PASCAL

JAMES L. RICHARDS

PASCAL

JAMES L. RICHARDS

Bemidji State University

ACADEMIC PRESS

A Subsidiary of Harcourt Brace Jovanovich

New York London

Paris San Diego San Francisco São Paulo

Sydney Tokyo Toronto

Academic Press, Inc.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
Academic Press, Inc. (London) Ltd.
24/28 Oval Road, London NW1 7DX

ISBN: 0-12-587520-7
Library of Congress Catalog Card Number: 81-66761

Printed in the United States of America

PREFACE

This book provides an introduction to computer programming and to the programming language called Pascal. No prior experience either programming or using computers is required. Some knowledge of basic algebra is necessary to understand several of the examples presented in the book and is helpful in learning the rules which govern the formation and use of mathematical expressions in the Pascal language. Readers who have completed a course in college algebra or have taken 1½–2 years of high school algebra should have little difficulty with the mathematical examples and concepts that are used.

Books on computer programming tend to emphasize the development of programs to solve real-world problems or the valid structures of instructions in a specific programming language. In an introductory programming course, a problem-solving approach is usually more attractive to students than one which centers on the constructs of a particular programming language, and yet the end product of a student's programming efforts must be a correct program written in some programming language. Thus, a parallel approach that incorporates programming methodology and the grammar of a programming language is most appropriate. In this book, I have tried to give a balanced presentation of programming methodology and the Pascal language.

In order to design and implement efficient and effective computer programs, a logical development strategy and an equally logical programming language are necessary. Pascal is the language presented in this book because it facilitates teaching good programming practices and because it is also being used widely for applications in business, industry, and education. Furthermore, versions of Pascal are available for nearly every popular large-scale computer and microcomputer.

Chapter One gives some background material on computer systems, programming methods, and programming languages. Students who have previous programming experience should be able to proceed through this chapter very rapidly. Some discussion of the differences between batch and time-sharing computer systems and their control languages is presented, but the instructor will have to supply students with the specific control language instructions they will need to use.

Chapters Two and Three cover basic elements of the Pascal language such as representations for numbers and other types of data, mathematical and nonmathematical operations on data, and the fundamental composition of a Pascal program. Many short and simple programs appear as examples in these two chapters, mostly to show how various types of data can be represented and used in a program. All four of the Pascal standard scalar data types (CHAR, INTEGER, REAL, and BOOLEAN) and simple expressions involving operations on such data are presented in Chapter Two. Expressions that involve several operations are discussed in Chapter Three.

Chapter Four is devoted entirely to Pascal instructions that are used for data input and output. The principles that govern the way that Pascal handles data input from an external source and displays the results of processing under program control are explained and illustrated using many examples and diagrams. While these principles are fundamentally the same for all versions of Pascal, some of the specifics concerning input and output (particularly, input) given in this chapter and illustrated in the examples are not exactly the same for all versions of Pascal. Your instructor or local computer center can provide the specific information you need to know about input and output for the version of Pascal available on your computer system.

Chapters Five and Six cover all of the Pascal instructions that control the sequencing of data processing. In Chapter Five, only those instructions that govern the selection of alternative processing activities are discussed (IF and CASE statements). Instructions that are used to control repetitive processing activities are presented in Chapter Six (WHILE, REPEAT, and FOR statements). The GOTO statement, which can be used to control either selection or repetition, is introduced at the end of Chapter Six (Section 6.4). Since Pascal provides a wealth of control structures that are far more attractive in style than any that use GOTO statements, Section 6.4 is included more for completeness than anything else and may be skipped without loss of continuity.

Chapter Seven provides an introduction to data and data structures that can be defined in a program. Only one type of structured data (the array) is discussed in Chapter Seven, but it is the structured data type that most readers having previous programming experience will recognize. After a slight detour to discuss subprograms (functions and procedures) and their role in the modular design of a program, additional structured data types are introduced in Chapter Nine (sets and records), Chapter Ten (files), and Chapter Eleven (dynamic data structures).

The material in Chapters One through Nine can easily be covered in a one-quarter, 4-credit course or a one-semester, 3-credit course. In fact, I have been able to include selected sections of Chapters Ten and Eleven in one quarter. A thorough discussion of files and dynamic data structures is usually reserved for an advanced programming course that concentrates on data structures. However, the rather elementary presentation of files, pointers, and dynamic variables in Chapters Ten and Eleven should be included in the course, if time permits, since it can ease the transition to a follow-up course on data structures.

To date, there is no universally accepted standard for Pascal. However, work toward an international standardization of Pascal has been in progress for several years. The International Standards Organization (ISO) is considering a proposed standard that was originally developed as Working Draft/3 by the British Standards Working Group DPS/13/14. To the best of my knowledge, the features of Pascal presented in this book are consistent with the proposed standard.

There are many people to whom I will always be indebted for their help, advice, and encouragement during the preparation of this book. First and foremost, I would like to thank my family for their understanding and encouragement. I am also very grateful to my colleague Professor Tom Richard at Bemidji State University, who reviewed several versions of my manuscript and helped me class-test

materials, and to Brent Cochran, Ron Elshaug, Jane Franz, Jim Herring, and Carol Mack, who devoted many hours to checking the examples, programs, and exercises that appear in this book. Finally, I would like to express my sincere appreciation to all the referees who reviewed the manuscript and made many helpful suggestions.

JAMES L. RICHARDS
Bemidji, Minnesota
August 1981

CONTENTS

ONE/AN INTRODUCTION TO COMPUTER SYSTEMS AND PROGRAMMING

Modern computers are powerful machines that are used to collect, analyze, and process enormous amounts of information rapidly and with a high degree of accuracy. The physical form of this information is referred to as **data.** A computer processes data by executing a sequence of precise instructions called a **program.** Computer programs are designed and constructed by people known as **programmers** to perform data processing tasks that will solve real world problems. Programming a computer is not really difficult, but it is meticulous work because every computer understands only a limited set of very exact instructions.

This book is about computer programming. All programmers need to know some fundamental terminology and facts about computers and programming. These basics are presented in this chapter. Much of the information in the following sections is not very detailed, since it is meant to provide only an overview of computers and programming. As you read and study the material in this chapter, try to develop some understanding of how computers, programs, and people interact to accomplish data processing tasks. Pay particular attention to the terminology that is introduced, because much of it will be used frequently in the remainder of the text. This chapter will not answer all the questions you may have about how computers operate and how they are programmed, but it will give you some basic knowledge that will help you develop programming skills.

1.1 FUNCTIONAL UNITS OF A COMPUTER

The physical machinery of which a computer is constructed is referred to as **hardware.** Each hardware device consists of electronic circuits and wires assembled to give that device certain data processing capabilities. The general organization of the hardware components for a typical computer is depicted in Figure 1–1. Basically, a computer has four functional units: a **central processing unit (CPU),** an **input unit,** an **output unit,** and a **memory.** The arrows in Figure 1–1 represent the flow of data between the various units.

THE CENTRAL PROCESSING UNIT

The central processing unit consists of circuitry that monitors and controls all the other hardware devices that are part of the computer. Actually, the CPU is composed of two units: the **control unit** and the **arithmetic and logic unit.** The control

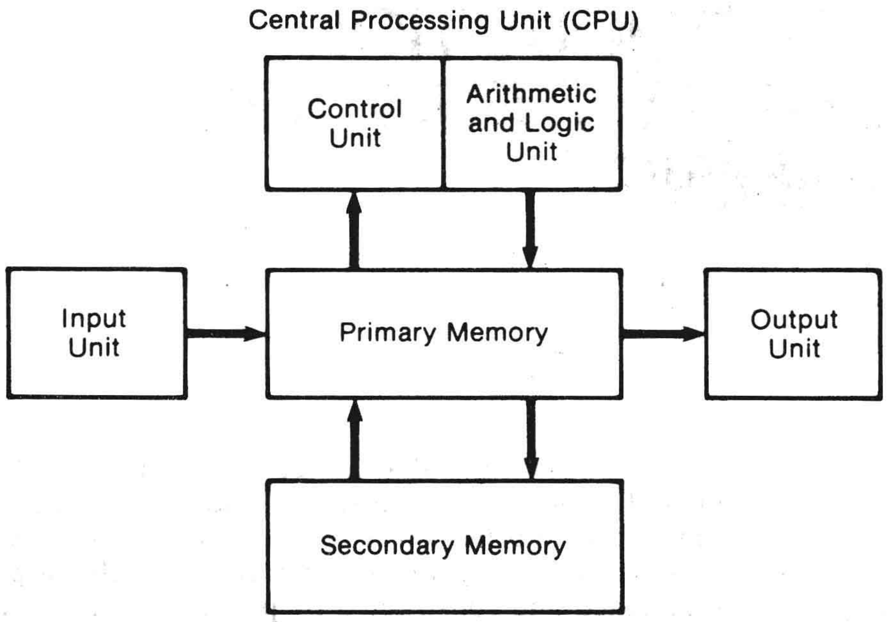

Figure 1–1
The functional units of a typical computer.

unit can access instructions from programs stored in memory, interpret those instructions, and then activate appropriate units of the computer to execute them. Other activities of the control unit include generating control and timing signals for the input and output units, entering and accessing data stored in memory, and routing data between memory and the arithmetic and logic unit.

The arithmetic and logic unit is a servant of the control unit. It can perform such simple arithmetic operations as addition and subtraction and it can perform certain logical operations such as comparing two numbers. The control unit provides the arithmetic and logic unit with appropriate data and then activates that unit to perform the desired operation.

MEMORY

As depicted in Figure 1–1, a computer has two types of memory: **primary memory** and **secondary memory.** Primary memory is sometimes called **internal memory** because it usually occupies the same physical enclosure as the central processing unit. A computer's primary memory consists of individually accessible storage cells, which we shall call **memory locations.** Each memory location can store exactly one data value, such as a number. A small computer may have only a few thousand of these memory locations, whereas large computers often have more than a million storage cells in their primary memory. Every memory location has a unique identification number, which serves as its **memory address.** We can think of an individual memory location as a box with a numbered lid whose contents are always visible through one end, as depicted in Figure 1–2. The central processing unit can access any memory location by using its memory address. Once the CPU has found a particular memory location, it can simply observe the con-

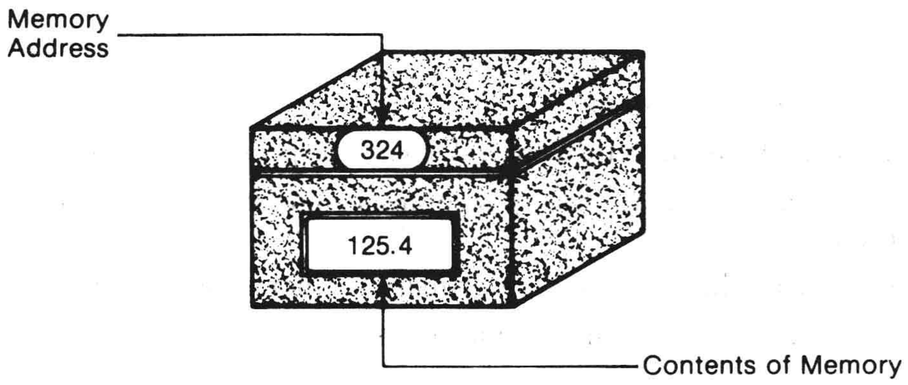

Figure 1–2
A memory location simulated as a box with a window at one end through which its contents are visible.

tents of that storage cell or it can store some value there. In the latter case, the new value replaces any value that is already in the memory location. The "old value" is destroyed because a memory location has the capacity to store only one value at a time.

The CPU accesses storage locations in primary memory very rapidly compared to those in secondary memory. Primary memory is normally used to store only information currently being processed by the CPU because the number of memory locations in primary memory is always limited. Secondary memory (also known as **mass storage**) provides more permanent data storage. Magnetic tapes and disks are common forms of secondary memory. Magnetic tape is a plastic ribbon coated with magnetic material on which information can be recorded in much the same way that a voice or music is recorded on sound tapes. A magnetic disk is a thin circular disk made of metal or plastic; it, too, is coated with magnetic material that serves as a recording medium. The amount of secondary memory is essentially unlimited, since tapes and disks can be removed from recording devices when they are filled with information and can be replaced by new tapes or disks.

INPUT AND OUTPUT UNITS

Input and output units link a computer with the outside world. Data and programs enter a computer's primary memory via some input device, and processed information is displayed on some output device. There are many types of input devices, and each one can "read" information represented in some physical form. Programs and data prepared on punched cards or paper tape, mark-sense cards, magnetic tape, or magnetic disks may be fed into an appropriate input device. Some input devices have typewriter-like keyboards that can be used to enter information. An output device is used to copy information from the computer's memory onto some recording medium. There are output devices that will print on paper, punch cards, or paper tape; record on magnetic tape or disks; or display information on a television screen. Although every computer normally uses at least one input device and one output device, it is not uncommon for the input and output units to have available several devices for input and output.

1.1 EXERCISES

1. What is a computer program?
2. To what does the term "hardware" refer with respect to computers?
3. Name the four functional units of a computer and briefly describe the function of each unit.
4. Why does a computer need both a primary memory and a secondary memory?
5. What does the phrase "memory address" mean?

1.2 COMPUTER PROGRAMS AND PROGRAMMING

The actual writing of a computer program is called **coding.** A program is simply a sequence of instructions for a computer that has been coded in a specific **programming language.** There are many programming languages and each one is a formal system of symbols, including rules for forming expressions, that can be used by a human being to communicate with a computer. Meaningful expressions are formed according to rigid **syntax rules** (or **grammar**) utilizing a well-defined vocabulary. Every program instruction must conform precisely to the syntax rules for the language in which the program is written. The rules are very rigid because a computer cannot "think" like a human being; it merely follows precise directions given in a program, and so those directions must be unambiguous. As with any language, the grammar for a programming language tells how to form "sentences" that are properly structured. There are rules of **semantics,** which tell when a syntactically correct instruction is also meaningful. Consider the following two English sentences.

Put the meat into the refrigerator.
Put the refrigerator into the meat.

Both sentences are syntactically correct according to the grammar of the English language, but the second sentence is semantically incorrect.

MACHINE-LEVEL LANGUAGES

The central processing unit can only execute instructions that are coded in **machine language.** In machine language, instructions and data are stored in the computer's memory as numbers composed solely of 1's and 0's. This is known as **binary coding** and the digits 0 and 1 are referred to as **bits** (short for **b**inary dig**its**). The number of bits that can be stored in a memory location is fixed for each computer. Suppose that our computer has memory locations that store 16-bit numbers. If we could look at the memory location whose addess is 327, we might see

```
327 | 0001010000001100 |
```

The contents of memory location 327 could represent a machine language instruction. If so, the control unit of the CPU can decode the instruction by examining groups of consecutive bits. For instance, the first six bits could be an operation code and the remaining ten bits could specify the source of data needed for the designated operation, as illustrated below.

000101	0000001100
Operation Code	Data Source

A machine language program consists of a sequence of binary coded instructions that the control unit is able to decode and execute.

In order to execute a program, the CPU uses special storage locations called **registers.** These registers are not part of primary memory. The computer executes the instructions in a program one at a time by repeating the same sequence of activities over and over again. That sequence of activities is called the **instruction execution cycle.** Two registers play central roles in the execution of an instruction: the program counter (register PC) and the instruction register (register IR). Here is a typical instruction execution cycle.

1. Fetch the instruction whose memory address is in register PC and copy it into register IR.
2. Increment the contents of register PC by 1 so that the memory address of the next instruction will be available at the start of the next cycle.
3. Decode the contents of register IR and fetch the data needed to perform the specified operation.
4. Execute the instruction.

When execution of the program is initiated, the program counter must contain the address of the first instruction. After that, each instruction execution cycle establishes the memory address for the instruction to be executed during the next cycle.

The CPU also has registers that it can use as "scratch pads" for numerical calculations. A register of this type is called an accumulator, and so we will refer to it as register AC. Some typical machine operations and the corresponding codes are given in the table below. The operations and codes listed in this table are for a hypothetical computer.

Operation Code		
Binary	Decimal	Operation
000000	0	Halt the execution of the program.
000101	5	Store the value of the data source in register AC.
010000	16	Load a copy of the value in register AC into the memory location whose address is the data source.
101011	43	Add the value of the data source to the contents register AC.

The table shows the operation codes in both binary and decimal form, the latter being the usual way to represent a whole number. Consider the following machine language program, which is stored in memory locations 327 through 330. The operation codes and data sources are also shown in decimal form so that we can follow the execution of the program.

Memory Address	Memory Contents	Operation Code	Data Source
327	0001010000001100	5	12
328	1010110000110011	43	51
329	0100000101000110	16	326
330	0000000000000000	0	0

When this program is executed, the numbers 12 and 51 will be added and the sum will be stored in memory location 326. Initially, the program counter will contain the memory address of the first instruction (327). A summary of the effects of each instruction is shown below. This summary traces the changes made to the contents of registers IR, PC, and AC. The contents of registers PC and AC would be in binary form inside the computer, but they are shown in decimal form here so that we can follow the progress of the program more easily.

	Instruction Register IR	Register PC	Register AC
		327	
Store the value 12 in the accumulator (register AC).	0001010000001100	328	12
Add the value 51 to the contents of register AC.	1010110000110011	329	63
Load memory location 326 with a copy of the contents of AC.	0100000101000110	330	63
Halt the program.	0000000000000000	331	63

At the end of the program, both register AC and memory location 326 (not shown in this table) will contain the value 63 (decimal).

Coding machine language instructions is difficult because it is necessary to remember specific combinations of 1's and 0's or to be constantly looking them up in some reference manual. Even one bit that is misplaced can totally change the meaning of an instruction. It is very easy to make errors when coding a machine language program and very difficult to locate the errors so that they can be corrected. In short, machine language programming is so complicated and time consuming that it is rarely used.

As an alternative to machine language coding, programs can be written by using abbreviations instead of binary codes to represent machine-level instructions. A language of this type is known as an **assembly language.** An assembly language program is written at the same level of detail as a machine language program, but the instructions are written in a form that makes them easier for humans to read. Consider the following sequence of hypothetical assembly language instructions.

```
SUM     CON  0
PROG    LDAC 12
        ADD  51
        STAC SUM
        HALT PROG
        END
```

As we know, a computer can only execute a program that is written in machine language. An assembly language program must be translated into machine language before it can be executed. This task is performed by another program, called an **assembler,** that resides in the computer's memory. The assembler treats an assembly language program as data and produces an equivalent version of that program in machine language. When we say that an assembly language program is executed, we mean that the assembled machine language version of that program is executed.

In the assembly language program shown above, the symbolic names LDAC, ADD, STAC, and HALT represent machine-level operations. The assembler bears the burden of constructing machine language instructions in which the corresponding binary operation codes appear. Relative to the operation codes we discussed earlier, these symbolic operation codes may have the meanings shown in the following table.

Operation Code		
Symbolic	Binary	Operation
HALT	000000	Halt the execution of the program.
STAC	000101	Store the value of the data source in register AC.
LDAC	010000	Load a copy of the value in register AC into the memory location whose address is the data source.
ADD	101011	Add the value of the data source to the contents of register AC.

The data sources appear after the symbolic operation codes in the assembly language program. SUM and PROG are symbolic names for memory addresses. SUM represents a memory location whose initial value is the CONstant 0. PROG is the symbolic name for the address of the first instruction in the program. When the program is assembled, the memory address corresponding to PROG is recorded so that the address of the first machine language instruction can be placed in the program counter when execution of the program is initiated. The word END marks the physical end of the assembly language program so that the assembler knows when to stop creating machine language instructions. If PROG has been associated with the memory address 327, the assembler language program and its corresponding machine language version will be as shown below.

Assembly Language Program			Machine Language Version	
			Memory Address	Memory Contents
SUM	CON	0	326	0000000000000000
PROG	LDAC	12	327	0001010000001100
	ADD	51	328	1010110000110011
	STAC	SUM	329	0100000101000110
	HALT	PROG	330	0000000000000000
	END			

The machine language program assembled here is identical to the one discussed earlier. Thus the purpose of the assembly language program is to add the numbers 12 and 51 and store the result in the memory location immediately preceding the one that contains the first program instruction (memory location 326, whose symbolic name is SUM).

Although an assembly language program looks different from a machine language program, programming in assembly language is still dominated by machine-oriented concepts. One of the major drawbacks to programming in machine language or assembly language is that there is no one machine language for all computers. In fact, machine languages (and hence assembly languages) vary considerably from computer to computer.

HIGH-LEVEL LANGUAGES

A computer's machine language is, unfortunately, far removed from languages that people use to communicate with other people. For this reason, modern computers are equipped with "built-in" programs called **systems programs** that enable them to communicate with people in a more human-like fashion. An assembler is a systems program that allows a programmer to create a machine-level program using symbols that are more descriptive of machine operations than binary code. The assembler's job is to translate a grammatically correct assembly language program into machine language. A systems program that takes a program written in one language and produces a version of that program in a different language is known as a **language processor.** There are three types of language processors: **assemblers, compilers,** and **interpreters.** Programs written in an assembly language specify operations at the machine level and so they must be coded with the hardware capabilities of the computer in mind. Other symbolic languages have been developed to take care of specific hardware requirements automatically so that the programmer can concentrate more on procedures and problem solving and less on the work of the computer. These so-called **high-level languages** allow program instructions to appear in an English-like form or with mathematical formulas. Compilers and interpreters are language processors used to produce translations of programs written in high-level languages.

While machine and assembly languages are tied to particular computers, high-level languages are not. Compilers and interpreters for a high-level language can be implemented as systems programs on a wide range of computers. This versatility makes possible the coding of "portable" programs; that is, a program coded in a high-level language can be translated and executed by many different computers. There are hundreds of high-level languages in existence, but only a few of