    WRITELN ('End of program')
END.
```

A sample run follows:

```
Program WALK2

Simulate tossing a coin
counting the number of heads.

Number of tosses ? 200
Starting seed      ? 0.678931

Number of heads  106
Percentage       53.00000

End of program
```

LOCATION AFTER RANDOM WALK

Consider a one-dimensional random walk in which the person takes each step randomly in one of two directions. The starting point is the origin and has the value 0. A step to the right lands on location 1. A step to the left of the origin is -1. Five steps to the right is 5. Three steps to the left of the origin is -3. Simulating the choice of direction is the same as that of simulating the toss of a coin.

The following program gives the location and the distance from the origin after a specified number of steps:

```
PROGRAM walk3 (INPUT, OUTPUT);
{ Distance from origin for random walk }
VAR
    number,     { Number of steps }
    loc,        { Location }
    distance,   { Distance from origin }
    counter     { Loop control variable }
      : INTEGER;
    random      Random number
      : REAL;
FUNCTION rnd (seed : REAL) : REAL;
{ Generate pseudo random number }
BEGIN
    seed := 17.1139 * seed + 0.513693;
    seed := seed - TRUNC(seed);
    rnd := seed
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program WALK3');
    WRITELN;
    WRITELN ('Simulate random walk giving');
    WRITELN ('location after a specified');
    WRITELN ('number of steps.');
    { Problem parameters }
    WRITELN;
    WRITE ('Number of steps ? ');
    READLN (number);
    WRITE ('Starting seed    ? ');
    READLN (random);
    { Generate }
    loc := 0;
    FOR counter := 1 TO number DO
        BEGIN
            random := rnd(random);
            IF random < 0.5 THEN
                loc := loc - 1
            ELSE
                loc := loc + 1
        END;
    distance := ABS(loc);
```

```
    WRITELN;
    WRITELN ('Location   ', loc);
    WRITELN ('Distance   ', distance);
    WRITELN;
    WRITELN ('End of program')
END.
```

A sample test run follows:

```
Program WALK3

Simulate random walk giving
location after a specified
number of steps.

Number of steps ? 100
Starting seed    ? 0.578471

Location         10
Distance         10

End of program
```

AVERAGE DISTANCE FROM STARTING POINT

Probability suggests that a person will tend to move away from a starting point. Although the direction is indeterminant, the expected distance grows with the number of steps. The following program computes the average distance from the origin for several experiments involving a specified number of steps:

```
PROGRAM walk4 (INPUT, OUTPUT);
{ Average distance from origin for random walks }
VAR
    nwalks,      { Number of random walks }
    walk,        { Current walk }
    number,      { Number of steps per walk }
    loc,         { Location }
    distance,    { Distance from origin }
    counter      { Loop control variable }
      : INTEGER;
    sum,         { Sum of distances }
    average,     { Average distance }
    random       { Random number }
      : REAL;
FUNCTION rnd (seed : REAL) : REAL;
{ Generate pseudo random number }
BEGIN
    seed := 17.1139 * seed + 0.513693;
```

```
        seed := seed - TRUNC(seed);
        rnd := seed
    END;
    BEGIN
        { Initial message }
        WRITELN;
        WRITELN ('Program WALK4');
        WRITELN;
        WRITELN ('Simulate random walk giving');
        WRITELN ('average distance from origin');
        WRITELN ('for several walks.');
        { Problem parameters }
        WRITELN;
        WRITE ('Number of steps per walk ? ');
        READLN (number);
        WRITE ('Number of walks          ? ');
        READLN (nwalks);
        WRITE ('Starting seed            ? ');
        READLN (random);
        { Generate }
        sum := 0.0;
        FOR walk := 1 TO nwalks DO
            BEGIN
                loc := 0;
                FOR counter := 1 TO number DO
                    BEGIN
                        random := rnd(random);
                        IF random < 0.5 THEN
                            loc := loc - 1
                        ELSE
                            loc := loc + 1
                    END;
                distance := ABS(loc);
                sum := sum + distance
            END;
        average := sum / nwalks;
        WRITELN;
        WRITELN ('Average distance ', average:10:3);
        WRITELN;
        WRITELN ('End of program')
    END.
```

A sample test run follows:

```
Program WALK4
```

```
Simulate random walk giving
average distance from origin
for several walks.

Number of steps per walk ? 10
Number of walks          ? 5
Starting seed            ? 0.325673

Average distance      3.200

End of program
```

EXERCISES

1. Write a program to simulate tossing a coin 1,000 times, counting the number of heads.
2. Write a program simulating 600 tosses of a six-sided die, counting the frequency for each of the six outcomes.
3. Write a program computing the average distance from the origin for several random-walk experiments. Draw conclusions about the relationship between the number of steps and the expected distance from the origin as a function of the number of steps.

12.3 MONTE CARLO SIMULATION

Gambling is synonymous with Monte Carlo, the kingdom for which simulation is named. Rolling the dice, spinning the roulette wheel, dealing the cards—the results are randomly determined.

One of the first applications of probabilistic simulation was that of estimating the area under the curve for functions that did not have known analytical solutions. The computer randomly generates a large number of values of x within the specified interval. The product of the interval width and the average height of the curve estimates the area.

The following program uses Monte Carlo simulation to estimate the area under the curve

$$f(x) = x^2 - 3x + 7$$

in the interval between 1 and 2:

```
PROGRAM mcsim1 (INPUT, OUTPUT);
{ Monte Carlo integration }
VAR
    number,         { Number of trials }
    counter         { Loop control variable }
      : INTEGER;
    seed,           { Random number seed }
```

```
        random,         { Random number }
        lower,          { Lower limit for x }
        upper,          { Upper limit for x }
        value,          { Value of f(x) }
        arg,            { Current value for x }
        sum,            { Sum of values for f(x) }
        average,        { Average value of f(x) }
        area            { Estimated area }
            : REAL;
FUNCTION fnc(x: REAL) : REAL;
{ Function to evaluate }
BEGIN
    fnc := x * x - 3.0 * x + 7.0
END;
FUNCTION rnd (VAR seed : REAL) : REAL;
{ Generate pseudo random number }
BEGIN
    seed := 17.1139 * seed + 0.513693;
    seed := seed - TRUNC(seed);
    rnd := seed
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program MCSIM1');
    WRITELN;
    WRITELN ('Use Monte Carlo simulation');
    WRITELN ('to estimate the area of f(x)');
    WRITELN ('within specified limits.');
    { Problem parameters }
    WRITELN;
    WRITE ('Lower limit for x =    ? ');
    READLN (lower);
    WRITE ('Upper limit for x =    ?.');
    READLN (upper);
    WRITE ('Number of trials       ? ');
    READLN (number);
    WRITE ('Random number seed ? ');
    READLN (seed);
    { Process }
    sum := 0.0;
    FOR counter := 1 TO number DO
        BEGIN
            random := rnd(seed);
            arg := lower + random * (upper - lower);
            value := fnc(arg);
```

```
            sum := sum + value
        END;
    average := sum / number;
    area := average * (upper - lower);
    WRITELN;
    WRITELN ('Estimated area = ',area:12:5);
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program MCSIM1

Use Monte Carlo simulation
to estimate the area of f(x)
within specified limits.

Lower limit for x =     ? 1.0
Upper limit for x =     ? 2.0
Number of trials        ? 100
Random number seed ? 0.325671

Estimated area =          4.82976
```

EXERCISES

1. Use Monte Carlo simulation to estimate the definite integral for the function

   $f(x) = \sin(x)//ln(x)$

   for x in the interval 1 to 3. Assume that x is in radians for the sine function.

2. Use Monte Carlo simulation to estimate the definite integral for the function

   $f(x) = 3x^2\sin(x) + 2x\cos(x)$

   for x in the interval 0 to 1.

12.4 MONTE CARLO OPTIMIZATION

The second example area of Monte Carlo simulation is optimization of a nonlinear objective function subject to nonlinear constraints. Consider the following problem statement:

$$
\begin{array}{lll}
\text{Max} & X1^2 + 30\sqrt{X2} + 1000/X3 + .25X1\,X2 & \\
\text{S.T.} & 10X1 + 8X2 + X3 & \leqslant 500 \\
 & (X1 - 20)^2 + Ln(X2) + X3^2 & \leqslant 300 \\
 & 0 \leqslant X1 \leqslant 50 & \\
 & 0 \leqslant X2 \leqslant 100 & \\
 & 0 \leqslant X3 \leqslant 20. &
\end{array}
$$

The search procedure consists of randomly choosing sets of values for the decision variables $X1$, $X2$, and $X3$. Note which feasible set has the optimum value for the objective function.

Use the following steps in the analysis:

1. Use Monte Carlo simulation to choose values for the decision variable.
2. Test the values with the constraint equations for joint feasibility.
3. Evaluate the objective function for that set of values, if feasible.
4. Compare the objective function value with the best found so far. If the objective function value is improved, save the current value and decision variables as the best found thus far.

Several approaches are possible. One is to evaluate thousands of sets of decision values by letting the computer run hours at a time. Another is to use several trial runs to estimate the approximate value of each decision variable. Refine the interval by narrowing the gap between the lower and upper limits for the random search.

High-speed digital computers make Monte Carlo simulation practical. Current microcomputers take one or two seconds to evaluate each set of decision values for problems similar to the example. Large, multimillion-dollar computers can evaluate several thousand sets per second. Future microcomputers will include high-speed arithmetic capability and approach large computers in raw number-crunching power. Monte Carlo simulation is an important weapon in the arsenal of the analyst.

The following program uses Monte Carlo optimization to find good solutions to the nonlinear constrained optimization problem of this section:

```
PROGRAM mcopt1 (INPUT, OUTPUT);
{ Monte Carlo optimization }
TYPE
    vector = ARRAY [ 1 .. 20] OF REAL;
VAR
    x,          { Vector of arguments }
    bestx,      { Best values of x found so far }
    lower,      { Lower limit for x }
    upper,      { Upper limit for x }
      : vector;
    nvars,      { Number of variables }
    i,          { Index for variable }
    number,     { Number of iterations }
    counter     { Loop control variable }
      : INTEGER;
    seed,       { Random number seed }
    fvalue,     { Function value }
```

```
    bestval      { Best objective function value }
       : REAL;
    response     { User response }
       : CHAR;
FUNCTION rnd : REAL;
BEGIN
    seed := 17.1139 * seed + 0.513693;
    seed := seed - TRUNC(seed);
    rnd := seed
END;
FUNCTION evaluate
    (VAR x : vector) : REAL;
BEGIN
    evaluate := x[1] * x[1] + 30.0 * SQRT(x[2])
        + 1000.0 / x[3] + 0.25 * x[1] * x[2]
END;
FUNCTION feasible
    (VAR x : vector) : BOOLEAN;
VAR
    cvalue1,     { Constraint value }
    cvalue2,     { Constraint value }
       : REAL;
    i            { Index }
       : INTEGER;
BEGIN
    cvalue1 := 10.0 * x[1] + 8.0 * x[2] + x[3];
    cvalue2 := (x[1] - 20.0) * (x[1] - 20.0)
                + LN(x[2]) + x[3] * x[3];
    feasible := (cvalue1 <= 500.0)
                AND (cvalue2 <= 300.0)
END;
PROCEDURE sample
{ Generate x's randomly }
    (nvars : INTEGER;
    VAR x, lower, upper : vector);
VAR
    i            { Index }
       : INTEGER;
    random       { Random number }
       : REAL;
BEGIN
    FOR i := 1 TO nvars DO
        x[i] := lower[i]
            + rnd * (upper[i] - lower[i])
END;
```

```
PROCEDURE limits
    (nvars : INTEGER;
    VAR lower, upper : vector);
VAR
    i           { Index }
      : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Lower limit, Upper limit for');
    FOR i := 1 TO nvars DO
        BEGIN
            WRITE ('Variable ', i:2, ' ? ');
            READLN (lower[i], upper[i])
        END
END;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program MCOPT1');
    WRITELN;
    WRITELN ('Monte Carlo optimization');
    WRITELN ('of nonlinear objective');
    WRITELN ('function subject to');
    WRITELN ('nonlinear constraints.');
    { Get problem parameters }
    WRITELN;
    WRITE ('Random number seed ? ');
    READLN (seed);
    WRITE ('Number of variables   ? ');
    READLN (nvars);
    limits (nvars, lower, upper);
    WRITELN;
    WRITE ('Number of iterations to try ? ');
    READLN (number);
    bestval := -1.0E30;
    WHILE number > 0 DO
        BEGIN
            FOR counter := 1 TO number DO
                BEGIN
                    sample (nvars, x, lower, upper);
                    IF feasible(X) THEN
                        BEGIN
                            fvalue := evaluate(x);
                            IF fvalue > bestval THEN
                                BEGIN
```

```
                                    bestval := fvalue;
                                    FOR i := 1 TO nvars DO
                                        bestx[i] := x[i]
                                END
                        END
                END;
            WRITELN;
            WRITELN ('Best objective value ', bestval:15:2);
            WRITELN;
            WRITELN ('Variable   Quantity');
            FOR i := 1 TO nvars DO
                WRITELN (i:5, bestx[i]:12:2);
            WRITELN;
            WRITELN ('Number of iterations to try');
            WRITE ('Use 0 to terminate          ? ');
            READLN (number);
            IF number > 0 THEN
                BEGIN
                    WRITELN;
                    WRITE ('Try another set of limits (Y/N)? ');
                    READLN (RESPONSE);
                    IF response = 'Y' THEN
                        limits (nvars, lower, upper)
                END
        END;
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run of this program follows:

```
Program MCOPT1

Monte Carlo optimization
of nonlinear objective
function subject to
nonlinear constraints.

Random number seed ? 0.325671
Number of variables   ? 3

Lower limit, Upper limit for
Variable  1 ? 0.0 50.0
Variable  2 ? 0.0 100.0
Variable  3 ? 1.0 20.0

Number of iterations to try ? 20
```

```
Best objective value          817.49

Variable   Quantity
    1        24.98
    2         6.67
    3        13.45

Number of iterations to try
Use 0 to terminate          ? 50

Try another set of limits (Y/N) ? Y

Lower limit, Upper limit for
Variable  1 ? 10.0 30.0
Variable  2 ? 1.0 10.0
Variable  3 ? 1.0 15.0

Best objective value         1444.55

Variable   Value
    1        29.98
    2         6.56
    3         2.38

Number of iterations to try
Use 0 to terminate          ? 0

End of program
```

EXERCISES

1. Continue exploring the problem of this section for its optimum objective function value.
2. Modify the routine that randomly generates the decision variable values to generate only integer solutions. This becomes an integer programming problem. Search for the optimum integer solution.

13 Data analysis

OVERVIEW The computer is a powerful tool for data analysis. Summary measures describe important characteristics and frequency distributions give a picture of the shape of the data. Predicting values of one variable on the basis of its association with another variable is a primary goal of data analysis.

Many programs are interactive, with data entry from the keyboard at run time. Other programs process data contained in disk data files. The programs of this chapter illustrate data analysis.

□

13.1 SUMMARY MEASURES

Common summary measures include the mean and the standard deviation. The arithmetic mean is the common average. The mean of the set of values

12 11 19 14 15

is the sum

71 = 12 + 11 + 19 + 14 + 15

divided by the number of values. The expression

14.2 = 71/5

gives the mean.

The mean divides the total equally among the parts. Five commune members pool their resources of $12, $11, $19, $14, and $15, respectively. Their total resources amount to $71.00. Dividing the total $71.00 equally among the five nets $14.20 each.

The variance of a set of data is the average squared deviation of the values around the mean. The standard deviation is the square root of the variance. Statisticians define two variances and two standard deviations. The population vari-

ance for a finite population of size N uses the divisor N for computing the average squared deviation. For a sample of size n, the sample variance requires the divisor $n - 1$.

FORMULAS

For a finite population of size N, the formula

$$\mu = (\Sigma x)/N$$

gives the population mean μ, and the formula

$$\sigma^2 = \Sigma (x - \mu)^2/N$$

gives the population variance. The standard deviation σ is the square root of the variance.

For a sample of size n, the formula

$$m = (\Sigma x)/n$$

gives the sample mean m, and the formula

$$s^2 = \Sigma (x - m)^2/(n - 1)$$

gives the sample variance. Again, the sample standard deviation is the square root of the variance.

The two variance formulas are not in a convenient form. Th alternate formula

$$\sigma^2 = (\Sigma x^2)/N - \mu^2$$

gives the population variance. The similar alternate formula

$$s^2 = (\Sigma x^2 - (\Sigma x)^2/n)/(n - 1)$$

gives the sample variance.

SAMPLE MEAN AND STANDARD DEVIATION

Five batteries give the following hours of service:

12 11 19 14 15.

Compute the sample mean and standard deviation for the hours of service. The following program computes the sample mean and sample standard deviation:

```
PROGRAM sum1 (INPUT, OUTPUT);
{ Sample mean and standard deviation }
VAR
    number,        { Number of observations }
    index          { Index for current observation }
```

```
        : INTEGER;
    value,          { Current value }
    sum,            { Sum of the values }
    sumsqr,         { Sum of squares }
    mean,           { Arithmetic mean (average) }
    std             { Standard deviation }
        : REAL;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SUM1');
    WRITELN;
    WRITELN ('Compute the mean and');
    WRITELN ('standard deviation for');
    WRITELN ('a set of sample data.');
    { Process }
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    sum := 0.0;
    sumsqr := 0.0;
    WRITELN;
    WRITELN ('Value for');
    FOR index := 1 TO number DO
        BEGIN
            WRITE ('Obs ', index:3, ' ? ');
            READLN (value);
            sum := sum + value;
            sumsqr := sumsqr + value * value
        END;
    { Display summary measures }
    mean := sum / number;
    std := SQRT ((sumsqr - sum * sum / number)
                /(number - 1.0));
    WRITELN;
    WRITELN ('Mean    ', mean:15:5);
    WRITELN ('Std dev ', std:15:5);
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

The test run follows:

```
Program SUM1
```

```
Compute the mean and
standard deviation for
a set of sample data.

Number of observations ? 5

Value for
Obs  1 ? 12.0
Obs  2 ? 11.0
Obs  3 ? 19.0
Obs  4 ? 14.0
Obs  5 ? 15.0

Mean        14.20000
Std dev      3.11448

End of program
```

The sample mean is 14.2 and the sample standard deviation is 3.11.

INTERPRETATION OF STANDARD DEVIATION

The mean is the average. It divides the total equally among the observations. The mean is the center of gravity of the data. The variance is the average squared deviation around the center of gravity. The variance is the moment of inertia of the data. If most of the data is close to the center of gravity, it has low inertia. If most of the data is at the extreme limits, it has high inertia, like a flywheel with a heavily weighted rim.

The largest concentration of values is around the center of gravity for most problems. For such bell-shaped distributions, 68 percent of the values will fall within one standard deviation of the mean and 95 percent within two standard deviations.

If the battery service life mean and standard deviation are representative, then about 68 percent of the batteries should last within 3.11 hours of the mean of 14.2 hours. About 95 percent of the batteries should last 14.2 ± 6.22 hours.

SAMPLE AND POPULATION MEASURES

The purpose of statistical analysis is usually to generalize from a small sample to a larger population. It is to draw valid conclusions on the basis of sample evidence. A census consists of the entire population; population measures are appropriate for census data. The purpose of a census is not to generalize to a large population.

A statistician may interpret a set of data as a sample in one study and a population in another. At times the sample standard deviation is significant, while at other times the population standard deviation is needed. For large sets of data there is no practical numerical difference between the two measures.

The following program computes the minimum value, maximum value, mean, population variance, population standard deviation, sample variance, and sample standard deviation for a set of data:

```
PROGRAM sum2 (INPUT, OUTPUT);
{ Summary measures }
VAR
    number,      { Number of observations }
    index        { Index for current observations }
      : INTEGER;
    value,       { Current value }
    minimum,     { Minimum value }
    maximum,     { Maximum value }
    sum,         { Sum of the values }
    sumsqr,      { Sum of the squares }
    mean,        { Arithmetic mean }
    popvar,      { Population variance }
    popstd,      { Population standard deviation }
    samvar,      { Sample variance }
    samstd       { Sample standard deviation }
      : REAL;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SUM2');
    WRITELN;
    WRITELN ('Compute minimum, maximum, ');
    WRITELN ('mean, sample variance, sample');
    WRITELN ('std dev, population variance');
    WRITELN ('and population std dev.');
    { Process data }
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    sum := 0.0;
    sumsqr := 0;0;
    minimum := 1.0E30;
    maximum := -1.0E30;
    WRITELN;
    WRITELN ('Value for');
    FOR index := 1 TO number DO
        BEGIN
            WRITE ('Obs ', index:3, ' ? ');
            READLN (value);
```

```
            sum := sum + value;
            sumsqr := sumsqr + value * value;
            IF value > maximum THEN
               maximum := value;
            IF value < minimum THEN
               minimum := value
         END;
      { Compute }
      mean := sum / number;
      popvar := sumsqr / number - mean * mean;
      popstd := SQRT(popvar);
      samvar := (sumsqr - sum * sum / number)
                / (number - 1.0);
      samstd := SQRT (samvar);
      { Display results }
      WRITELN;
      WRITELN ('Mean        ', mean:15:5);
      WRITELN ('Minimum     ', minimum:15:5);
      WRITELN ('Maximum     ', maximum:15:5);
      WRITELN ('Sample');
      WRITELN ('variance    ', samvar:15:5);
      WRITELN ('Std dev     ', samstd:15:5);
      WRITELN ('Population');
      WRITELN ('variance    ', popvar:15:5);
      WRITELN ('Std dev     ', popstd:15:5);
      { Final message }
      WRITELN;
      WRITELN ('End of program');
END.
```

A sample test run follows.

```
Program SUM2

Compute the minimum, maximum,
mean, sample variance, sample
std dev, population variance,
and population std dev.

Number of observations ? 5

Value for
Obs  1 ? 12.0
Obs  2 ? 11.0
Obs  3 ? 19.0
Obs  4 ? 14.0
Obs  5 ? 15.0
```

```
Mean                14.20000
Minimum             11.00000
Maximum             19.00000
Sample
Variance             9.70000
Std dev              3.11448
Population
Variance             7.75999
Std dev              2.78568

End of program
```

DISK DATA FILE REDIRECTION FOR KEYBOARD ENTRIES

Interactive data entry is appropriate for small sets of data, but is not practical for large sets. Alcor Pascal allows redirection of keyboard data entries to a disk file. The same response that would have come from the keyboard now comes from the designated data file. The Pascal program editor or word processing program can create and edit the disk data file.

The program SUM2 requests the number of observations and then obtains those values one by one. The following disk data file provides this information:

```
5
12.0
11.0
19.0
14.0
15.0
```

The following test run includes the response to the INPUT prompt, giving the disk data file name DATA1/DAT:

```
INPUT     = DATA1/DAT
OUTPUT    =

Program SUM2

Compute the minimum, maximum,
mean, sample variance, sample
std dev, population variance,
and population std dev.

Number of observations ?
Value for
Obs  1 ? Obs  2 ? Obs  3 ? Obs  4 ? Obs  5 ?
Mean                14.20000
Minimum             11.00000
Maximum             19.00000
```

```
Sample
Variance          9.70000
Std dev           3.11448
Population
Variance          7.75999
Std dev           2.78568
End of program
```

PROGRAM DESIGNED TO USE DISK DATA FILE

Normal Pascal files include INPUT from keyboard and OUTPUT to video display. Interactive programs make primary use of these files. Other common devices are printers and disk drives. The printer provides a permanent record of the output in a form people can read. The term for this is hard copy. Disk data files are also permanent but are not in a form which people can read.

The following program SUM3 is similar to SUM2 except that it obtains its data from a disk data file:

```
PROGRAM sum3 (INPUT, OUTPUT, diskfile);
{ Summary measures }
VAR
    diskfile      { Disk data file }
      : TEXT;
    number        { Number of observations }
      : INTEGER;
    value,        { Current value }
    minimum,      { Minimum value }
    maximum,      { Maximum value }
    sum,          { Sum of the values }
    sumsqr,       { Sum of the squares }
    mean,         { Arithmetic mean }
    popvar,       { Population variance }
    popstd,       { Population standard deviation }
    samvar,       { Sample variance }
    samstd        { Sample standard deviation }
      : REAL;
BEGIN
    { Initial message }
    RESET (diskfile);
    WRITELN;
    WRITELN ('Program SUM3');
    WRITELN;
    WRITELN ('Compute minimum, maximum,');
    WRITELN ('mean, sample variance, sample');
```

```
    WRITELN ('std dev, and population');
    WRITELN ('variance and std dev');
    WRITELN ('using a disk data file.');
    { Process data }
    number := 0;
    sum := 0.0;
    sumsqr := 0.0;
    minimum := 1.0E30;
    maximum := -1.0E30;
    REPEAT
        BEGIN
            READLN (diskfile, value);
            number := number + 1;
            sum := sum + value;
            sumsqr := sumsqr + value * value;
            IF value > maximum THEN
               maximum := value;
            IF value < minimum THEN
               minimum := value
        END;
    UNTIL EOF (diskfile);
    CLOSE (diskfile);
    { Compute }
    mean := sum / number;
    popvar := sumsqr / number - mean * mean;
    popstd := SQRT(popvar);
    samvar := (sumsqr - sum * sum / number)
            / (number - 1.0);
    samstd := SQRT(samvar);
    { Display results }
    WRITELN;
    WRITELN ('Number    ', number:10);
    WRITELN ('Mean      ', mean:15:5);
    WRITELN ('Minimum   ', minimum:15:5);
    WRITELN ('Maximum   ', maximum:15:5);
    WRITELN ('Sample');
    WRITELN ('variance  ', samvar:15:5);
    WRITELN ('Std dev   ', samstd:15:5);
    WRITELN ('Population');
    WRITELN ('variance  ', popvar:15:5);
    WRITELN ('Std dev   ', popstd:15:5);
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A listing of the data file DATA2/DAT for this program follows:

```
12.0
11.0
19.0
14.0
15.0
```

The test run using this data file follows:

```
INPUT    =
OUTPUT   =
DISKFILE = DATA2/DAT
Program SUM3

Compute minimum, maximum,
mean, sample variance, sample
std dev, and population
variance and std dev
using a disk data file.

Number                5
Mean            14.20000
Minimum         11.00000
Maximum         19.00000
Sample
Variance         9.70000
Std dev          3.11448
Population
Variance         7.75999
Std dev          2.78568

End of program
```

The data file contains only data values and does not need to specify the number of observations as its initial value. The program reads the entire file and computes the number of observations in the process.

EXERCISES

1. Write a program to compute the sample mean and sample standard deviation. Read data from a disk data file.
2. Write an interactive program that computes the sample and population measures. Use definitional formulas. Read the values into an array and compute the sum of the squares of the deviations of the values around the mean.

13.2 TABULATING FREQUENCIES

A frequency distribution gives the distribution of the data values. Where do most of the values fall? How many fall to the far right and the far left?

The first step is to divide the range of values into classes. The second step is to tally the frequencies by counting the number of values that fall into each class. Class boundaries separate the classes, and if possible, they should uniquely separate the classes. Use boundaries similar to 9.5, 19.5, 29.5 for values recorded to the nearest whole number. Use boundaries such as 24.995 and 29.995 for data recorded to the nearest hundredth. Class widths should be equal.

FREQUENCY DISTRIBUTION

The following program forms a frequency distribution for a set of data:

```
PROGRAM freq1 (INPUT, OUTPUT);
{ Tabulate frequency distribution }
VAR
    table      { Table of frequencies }
      : ARRAY [1 .. 20] OF INTEGER;
    class,     { Class index }
    lower,     { Lowest class index }
    upper,     { Highest class index }
    sum,       { Cumulative frequency }
    number,    { Number of observations }
    index      { Index for current observation }
      : INTEGER;
    value,     { Value of current observation }
    lowest,    { Value of lowest boundary }
    width,     { Class width }
    bound      { Boundary of current class }
      : REAL;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program FREQ1);
    WRITELN;
    WRITELN ('Form frequency distribution');
    WRITELN ('for a set of data.');
    { Initialize }
    FOR class := 1 TO 20 DO
        table[class] := 0;
    WRITELN;
    WRITE ('Lowest class boundary  ? ');
    READLN (lowest);
```

```
    WRITE ('Class width                ? ');
    READLN (width);
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    { Process }
    WRITELN;
    WRITELN ('Value for');
    FOR index := 1 TO number DO
        BEGIN
            WRITE ('Obs ', index:3, ' ? ');
            READLN (value);
            class := 1 + TRUNC((value - lowest) / width);
            table[clas] := table[class] + 1
        END;
    { Determine limits }
    lower := 20;
    upper := 1;
    FOR class := 1 TO 20 DO
        BEGIN
            IF table[class] > 0 THEN
                BEGIN
                    IF class < lower THEN
                        lower := class;
                    IF class > upper THEN
                        upper := class
                END
        END;
    { Display frequency distribution }
    sum := 0;
    WRITELN;
    WRITELN ('Lowest        Frequency        Cumulative');
    FOR class := lower TO upper DO
        BEGIN
            bound := lowest + (class - 1.0) * width;
            sum := sum + table[class];
            WRITELN (bound:15:5, table[class]:8, sum:12)
        END;
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program FREQ1
```

```
Form frequency distribution
for a set of data.

Lowest class boundary  ? 9.5
Class width            ? 2.0

Number of observations ? 5

Value for
Obs  1 ? 12.0
Obs  2 ? 11.0
Obs  3 ? 19.0
Obs  4 ? 14.0
Obs  5 ? 15.0

Lowest          Frequency       Cumulative
     9.50000            1                1
    11.50000            1                2
    13.50000            2                4
    15.50000            0                4
    17.50000            1                5

End of program
```

HISTOGRAM

A histogram is a bar chart representing the frequencies of a frequency distribution. It gives a graphic picture of the distribution of the data values. The simplest charts consist of lines of asterisks or other characters. The Pascal

```
WRITE ('*')
```

command forms the line of asterisks one symbol at a time. The following program generates three lines per frequency class and makes the longest bar 40 column positions in length:

```
PROGRAM freq2 (INPUT, OUTPUT);
{ Frequency distribution and histogram }
VAR
    table       { Table of frequencies }
      : ARRAY [1 .. 20] OF INTEGER;
    class,      { Class index }
    lower,      { Lowest class index }
    upper,      { Highest class index }
    sum,        { Cumulative frequency }
    max,        { Maximum frequency }
    pos,        { Number of column positions for bar }
    col,        { Current column for bar }
    number,     { Number of observations }
    index       { Index for current observation }
```

```
        : INTEGER;
    value,      { Value of current observation }
    lowest,     { Value of lowest boundary }
    width,      { Class width }
    bound       { Boundary of current class }
        : REAL;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program FREQ2');
    WRITELN;
    WRITELN ('Form frequency distribution');
    WRITELN ('and histogram for a set of data.');
    { Initialize }
    FOR class := 1 TO 20 DO
        table[class] := 0;
    WRITELN;
    WRITE ('Lowest class boundary  ? ');
    READLN (lowest);
    WRITE ('Class width            ? ');
    READLN (width);
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    { Process }
    WRITELN;
    WRITELN ('Value for');
    FOR index := 1 TO number DO
        BEGIN
            WRITE ('Obs ', index:3, ' ? ');
            READLN (value);
            class := 1 + TRUNC((value - lowest) / width);
            IF class >= 1 AND class <= 20 THEN
                table[class] := table[class] + 1
        END;
    { Determine limits }
    lower := 20;
    upper := 1;
    max := 0;
    FOR class := 1 TO 20 DO
        BEGIN
            IF table[class] > 0 THEN
                BEGIN
                    IF table[class] > max THEN
                        max := table[class];
```

```
                    IF class < lower THEN
                        lower := class;
                    IF class > upper THEN
                        upper := class
                END
        END;
    { Display frequency distribution }
    sum := 0;
    WRITELN;
    WRITELN ('Lowest        Frequency        Cumulative');
    FOR class := lower TO upper DO
        BEGIN
            bound := lowest + (class - 1.0) * width;
            sum := sum + table[class];
            WRITELN (bound:15:5, table[class]:8, sum:12)
        END;
    { Histogram }
    WRITELN;
    FOR class := lower TO upper DO
        BEGIN
            pos := (40 * table[class]) DIV max;
            FOR index := 1 TO 3 DO
                BEGIN
                    FOR col := 1 TO pos DO
                        WRITE ('*');
                    WRITELN
                END
        END;
      Final message
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program FREQ2

Form frequency distribution
and histogram for a set of data.

Lowest class boundary   ? 9.5
Class width             ? 2.0

Number of observations ? 5

Value for
Obs  1 ? 12.0
Obs  2 ? 11.0
```

```
Obs  3 ? 19.0
Obs  4 ? 14.0
Obs  5 ? 15.0
Lowest          Frequency       Cumulative
      9.50000           1                1
     11.50000           1                2
     13.50000           2                4
     15.50000           0                4
     17.50000           1                5
**********************
**********************
**********************
**********************
**********************
**********************
****************************************
****************************************
****************************************
**********************
**********************
**********************
End of program
```

HARD COPY

Hard copy refers to printed records from the computer. The Radio Shack TRSDOS command DUAL sends output to both the screen and the printer. This important command permits hard copy output from interactive programs that make predominant use of the video screen. Alternately, the key combination Shift, down arrow, asterisk prints the contents of the video screen. These two methods provide hard copy when needed.

Programs that produce extensive output may send output to the printer; the user then does not need to switch back and forth between the DUAL (ON) and DUAL (OFF) modes. The following program obtains data from a disk data file and sends the results to the printer:

```
PROGRAM freq3 (INPUT, OUTPUT, diskfile, printer);
{ Frequency distribution and histogram using disk file }
VAR
    printer,    { Printer file }
    diskfile    { Disk data file }
      : TEXT;
    table       { Table of frequencies }
      : ARRAY [1 .. 20] OF INTEGER;
```

```
      class,      { Class index }
      lower,      { Lowest class index }
      upper,      { Highest class index }
      sum,        { Cumulative frequency }
      max,        { Maximum frequency }
      pos,        { Number of column positions for bar }
      col,        { Current column for bar }
      number,     { Number of observations }
      index       { Index for current observation }
         : INTEGER;
      value,      { Value of current observation }
      lowest,     { Value of lowest boundary }
      width,      { Class width }
      bound       { Boundary of current class }
         : REAL;
BEGIN
      { Initial message }
      REWRITE (printer);
      RESET (diskfile);
      WRITELN;
      WRITELN ('Program FREQ3');
      WRITELN;
      WRITELN ('Form frequency distribution');
      WRITELN ('and histogram for a set of data.');
      { Initialize }
      FOR class := 1 TO 20 DO
           table[class] := 0;
      WRITELN;
      WRITE ('Lowest class boundary  ? ');
      READLN (lowest);
      WRITE ('Class width            ? ');
      READLN (width);
      { Process }
      number := 0;
      REPEAT
           READLN (diskfile, value);
           number := number + 1;
           class := 1 + TRUNC((value - lowest) / width);
           IF class >= 1 AND class <= 20 THEN
              table[class] := table[class] + 1
      UNTIL EOF (diskfile);
      CLOSE (diskfile);
      { Determine limits }
      lower := 20;
      upper := 1;
      max := 0;
```

```
    FOR class := 1 TO 20 DO
        BEGIN
            IF table[class] > 0 THEN
                BEGIN
                    IF table[class] > max THEN
                        max := table[class];
                    IF class < lower THEN
                        lower := class;
                    IF class > upper THEN
                        upper := class
                END
        END;
    { Display frequency distribution }
    sum := 0;
    WRITELN (printer);
    WRITELN (printer,
        'Number of observations = ', number);
    WRITELN (printer);
    WRITELN (printer,
        'Lowest        Frequency        Cumulative');
    FOR class := lower TO upper DO
        BEGIN
            bound := lowest + (class - 1.0) * width;
            sum := sum + table[class];
            WRITELN (printer,
                bound:15:5, table[class]:8, sum:12)
        END;
    { Histogram }
    WRITELN (printer);
    FOR class := lower TO upper DO
        BEGIN
            pos := (40 * table[class]) DIV max;
            FOR index := 1 TO 3 DO
                BEGIN
                    FOR col := 1 TO pos DO
                        WRITE (printer, '*');
                    WRITELN (printer)
                END
        END;
    CLOSE (printer);
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample terminal session follows:

```
INPUT    =
OUTPUT   =
DISKFILE = DATA2/DAT
Program FREQ3
Form frequency distribution
and histogram for a set of data.
Lowest class boundary  ? 9.5
Class width            ? 2.0
End of program
```

The printed output follows:

```
Number of observations =     5
Lowest          Frequency          Cumulative
      9.50000           1                   1
     11.50000           1                   2
     13.50000           2                   4
     15.50000           0                   4
     17.50000           1                   5
**********
**********
**********
**********
**********
**********
****************************************
****************************************
****************************************
**********
**********
**********
```

EXERCISES

1. Write a program that computes the sample mean, the standard deviation, and forms a frequency distribution.
2. Write a program that computes the minimum, maximum, mean, standard deviation, forms a frequency distribution, and displays a histogram.

13.3 SORTING

Sorting data is an important task assigned to the computer; it brings order out of chaos. This section discusses three simple sorting algorithms that are adequate for small sets of data. Large sets of data require more complex algorithms.

SORT BY INSERTION INTO AN ORDERED LIST

The first method is appropriate for programs using interactive data entry. Each data item is placed in its proper position relative to the other values. The new value to be inserted into the list is first placed at its tail. If the value to its left is larger, it is interchanged and compared to the new value to its immediate left. This interchanging continues until there is no value to its left or until the value to its left is smaller and does not require an interchange.

This is actually the bubble sort algorithm embedded in a program using interactive data entry. The set of data can grow quite large before the user senses that the program response is slowing down because of the insertion activity. The following program uses the sort by insertion method:

```
PROGRAM sort1 (INPUT, OUTPUT);
{ Sort by insertion }
VAR
    data            { Vector of data values }
      : ARRAY [1 .. 200] OF REAL;
    temp            { Temporary value for exchange }
      : REAL;
    number,         { Number of observations }
    index,          { Index of data vector }
    loc             { Location for interchange }
      : INTEGER;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SORT1');
    WRITELN;
    WRITELN ('Sort a set of data');
    WRITELN ('into ascending order');
    { Initialize }
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    WRITELN ('Value for');
    WRITE ('Obs  1 ? ');
    READLN (data[1]);
```

```
    { Get value and insert in proper place }
    FOR index := 2 TO number DO
        BEGIN
            WRITE ('Obs ', index:3, ' ? ');
            READLN (data[index]);
            loc := index - 1;
            WHILE loc > 0 AND data[loc] > data[loc + 1] DO
                BEGIN
                    temp := data[loc];
                    data[loc] := data[loc + 1];
                    data[loc + 1] := temp;
                    loc := loc - 1
                END
        END;
    { Display sorted list }
    WRITELN;
    WRITELN ('Values in ascending order');
    FOR index := 1 TO number DO
        WRITELN (data[index]:15:5);
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program SORT1

Sort a set of data
into ascending order

Number of observations ? 5

Value for
Obs  1 ? 12.0
Obs  2 ? 11.0
Obs  3 ? 19.0
Obs  4 ? 14.0
Obs  5 ? 15.0

Values in ascending order
        11.00000
        12.00000
        14.00000
        15.00000
        19.00000

End of program
```

SORT BY SELECTION AND EXCHANGE

Sort by selection and exchange is one of the easiest to understand. The first task is to place the smallest value in the first location. The program searches the entire list for the smallest value and exchanges places between the smallest value and the value in the first position. The next task is to place the next smallest value in the second position. The program searches the remaining $n-1$ items for the second smallest value and exchanges that with the value in the second position.

This process of searching the remaining list for the next smallest remaining value continues until the list is sorted. The following program uses the sort by selection and exchange:

```
PROGRAM sort2 (INPUT, OUTPUT);
VAR
    data          { Data values }
      : ARRAY [1 .. 200] OF REAL;
    temp          { Temporary value for exchange }
      : REAL;
    number,       { Number of observations }
    i,            { Index for data values }
    j,            { Index for data values }
    loc           { Location of smallest value }
      : INTEGER;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SORT2');
    WRITELN;
    WRITELN ('Sort a set of values using');
    WRITELN ('selection and exchange.');
    { Get data }
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    WRITELN;
    WRITELN ('Value for');
    FOR i := 1 TO number DO
        BEGIN
            WRITE ('Obs ', i:3, ' ? ');
            READLN (data[i])
        END;
    { Sort into ascending order }
    FOR i := 1 TO number - 1 DO
        BEGIN
            loc := i;
            FOR j := i + 1 TO number DO
```

```
                IF data[j] < data[loc] THEN
                    loc := j;
            temp := data[i];
            data[i] := data[loc];
            data[loc] := temp
        END;
    { Display sorted values }
    WRITELN;
    WRITELN ('Values in ascending order');
    FOR i := 1 TO number DO
        WRITELN (data[i]:15:5);
    { Final message }
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program SORT2

Sort a set of values using
selection and exchange.

Number of observations ? 5

Value for
Obs  1 ? 12.0
Obs  2 ? 11.0
Obs  3 ? 19.0
Obs  4 ? 14.0
Obs  5 ? 15.0

Values in ascending order
        11.00000
        12.00000
        14.00000
        15.00000
        19.00000

End of program
```

BUBBLE SORT

The bubble sort is another sort that is easy to understand. The bubble sort nibbles on the mass of numbers one at a time, moving the new value into its correct position relative to the others already placed. This is really the same algorithm used by the insertion program SORT1.

The following program illustrates the bubble sort:

```
PROGRAM sort3 (INPUT, OUTPUT);
```

```
VAR
    data          { Data values }
      : ARRAY [1 .. 200] OF REAL;
    temp          { Temporary value for exchange }
      : REAL;
    number,       { Number of observations }
    i,            { Index for data values }
    j,            { Index for data values }
      : INTEGER;
BEGIN
    { Initial message }
    WRITELN;
    WRITELN ('Program SORT3');
    WRITELN;
    WRITELN ('Sort a set of values');
    WRITELN ('using bubble sort.');
    { Get data }
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    WRITELN;
    WRITELN ('Value for');
    FOR i := 1 TO number DO
        BEGIN
            WRITE ('Obs ', i:3, ' ? ');
            READLN (data[i])
        END;
    { Sort into ascending order }
    FOR i := 2 TO number DO
        BEGIN
            j := i - 1;
            WHILE j > 0 AND data[j] > data[j + 1] DO
                BEGIN
                    temp := data[j];
                    data[j] := data[j + 1];
                    data[j + 1] := temp;
                   j := j - 1
                END
        END;
    { Display sorted values }
    WRITELN;
    WRITELN ('Values in ascending order');
    FOR i := 1 TO number DO
        WRITELN (data[i]:15:5);
    { Final message }
```

```
    WRITELN;
    WRITELN ('End of program');
END.
```

A sample test run follows:

```
Program SORT3

Sort a set of values
using bubble sort.

Number of observations ? 5

Value for
Obs  1 ? 12.0
Obs  2 ? 11.0
Obs  3 ? 19.0
Obs  4 ? 14.0
Obs  5 ? 15.0

Values in ascending order
        11.00000
        12.00000
        14.00000
        15.00000
        19.00000

End of program
```

EXERCISES

1. Write a program that computes the mean, median, and standard deviation for a set of sample data. The median is the middle value or values of a set of data that has been sorted into order. It is the middle value for an odd number of values. It is the average of the two middle values for an even number of values.
2. Write a program that computes the mean and standard deviation for a set of data, sorts the data into order, forms a frequency distribution, and displays a histogram.

13.4 CURVE FITTING

A function of the form

$y = f(x)$

computes the value y as a function of the value x. The linear function

$y = f(x) = a + bx$

describes a straight line crossing the Y-axis at point a and having slope b.

The function

$$y = f(x) = \text{Log}\ (x)$$

gives y as the logarithm of x. Many other functional representations include quadratic, cubic, exponential, etc.

Regression analysis, also called curve fitting, predicts the dependent y as a function of the independent variable x. The functional relationship is not known beforehand. On the basis of sample evidence of pairs of values, the analyst tries to determine the appropriate relationship.

LINEAR REGRESSION

Linear regression attempts to fit a linear function to the data. The Y-intercept a and slope b are the two parameters to be estimated. The formula

$$b = (\Sigma xy - (\Sigma x)(\Sigma y)/n)/(\Sigma x^2 - (\Sigma x^2)/n)$$

gives the slope and

$$a = m_y - bm_x$$

gives the Y-intercept.

The three measures of goodness of fit are the standard error of estimate, the correlation coefficient, and the coefficient of determination. The standard error of estimate is the standard deviation among the residual errors. The errors have a mean of 0. If the errors are normally distributed, 68 percent of the observations fall within one standard error of the regression line and 96 percent fall within two standard errors.

The coefficient of determination gives the fraction of the variation of the dependent variable y explained by its association with the independent variable x. The variation of y is the sum of the squares of the deviations of y around its mean. The error variation is the sum of the squares of the deviations of y around the regression line. The explained variation is the difference between the total variation and the error variation.

The formula

$$SST = \Sigma (y - m_y)^2$$

gives the total variation. SST stands for Total Sum of Squares. If y' is the predicted value of y, the formula

$$SSE = \Sigma (y - y')^2$$

gives the error variation. SSE stands for Error Sum of Squares. The formula

$$SSR = \Sigma (y' - m_y)^2$$

gives the explained variation. SSR stands for Regression Sum of Squares. In least

squares regression the slope and Y-intercept are chosen to minimize the error variation. For least squares regression the relationship

$$SST = SSR + SSE$$

holds.

The ratio

$$SSR/SST$$

gives the coefficient of determination. The correlation coefficient is the square root of this coefficient of determination. The expression

$$1 - r^2$$

gives the fraction of the variation of Y not explained. Multiply by the variance of Y and take the square root to obtain the standard error of estimate.

The following program computes the regression parameters and measures of goodness of fit for a linear equation:

```
PROGRAM curve1 (INPUT, OUTPUT);
{ Fit linear function }
TYPE
    vector = ARRAY [1 .. 200] OF REAL;
VAR
    dep,            { Dependent variable }
    ind             { Independent variable }
      : vector;
    number          { Number of observations }
      : INTEGER;
    slope,          { Slope of function }
    intercept       { Y-intercept }
      : REAL;
PROCEDURE initial;
{ Initial message }
BEGIN
    WRITELN;
    WRITELN ('Program CURVE1');
    WRITELN;
    WRITELN ('Fit linear function estimating');
    WRITELN ('dependent variable as a function');
    WRITELN ('of independent variable.');
END;
PROCEDURE getdata
    (VAR number : INTEGER;
     VAR dep, ind : vector);
{ Get values for dependent and independent variables }
```

```
VAR
    index          { Index for data vectors }
       : INTEGER;
BEGIN
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    WRITELN;
    WRITELN ('Value for Dep var, Ind var for');
    FOR index := 1 TO number DO
        BEGIN
            WRITE ('Obs ', index:3, ' ? ');
            READLN (dep[index], ind[index])
        END
END;
PROCEDURE solve
    (number : INTEGER;
     VAR intercept, slope : REAL;
     VAR dep, ind : vector);
{ Solve for slope, Y-intercept, and fit }
VAR
    index          { index }
       : INTEGER;
    y,             { Current value of dep variable }
    x,             { Current value of ind variable }
    sumdep,        { Sum for dependent variable }
    sumind,        { Sum for independent variable }
    sumsqrdep,     { Sum of squares for dep variable }
    sumsqrind,     { Sum of squares for ind variable }
    sumcross,      { Sum of cross products }
    vardep,        { Variation for dependent variable }
    varind,        { Variation for independent variable }
    covar,         { Covariation between dep and ind variables }
    stderr,        { Standard error of estimate }
    correl,        { Correlation coefficient }
    determ         { Coefficient of determination }
        : REAL;
BEGIN
    { Clear accumulators }
    sumdep := 0.0;
    sumind := 0.0;
    sumsqrdep := 0.0;
    sumsqrind := 0.0;
    sumcross := 0.0;
    { Form sums and sums of squares }
    FOR index := 1 TO number DO
```

```
            BEGIN
                 y := dep[index];
                 x := ind[index];
                sumdep := sumdep + y;
                sumind := sumind + x;
                sumsqrdep := sumsqrdep + y * y;
                sumsqrind := sumsqrind + x * x;
                sumcross := sumcross + y * x
            END;
        vardep := sumsqrdep - sumdep * sumdep / number;
        varind := sumsqrind - sumind * sumind / number;
        covar := sumcross - sumdep * sumind / number;
        slope := cover / varind;
        intercept := sumdep / number
                    - slope * sumind / number;
        correl := covar / SQRT(vardep * varind);
        determ := correl * correl;
        stderr := SQRT((1.0 - determ) * vardep);
        WRITELN;
        WRITELN ('Y-intercept       ', intercept:15:5);
        WRITELN ('Slope             ', slope:15:5);
        WRITELN;
        WRITELN ('Std error         ', stderr:15:5);
        WRITELN ('Correlation       ', correl:15:5);
        WRITELN ('Determination     ', determ:15:5);
END;
PROCEDURE residuals
        (number : INTEGER;
         VAR intercept, slope : REAL;
         VAR dep, ind : vector);
{ Compute residuals }
VAR
        i               { Index for variables }
            : INTEGER;
        pred,           { Predicated value of dep variable }
        error           { Residual error }
            : REAL;
BEGIN
        WRITELN;
        WRITELN ('       Actual       Predicted       Residual');
        FOR i := 1 TO number DO
            BEGIN
                pred := intercept + slope * ind[i];
                error := dep[i] - pred;
                WRITELN (dep[i]:15:5, pred:15:5, error:15:5)
            END
```

```
END;
BEGIN
{ Body of main program }
    initial;
    getdata (number, dep, ind);
    solve (number, intercept, slope, dep, ind);
    residuals (number, intercept, slope, dep, ind)
END.
```

Use the program to estimate the dependent variable Y as a function of the independent variable *X*:

Dep. var.	*Ind. var.*
9	1
7	1
5	2
4	2
4	3
1	3

```
Program CURVE1

Fit linear function estimating
dependent variable as a function
of independent variable.

Number of observations ? 6

Value for Dev var, Ind var for
Obs  1 ? 9.0 1.0
Obs  2 ? 7.0 1.0
Obs  3 ? 5.0 2.0
Obs  4 ? 4.0 2.0
Obs  5 ? 4.0 3.0
Obs  5 ? 1.0 3.0

Y-intercept          10.50000
Slope                -2.75000

Std error             2.78388
Correlation          -0.89222
Determination         0.79605

      Actual          Predicted          Residual
     9.00000            7.75000           1.25000
     7.00000            7.75000          -0.75000
     5.00000            5.00000           0.00000
     4.00000            5.00000          -1.00000
     4.00000            2.25000           1.75000
     1.00000            2.25000          -1.25000
```

NONLINEAR CURVE FITTING

Some wag has said, "But the world is not linear." Linear regression is not always appropriate. One of the most productive methods of handling nonlinear relationships is to transform the independent variable into a form for which a linear fit is appropriate. Common transformations are square, square root, cube, logarithm, exponential, and reciprocal.

Standard linear regression analysis of the tranformed data gives the Y-intercept, slope, and measures of goodness of fit. The Y-intercept and slope apply only for the transformed independent variable. The measures of goodness of fit and the table of residuals aid in choosing an appropriate transformation. The best transformation is the one with the smallest standard error. This is also the one with the correlation coefficient magnitude closest to 1.0.

The following program performs nonlinear curve fitting by fitting a linear line to transformed data:

```
PROGRAM curve2 (INPUT, OUTPUT);
{ Fit nonlinear function }
TYPE
    vector = ARRAY [1 .. 200] OF REAL;
VAR
    dep,          { Dependent variable }
    ind,          { Independent variable }
    trans         { Transformed ind variable }
      : vector;
    number        { Number of observations }
      : INTEGER;
    slope,        { Slope of function }
    intercept     { Y-intercept }
      : REAL;
    response      { User response }
      : CHAR;
PROCEDURE initial;
{ Initial message }
BEGIN
    WRITELN;
    WRITELN ('Program CURVE2');
    WRITELN;
    WRITELN ('Fit nonlinear function estimating');
    WRITELN ('dependent variable as a function');
    WRITELN ('of independent variable.');
END;
PROCEDURE getdata
    (VAR number : INTEGER;
     VAR dep, ind : vector);
```

```
{ Get values for dependent and independent variables }
VAR
    index          { Index for data vectors }
      : INTEGER;
BEGIN
    WRITELN;
    WRITE ('Number of observations ? ');
    READLN (number);
    WRITELN;
    WRITELN ('Value for Dep var, Ind var for');
    FOR index := 1 TO number DO
        BEGIN
            WRITE ('Obs ', index:3, ' ? ');
            READLN (dep[index], ind[index])
        END
END;
PROCEDURE transform
    (number : INTEGER;
     VAR ind, trans : vector);
{ Transform independent variable }
VAR
    i,     { Index for variables }
    code          { Code number of transformation }
      :INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Code  Transformation for ind var');
    WRITELN ('  1   Linear');
    WRITELN ('  2   Square');
    WRITELN ('  3   Cube');
    WRITELN ('  4   Square root');
    WRITELN ('  5   Logarithm');
    WRITELN ('  6   Exponential');
    WRITELN ('  7   Reciprocal');
    WRITELN;
    WRITE ('Number of transformation ? ');
    READLN (code);
    FOR i := 1 TO number DO
        CASE code of
            1 : trans[i] := ind[i];
            2 : trans[i] := ind[i] * ind[i];
            3 : trans[i] := ind[i] * ind[i] * ind[i];
            4 : trans[i] := SQRT(ind[i]);
            5 : trans[i] := LN(ind[i]);
            6 : trans[i] := EXP(ind[i]);
```

```
            7 : trans[i] := 1.0 / ind[i]
          END;
END;
PROCEDURE solve
     (number : INTEGER;
      VAR intercept, slope : REAL;
      VAR dep, ind : vector);
{ Solve for slope, Y-intercept, and fit }
VAR
     index          { index }
        : INTEGER;
     y,             { Current value of dep variable }
     x,             { Current value of ind variable }
     sumdep,        { Sum for dependent variable }
     sumind,        { Sum for independent variable }
     sumsqrdep,     { Sum of squares for dep variable }
     sumsqrind,     { Sum of squares for ind variable }
     sumcross,      { Sum of cross products }
     vardep,        { Variation for dependent variable }
     varind,        { Variation for independent variable }
     covar,         { Covariation between dep and ind variables }
     stderr,        { Standard error of estimate }
     correl,        { Correlation coefficient }
     determ         { Coefficient of determination }
        : REAL;
BEGIN
     { Clear accumulators }
     sumdep := 0.0;
     sumind  := 0.0;
     sumsqrdep := 0.0;
     sumsqrind := 0.0;
     sumcross := 0.0;
     { Form sums and sums of squares }
     FOR index := 1 TO number DO
          BEGIN
               y := dep[index];
               x := ind[index];
               sumdep := sumdep + y;
               sumind := sumind + x;
               sumsqrdep := sumsqrdep + y * y;
               sumsqrind := sumsqrind + x * x;
               sumcross := sumcross + y * x
          END;
     vardep := sumsqrdep - sumdep * sumdep / number;
     varind := sumsqrind - sumind * sumind / number;
```

```
    covar := sumcross - sumdep * sumind / number;
    slope := covar / varind;
    intercept := sumdep / number
                 - slope * sumind / number;
    correl := covar / SQRT(vardep * varind);
    determ := correl * correl;
    stderr := SQRT((1.0 - determ) * vardep);
    WRITELN;
    WRITELN ('Y-intercept      ', intercept:15:5);
    WRITELN ('Slope            ', slope:15:5);
    WRITELN;
    WRITELN ('Std error        ', stderr:15:5);
    WRITELN ('Correlation      ', correl:15:5);
    WRITELN ('Determination    ', determ:15:5);
END;
PROCEDURE residuals
    (number : INTEGER;
     VAR intercept, slope : REAL;
     VAR dep, ind : vector);
{ Compute residuals }
VAR
    i               { Index for variables }
       : INTEGER;
    pred,           { Predicated value of dep variable }
    error           { Residual error }
       : REAL;
BEGIN
    WRITELN;
    WRITELN ('      Actual      Predicted      Residual');
    FOR i := 1 TO number DO
        BEGIN
            pred := intercept + slope * ind[i];
            error := dep[i] - pred;
            WRITELN (dep[i]:15:5, pred:15:5, error:15:5)
        END
END;
BEGIN
{ Body of main program }
    initial;
    getdata (number, dep, ind);
    REPEAT
        transform (number, ind, trans);
        solve (number, intercept, slope, dep, trans);
        WRITELN;
        WRITE ('Display residuals (Y/N) ? ');
```

```
        READLN (response);
        IF response = 'Y' THEN
           residuals (number, intercept, slope, dep, trans);
        WRITELN;
        WRITE ('Try another transformation (Y/N) ? ');
        READLN (response)
    UNTIL response = 'N'
END.
```

The following data has a strong nonlinear association:

Dep. var.	*Ind. var.*
2	1
3	2
8	3
15	4
30	5

Try several transformations and compare their measures to fit.

Program CURVE2

Fit nonlinear function estimating
dependent variable as a function
of independent variable.

Number of observations ? 5

Value for Dev var, Ind var for
Obs 1 ? 2.0 1.0
Obs 2 ? 3.0 2.0
Obs 3 ? 8.0 3.0
Obs 4 ? 15.0 4.0
Obs 5 ? 30.0 5.0

Code	Transformation
1	Linear
2	Square
3	Cube
4	Square root
5	Logarithm
6	Exponential
7	Reciprocal

Number of transformation ? 1

Y-intercept	-8.80000
Slope	6.80000
Std error	8.17313

```
Correlation          0.93476
Determination        0.87377
Display residuals (Y/N) ? Y
      Actual          Predicted          Residual
     2.00000          -2.00000           4.00000
     3.00000           4.80000          -1.80000
     8.00000          11.60000          -3.60000
    15.00000          18.40000          -3.40000
    30.00000          25.20000           4.80000
Try another transformation (Y/N) ? Y
Code  Transformation
  1   Linear
  2   Square
  3   Cube
  4   Square root
  5   Logarithm
  6   Exponential
  7   Reciprocal
Number of transformation ? 2
Y-intercept          -1.28235
Slope                 1.17112
Std error             4.03091
Correlation           0.98453
Determination         0.96930
Display residuals (Y/N) ? Y
      Actual          Predicted          Residual
     2.00000          -0.11123           2.11123
     3.00000           3.40214          -0.40214
     8.00000           9.24776          -1.25776
    15.00000          17.45560          -2.45562
    30.00000          27.99570           2.00428
Try another transformation (Y/N) ? Y
Code  Transformation
  1   Linear
  2   Square
  3   Cube
  4   Square root
  5   Logarithm
  6   Exponential
  7   Reciprocal
Number of transformation ? 6
```

```
Y-intercept          2.79817
Slope                0.18872

Std error            3.06374
Correlation          0.99109
Determination        0.98226

Display residuals (Y/N) ? Y

     Actual          Predicted          Residual
    2.00000            3.31115          -1.31115
    3.00000            4.19259          -1.19259
    8.00000            6.58861           1.41139
   15.00000           13.10170           1.89833
   30.00000           30.80600          -0.80598

Try another transformation (Y/N) ? N
```

EXERCISES

1. Convert the linear regression program to read data from a disk data file.
2. Convert the nonlinear regression program to read data pairs from a disk data file, but make the choice of transformation interactive.

14 Lists

OVERVIEW A list is a set of elements, usually records. A cell is the memory location containing one element. This list may be internal or external; external lists in Pascal are sequential files. Each cell is one record of the file.

Each element of the list has an identifier and one or more attributes. For most lists the identifier is unique, no two elements have the same identifier. For some applications several elements may have the same identifier value.

Common list operations include deleting elements that are no longer needed, inserting new elements, and modifying the contents of existing elements. Additional operations include building the initial list and listing the contents to the printer.

Inserting, deleting, and changing list elements is called maintenance. The programs of this chapter maintain external files using internal lists. The programs load the sequential external file into an internal list for ease of maintenance. The programs save the updated list as an external sequential file after completing the changes.

□

14.1 DENSE ORDERED LIST

A dense list contains no empty cells in its occupied region. A loose list contains empty cells scattered throughout its occupied region. Keeping the list dense while carrying out insertions and deletions requires effort.

An ordered list is one maintained in the order of its identifier (key). Inserting a new record into a dense ordered list requires making room for it by bumping the following records down one position. Deleting a record from a dense ordered list requires filling in the gap by moving the following records forward one position.

An external file contains a list of customer account balances. Each record contains an identifier, called a key, and an account balance. These records are in order by ID number and must be kept that way. The following program contains routines for building and maintaining the list of customer account balances:

```
PROGRAM dense1 (INPUT, OUTPUT, diskfile);
{ Internal dense ordered list }
TYPE
    list = RECORD
               key : INTEGER;
               balance : REAL;
           END;
    table = ARRAY [1 .. 100] OF list;
VAR
    internal            { Internal list of records }
       : table;
    number              { Number of records }
       : INTEGER;
    code                { Code for transaction }
       : INTEGER;
    response            { User response }
       : CHAR;
PROCEDURE initial;
BEGIN
    WRITELN;
    WRITELN ('Program DENSE1');
    WRITELN;
    WRITELN ('Maintain sequential file as');
    WRITELN ('dense ordered internal list.')
END;
PROCEDURE build
    (VAR number : INTEGER;
     VAR internal : table);
{ Build initial internal list }
VAR
    index               { Index for record position }
       : INTEGER;
BEGIN
    WRITELN;
    WRITE ('Number of records ? ');
    READLN (number);
    WRITELN;
    WRITELN ('Information for');
    FOR index := 1 TO number DO
        BEGIN
            WRITELN;
            WRITELN ('Record ', index);
            WRITE ('Id number ? ');
            READLN (internal[index].key);
            WRITE ('Balance ? ');
```

```
                READLN (internal[index].balance)
            END;
        WRITELN;
        WRITELN ('End of data entry')
END;
PROCEDURE print
        (number : INTEGER;
         VAR internal : table);
{ Print contents of records }
VAR
        printer             { File for printed output }
            : TEXT;
        index               { Index for record }
            : INTEGER;
BEGIN
        WRITELN;
        WRITELN ('Use device :L to send to printer');
        REWRITE (printer);
        WRITELN (printer, 'Id number  Balance');
        FOR index := 1 TO number DO
            WRITELN (printer, internal[index].key,
                        internal[index].balance:12:2);
        WRITELN (printer);
        WRITELN (printer, 'End of output');
        CLOSE (printer);
        WRITELN;
        WRITELN ('End of output')
END;
PROCEDURE display
        (number : INTEGER;
         VAR internal : table);
{ Display contents of one record }
VAR
        index,              { Index of record in list }
        idnumber            { ID number of desired record }
            : INTEGER;
        found               { Flag indicating whether record is found }
            : BOOLEAN;
BEGIN
        WRITELN;
        WRITE ('ID number of desired record ? ');
        READLN (idnumber);
        index := 0;
        found := FALSE;
        REPEAT
```

```
            index := index + 1;
            found := idnumber = internal[index].key;
        UNTIL found OR index >= number
                OR idnumber < internal[index].key;
        IF found THEN
            WRITELN ('Balance', internal[index].balance:12:2)
        ELSE
            WRITELN ('Not in file')
END;
PROCEDURE add
        (VAR number : INTEGER;
         VAR internal : table);
{ Insert record into internal list }
VAR
        index,              { Index to records }
        location,           { Location for insertion }
        idnumber            { ID number to insert }
            : INTEGER;
        stop,               { Flag to stop search }
        good                { Flag indicating insertion possible }
            : BOOLEAN;
BEGIN
        WRITELN;
        WRITELN ('Insert record');
        REPEAT
            WRITE ('ID number ? ');
            READLN (idnumber);
            index := 0;
            good := FALSE;
            stop := FALSE;
            REPEAT
                index := index + 1;
                IF index > number THEN
                    BEGIN
                        good := TRUE;
                        stop := TRUE
                    END
                ELSE IF idnumber < internal[index].key THEN
                    BEGIN
                        good := TRUE;
                        stop := TRUE
                    END
                ELSE IF idnumber = internal[index].key THEN
                    BEGIN
                        good := FALSE;
                        stop := TRUE;
```

```
                    WRITELN ('Try another ID number')
                END
            UNTIL stop
        UNTIL good;
        { Make insertion }
        location := index.
        IF location <= number THEN
            FOR index := number DOWNTO location DO
                internal[index + 1] := internal[index] ;
        internal[location].key := idnumber;
        WRITE ('Balance ? ');
        READLN (internal[location].balance);
        number := number + 1
END;
PROCEDURE remove
    (VAR number : INTEGER;
     VAR internal : table);
{ Delete record from list }
VAR
    index,              { Index for records }
    idnumber,           { ID number to delete }
    loc                 { Location of record to delete }
        : INTEGER;
    found               { Flag indicating whether record is found }
        : BOOLEAN;
    response            { User response }
        : CHAR;
BEGIN
    WRITELN;
    WRITE ('Id number to delete ? ');
    READLN (idnumber);
    index := 1;
    found := FALSE;
    WHILE index <= number AND found = FALSE DO
        BEGIN
            found := idnumber = internal[index].key;
            index := index + 1
        END;
    loc := index - 1;
    IF found THEN
        BEGIN
            WRITELN;
            WRITELN ('Balance = ', internal[loc].balance:12:2);
            WRITE ('Do you still wish to delete (Y/N) ? ');
            READLN (response);
            IF response = 'Y' THEN
```

```
                BEGIN
                    FOR index := loc TO number - 1 DO
                        internal[index] := internal[index + 1];
                    number := number - 1
                END
            ELSE
                WRITELN ('Record not deleted')
        END
    ELSE
        WRITELN ('Record not in file')
END;
PROCEDURE change
    (number : INTEGER;
     VAR internal : table);
VAR
    index,              { Index to record }
    idnumber,           { Id number to record to change }
    code                { Transaction code }
       : INTEGER;
    found               { Flag indicating whether found }
       : BOOLEAN;
    amount              { Amount of transaction }
       : REAL;
BEGIN
    WRITELN;
    WRITE ('ID number of record to change ? ');
    READLN (idnumber);
    found := false;
    index := 0;
    REPEAT
        index := index + 1;
        found := idnumber = internal[index].key
    UNTIL found OR index >= number
        OR idnumber < internal[index].key;
    IF found THEN
        BEGIN
            WRITELN ('Current balance = ',
                        internal[index].balance:12:2);
            WRITE ('Transaction code 1=payment, 2=purchase ? ');
            READLN (code);
            WRITE ('Amount of transaction                    ? ');
            READLN (amount);
            IF code = 2 THEN
               amount := amount + internal[index].balance
            ELSE
```

```
                amount := internal[index].balance - amount;
            internal[index].balance := amount;
            WRITELN ('new balance
                        amount:12:2)
        END
    ELSE
        WRITELN ('Record not in file');
END;
PROCEDURE load
    (VAR number : INTEGER;
     VAR internal : table);
{ Load internal file from external file }
VAR
    diskfile            { Diskfile for records }
      : TEXT;
    index               { Index for record }
      : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Name of external disk file');
    RESET (diskfile);
    index := 0;
    REPEAT
        index := index + 1;
        READLN (diskfile, internal[index].key,
                          internal[index].balance)
    UNTIL EOF (disfile);
    number := index;
    CLOSE (diskfile);
    WRITELN;
    WRITELN ('Number of records ', number)
END;
PROCEDURE save
    (number : INTEGER;
     VAR internal : table);
{ Save internal file to external disk file }
VAR
    diskfile            { Disk file for records }
      : TEXT;
    index               { Index for record }
      : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Name of disk file to contain internal file');
    REWRITE (diskfile);
```

```
    FOR index := 1 TO number DO
        WRITELN (diskfile, internal[index].key,
                           internal[index].balance);
    CLOSE (diskfile)
END;
{ Body of main program }
BEGIN
    initial;
    code := 0;
    REPEAT
        WRITELN;
        WRITELN ('Code  Operation');
        WRITELN ('    1  Build new file');
        WRITELN ('    2  Load existing file');
        WRITELN ('    3  Save to external file');
        WRITELN ('    4  Update existing record');
        WRITELN ('    5  Insert new record');
        WRITELN ('    6  Delete old record');
        WRITELN ('    7  Display current record');
        WRITELN ('    8  Print contents of records');
        WRITELN ('    9  Terminate processing');
        WRITELN;
        WRITE ('Code number ? ');
        READLN (code);
        CASE code OF
            1 : build(number, internal);
            2 : load(number, internal);
            3 : save (number, internal);
            4 : change (number, internal);
            5 : add(number, internal);
            6 : remove(number, internal);
            7 : display(number, internal);
            8 : print(number, internal);
            9 : BEGIN
                    WRITELN;
                    WRITELN ('Do not terminate unless');
                    WRITELN ('internal list is saved');
                    WRITELN ('to disk file.');
                    WRITELN;
                    WRITE ('Still want to terminate (Y/N) ? ');
                    READLN (response);
                    IF response = 'N' THEN
                       code := 0
                END;
            OTHERWISE WRITELN ('Invalid code number')
        END
```

```
    UNTIL code = 9;
    WRITELN;
    WRITELN ('End of program')
END.
```

The following test run builds an initial file:

```
Program DENSE 1

Maintain sequential file as
dense ordered internal list.

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records

Code number ? 1

Number of records ? 2

Information for

Record          1
Id number ? 101
Balance    ?  75.25

Record          2
Id number ? 107
Balance    ? 123.90

End of data entry

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records

Code number ? 3

Name of disk file to contain internal file
DISKFILE = DATA1/DAT
```

```
Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records

Code number ? 9

Do not terminate unless
internal list is saved
to disk file.

Still want to terminate (Y/N) ? Y

End of program
```

The following program loads the diskfile into the internal list and updates it:

```
Program DENSE1

Maintain sequential file as
dense ordered internal list.

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records

Code number ? 2

Name of external disk file
DISKFILE = DATA1/DAT

Number of records  2

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
```

```
Code number ? 8

Use device   :L to send to printer
PRINTER = :L

End of output

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records

Code number ? 4

Id number of record to change ? 107
Current balance =          123.90
Transaction code 1=payment, 2=purchase ? 1
Amount of transaction                  ? 74.00
new balance                              48.90

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records

Code number ? 5

Insert new record
ID number ? 105
Balance   ? 119.95

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records

Code number ? 8
```

```
Use device   :L to send to printer
PRINTER = :L

End of output

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records

Code number ? 3

Name of disk file to contain internal file
DISKFILE = DATA2/DAT

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records

Code number ? 9

Do not terminate unless
internal list is saved
to disk file.

Still want to terminate (Y/N) ? Y

End of program

First printed listing:

Id number  Balance
      101     75.25
      107    123.90

End of output

Second printed listing:

Id number  Balance
      101     75.25
      105    119.95
      107     48.90

End of output
```

EXERCISES

1. Write a program using the method of this section to maintain a telephone directory.
2. Write a program using the method of this section to maintain a name and address file.

14.2 LOOSE RANDOM LIST

Maintaining a dense ordered list requires much shuffling of elements to make room for insertions and to close up after deletions. A loose ordered list contains empty cells. An unused ID number such as -9999 flags each empty cell. Inserting a new record into a loose ordered list may require bumping cells down to make room, but deleting an existing cell requires only flagging the cell location as empty.

A random list makes no pretense of keeping the elements in order. Insertions are made by appending the new record at the end. Deletions are made by flagging the empty cell. A loose random list requires no shuffling of cells to make room for insertions or close up after deletions.

An external file contains customer ID numbers and account balances maintained in random order. The following program maintains the file using an internal loose random list, but does not copy the empty cells to the external file which remains a dense random list:

```
PROGRAM loose 1 (INPUT, OUTPUT, diskfile);
{ Interal loose random list }
TYPE
    list = RECORD
               key : INTEGER;
               balance : REAL;
           END;
    table = ARRAY [1..100] OF list;
VAR
    internal            { Internal list of records }
       : table;
    number              { Number of records }
       : INTEGER;
    code                { Code for transaction }
       : INTEGER;
    response            { User response }
       : CHAR;
PROCEDURE initial;
BEGIN
    WRITELN;
    WRITELN ('Program LOOSE1');
    WRITELN;
    WRITELN ('Maintain sequential file as');
```

```
    WRITELN ('loose random internal list.')
END;
PROCEDURE build
    (VAR number : INTEGER;
     VAR internal : table);
{ Build initial internal list }
VAR
    index               { Index for record position }
      : INTEGER;
BEGIN
    WRITELN;
    WRITE ('Number of records ? ');
    READLN (number);
    WRITELN;
    WRITELN ('Information for');
    FOR index := 1 TO number DO
        BEGIN
            WRITELN;
            WRITELN ('Record ', index);
            WRITE ('Id number ? ');
            READLN (internal[index].key);
            WRITE ('Balance    ? ');
            READLN (internal[index].balance)
        END
    WRITELN;
    WRITELN ('End of data entry')
END;
PROCEDURE print
    (number : INTEGER;
     VAR internal : table);
{ Print contents of records }
VAR
    printer             { File for printed output }
      : TEXT;
    index               { Index for record }
      : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Use device :L to send to printer');
    REWRITE (printer);
    WRITELN (printer, 'Id number   Balance');
    FOR index := 1 TO number DO
        IF internal[index].key <> - 9999 THEN
          WRITELN (printer, internal[index].key,
                   internal[index].balance:12:2);
```

```
        WRITELN (printer);
        WRITELN (printer, 'End of output');
        CLOSE (printer);
        WRITELN;
        WRITELN ('End of output')
    END;
    PROCEDURE display
        (number : INTEGER;
         VAR internal: table);
    { Display contents of one record }
    VAR
        index,              { Index of record in list }
        idnumber            { ID number of desired record }
            : INTEGER;
        found               { Flag indicating whether record is found }
            : BOOLEAN;
    BEGIN
        WRITELN;
        WRITE ('ID number of desired record ? ');
        READLN (idnumber);
        index := 0;
        found := FALSE;
        REPEAT
            index := index + 1;
            found := idnumber = internal[index].key;
        UNTIL found OR index >= number;
        IF found THEN
            WRITELN ('Balance', internal[index].balance:12:2)
        ELSE
            WRITELN ('Not in file')
    END;
    PROCEDURE add
        (VAR number : INTEGER;
         VAR internal : table);
    { Insert record into internal list }
    VAR
        idnumber            { ID number of record }
            : INTEGER;
    BEGIN
        WRITELN;
        WRITELN ('insert record');
        WRITE ('ID number ? ');
        READLN (idnumber);
        number := number + 1;
        internal[number].key := idnumber;
```

```
    WRITE ('Balance  ? ');
    READLN (internal[number].balance);
END;
PROCEDURE remove
    (VAR number : INTEGER;
     VAR internal : table);
{ Delete record from list }
VAR
    index,              { Index for records }
    idnumber            { ID number to delete }
       : INTEGER;
    found               { Flag indicating whether record is found }
       : BOOLEAN;
    response            { User response }
       : CHAR;
BEGIN
    WRITELN;
    WRITE ('Id number to delete ? ');
    READLN (idnumber);
    index := 0;
    found := FALSE;
    REPEAT
        index := index + 1;
        found := idnumber = internal[index].key;
    UNTIL found OR index >= number;
    IF found THEN
        BEGIN
            WRITELN;
            WRITELN ('Balance = ', internal[index].balance:12:2);
            WRITE ('Do you still wish to delete (Y/N) ? ' );
            READLN (response);
            IF response = 'Y' THEN
               internal[index].key := -9999
            ELSE
               WRITELN ('Record not deleted'0
        END
    ELSE
        WRITELN ('Record not in file')
END;
PROCEDURE change
    (number : INTEGER;
     VAR internal : table);
VAR
    index,              { Index to record }
    idnumber,           { Id number of record to change }
    code                { Transaction code }
```

```
        : INTEGER;
    found               { Flag indicating whether found }
        : BOOLEAN;
    amount              { Amount of transaction }
        : REAL;
BEGIN
    WRITELN;
    WRITE ('ID number of record to change ? ');
    READLN (idnumber);
    found := false;
    index := 0;
    REPEAT
        index := index + 1;
        found := idnumber = internal[index].key
    UNTIL found OR index >= number;
    IF found THEN
        BEGIN
            WRITELN ('Current balance = ',
                        internal[index].balance:12:2);
            WRITE ('Transaction code 1=payment, 2=purchase ? ');
            READLN (code);
            WRITE ('Amount of transaction                    ? ');
            READLN (amount);
            IF code = 2 THEN
               amount := amount + internal[index].balance
            ELSE
               amount := internal[index].balance - amount;
            internal[index].balance := amount;
            WRITELN ('new balance                          ',
                        amount:12:2)
        END
    ELSE
        WRITELN ('Record not in file');
END;
PROCEDURE load
    (VAR number : INTEGER;
     VAR internal : table);
{ Load internal file from external file }
VAR
    diskfile            { Disk file for records }
        : TEXT;
    index               { Index for record }
        : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Name of external disk file');
```

```
    RESET (diskfile);
    index := 0;
    REPEAT
        index := index + 1
        READLN (diskfile, internal[index].key,
                          internal[index].balance)
    UNTIL EOF (diskfile);
    number := index;
    CLOSE (diskfile);
    WRITELN;
    WRITELN ('Number of records ', number)
END;
PROCEDURE save
    (number : INTEGER;
     VAR internal : table);
{ Save internal file to external disk file }
VAR
    diskfile            { Disk file for records }
      : TEXT;
    index               { Index for record }
      : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Name of disk file to contain internal file');
    REWRITE (diskfile);
    FOR index := 1 TO number DO
        IF internal[index].key <> -9999 THEN
          WRITELN (diskfile, internal[index].key,
                             internal[index].balance);
    CLOSE (diskfile)
END;
{ Body of main program }
BEGIN
    initial;
    code := 0;
    REPEAT
        WRITELN;
        WRITELN ('Code  Operation');
        WRITELN ('    1  Build new file');
        WRITELN ('    2  Load existing file');
        WRITELN ('    3  Save to external file');
        WRITELN ('    4  Update existing record');
        WRITELN ('    5  Insert new record');
        WRITELN ('    6  Delete old record');
        WRITELN ('    7  Display current record');
        WRITELN ('    8  Print contents of records');
```

```
            WRITELN ('    9  Terminate processing');
            WRITELN;
            WRITE ('Code number ? ');
            READLN (code);
            CASE code OF
                1 : build(number, internal);
                2 : load(number, internal);
                3 : save(number, internal);
                4 : change(number, internal);
                5 : add(number, internal);
                6 : remove(number, internal);
                7 : display(number, internal);
                8 : print(number, internal);
                9 : BEGIN
                        WRITELN;
                        WRITELN ('Do not terminate unless');
                        WRITELN ('internal list is saved');
                        WRITELN ('to disk file.');
                        WRITELN;
                        WRITE ('Still want to terminate (Y/N) ? ');
                        READLN (response);
                        IF response = 'N' THEN
                           code := 0
                    END;
               OTHERWISE WRITELN ('Invalid code number')
            END
        UNTIL code = 9;
        WRITELN;
        WRITELN ('End of program')
END.
```

A test run to create the initial file follows:

```
Program LOOSE1

Maintain sequential file as
loose random internal list.

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
   9  Terminate processing
```

```
Code number ? 1

Number of records ? 3

Information for

Record          1
Id number ? 101
Balance     ? 123.95

Record          2
Id number ? 105
Balance     ? 57.75

Record          3
Id number ? 107
Balance     ? 175.50

End of data entry

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
   9  Terminate processing

Code number ? 3

Name of disk file to contain internal file
DISKFILE = DATA1/DAT

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
   9  Terminate processing

Code number ? 9

Do not terminate unless
internal list is saved
to disk file.
```

```
Still want to terminate (Y/N) ? Y

End of program
```

A second test run using this file follows:

```
Program LOOSE1

Maintain sequential file as
loose random internal list.

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
   9  Terminate processing

Code number ? 2

Name of external disk file
DISKFILE = DATA1/DAT

Number of records       3

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
   9  Terminate processing

Code number ? 8

Use device   :L to send to printer
PRINTER = :L

End of output

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
```

```
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 5

Insert record
ID number ? 102
Balance    ? 99.95

Code  Operation
    1  Build new file
    2  Load existing file
    3  Save to external file
    4  Update existing record
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 6

Id number to delete ? 105

Balance =          57.75
Do you still want to delete (Y/N) ? Y

Code  Operation
    1  Build new file
    2  Load existing file
    3  Save to external file
    4  Update existing record
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 8

Use device   :L to send to printer
PRINTER = :L

End of output

Code  Operation
    1  Build new file
    2  Load existing file
    3  Save to external file
```

```
    4  Update existing record
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 3

Name of disk file to contain internal file
DISKFILE = DATA2/DAT

Code  Operation
    1  Build new file
    2  Load existing file
    3  Save to external file
    4  Update existing record
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 9

Do not terminate unless
internal list is saved
to disk file.

Still want to terminate (Y/N) ? Y

End of program
```

EXERCISES

1. Write a program to maintain an inventory price list. Use this section's method.
2. Write a program to maintain a telephone directory. Use this section's method.

14.3 LINKED LIST

A linked list contains a pointer (index) for each element which gives the location of the next element in logical sequence. The logical order may differ from the physical order. It is not necessary to shuffle elements around to make room for insertions when using a linked list. The chain of links is broken and relinked through the new element.

A space list consists of a linked list of empty cells handled as a stack. A stack follows a last-in-first-out discipline. A record to be inserted into the linked list is placed in the top cell of the space list. It is then linked into its proper place

in the linked list. To delete a record from the linked list, link its predecessor to its sucessor and place the deleted cell as the new top of the space list.

The first, second, and third cells of the list area have special uses. They contain dummy records which make the linked list processing easier. The first cell is the header cell pointing to the first linked list element. The key of the header cell is -9999 which is always less than any valid record ID number. The second cell is the space list cell and points to the top of the stack of empty cells. The third cell is a dummy cell containing the tail of the linked list. It contains the key 9999 which is larger than any valid ID number.

Any valid ID number is greater than the initial dummy element key and less than the final dummy element key. This facilitates programming the routines for searching, inserting, and deleting elements. The following program maintains an external file in sequential order using an internal linked list:

```
PROGRAM linked1 (INPUT, OUTPUT, diskfile);
{ Internal linked list }
TYPE
    list = RECORD
               key : INTEGER;
               link : INTEGER;
               balance : REAL;
           END;
    table = ARRAY [1..100] OF list;
VAR
    internal          { Internal list of records }
       : table;
    header,           { Pointer to first record }
    space             { Pointer to space list }
       : INTEGER;
    code              { Code for transaction }
       : INTEGER;
    response          { User response }
       : CHAR;
PROCEDURE initial;
BEGIN
    WRITELN;
    WRITELN ('Program LINKED1);
    WRITELN;
    WRITELN ('Maintain sequential file');
    WRITELN ('as internal linked list.')
END;
PROCEDURE build
    (VAR internal : table);
{ Build initial internal list }
```

```
VAR
    number,          { Number of records }
    space,           { Top of space list }
    previous,        { Location of previous record }
    next,            { Location of next record }
    idnumber,        { Record ID number }
    loc,             { Location for new record }
    index            { Index for record position }
      : INTEGER;
BEGIN
    { Generate initial space list }
    internal[1].key := -9999;
    internal[1].link := 2;
    internal[2].key := 9999;
    internal[2].link := 3;
    space := 3;
    FOR index := 3 TO 99 DO
        internal[index].link := index + 1;
    internal[100].link := -9999;
    { Build linked list }
    WRITELN;
    WRITE ('Number of records ? ');
    READLN (number);
    WRITELN;
    WRITELN ('Information for');
    index := 0;
    REPEAT
        index := index + 1;
        loc := space;
        space := internal[space].link;
        WRITELN;
        WRITELN ('Record ', index);
        WRITE ('Id number ? ');
        READLN (idnumber);
        internal[loc].key := idnumber;
        WRITE ('Balance  ? ');
        READLN (internal[loc].balance);
        next := 1;
        REPEAT
            previous := next;
            next := internal[next].link
        UNTIL internal[next].key > idnumber;
        internal[loc].link := next;
        internal[previous].link := loc
    UNTIL index >= number OR space = -9999;
```

```
    internal[2].link := space;
    IF space = -9999 THEN
        BEGIN
            WRITELN;
            WRITELN ('Internal list is full')
        END;
    WRITELN;
    WRITELN ('End of data entry')
END;
PROCEDURE print
    (VAR internal : table);
{ Print contents of records }
VAR
    printer             { File for printed output }
      : TEXT;
    index               { Index for record }
      : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Use device :L to send to printer');
    REWRITE (printer);
    WRITELN (printer, 'Id number   Balance');
    index := internal[1].link;
    REPEAT
        WRITELN (printer, internal[index].key,
                  internal[index].balance:12:2);
        index := internal[index].link
    UNTIL internal[index].key = 9999;
    WRITELN (printer);
    WRITELN (printer, 'End of output');
    CLOSE (printer);
    WRITELN;
    WRITELN ('End of output')
END;
PROCEDURE display
    (VAR internal : table);
{ Display contents of one record }
VAR
    index,              { Index of record in list }
    idnumber            { ID number of desired record }
      : INTEGER;
    found               { Flag indicating whether record is found }
      : BOOLEAN;
BEGIN
    WRITELN;
```

```
    WRITE ('ID number of desired record ? ');
    READLN (idnumber);
    index := internal[1].link;
    found := FALSE;
    REPEAT
        index := internal[index].link;
        found := idnumber = internal[index].key
    UNTIL found OR internal[index].key > idnumber;
    IF found THEN
        WRITELN ('Balance', internal[index].balance:12:2)
    ELSE
        WRITELN ('Not in file')
END;
PROCEDURE add
    (VAR internal : table);
{ Insert record into internal list }
VAR
    index,              { Index to records }
    previous,           { Location for insertion }
    loc,                { Location for insertion }
    idnumber            { ID number to insert }
      : INTEGER;
    stop,               { Flag to stop search }
    good                { Flag indicating insertion possible }
      : BOOLEAN;
BEGIN
    WRITELN;
    WRITELN ('Insert record');
    WRITE ('ID number ? ');
    READLN (idnumber);
    loc := internal[2].link;
    space := internal[loc].link;
    internal[2].link := space;
    index := 1;
    REPEAT
        previous := index;
        index := internal[index].link
    UNTIL idnumber < internal[index].key;
    { Make insertion }
    internal[loc].key := idnumber;
    internal[loc].link := index;
    internal[previous].link := loc;
    WRITE ('Balance ? ');
    READLN (internal[loc].balance);
    IF space = -9999 THEN
```

```
        BEGIN
            WRITELN;
            WRITELN ('No more room for insertions')
        END
END;
PROCEDURE remove
    (VAR internal : table);
{ Delete record from list }
VAR
    previous,           { Previous record }
    index,              { Index for records }
    idnumber            { ID number to delete }
       : INTEGER;
    found               { Flag indicating whether record is found }
       : BOOLEAN;
    response            { User response }
       : CHAR;
BEGIN
    WRITELN;
    WRITE ('ID number to delete ? ');
    READLN (idnumber);
    index := 1;
    found := FALSE;
    REPEAT
        previous := index;
        index := internal[index].link;
        found := idnumber = internal[index].key
    UNTIL found OR internal[index].key > idnumber;
    IF found THEN
        BEGIN
            WRITELN;
            WRITELN ('Balance = ', internal[index].balance:12:2);
            WRITE ('Do you still wish to delete (Y/N) ? ');
            READLN (response);
            IF response = 'Y' THEN
                BEGIN
                    internal[previous].link := internal[index].link;
                    internal[index].link := internal[2].link;
                    internal[index].link := index
                END
            ELSE
                WRITELN ('Record not deleted'0
        END
    ELSE
        WRITELN ('Record not in file')
END;
```

```
PROCEDURE change
    (VAR internal : table);
VAR
    index,              { Index to record }
    idnumber,           { Id number of record to change }
    code                { Transaction code }
       : INTEGER;
    found               { Flag indicating whether found }
       : BOOLEAN;
    amount              { Amount of transaction }
       : REAL;
BEGIN
    WRITELN;
    WRITELN ('ID number of record to change ? ');
    READLN (idnumber);
    found := false;
    index := 1;
    REPEAT
        index := internal[index].link;
        found := idnumber = internal[index].key
    UNTIL found OR internal[index].key > idnumber;
    IF found THEN
        BEGIN
            WRITELN ('Current balance = ',
                        internal[index].balance:12:2);
            WRITE ('Transaction code 1=payment, 2=purchase ? ');
            READLN (code);
            WRITE ('Amount of transaction                    ? ');
            READLN (amount);
            IF code = 2 THEN
               amount := amount + internal[index].balance
            ELSE
               amount := internal[index].balance - amount;
            internal[index].balance := amount;
            WRITELN ('new balance
                        amount:12:2)
        END
    ELSE
        WRITELN ('Record not in file');
END;
PROCEDURE load
    (VAR internal : table);
{ Load internal file from external file }
VAR
    diskfile            { Disk file for records }
       : TEXT;
```

```
    number,             { Number of records }
    index               { Index for record }
      : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Name of external disk file');
    RESET (diskfile);
    index := 2;
    REPEAT
        index := index + 1;
        READLN (diskfile, internal[index].key,
                          internal[index].balance)
    UNTIL EOF (diskfile);
    CLOSE (diskfile);
    number := index - 2;
    internal[1].key := -9999;
    internal[1].link := 3;
    internal[number + 3].key := 9999;
    internal[2].link := number + 4;
    FOR index := 3 TO 99 DO
        internal[index].link := index + 1;
    internal[100].link := -9999;
    WRITELN;
    WRITELN ('Number of records ', number)
END;
PROCEDURE save
    (VAR internal : table);
{ Save internal file to external disk file }
VAR
    diskfile            { Disk file for records }
      : TEXT;
    index               { Index for record }
      : INTEGER;
BEGIN
    WRITELN;
    WRITELN ('Name of disk file to contain internal file');
    REWRITE (diskfile);
    index := internal[1].link;
    REPEAT
        WRITELN (diskfile, internal[index].key,
                           internal[index].balance);
        index := internal[index].link
    UNTIL internal[index].key = 9999;
    CLOSE (diskfile)
END;
```

```
{ Body of main program }
BEGIN
    initial;
    code := 0;
    REPEAT
        WRITELN;
        WRITELN ('Code  Operation');
        WRITELN ('    1  Build new file');
        WRITELN ('    2  Load existing file');
        WRITELN ('    3  Save to external file');
        WRITELN ('    4  Update existing record');
        WRITELN ('    5  Insert new record');
        WRITELN ('    6  Delete old record');
        WRITELN ('    7  Display current record');
        WRITELN ('    8  Print contents of records');
        WRITELN ('    9  Terminate processing');
        WRITELN;
        WRITE ('Code number ? ');
        READLN (code);
        CASE code OF
            1 : build (internal);
            2 : load (internal);
            3 : save(internal);
            4 : change(internal);
            5 : add(internal);
            6 : remove(internal);
            7 : display(internal);
            8 : print(internal);
            9 : BEGIN
                    WRITELN;
                    WRITELN ('Do not terminate unless');
                    WRITELN ('internal list is saved');
                    WRITELN ('to disk file.');
                    WRITELN;
                    WRITE ('Still want to terminate (Y/N) ? ');
                    READLN (response);
                    IF response = 'N' THEN
                       code := 0
                END;
           OTHERWISE WRITELN ('Invalid code number')
        END
    UNTIL code = 9;
    WRITELN;
    WRITELN ('End of program')
END.
```

The following test run creates the initial file:

```
Program LINKED1

Maintain sequential file
as internal linked list.

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
   9  Terminate processing

Code number ? 1

Number of records ? 3

Information for

Record          1
Id number ? 101
Balance     ? 75.95

Record          2
Id number ? 109
Balance     ? 145.25

Record          3
Id number ? 105
Balance     ? 25.99

End of data entry

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
   9  Terminate processing

Code number ? 3

Name of disk file to contain internal file
DISKFILE = DATA1/DAT
```

```
Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
   9  Terminate processing

Code number ? 9

Do not terminate unless
internal list is saved
to disk file.

Still want to terminate (Y/N) ? Y

End of program
```

The following test run updates the file:

```
Program LINKED1

Maintain sequential file
as internal linked list.

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
   6  Delete old record
   7  Display current record
   8  Print contents of records
   9  Terminate processing

Code number ? 2

Name of external disk file
DISKFILE = DATA1/DAT

Number of records      3

Code  Operation
   1  Build new file
   2  Load existing file
   3  Save to external file
   4  Update existing record
   5  Insert new record
```

```
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 8

Use device   :L to send to printer
PRINTER = :L

End of output

Code  Operation
    1  Build new file
    2  Load existing file
    3  Save to external file
    4  Update existing record
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 5

Insert record
ID number ? 102
Balance   ? 83.48

Code  Operation
    1  Build new file
    2  Load existing file
    3  Save to external file
    4  Update existing record
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 6

ID number to delete ? 105

Balance =          25.99
Do you still wish to delete (Y/N) ? Y

Code  Operation
    1  Build new file
    2  Load existing file
    3  Save to external file
```

```
    4  Update existing record
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 8

Use device   :L to send to printer
PRINTER = :L

End of output

Code  Operation
    1  Build new file
    2  Load existing file
    3  Save to external file
    4  Update existing record
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 3

Name of disk file to contain internal file
DISKFILE = DATA2/DAT

Code  Operation
    1  Build new file
    2  Load existing file
    3  Save to external file
    4  Update existing record
    5  Insert new record
    6  Delete old record
    7  Display current record
    8  Print contents of records
    9  Terminate processing

Code number ? 9

Do not terminate unless
internal list is saved
to disk file.

Still want to terminate (Y/N) ? Y

End of program

First printed listing:
```

```
Id number   Balance
      101     75.95
      105     25.99
      109    145.25
```

End of output

Second printed listing:

```
Id number   Balance
      101     75.95
      102     83.48
      109    145.25
```

End of output

EXERCISES

1. Write a program to maintain a telephone directory. Use this section's method.
2. Write a program to maintain a customer name and address file. Use this section's method.

Index

Petrocelli's "INVITATION TO" Series

Invitation to FORTRAN for the TRS-80
Invitation to COBOL for the TRS-80
Invitation to PASCAL for the TRS-80

by Lawrence L. McNitt

Invitation to Ada & Ada Reference Manual
Invitation to FORTH
Invitation to PASCAL
Invitation to MAPPER

by Harry Katzan, Jr.

Invitation to MVS: Logic & Debugging

by Harry Katzan, Jr. and Davis Tharayril

Invitation to BASIC Fitness

by Stephen J. Mecca & Cemal Ekin

089433-253-8