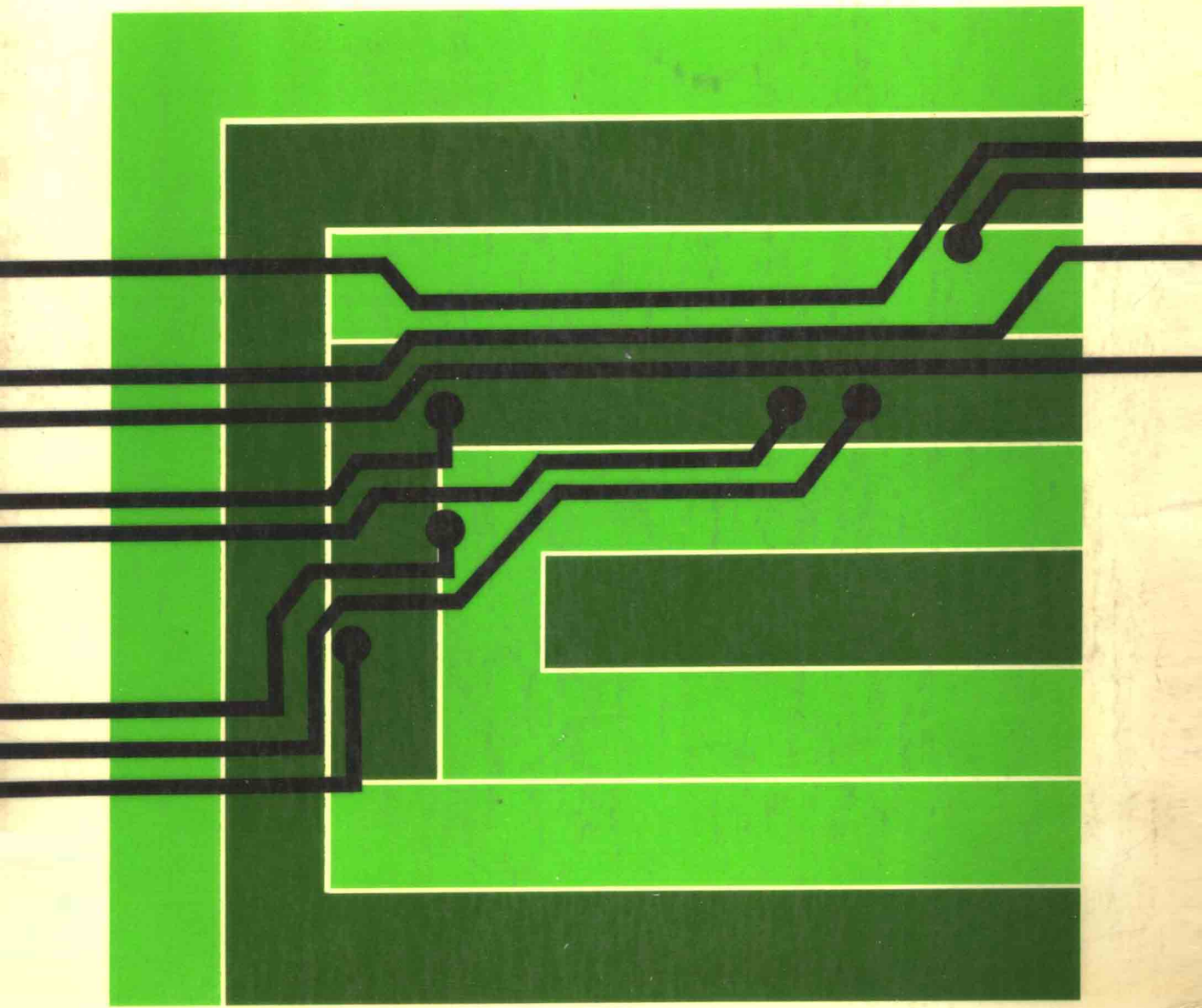
PASCAL

GENEVA G. BELFORD AND C. L. LIU



PASCAL

GENEVA G. BELFORD
C. L. LIU

UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

McGRAW-HILL BOOK COMPANY

NEW YORK ST. LOUIS SAN FRANCISCO AUCKLAND BOGOTÁ HAMBURG JOHANNESBURG LONDON
MADRID MEXICO MONTREAL NEW DELHI PANAMA PARIS SÃO PAULO SINGAPORE
SYDNEY TOKYO TORONTO

PASCAL

Copyright © 1984 by McGraw-Hill, Inc.
All rights reserved. Printed in the United States of America.
Except as permitted under the United States Copyright Act of 1976,
no part of this publication may be reproduced or distributed
in any form or by any means, or stored in a data base or retrieval system,
without the prior written permission of the publisher.

34567890DOCDOC8987654

ISBN 0-07-038138-0

This book was set in Bookman Light by The Saybrook Press, Inc.
The editors were Eric M. Munson and Linda A. Mittiga;
the designer was Anne Canevari Green;
the production supervisor was Charles Hess.
The drawings were done by Burmar.
R. R. Donnelley & Sons Company was printer and binder.

Pascal syntax diagrams are reprinted with permission of
Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*,
2d ed., copyright © 1974 by Springer-Verlag, Heidelberg, pp. 116–118.
All rights reserved.

Library of Congress Cataloging in Publication Data

Belford, Geneva G.
Pascal.

Includes index.
1. PASCAL (Computer program language) 2. Structured programming. I. Liu, C. L. (Chung Laung), date
II. Title.
QA76.73.P2B45 1984 001.64'24 83-19927
ISBN 0-07-038138-0

PREFACE

This book is intended to be an introduction to programming in the Pascal language. It was written to fill the need for a book that a nontechnically oriented beginner would be able to read—and, we hope, even enjoy reading. We have not aimed at a textbook for majors in computer science; consequently we do not treat topics such as data structures and algorithms in any systematic manner.

One of the features of the book is that the topics are arranged in such a way that the exposition is always sequential, in the sense that understanding never depends on material to be covered later. We hope thereby to encourage students to read carefully as they go and not postpone understanding of some concepts until later. At the same time, the material is arranged so that students can quickly start to write simple programs and actually use the computer as early as the first week of the course.

Examples of complete programs are included at the ends of many of the chapters to expose students to some of the important concepts of program development, structured programming, and flowcharting. We chose to utilize one of the modular styles of flowcharting (essentially Nassi-Schneiderman), since we believe that this is helpful in gaining a good feel for structured programming. We have also waited to introduce flowcharting until after loops are discussed, since we find that students tend to feel that flowcharts are useless if they first see them used in discussions of straight-line code.

Even though this is an elementary book, we decided to present syntax in a formal way by using, with minor alterations, the syntax diagrams from Jensen and Wirth's *Pascal User Manual and Report*. We believe that the clarity and lack of ambiguity of this approach outweigh the small amount of difficulty that some students may initially experience in understanding the

diagrams. Also, the diagrams, once understood, are useful as a quick summary of the syntax for later reference. (Furthermore, a beginner could choose to skip the syntax diagrams in a first reading of the book.)

The proliferation of versions and extensions of the Pascal language poses something of a problem in writing a text on "Pascal." We have omitted discussions of extensions (e.g., predefined character string types) and have tried to describe the language as originally presented in Jensen and Wirth and clarified in the 1979 draft standard [B. W. Ravenal, "Toward a Pascal Standard," Computer, Vol. 12, No. 4 (April 1979) pp. 68–83.] In the interests of discouraging sloppy programming, we have, however, avoided any use (and indeed mention) of the GOTO statement and statement labels, although GOTO and LABEL have perforce been included in the list of reserved words in Appendix A. Examples in the text were run on a CDC CYBER 175, using the E.T.H. Zurich/University of Minnesota Pascal 6000 Compiler. We have attempted to make it clear to the reader that Pascal implementations vary, so that some experimentation and consultation of local manuals is advisable.

The first eleven chapters of the book cover those features of Pascal with which one can program tasks like those one would learn how to program in FORTRAN or BASIC. Chapters 12–16 deal with what might be called "advanced" features of the Pascal language—recursion, type definition, structured data types, etc. After completing Chapter 11, the student should have developed a reasonable amount of programming sophistication. In the later chapters, therefore, topics are covered at a somewhat brisker pace, although we hope with equal clarity.

We wish to express our appreciation to the numerous friends, students, and reviewers who read various parts of the manuscript at various stages and provided many helpful comments and suggestions. Special thanks go to Arthur Liestman, Sandra Pritchard, and Andrew Spry, who were responsible for writing and working out solutions to most of the exercises, and to Cindy Robins, who, as a high-school-age "beginner," tested the book for readability and suitability for self-study.

GENEVA G. BELFORD
C. L. LIU

此为试读,需要完整PDF请访问: www.ertongbook.com

CONTENTS

1 INTRODUCTION

1-1 INTRODUCTION

The electronic computer was invented slightly more than 30 years ago. Seldom in the history of the world has a technology developed at such an explosive rate. By now computers play a role in almost every aspect of our daily lives. Computers are used to record our banking transactions, to check our income tax returns, and to control the inventories of the stores where we shop. Computers are used to control the flight of spacecraft, to regulate the processes in a chemical factory, and to analyze the results of complex physics experiments.

We are now beginning to see computers appearing in our homes. Computers are built into microwave ovens to carry out complex sequences of cooking operations without our having to be on hand to turn the dials. Ultramodern sewing machines have built-in computers to allow a wide choice of fancy stitches without mechanical cams. Many of us have enjoyed so-called video games, which are really *computer* games that use our TV screen for a display. If we're anxious to save on gas, we could buy a computer for our car that tells us, with merely the touch of a button, what gas mileage we are getting at the moment. And many of the readers of this book are wearing computers on their wrists to tell them the time and carrying small computers in their pockets to relieve them of the need to remember how to multiply.

Given this pervasiveness of computers, anyone with any curiosity at all should want to learn something about them. There are many questions one might raise: First, what *is* a computer? (Clearly the computers inside our watches and those that manage our bank accounts are rather different machines!) How is a computer built? What kinds of jobs can a computer do? If we want a computer to do one of those jobs, what one should we buy—and

此为试读,需要完整PDF请访问: www.ertongbook.com

how do we go about using that computer to get the job done? It is not possible for us to cover all aspects of computing in this book. Our goal is the limited one of teaching you to use a general-purpose computer, that is, one that can be used to carry out a wide variety of tasks involving computations and data manipulation.

If you are interested in learning all about automobiles, an obvious first step is to learn to drive one. This will give you a good feel for the capabilities and limitations of cars. Similarly, if you learn how to use a computer, you will not only be able to use it to solve various kinds of problems, but you also should gain a reasonably good understanding of what the capabilities and limitations of a computer are. Your view, as a beginner, will necessarily be a bit limited. Just as when you first learn to drive a car you are unlikely to have a feel for what it's like to drive in the Indianapolis 500, so when you first learn to use a computer you may not be able to imagine how you could get the computer to do such complex tasks as maintaining the records of a large corporation or guiding a spacecraft to the moon. However, as you progress through this book, you will find yourself gaining a basic understanding of how computers can help solve real problems.

1-2 PROGRAMMING A COMPUTER

A visitor from Mars has just presented us with a robot which he claims can do all sorts of wonderful things. Two immediate questions arise: What are the wonderful things that the robot can do? How do we instruct the robot to do these wonderful things for us?

Similarly, when you discover the availability of a computer at school or at work, you might ask: What are the wonderful things that a computer can do? How do I instruct the computer to do these wonderful things for me?

This book is mainly concerned with answering these two questions. To do this, we will concentrate on teaching you how to give instructions to a computer, since providing an answer to the second question should go a long way toward answering the first. If you are capable of giving instructions to the computer you must have a reasonably good idea of what the computer can and cannot do. To return to the car analogy, when you learn how to put a car in forward and backward gears, you know that a car can move forward or backward. Not having been taught to move a car sideways, you might feel reasonably safe in concluding that cars do not have that capability—although it might occur to a bright student that making a car hop sideways could just be too tricky a maneuver for a beginning course!

The first thing to get clear in learning to use a computer is that a computer is just a machine. Any instruction given to a computer must be given in such a way that the computer can understand exactly what the instruction is. Indeed, a computer can only follow instructions given in a code known as *machine language*. Most computers are built in such a way that these machine language instructions are in the form of sequences of 0's and 1's. (For example, 0111011010 might mean "multiply 3 times 2.") It is

clear that we ordinary human beings would have some difficulty remembering instructions coded like this. Unfortunately, there are compelling engineering reasons for building computers this way. At first glance it seems that we are in a dilemma; namely, a computer can only understand instructions given in the form of sequences of 0's and 1's, while a human being would find it extremely cumbersome to give instructions in this form. The situation, however, is not an unfamiliar one. If we wish to talk to a Martian in English while the Martian can only understand Martian, we need an interpreter. Indeed, the same idea works for communication with a computer, as Figure 1-1 illustrates.

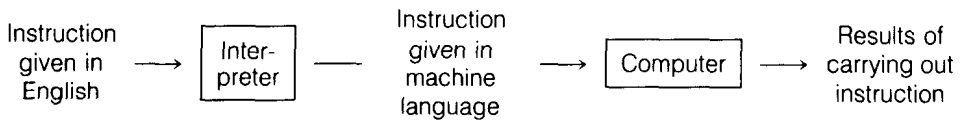


Figure 1-1

The illustration is not accurate in one particular; namely, we really cannot give instructions in plain English to the computer. There are two reasons for this: First, ordinary English is too complicated for us to "train" an interpreter to understand the whole language. (As will be seen, the interpreter is less intelligent than a human.) Second, ordinary English can be ambiguous. There are obvious examples such as, "Find the square root of 4 times 5." (Is the answer 10 or $2\sqrt{5}$?) But even carefully written instructions can leave something to be guessed at. For example, the instruction "Add to each salesperson's salary a bonus of 5 percent of the difference between his total sales and his target sales" leaves open the question as to whether there can be a *negative bonus*, that is, a deduction from the regular salary, if total sales are below target sales. To avoid these problems, what has actually happened is that simple, English-like languages have been designed with the following requirements in mind:

1. They should be close enough to English so that they are easy for human beings to learn, understand, and remember.
2. They should be flexible enough so that it is convenient to instruct the computer to carry out quite complicated tasks.
3. They should be precise enough so that there can be no ambiguity as to the intention of the user.
4. They should make translation to machine language not only possible but also reasonably efficient.

Such artificial languages are referred to as *programming languages*. (We should note that a set of instructions given to a computer to define some task is known as a *program*; writing such a set of instructions is known as *programming* the computer.) Since the early 1950s many programming languages have been proposed. The more important and widely used ones include the languages called FORTRAN, PL/I, COBOL, BASIC, ALGOL, APL, and Pascal. It is not possible, nor is it necessary, for us to teach a student all

these languages in one course. The best approach for a beginner is to pick one language and learn to use it proficiently. Since most of these languages bear a strong resemblance to each other, both in appearance and in underlying design philosophy, students should have little difficulty in learning new languages on their own after they have mastered one.

The language we have picked to describe in this book is called *Pascal*. Pascal is a relative newcomer among programming languages, having been designed by Niklaus Wirth (a professor at the Federal Institute of Technology in Zurich, Switzerland) in 1969. Having been built on some 15 years of previous development and experimentation in the design of programming languages, Pascal is in many ways easier to use—and has fewer pitfalls for the unwary—than many of the older languages. It is therefore a good choice as a language for a beginner to learn.

The next question is: How are interpreters trained to translate instructions given in the Pascal language into instructions in machine language for the computer to execute? As we hinted earlier, human interpreters are not used. There are many obvious reasons for this: Humans are not always reliable; they won't work 24 hours a day; they are slow; they might get sick; they take vacations and coffee breaks; and it takes a long time to train them. A much better idea is to build a machine that will act as an interpreter. As a matter of fact, the best candidate for such a machine is the computer itself. When provided with a set of instructions detailing how to carry out the translation, the computer can perform the task of translating instructions given in Pascal into instructions in machine language. Figure 1-2 illustrates the situation.

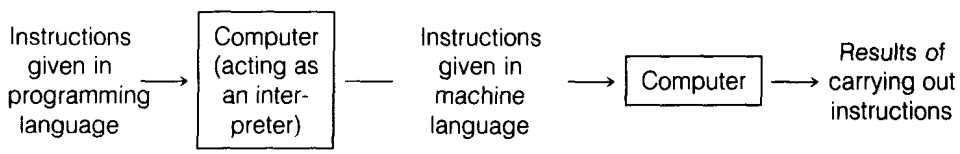


Figure 1-2

The set of instructions (or program) given to a computer to tell it how to translate from an English-like programming language to machine language is usually referred to as a *compiler*. Luckily for those of us who want to use computers, compilers have been written (and are readily available) to translate from almost any popular programming language to the machine language of whatever computer we might have available. Just as human translators may translate the same sentence in slightly different ways, so two different compilers on different computers may translate the same program slightly differently. Programming language features, the detailed effects of which may vary from computer to computer, are referred to as being *implementation-dependent*. Because of this dependence on implementation, we cannot always describe to you in perfect detail what a program written in Pascal will do on your computer. But as we describe the Pascal language, we will try to warn you when your computer may behave slightly differently from ours.

Actually, compilers are only one of a number of programs that automatically come into play to make your use of the computer easy. Another program (sometimes called an *executive* or *monitor*) looks at your job, sees what programming language you have written it in, calls on the proper compiler to translate it, and then sees to it that the resulting machine language instructions are carried out. Still other programs handle the details of fetching the data you want worked on from wherever it may happen to be stored and of returning results to you in the form you ask for—e.g., displayed on a TV screen or printed on paper. On large central computers, such as those maintained at colleges and universities, accounting programs check whether you have money to use the computer and, if so, bill the cost of your job to your account (which, like your bank account, is maintained inside the computer). In short, one no longer should talk about "the computer" but about "the computer system," where the system includes not only the *hardware* (the electronic machine itself) but also *software* (the set of prewritten programs that make it convenient for you to get your jobs done).

SIMPLE PASCAL PROGRAMS

2-1 INTRODUCTION

As we noted in the preceding chapter, a computer program is a set of instructions given to the computer. This book explains how to write programs in the particular language known as Pascal. In this chapter, we shall show you what a Pascal program looks like by introducing one of the simplest (and one of the most important) instructions we can give to the computer.

Instructions given to a computer, like those written in ordinary English, are divided into sentences, which in Pascal are referred to as *statements*. The major part of a Pascal program consists of a number of statements arranged in sequential order. This corresponds to a set of instructions to be followed in that same order. In addition to these statements, which are explicit instructions to the computer to perform particular tasks, the program must also include other information that the computer needs in order to carry out these tasks successfully. The situation is analogous to that of a truck driver ordered to drive a particular truck from Chicago to New York. In addition to explicit instructions on how to get to New York, the trucker might also like to know the truck's weight so that, for example, he or she can avoid state weighing stations if the truck is overweight.

2-2 STRUCTURE OF SIMPLE PROGRAMS

The first information that needs to be given to the compiler in a Pascal program is the program's *name*. Every program must have a name to be used

for identification purposes. A name is made up of characters. The characters may be either letters (A, B,..., Z) or digits (0, 1, 2,..., 9), but the first character must be a letter. Thus,

REPORT TAX5 Q74J AAA

are all possible names for programs, while

7RATE AB* JOHN DOE

are illegal names for programs. 7RATE is illegal because it begins with a digit. AB* is illegal because it contains an *, which is a character that is neither a letter nor a digit. JOHN DOE is illegal because it contains a blank, which is neither a letter nor a digit but *is* a character. You can make up names that are as long as is necessary to identify your program descriptively, for example, TAXCOMPUTATION or INVESTMENTUPDATE. But you should be warned that some compilers will only look at the first eight characters of a name, and they will confuse two programs named, for example, TAXCOMPUTATIONFED and TAXCOMPUTATIONSTATE.

There is one more restriction—a name cannot be a *reserved word*. The reserved words are those that have special roles to play in the language, so that it would confuse the compiler if you used them for other purposes. For handy reference we have listed all of the Pascal reserved words in Appendix A.

Every Pascal program begins with a heading that gives its name, e.g.,

```
PROGRAM PAYROLL;
```

Note that the heading begins with the word PROGRAM. PROGRAM is a reserved word indicating that this is the beginning of a program and that the program's name will follow. In this case the program's name is PAYROLL. (Since the word PROGRAM is reserved, PROGRAM PROGRAM is illegal.) The heading is then followed by a program *block*, which, in the case of very simple programs, consists of the list of instructions to be carried out by the computer. This list of instructions, or statements, is prefaced by the word BEGIN and followed by the word END. Thus a simple program looks like this:

```
PROGRAM SIMPLE;
BEGIN

  (list of statements)

END.
```

Notice that a period follows the END to indicate the end of the program. (It would appear at first sight that END itself should play this role. But you will see later that ENDs can also appear in the middle of a program.) The other punctuation appearing here is the semicolon, used to separate the heading

此为试读,需要完整PDF请访问: www.ertongbook.com