Speaking Pascal

A Computer Language Primer

Kenneth A. Bowen

HAYDEN

SPEAKING PASCAL

Kenneth A. Bowen
Syracuse University

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

*For Alexandra
and Melissa*

Library of Congress Cataloging in Publication Data

```
Bowen, Kenneth A.
   Speaking Pascal.

   Bibliography: p.
   Includes index.
   1. PASCAL (Computer program language)  I. Title.
QA76.73.P2B66       001.64'24       81-5390
ISBN 0-8104-5164-6                  AACR2
```

Printed in the United States of America

2 3 4 5 6 7 8 9 PRINTING

81 82 83 84 85 86 87 88 89 YEAR

SPEAKING PASCAL

Preface

Learning to program should be fun! This book makes learning to program in the language Pascal a pleasant experience. It is designed for beginners with no previous programming experience. The pace and approach are slow and careful, yet light-hearted and enjoyable. There are many carefully chosen examples which are used to motivate and illustrate the important ideas. These examples are chosen to reflect concerns from everyday life, such as diagnosing and treating winter colds, or choosing breakfast from a menu. Thus they are immediately meaningful and interesting to everyone. In this way, learning the fundamental ideas of Pascal becomes a pleasant task.

While presenting the elements of the particular programming language Pascal, I have tried to convey an understanding and appreciation of the top-down, structured approach to program design and construction. All of the principal examples are carefully developed with these methods. Experience has shown this approach to be the most reliable method of producing correct and understandable programs.

The version of Pascal presented is extremely close to the international standard for Pascal, and thus is widely available on many computers. The few deviations from the proposed standard are noted in the text. The version presented covers all the important constructs used in everyday programming. These are widely available on every computer which supports Pascal at all, and include procedures and functions, arrays, records, enumeration types, iteration and control, and input-output. A few constructs which are not widely used in everyday programming have been omitted.

This book has grown out of my experiences teaching Pascal and other programming languages at Syracuse University. To my colleagues at the School of Computer and Information Science I am indebted for

many fruitful conversations over the years, and in particular to Lockwood Morris, with whom I first learned Pascal as we jointly taught the first course in this language at Syracuse.

To Henry Ledgard, I owe a very special thanks. As my editor, he has been both a wonderful source of ideas and a vigilant critic. In particular, he suggested that I transfer the top-down approach to the design and execution of the book itself, and was a great aid in actually accomplishing this. The contribution of Randall Bond, fine arts librarian at Syracuse, is immediately evident: he led me to the sources of the lovely illustrations in the text. Betsy Clarke helped with both the typing and final production – to her, many thanks.

And last, but far from least, I owe tremendous gratitude to my wife and daughters for their support and forbearance during the writing of this book.

Kenneth A. Bowen
Syracuse, New York
April, 1981

Contents

Preface v

First Words

Chapter 1 *Talking to Computers: Introduction* 1

1.1 A Program to Read Aloud 2
1.2 Your Conversational Partner 5
1.3 Behind the Conversation 8

Simple Conversations

Chapter 2 *Money Talk: Integer Expressions* 11

2.1 Striking Up a Conversation 12
2.2 Your Response 14
2.3 Program Structure 21
2.4 Comments and Documentation 31

Chapter 3 *But There's Much More to Talk About: Real Numbers, Characters, and Other Things* 34

3.1 Taking the World's Measure 35
3.2 Messages and Codes 42
3.3 Other Things 45

Controlling the Conversation

Chapter 4 *Repetitive Conversations: Iteration* 52

4.1 Limited Repetition 53
4.2 Going on Forever 59

Chapter 5 *Consultation and Diagnosis: Conditionals* 67

5.1 Decisions, Decisions 68
5.2 Multiple Alternatives 75

Chapter 6 *Recording Conversations: I-O and Files* 81

6.1 Recording Devices 82
6.2 Organizing Your Files 83
6.3 Formatted I-O 94

Sophisticated Conversations

Chapter 7 *Table Talk: Arrays* 98

7.1 Listing Your Points 99
7.2 Speaking Literally 107
7.3 Expanding Your Horizons 117

Chapter 8 *Steering the Conversation: Procedures and Functions* 122

8.1 Navigating the Shoals of Complexity 123
8.2 Packaged Conversations 130
8.3 Telling Them What They Need to Know 145

Chapter 9 *Talented Talk: Records* 158

9.1 Picturing the World 159
9.2 Filing the Pictures 162

Last Words

Chapter 10 *Elegant Discourse: Program Design Methodology* ... 177

10.1 Top Down Design 178
10.2 Pretty Comments 179

Appendix A Pascal at a Glance 216
Appendix B Reserved Words for Pascal 221
Appendix C Intrinsic Functions and Procedures 222
Appendix D Character Code Tables 227
Appendix E UCSD Pascal 230

Index 234

Chapter One

Talking to Computers: Introduction

In the 35 odd years of their existence, electronic computers have undergone a remarkable growth and development that still continues today. The early legendary machines such as the ENIAC, the IAS computer, and the UNIVAC I were behemoth arrays of thousands of vacuum tubes that occupied large rooms, constantly broke down, and cost hundreds of thousands of dollars. Today far more powerful machines built out of miniaturized transistors occupy no more space than a typewriter, run reliably for years, and can be bought for less than $1,000 at retail stores throughout the United States. Despite this dramatic development, these modern machines are conceptually quite similar to their early counterparts. Both are sophisticated machines for manipulating electronic representations of data at incredibly high speeds. And both must be told explicitly what to do in these manipulations.

Popular myths to the contrary, computers themselves are not at all intelligent. They can no more think through solutions of problems on their own than a semi-trailer truck is capable of driving itself from New York to Los Angeles. In both cases, intelligent human guidance is needed. In the case of the semi-trailer truck, the human operator is always present, directing the actions of the truck at each moment. In the case of the computer, the human operator generally prepares a set of instructions (a program) for the machine in advance, and the computer follows these instructions as it goes about the manipulations to compute a solution to the problem at hand. In each case the intelligence is human; the expenditure of effort is mechanical or electronic.

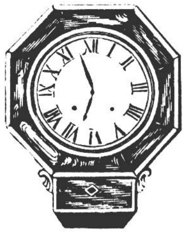
The actual instructions that a computer is prepared to accept and execute are extremely primitive. Consequently the preparation of programs using these actual machine instructions is not only tedious and difficult, but also highly error-prone. This difficulty has led to the development of methods for controlling computers which use instructions that are more suitable for human use. These methods are generally termed *higher-order lanaguages* for computer programming.

The number and diversity of these languages is bewildering. A fragmentary list of some of the better-known languages includes ALGOL, APL, BASIC, COBOL, FORTRAN, JOVIAL, LISP, LOGO, Pascal, PL/I, PROLOG, SAIL, and SNOBOL. These various programming languages have been designed with differing goals. Some are intended to be general-purpose, while others are intended to be more suitable for business data processing, scientific numerical calculation, research in artificial intelligence, text processing, and so forth. In this book we will study the programming language Pascal. This is a modern, general-purpose programming language. Its design has benefited from experiences with earlier programming languages and is oriented towards so-called *structured programming* methods. At the present time Pascal is regarded as a paradigm among programming languages. As such, it has become widely available on most computers.

1.1 A Program to Read Aloud

One of the intents of the designer of Pascal (Nicklaus Wirth) was to produce a computer language that was easy to learn. He succeeded at this, and one of the consequences of his success is that Pascal makes possible the writing of programs that are clear and easy to understand. To illustrate his success in doing this, consider Program 1-1. Its purpose is to set up a small dialogue between the computer and the user at the terminal. The computer will ask the user for the time of

```
program TIMEOFDAY(INPUT,OUTPUT);
(* ------------------------------------------------------------ *
 *      This program accepts as input a so-called military      *
 *  time specification, such as 0745 or 2130, and outputs the   *
 *  corresponding time in the usual am/pm format.               *
 * ------------------------------------------------------------ *)

   var
      MILTIME, HOURS, MINS: INTEGER;

begin

   WRITE('Please type the time in military format: ');
   READ(MILTIME);

   HOURS := MILTIME div 100;
   MINS  := MILTIME mod 100;

   WRITE('The time at the tone is: ');

   if (HOURS = 0) and (MINS = 0) then
     WRITE(' MIDNIGHT')
   else if HOURS < 1 then
     WRITE('12:, MINS, 'a.m.')
   else if HOURS < 12 then
     WRITE(HOURS, ':', MINS, 'a.m.')
   else if (HOURS = 12) and (MINS = 0) then
     WRITE(' NOON')
   else if (HOURS = 12) and (MINS > 0) then
     WRITE(HOURS, ':', MINS, 'p.m.')
   else
     WRITE((HOURS - 12), ':', MINS, 'p.m.');

end.
```



Program 1-1 A Simple Program to Compute the Time of Day.

day expressed in the 24-hour or "military" format. The user supplies it, and the computer responds with the time expressed in the common a.m./p.m. format.

Though we have not yet studied the details of Pascal, it is possible to read this program (aloud) and follow its operation.

The first line simply identifies the start of the program. The next five lines, from the opening (* to the closing *) are a comment, which has no effect on the machine but simply describes the action of the program for the benefit of human readers. Then in lines 8 and 9 there occurs a *variable declaration*. This simply asserts that we intend to use the words MILTIME, HOURS, MINS as names of "containers" or variables for holding integers. The action part of the program starts in line 11. (Line numbers are present in this program for reference only. Normally Pascal programs do not contain line numbers.)

The first action is to write out the following message on the terminal:

```
    Please type the time in military format
```

The next action (in line 14) is to read in the integer typed on the terminal by the user and store it in the variable or container called MILTIME. Then the hours in this military time are obtained by dividing the value of MILTIME by 100 and obtaining the integer quotient. The result is stored in the variable HOURS. The remaining minutes in MILTIME are obtained (in line 17) as the remainder when the value in MILTIME is divided by 100. This result is stored in the variable MINS. Next the message

```
    The time at the tone is
```

is printed out on the terminal (and no new line is started).

Finally, beginning at line 21, the program must make a decision as to what to print out for the time. It proceeds as follows:

1. If both HOURS and MINS contain value 0, the time must be midnight, and so it prints this.
2. However, if either the value in HOURS is not 0 or the value in MINS is not 0, but the value in HOURS is less than 12, it must be morning; and so the program prints out the value in HOURS followed by a colon (:), in turn followed by the value in MINS, and finally the expression `'a.m.'`.
3. Now, if the value in HOURS is precisely 12 and the value in MINS is 0, it must be noon and the program prints this.
4. But if the value in HOURS is still 12 and the value in MINS is not 0, it must be between noon and 1 p.m.; and so the

program prints out 12 followed by a colon followed by the value in MINS, which is finally followed by the expression `'p.m.'`.

5. In the final case, the value in HOURS must be greater than 12. Therefore, we must correct it by subtracting 12 from it. The program does this as it prints out the time.

Notice how much English it took to explicate the simple Pascal text for this decision. Yet the English is no clearer! We will find this quite often to be true: if we know precisely what we wish to do, it is often easier to say it in Pascal than in English.

1.2 Your Conversational Partner

Just as it is possible to operate a car or truck with no knowledge of its inner mechanical workings, it is possible to program and operate computers with no knowledge of their inner electronic workings. But a vehicle can be driven more effectively and efficiently if one possesses some knowledge of its inner design. Moreover, this knowledge becomes even more valuable in dealing with the inevitable minor malfunctions and difficulties that can arise during a trip.

The same is true for computers. The person who has some knowledge of the inner construction of computers can design more efficient and effective programs, as well as deal more effectively with the problems ("bugs") which seem to inevitably arise in programming. This knowledge need not be at the level of the engineering and physical details of the machine's function, but rather at the logical or conceptual level.

Viewed quite simplistically, a computer can be seen as a device that accepts certain data as input and gives forth other data as output, as indicated in Figure 1-1.

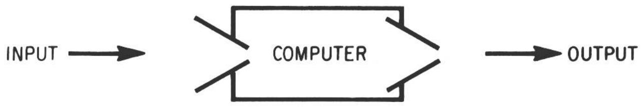


Figure 1-1. The Computer as a Black Box.

However, to use the machine effectively, we will need to look inside it a bit. If we begin breaking down the logical structure of a computer, we discover that it has the following requirements:

- There must be devices for inputting both data and instructions to the machine and for outputting the results of its computations. These are collectively known as input-output devices, or I-O for short.
- There must be a *central processor* that interprets the instructions and carries out the arithmetic and logical computations.
- There must be a *memory* in which both data and instructions can be stored and retrieved by the central processor.

Our diagram of the computer might now appear as in Figure 1-2.

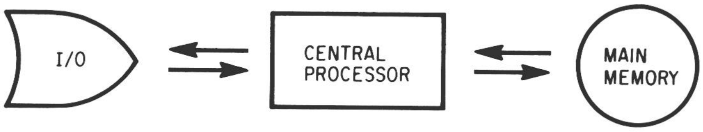


Figure 1-2. Elementary Computer Structure.

To fully grasp some of what follows, we will need to break this diagram down a bit further. First we note that there exist a variety of input-output devices, some of which can do both input and output, and others that are strictly limited to one function or the other. Both the teletype (which for our purposes includes Decwriters, IBM 2741 terminals, and so forth) and the graphics display terminal, as well as punched paper tape, can do both input and output. On the other hand, the card reader can only input data, while the card punch, high-speed line printer, and plotter can only output data.

Computers also have several types of memory. The fastest memory, in the sense that data can be entered into it and recopied from it faster than in any other type, is known as *primary storage* memory. This is the memory unit used directly by the central processor. (It is sometimes called *core* memory, reflecting the fact that in many of the early large-scale machines this memory was made up of many small magnetic iron "cores.") Unfortunately, primary memory units are very expensive and tend to be bulky relative to the amount of information they can store. Hence large-scale machines make use of several other slower, but more economical and compact, memory units. The first of these is the *magnetic disk*, which is the fastest memory after primary memory. These disks resemble large phonograph records that have been coated on both sides with a brown magnetic substance similar to that used on ordinary magnetic tape. Next in order of memory speed comes magnetic tape itself. And last of all is the ubiquitous punched card. Recently, magnetic tape in the form of cassettes has also come into use, especially with smaller computer systems.

Our logical diagram of a large computer might now appear as in Figure 1-3.

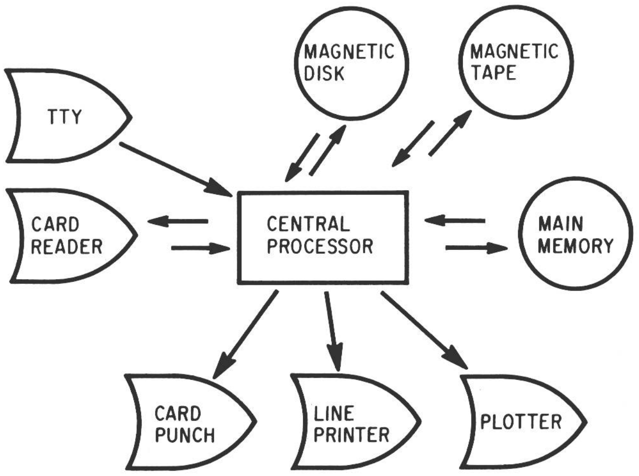


Figure 1-3. Logical Structure of a Computer.

The input-output devices indicated simply form the basic complement for a general-purpose installation. In special-purpose installations, sensors and control motors are added. By means of these additions, computers can be used to monitor and control industrial and manufacturing processes, direct environmental control in buildings and homes, and run games and toys.

As we indicated, in such a computer we store not only numbers and other data in the memory but also the instructions for a computation. Thus the machine is able to obtain and read its instructions just as fast as it can obtain its data. The question obviously arises as to what form we can imagine the data and instructions to take when they are stored in memory. Are they represented in a form resembling our ordinary ways of writing or in some other manner? Unfortunately for the beginning student, the form in which they appear in memory is quite different from our ordinary representations. In fact, one can say that all of the items – numbers, other data, and instructions – are represented in a language whose only symbols are the digits 0 and 1. This apparently bizarre state of affairs is caused by engineering limitations in the construction of computers. Needless to say, programming in "machine language" is tedious in the extreme.

The central processing unit has capabilities for manipulating the words of this machine language. Among some of its capabilities are the following:

- Comparing two words to see if they are equal.
- Taking two words that represent numbers and adding, subtracting, multiplying, or dividing them to produce the word representing the sum, difference, product, or quotient, respectively.
- Taking a word representing a letter or digit and causing the letter or digit to be output, say on a teletype.
- Sensing that a key has been typed on a teletype, and obtaining the word representing the character on the key.

These capabilities are really quite primitive. The power of computers arises from their ability to perform such operations over and over at incredibly high speeds (thousands or millions of operations per second). High-level languages such as Pascal provide the capability to control such runaway speed and power without the tedium and difficulty of machine-language programming.

1.3 Behind the Conversation

When the statements of a higher level programming language such as Pascal are typed in or read in from the punched card reader, they are automatically coded into a form that the machine can manipulate.

Though the higher level statements are in a binary code, the coded form usually is not in the form of a machine-language instruction; often, from the computer's point of view, the coded form is just gibberish. So the second stage of the translation procedure involves the translation of these coded statements into instructions intelligible by the machine. This second stage translation is not physically built into the machine, but is accomplished by a complex program called an interpreter or compiler. Each higher level programming language has its own interpreter, compiler, or both.

The difference between an interpreter and a compiler lies in their approach to the translation of the statements of the higher order language. Roughly, an interpreter reads each individual statement or command of the program and immediately translates and executes it, while a compiler translates the entire program before actual execution begins. There are advantages and disadvantages to each approach. In general, once a compiler has finished the total translation of a program, the actual execution speed of this *compiled* program is faster than that of the same original (*source*) program executed by an interpreter. Moreover, the amount of memory occupied by the compiled program is much less than the combined space occupied by the source program together with the interpreter program. Since main memory is a scarce and expensive commodity, this is a serious consideration. On the other hand, compilers as programs themselves are usually much larger and more complex than interpreter programs; they can be so big as to be impossible to fit in the memory of a microcomputer. Moreover, when the source program contains errors (*bugs*), it is often much easier to diagnose these errors with the assistance of an interpreter than with a compiler.

You will recall that an interpreter or a compiler for a higher level language is itself just a computer program. This program, which *is* the interpreter or compiler, is called an *implementation* of the higher level language. Since it is usually a program for a particular kind of computer, it is called an implementation of the higher level language *on* that computer. For example, one speaks of implementations of Pascal on IBM 370 computers, on CDC 6600 computers, and so forth.

Some languages (such as LISP) are implemented using both compilers and interpreters. Some (such as BASIC and LOGO) primarily use interpreters, and others (including Pascal) primarily use compilers. Pascal has been implemented on most mainframe and minicomputers available today, as well as on a wide range of microcomputers. All of these implementations differ in one degree or another. The fine details of these differences are usually not of concern and can be determined by consulting the appropriate computer's manuals.