Pascal Programming

Laurence V. Atkinson
Department of Computer Science, University of Sheffield

A Wiley–Interscience Publication

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto

***British Library Cataloguing in Publication Data*:**

Atkinson, Laurence V.
Pascal programming.—(Wiley series in computing).
1. PASCAL (Computer program language)
I. Title
001.6424 QA76.73.P2 80–40126
ISBN 0 471 27773 8
ISBN 0 471 27774 6 Pbk

Typeset in Northern Ireland at The Universities Press (Belfast) Ltd. and printed by Page Bros. (Norwich) Ltd., Norwich.

Pascal Programming

Introduction

This book is intended both as an introductory text for those with no previous knowledge of programming and as a conversion text for those with experience of some high level programming language other than Pascal. The book's dual role has influenced its layout.

Some introductory remarks are presented for the benefit of those with no knowledge of computers.

Part A (Chapters 1–8) introduces some programming concepts fundamental to all high level sequential programming languages. These are presented here in the context of Pascal but most features will have equivalents in other languages. The lessons of good programming style are relevant whatever programming language is used. This is stressed in the discussion of loops (Chapter 6). Loops abound in most programs but are a noted source of trouble for beginners. The methodology of loop construction is given greater emphasis than in most programming texts.

Experienced programmers should be able to read these chapters quickly and acquaint themselves with the Pascal interpretation of familiar concepts. Novices should acquire a solid foundation for succeeding chapters to build upon.

Part B (Chapter 9) presents two features of Pascal which have no counterpart in any other programming language (except those developed from Pascal). These are the two most important new concepts in Pascal. They have had a far-reaching effect upon the way we think about problems, the way we express algorithms, the confidence we can place in our programs, and the speed with which programs run on the computer. Coverage of these two features, together with a discussion of their relevance to program security, constitutes a complete *Part* to emphasize their importance. The lessons to be learned from this chapter pervade the remainder of the book.

Part C (Chapters 10–14) describes the data structuring facilities available in Pascal.

Part D (Chapter 15) embodies the shortest chapter of the book. It introduces a Pascal statement you should rarely have to use.

Each chapter concludes with a set of programming exercises. Within each set these tend to be in increasing order of difficulty. If you do not have time to complete all the excercises you should attempt at least those marked with an asterisk. These form a representative sample. Because many examples appear in the book no solutions to exercises are given, but notes and hints for some exercises (marked with a dagger) are presented in Appendix F.

The book is essentially pedagogical and so should present information in the order in which it can best be assimilated by the beginner; however, it must also serve as a reference text. Inevitably, therefore, some compromises have been made. The basic ordering of the topics is sensible from a teaching (and, hence, learning) viewpoint but the depth of coverage is often greater than the novice needs during a first reading. When teaching programming, I introduce enumerated and subrange types (Chapter 9) after a shallow coverage of the first six chapters but treat recursion and backtrack programming (Chapter 7) as the last topic, while top-down design (Chapter 8) influences the whole presentation.

Programming is best learned by practice. To this end complete programs are introduced in Chapter 1 and throughout the book the emphasis is on illustration by example. The examples are taken from a variety of application areas. Few examples require more than an elementary knowledge of mathematics.

The computing world has its own jargon, familiar English words often being used but with new meanings. Each new term introduced will be enclosed within "quotation marks". Pascal text will appear in **bold** or *italicized* type.

Computers and Computing: Some Introductory Remarks

A computer is a machine. Our aim is to make the machine perform some specified actions. With some machines we might express our intentions by depressing keys, pushing buttons, rotating knobs, etc. For a computer we construct a sequence of instructions (this is a "program") and present this sequence to the machine. If we have a "terminal" (rather like an electric typewriter, possibly with a display screen in place of paper) connected to the computer we can type our program directly. An alternative method is to type the information onto a "punched card", a rectangular piece of high quality card (typically 19 cm by 8 cm). A "card punch" is a machine capable of punching patterns of holes in any of 80 columns across the card. In any column of the card is a pattern of holes and no-holes which can be sensed by a "card reader". A light shines onto one side of the card and the positions of the holes are determined by a bank of photoelectric cells on the other. Thus the information we type is recorded in a sort of Morse code. Many such cards can be collected together and fed to the computer as a "deck". There are other ways of getting information into a computer but you are unlikely to meet them in the context of an introductory programming course.

When the computer has done something, hopefully what we wanted it to, we require it to produce some "output". If we are communicating with the machine via a terminal we would expect the output to appear at the terminal. If we are communicating via any other medium, output will probably be produced by a "line printer". This machine handles continuous stationery folded into pages (typically 36 cm wide and 28 cm high) and is so called because it produces its print in units of a line (unlike a typewriter which produces its type one character at a time).

The particular system available to you will depend upon the "operational environment" of your computer installation. Pascal programs as presented in this book are independent of the operational environment but the conventions of your installation will dictate the way programs are supplied to the computer.

When specifying our intentions to the computer we must be precise. The computer has no intelligence but merely responds in a certain well-defined way to each correctly supplied instruction. Natural English is prone to ambiguity and so is not a suitable language in which to express our instructions. Accordingly languages have been developed for the specific purpose of communicating with computers. These languages are called "programming languages".

Every computer has a basic set of instructions which it can obey directly but writing programs in this "machine code" is very tedious and error prone. Accordingly "high level languages" have been developed to make life easier for us. Some are designed for use in particular problem areas but some are called "general purpose languages". Two of the earliest languages developed (in the late 1950s) were *Fortran* for numerical computing and *Cobol* for commercial applications. In 1960 the language *Algol 60* appeared and although intended for scientific use it paved the way for the general purpose languages which followed it during the 1960s. Prominent among these are *PL/I, Simula 67,* and *Algol 68.*

In 1971 we have the appearance of *Pascal* designed by Professor Niklaus Wirth of Eidgenossische Technische Hochschule in Zurich. Pascal was intended as a vehicle for teaching programming systematically and to provide means of efficient implementation on existing computers. In the eyes of many it is superior to any other programming language in general use today and its spread throughout the computing world has been extensive. The language, as described in this book, conforms to the BSI/ISO draft Standard for Pascal.

Contents

APPENDIXES

Part A

SOME FUNDAMENTAL CONCEPTS

1

Programs and Data

1.1 Programs

1.1.1 Algorithms, programs, and sequential flow

An "algorithm" is a description of a process expressed as a sequence of steps. For example, the following constitutes an algorithm:

Step 1. Mix eggs, flour, sugar, margarine, and butter.
Step 2. Place mixture in a cake tin.
Step 3. Bake in a moderate oven for 30 minutes.

In computing we are concerned with presenting algorithms to computers. We must therefore adopt some notation in which to express our algorithms. This notation is called the "programming language". The actions to be carried out for each step of an algorithm are described by what programming languages call "statements". A complete sequence of statements is called a "program" and when a computer "runs" a program it "obeys" each statement in turn.

1.1.2 Compilation and execution

The computer does not obey each statement as it is presented; instead it attempts to translate each statement into machine code. This translation process (called "compilation") is carried out by a program written specially for the purpose. It is possible to define an algorithm for translating from one specified computer language (the "source language") into another (the "target language"). Hence it is possible to write a program to perform the translation. Such a program is called a "compiler".

A "Pascal compiler" for a particular computer is a program which translates Pascal programs into the machine code of that computer. Pascal programs can be run on any computer with a Pascal compiler. The compiler will usually "list" (i.e. print out) the program, numbering the lines consecutively. Throughout this book programs are often presented in this way: the line numbers are not supplied by the programmer.

A program may fail to compile. This happens if the program is incomplete or contains some misuse of the source language. The rules of grammar of a language are called the "syntax" of the language and, in programming terms, a violation of these constitutes a "syntax error" or "syntactic error". Thus the English sentence

Programming (with Pascal is, fun.

contains two syntax errors:

1. A comma should not appear between *is* and *fun.*
2. Either the opening bracket should not be present or a closing bracket should follow *Pascal.*

Notice, as in the second case above, that there may be several ways to correct a syntax error. Thus a compiler may detect the *presence* of an error but be unable to inform you as to the precise *location* or *nature* of the error.

A sentence, though syntactically correct, may be meaningless. Consider an example. A simple English sentence may have the form:

Noun phrase – *verb* – *noun phrase.*

Both the following sentences are therefore syntactically correct:

1. *A girl recites a poem.*
2. *A poem recites a girl.*

The meaning of the second is somewhat obscure! The meanings of programs are described as "semantics". More will be said later about syntactic and semantic errors.

If the compiler detects the presence of syntactic or semantic errors the computer will make no attempt to run the program. No statement will be obeyed until the complete program has been successfully compiled. The program must be corrected and recompiled. When a complete program has been successfully compiled "execution" may be attempted (i.e. the program can be run). Most computer systems can be arranged to run a program automatically upon successful compilation. Your particular installation will have details of this facility.

Errors may occur during execution. For instance, one may (presumably unintentionally) ask a program to print out the letter following *Z* in the alphabet. You will see later why this would not usually be detected until run-time. If you are running your program from a terminal you may be allowed to correct some run-time errors as they occur and continue execution. If you are not using a terminal, program execution will be terminated in the event of a run-time error. The program must then be corrected, recompiled, and (when successfully compiled) run again.

1.1.3 Program output

All your programs should generate output and, as mentioned earlier, it will probably be produced by a terminal or line printer. It is possible that no lower case letters will be available on these devices, in which case all letters printed will be upper case (i.e. capitals).

In Pascal, output is generated by the

```
write
```

and

```
writeln
```

procedures. A "procedure" is a set of instructions which performs some specified task.

"Parameters", enclosed within parentheses and separated by commas, may be supplied to indicate any entities needed for the task. For *write* and *writeln* the parameters specify what is to be printed. The procedures are supplied by the system; the parameters must be supplied by the user.

Output is discussed in more detail at the end of this chapter; for the present an introduction to *writeln* will suffice. No parameters need be supplied. When the statement

```
writeln
```

is obeyed at run-time the print head of the output device will position itself at the start of the next line. If parameters are supplied, the "value" of each in turn is printed along the line from left to right; the print head then returns to the beginning of the next line. In the simplest case the parameters might be numbers or strings of one or more characters. In Pascal, "strings" are enclosed within "quotes", (').

Examples of writeln-statements are as follows:

```
1. writeln (2001)
2. writeln (-9)
3. writeln ('I think therefore I am')
4. writeln
5. writeln ('A prime number:', 17, 'two more:', 5, 7)
```

If the five writeln-statements above were obeyed one after another the output would be:

```
2001
-9
I think therefore I am

A prime number:        17two more:        5      7
```

Integers (whole numbers) are output to a predefined "field width", but this varies from one compiler to another. The number of spaces preceding each integer printed will therefore be dependent upon your particular implementation. No extra spaces are output with strings but, as in examples 3 and 5, spaces may be included within a string.

Numbers with fractional parts are called "real numbers" or simply "reals" and are printed in "standard floating point form". This comprises a "coefficient" and a "scale factor" separated by the letter E. The form $\alpha E\beta$ corresponds to the value $\alpha \times 10^{\beta}$. The coefficient is a number with absolute magnitude greater than or equal to unity but less than ten, expressed to some predefined, but implementation dependent, number of decimal places. The scale factor is a signed integer. Here are some real numbers and, assuming coefficients contain six decimal places, their standard floating point equivalents:

Real number	S.f.p. form
14.2	*1.420000E+01*
−613.94173258	*−6.139417E+02*
3.14159265358979323846	*3.141593E+00*
0.00000123456789	*1.234568E−06*
97.0	*9.700000E+01*

It is possible to output the values of "arithmetic expressions". The statement

```
writeln (4, '+', 19, '=', 4+19)
```

produces the output

```
4+          19=          23
```

Note that digits may be interpreted as characters as well as constituting integers:

```
writeln ('4 * 19=', 4 * 19)
```

would produce

```
4 * 19=          76
```

When *4 * 19* appears between quotes it is just a character string ("four", "asterisk", "one", "nine"); when it appears without quotes it means "4 times 19".

Arithmetic expressions are fully explained in Chapter 2. It is also possible to output the value of a "boolean expression"; these are described in Chapter 5.