PASCAL at Work and Play

An Introduction to Computer Programming in Pascal

RICHARD S. FORSYTH

Pascal at Work and Play

AN INTRODUCTION TO COMPUTER PROGRAMMING IN PASCAL

RICHARD S. FORSYTH

Polytechnic of North London

LONDON NEW YORK

CHAPMAN AND HALL

First published 1982 by
Chapman and Hall Ltd
11 New Fetter Lane, London EC4P 4EE
Published in the USA by
Chapman and Hall
733 Third Avenue, New York NY 10017

Printed in Great Britain
at the University Press, Cambridge

ISBN 0 412 23370 3 (cased)
ISBN 0 412 23380 0 (Science Paperback)

Library of Congress Cataloging in Publication Data

Forsyth, Richard S.
Pascal at work and play.

Bibliography: p.
Includes index.
1. PASCAL (Computer program language)
I. Title.
QA76.73.P2F67 1982 001.64'24 82–4515
ISBN 0–412–23370–3 AACR2
ISBN 0–412–23380–0 (Science paperbacks: pbk.)

British Library Cataloguing in Publication Data

Forsyth, Richard S.
Pascal at work and play.
1. PASCAL (Computer program language)
I. Title
001.64'24 QA76.73.P2

ISBN 0–412–23370–3
ISBN 0–412–23380–0 Pbk

Pascal
at Work and Play

Preface

This is both a first and a second level course in Pascal. It starts at an elementary level and works up to a point where problems of realistic complexity can be tackled. It is aimed at two audiences: on the one hand the computer professional who has a good knowledge of Cobol or Fortran but needs convincing that Pascal is worth learning, and on the other hand the amateur computer enthusiast who may have a smattering of Basic or may be an absolute beginner.

Its approach is based on two principles that are not always widely recognized.

The first is that computing is no longer a specialist subject. In the early days of computing a priesthood arose whose function was to minister to those awesome, and awesomely expensive, machines. Just as in the ancient world, when illiteracy was rife, the scribes formed a priestly caste with special status, so the programmers of yesteryear were regarded with reverence. But times are changing: mass computer-literacy is on its way. We find already that when a computer enters a classroom it is not long before the pupils are explaining the finer points of its use to their teacher – for children seem to have greater programming aptitude than adults. This book, it is hoped, is part of that process of education by which the computer is brought down to earth; and therefore it attempts to divest computing of the mystique (and deliberate mystification) that still tends to surround the subject.

The other principle is that the second best way to achieve competence as a programmer is to read non-trivial programs and see how they work. (The best way is of course to write programs, and plenty of them.) So a large proportion of this book is taken up by full descriptions and listings of four good-sized programs that are far removed from the toy examples sometimes shown in programming textbooks. This aspect, based on the fact that people learn by example, is too often neglected by authors on programming, who tend to feel they have done their duty once they have presented the rules of the language.

Pascal has been chosen because it is elegant enough to appeal to computer scientists and professional programmers while still being simple enough to be taught to almost anyone who is genuinely interested in programming. It is also compact enough to run on many microcomputers – those Volkswagens of the computer age. Indeed there are signs that Pascal will become the lingua franca of the coming 'computer revolution' in schools, businesses and the home. This book is for anyone who wants to join that revolution.

October 1981 Richard Forsyth

Acknowledgements

Various people have contributed in various ways to the making of this book. My debt to other published authors is acknowledged in the Bibliography. Here I would just like to thank the following people for their advice and assistance. (The order is purely alphabetic.)

Maureen Ashman
Ashesh Datta
Cyril Drimer
Mei Lynn Forsyth
Tim Harris
Sallahe Hassan
Jeff Hillmore
Declan Kelly
Joan New
Maria Panayi
Therese Rieul
Bill Tuck

I am also grateful to the Polytechnic of North London Computer Service, without whose co-operation this book could not have been written.

Contents

PART ONE

Pascal at Large

'The limits of my language mean the limits of my world.'
Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*

Introduction

On choosing a programming language

I have never actually seen anyone come to blows over the choice of a programming language, but it is an issue that can inspire heated argument. I have heard high-brow Lisp addicts snidely denigrating Algol 68 (not Algol 60, that was even beneath contempt) at an Artificial Intelligence symposium; while on another occasion I listened meekly to an experienced systems programmer pouring torrents of abuse on the benighted fools who persisted with 'infantile' and 'mind-polluting' languages like Fortran and Cobol long after the true way (Algol 68 again) had been revealed to the world. In the circumstances I thought it prudent to keep quiet about the fact that I used such a demotic language as Basic.

Indeed few subjects arouse such passions in the data-processing fraternity. If a letter appears one week in the correspondence columns of *Computer Weekly* or another journal of that ilk, championing, say, PL/I and describing Cobol, for example, in unfavourable terms, it is sure to provoke an immediate storm of protest from a legion of Cobol loyalists which will take months to subside. Others will join the fray on both sides and the controversy will probably not die down until the editor intervenes to halt it. Much the same applies to other languages, each of which has its devotees and detractors – some so zealous as to suggest quasi-religious fervour. Pascal is particularly prone to inspire missionary zeal.

Against this background I do not wish to pretend that I can offer a rational and unbiased evaluation of the merits of Pascal. Nevertheless I believe it is important to persuade you that it has some notable advantages that make it worth the effort of learning. The fact that people get emotional about the relative strengths and weaknesses of programming languages shows that the question is not a trivial one.

The importance of language in moulding thought has become known in linguistics as the 'Whorfian Hypothesis' after Benjamin Lee Whorf, a scholar who drew many of his conclusions from a study of the different modes of thought between American Indian languages and European ones. It is perhaps best expressed by Edward Sapir, Whorf's teacher and colleague (Carroll, 1956).

'Human beings do not live in the objective world alone, nor alone in the world of social activity as ordinarily understood, but are very much at the mercy of the

particular language which has become the medium of expression in their society. It is quite an illusion to imagine that one adjusts to reality essentially without the use of language and that language is merely an incidental means of solving specific problems of communication or reflection.'

Applied to computing, this implies that a programming language is not a passive instrument but an active collaborator. It means that when you write a Pascal program you are standing on the shoulders of Niklaus Wirth, the inventor of that language. And, quite clearly, the further you want to see, the taller should be the giant on whose shoulders you stand. For whether you notice it or not, the programming language you use lays down a style of approach which necessarily constrains its user by making some techniques easy and others difficult.

Plan of this book

This book is divided into four parts. The first part describes the rules of the Pascal language, and shows how its features may be put to use. It also outlines some precepts of good programming practice.

But merely learning the rules of Pascal is not the same as knowing how to write a Pascal program, so the 'meat' of the book comes in Parts 2 and 3. Part 2 concentrates on two medium-to-large programs, one to sort a file of data into order, the other to find the shortest path through a network. Both are serious problems with many real-life applications and both are explained in detail. These programs were written and run on a large mainframe computer (the DEC System-10) using a Pascal compiler from the University of Hamburg.

Computing is not all hard work, however. The microprocessor has liberated the computer from the fortified citadel of the commercial data-processing department and from the cloistered sanctuary of the big university installation; and now that computers have come to the people, people (especially children) find they are fun to play with. The personal computers of today already have facilities for colour graphics and sound output that make games more lively – the best example being 'Space Invaders' – and the personal computers of tomorrow will have all this and more.

Thus it is appropriate that the programs in Part 3, which is concerned with the more frivolous aspects of computing, were written and tested on the Research Machines 380Z, a microcomputer which runs the popular CP/M operating system, using Pascal/Z. The first program in Part 3 simulates a soccer game and the second plays Go-Moku, an ancient oriental game. Both are significant pieces of work and the reader will find complete program listings, printout from trial runs and descriptions of methods used to help him or her come to grips with them. Reading and understanding programs which are not just 'Mickey Mouse' examples is the fastest way to appreciate the power of a programming language.

The fact that Pascal can cope with such a variety of tasks is, in my view, one of the strongest arguments in its favour.

The ideal reader will first peruse this book, then use it. Programming is learnt by doing. To derive any benefit from the book, you will need to read it in conjunction with practical work on a computer. If you have no prospect of access to a computer system with Pascal, you may as well stop reading now.

The crunch will come at the end of Chapter 5. Up to that point it is all reading; from then on there are plenty of exercises to keep you on your toes. If you do not attempt the exercises (or some equivalent pieces of your own) you will gradually lose contact, so that by about half-way through you will not understand what you are reading. Even just typing in the example programs and getting them to work on your system is better than no programming at all.

(Typographical note: for clarity all computer output is shown in capitals throughout the book; user input and program listings appear in lower case.)

1
Writing programs

A computer is simply a machine for processing information. Since it cannot answer questions but can only obey orders it must be given a sequence of instructions in a language it can understand – a program – to make it do anything useful. Someone has to write that program; you, for instance.

There is not time here to give a potted history of computers from the abacus to the silicon chip (as some old-fashioned programming texts did) or to delve into the electronics of how they work (as some of the newer books do), but a little background information about compilers and high-level languages should be useful to the prospective Pascal programmer.

(If you do want to read about computer history and hardware two excellent books are *Using Computers* and *The Making of the Micro* (Meek and Fairthorne, 1977; Evans, 1981).)

1.1 High level languages

The central processing unit of a computer can only obey instructions that are encoded as groups of binary digits, termed 'bits', i.e. as sequences of zeroes and ones, such as 01110110. Such instructions are said to be in machine code. Two-state devices are easy to construct – for instance a transistorized switch may be on or off, a voltage may be high or low, a tiny piece of magnetized material may have polarity north–south or south–north, a current may be flowing or not – and so binary code is very convenient for computing machines. But it is extremely inconvenient for humans, and no one in their right mind writes programs in machine code these days.

The task of translating from a more symbolic notation to machine code is one that can be mechanized, and historically the first programs to carry out this task were 'assemblers'. The main advantage of the assembler is that its user can employ symbolic names composed of letters and digits instead of having to remember their binary equivalents. The assembler then converts from something like

```
add   a,100
```

to something like

```
11000110  01100100
```

(Zilog Z-80 example).

Assembly language is still restrictive in two respects however: firstly it is tied to the particular machine used, each computer ordinarily having its own machine code, which means that assembly language programs cannot normally be transferred from one machine to another; secondly the operations available are rather primitive since they correspond to things that the processor can carry out directly – things like comparing two characters or adding two numbers together. It is said to be a 'low level' language.

By contrast each instruction in a 'high level' language typically involves many machine-code instructions; and because the language is defined on paper (not built into a specific processor) it can, in theory, be common to a variety of different machines. A more complex translator program, known as a 'compiler', is required to implement a high level language. It is important to realize that a compiler is just another program: its distinctive feature is that its input is a program in one (high level) language and its output is a version of the same program in a different (low level) language (Fig. 1.1).

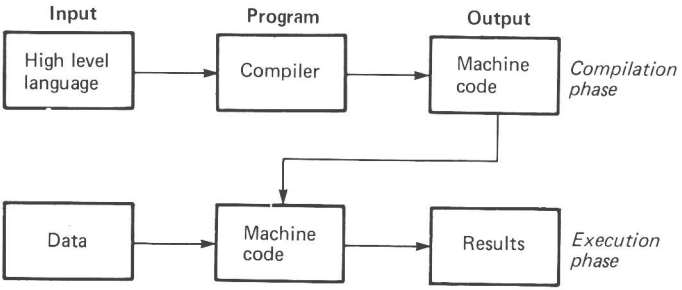

Figure 1.1 A compiler is a program

The first high level language to be widely used was Fortran, developed in 1956. In scientific circles Fortran is still widely used today. The next important high level language to arrive on the scene was Cobol, first defined in 1960. The US government made the provision of a Cobol compiler mandatory for computer suppliers competing for government contracts, and this backing ensured its success. It has been said that of all programs written by programmers who are paid for their work the number of lines in Cobol far exceeds those in all other languages added together (and not just because Cobol is rather verbose). In fact, most advertisements recruiting programmers in the computer trade press tend to include the magic formula 'two years Cobol experience required'.

In 1964 Basic hit the scene and the world has never been the same. It was