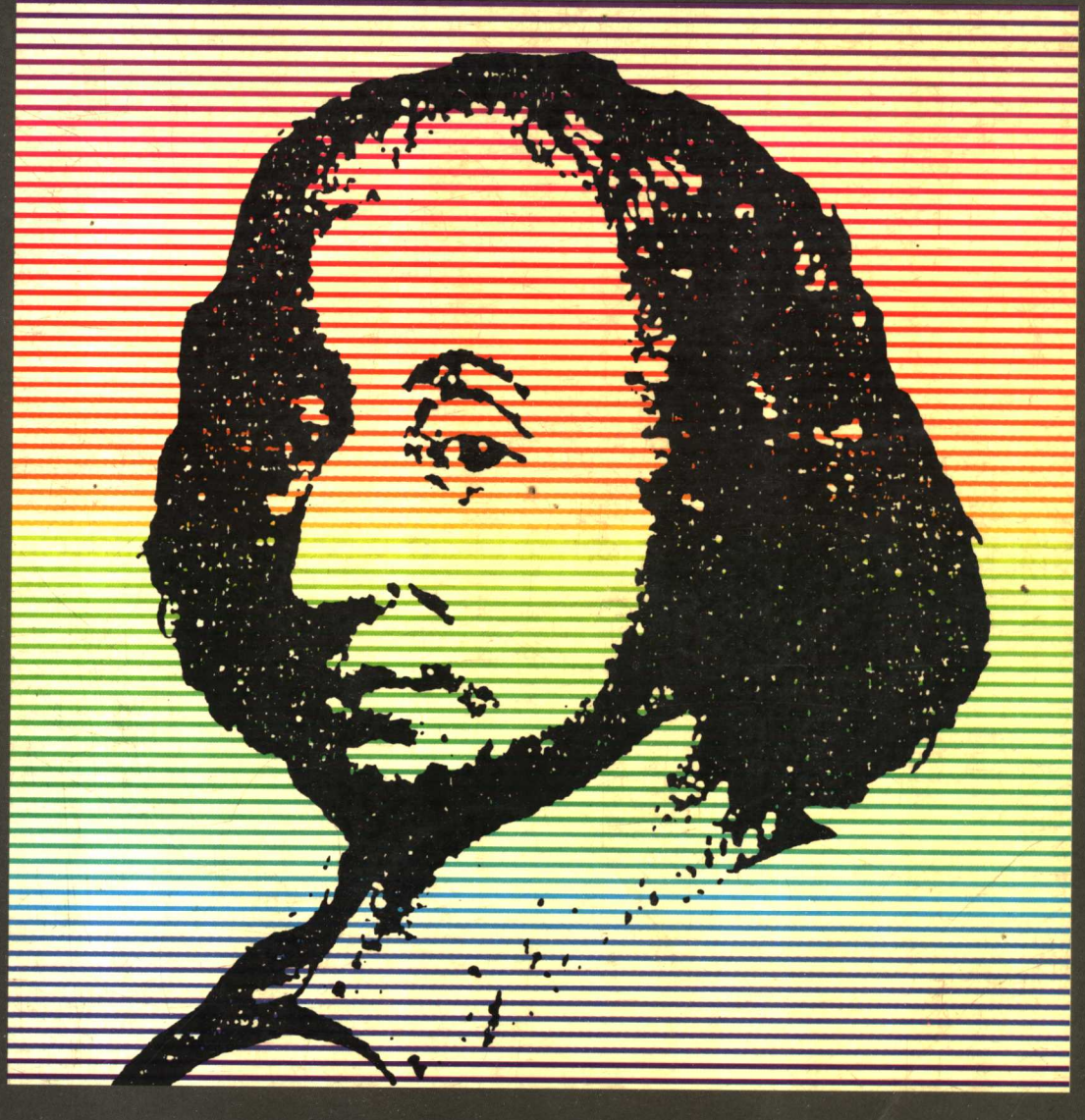
PASCAL

JAMES L. RICHARDS



SECOND EDITION

PASCAL

SECOND EDITION

James L. Richards
Bemidji State University

ACADEMIC PRESS COLLEGE DIVISION
Harcourt Brace Jovanovich, Publishers

Orlando San Diego San Francisco New York London
Toronto Montreal Sydney Tokyo São Paulo

Academic Press, Inc.
Orlando, Florida 32887

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

ISBN: 0–12–587522–3
Library of Congress Catalog Card Number: 85–71911
Printed in the United States of America

Preface

Pascal, Second Edition, reconfirms my philosophy of teaching computer science. The guiding principles of the first edition remain at work in this revision. Particular sections have been changed to improve the original or to bring it up to date. The strengths of the first edition, which depend greatly on my own beliefs that time is well spent in careful preparation, frequent practice of skills, and gradual accumulation of those skills, are still in evidence. In its first edition, *Pascal* demonstrated its worthiness in the classroom. With this new edition, my concern was not how to teach, but what to teach. Specific changes have been made in line with contemporary trends, making this new edition especially effective and thorough.

One of the strengths retained from the first edition is that the student need not have prior programming experience. The beginning chapters are paced to provide a gradual introduction to programming concepts and fundamental elements of the Pascal language. Alternately, the student with some programming experience can proceed fairly rapidly through at least the first four chapters. Students who have successfully studied high school algebra and geometry should easily understand the mathematical examples.

This book's main purpose is to explain how to design and construct computer programs using Pascal. Learning the specifics of a programming language is a substantial task for anyone. Beginning programmers are faced with having to learn programming methodology simultaneously with the syntax of a programming language. The development of efficient and effective programs requires problem-solving skills that are reinforced by studying examples and by working on programming problems. This new edition, even more than the first, emphasizes these necessary elements.

Algorithms

Pascal, Second Edition, repeatedly encourages the development and stepwise refinement of algorithms that serve as program models. All algorithms in the examples are in a narrative form that incorporates elements resembling specific

Pascal constructs (a form of pseudocode), thereby easing the transition from an informal outline to a precisely worded program. Only a few formal flowcharts remain from the first edition to demonstrate the effects of particular Pascal instructions, not to serve as outlines for program segments or modules. Although I have used a consistent style in presenting the algorithms, the student need not adopt this style. Teachers of programming should be aware that students face a formidable task in remembering formal rules imposed by a programming language. To expect the student to conform to rigid rules for preparing preliminary algorithms often discourages the student from outlining before coding.

Programming Examples

Many program segments and short program examples demonstrate the use of particular Pascal instructions as they are introduced. Most program listings are followed by one or more samples of printed output generated by the program. Because the various implementations of Pascal lack uniformity for reading and writing data, the student may need to slightly modify some of the example programs before running them on a particular system.

Case Studies

Fourteen case studies provide extensive examples of methodical program development. These case studies appear throughout the text, beginning as early as Chapter 3. Each one details the step-by-step design and implementation of a complete Pascal program, starting with a broad description of the requirements for the program.

Exercises and Programming Problems

Nearly 900 class-tested exercises and programming problems appear; many are new or revised forms of exercises and problems in the first edition. The exercises are grouped at the ends of the sections so the student can test his or her understanding of that section. Appendix E contains answers for selected exercises. More extensive programming exercises appear as Programming Problems in a separate section at the end of the chapter—many of these problems are new. The problems cover a wide range of programming applications (e.g., processing television ratings, creating a technical dictionary) so that each student may select problems of special interest.

Programming Projects

Appendix F contains descriptions for eight programming projects. The project requirements, suggested enhancements, and general descriptions have been restated to more clearly delineate the scope of each project. Even more extensive than the programming problems at the ends of chapters, these projects are more open-ended so the instructor can vary the breadth and complexity of project assignments.

Character Sets

Example programs and program segments use the full ASCII character set. Students using Pascal implementations that employ some subset of the ASCII characters or some other character set may need to revise the example programs to run them on their systems. Appendix A lists the ASCII, CDC, and EBCDIC characters.

Program Modules as Procedures

Pascal procedures (without parameters), introduced as early as Chapter 3, facilitate the modular development of programs. This represents a significant change in the approach taken in the first edition of *Pascal* in which subprogram (procedure and function) declarations were not discussed until Chapter 8. In this new edition, Chapter 7 fully covers subprograms.

Syntax Diagrams

Syntax diagrams occur throughout the text to define precisely elements of the Pascal language. The current popularity of these diagrams necessitates that potential computer scientists understand what they mean. Beginning programmers are likely to be intimidated by a syntax diagram. Thus, instructors may wish to spend extra time explaining syntax diagrams and their use in verifying the validity of program entities.

Print Enhancements

Boldface is used to identify Pascal reserved words in programs appearing in the text. In Chapter 2, which introduces string data, apostrophes—not single quotation marks—have been used in the regular text to show that char constants are

formed by enclosing the data with the ASCII decimal 39 character. This has been done to avoid confusing the student with different variations of the same ASCII character when he or she is first learning the syntax of character strings. Color is used functionally to indicate operations performed by the user.

Pascal Implementations

No one text is the final authority for every implementation of the Pascal language. This text follows the standard for Pascal established by the International Standards Organization (ISO), which allows certain extensions and variations from one implementation of Pascal to another. Two appendixes provide additional information about the Pascal language:

1. Appendix G describes several advanced features of Pascal that are included in the ISO Standard (e.g., statement labels, the goto statement, procedures and functions as parameters, and conformant arrays).
2. Appendix H lists peculiarities of three popular Pascal implementations (Apple Pascal, Turbo Pascal, and VAX-11 Pascal).

Course Structure

One ten-week quarter should allow sufficient time to cover material in the first eight chapters as well as selected topics from the remaining three chapters. When this book is used for a semester course, there should be ample time for a detailed discussion of record structures in Chapter 9. Courses extending to two terms can include all eleven chapters and at least some of the advanced features mentioned in Appendix G. Individual or group programming assignments using the projects described in Appendix F can be formulated to give students practice in developing programs that require extensive design.

Beyond Pascal

For the student pursuing the study of computer science beyond a single course in programming with Pascal, the recommended next course is one emphasizing the design and use of data structures, guided by data abstractions and software-engineering principles. The closing chapters of *Pascal,* Second Edition, provide the groundwork for an in-depth study of more complex data structures and their applications. A follow-up text building on this groundwork is in preparation.

Instructor's Manual

A supplementary manual for instructors is available from Academic Press. This manual has four parts: (1) answers to exercises that are not in Appendix E, (2) notes for teachers, (3) a collection of test questions, and (4) transparency masters for selected diagrams and algorithms.

Acknowledgments

Many people provided help, advice, and encouragement that aided me in the preparation of this text. I am indebted to those who carefully reviewed the manuscript and offered many suggestions that resulted in significant improvements in the accuracy and presentation of the material. They are George Beekman, Oregon State University; Howard Binnick, City University of New York & Bramson ORT Technical Institute; Sarah Brooks, Mohawk Valley Community College; Wilber P. Dershimer, Jr., Seminole Community College; Louis Gioia, Nassau Community College; Jim Ingram, Amarillo College; Russell Lee, Allan Hancock College; C. William Marsh, University of Cincinnati and Raymond Walters College; Bro. Ernest Paquet, f.i.c., Brothers of Christian Instruction at Walsh College; Margaret Anne Pierce, Georgia Southern College; and Tom Richard, Bemidji State University. I am especially grateful to Professor Susan Hickey for preparing the summaries of Apple Pascal, Turbo Pascal, and VAX-11 Pascal that constitute an invaluable Appendix H, and for writing the *Instructor's Manual.* Finally, I must thank my family for their continuing encouragement, understanding, and patience.

Contents

1

An Introduction to Computer Systems and Programming

Modern computers are high-speed electronic devices capable of collecting, storing, analyzing, and processing enormous amounts of factual information (or **data**) quickly and accurately. We humans can accomplish very little data processing per second compared to today's computers, which are capable of performing thousands of operations per second. Although we cannot match the information gathering and processing speeds of computers, we can direct computer operations toward data processing goals by means of **computer programs** that give step-by-step instructions to the computer. Those who develop computer programs are called **computer programmers.** Unfortunately, we cannot yet talk to computers as we talk to one another. The list of directions we want a computer to follow must be coded in a form transmittable to the computer. The coded program must meet the rigid requirements of a **programming language** that the computer recognizes, like the language **Pascal** described in this book.

Computer programming can be interesting and enjoyable work. Learning to do computer programming can also be an interesting and enjoyable experience, but there are some obstacles that a prospective programmer must overcome. First, there is the problem of communicating with a computer. A programmer must learn how to give fundamental instructions to a computer so that the machine will pay attention. A person need not know everything there is to know about the inner workings of a computer to be able to program it, but some basic familiarity with the computer must be achieved in the early stages of learning to construct and implement programs. Another problem a beginning programmer faces is learning the particulars of the programming language that will be used to code programs. Many hours of practice using a language like Pascal are necessary to become proficient at program coding. For those who know how to write Pascal-coded instructions, the actual writing of the program is easy. The most difficult and most interesting part of programming is planning a program.

Naturally, the planning should come first; if the planning is thorough and accurate, the program should be too.

From Chapter 2 on in this book, programming methodology and the particulars of the Pascal language are examined. The purpose of this first chapter is to present an overview of computer systems and of the programming process. Most of the computer terminology used throughout the book is defined in the next few pages. As you read this chapter, try to develop a general perspective on computers and programming.

1.1 Computer Systems

The electronic and mechanical devices that constitute the tangible components of a computer are referred to as **hardware;** computer programs are said to be **software.** It is an appropriate blend of hardware devices and software that gives a computer life as a data processor. First, we will examine the major hardware components of a computer.

Functional Units of a Computer

Figure 1–1 shows the general organization of the four main functional units of a computer: a **central processing unit** (CPU), a **memory,** an **input unit,** and an **output unit.** The arrows in the diagram show the directions in which data may flow from one unit to another.

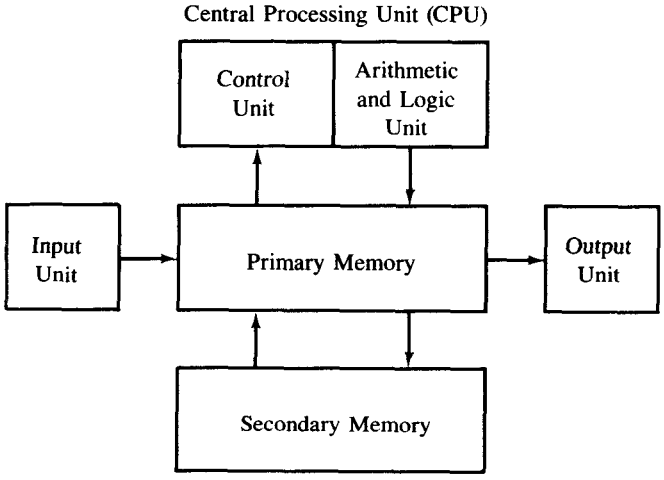


FIGURE 1—1 The functional units of a typical computer.

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 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.

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 will 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; 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 contents 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.

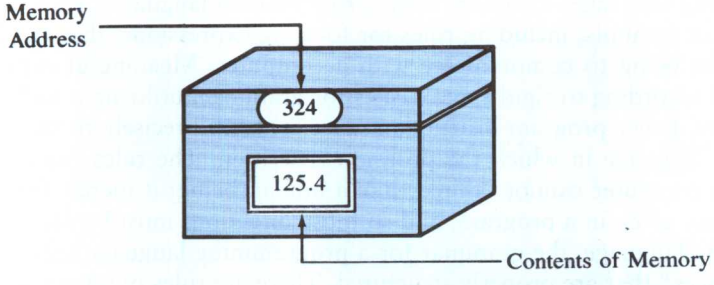


FIGURE 1—2 A memory location simulated as a box with a window at one end through which its contents are visible.

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 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, marksense cards, magnetic tape, or magnetic disks can 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 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.

Programming Languages

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, each one 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:

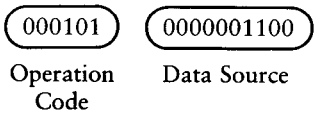
Put the meat into the refrigerator.
Put the refrigerator into the meat.

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

The central processing unit can execute only 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 1s and 0s. This is known as **binary coding** and the digits 0 and 1 are referred to as **bits** (short for binary digits). 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 address 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 decodes 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.



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

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
```

In assembly language instructions, special word symbols (called **mnemonics**) like ADD, STAC, and HALT in the above example replace binary operation codes. This makes an assembly language program easier for people to read than a machine language program once they learn the assembly language. As we know, a computer can execute only a program written in machine language. An assembly language program must be translated into machine language before it can be executed. This task is perfomed 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.

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.

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 nearly human 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.

Machine and assembly languages are tied to particular computers, but high-level languages are not. Compilers or interpreters for a popular high-level language are implemented as systems programs on a wide variety of computers. Furthermore, a high-level language is easier to learn and makes programs more human-readable than they would be in either machine or assembly language. Consider the following example of a simple, yet complete, Pascal program.

The purpose of the above program is made clear by its name, *Add35and64.* Certain words like **program, var, begin,** and **end** in our example have specific