Cambridge Computer Science Texts · 12

Programming via Pascal

J. S. ROHL and H. J. BARRETT

Programming via Pascal

J.S. ROHL and H.J. BARRETT

Department of Computer Science
University of Western Australia

CAMBRIDGE UNIVERSITY PRESS
Cambridge
London New York New Rochelle
Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
32 East 57th Street, New York, NY 10022, USA
296 Beaconsfield Parade, Middle Park, Melbourne 3206, Australia

First published 1980

Printed in the United States of America

Library of Congress Cataloguing in publication data

Rohl, Jeffrey Soden

Programming via Pascal

Includes index

1. PASCAL (Computer program language) I. Barrett,
H.J., joint author. II. Title.
QA76.73.P2R63 001.6'424 79-17433
ISBN 0 521 22628 7 hard covers
ISBN 0 521 29583 1 paperback

PREFACE

What, another book on Pascal! What can you possibly add which has not been said already? These questions have been foremost in our minds for some time now. When we decided to start this project there were no acceptable textbooks. We were obliged to use Jensen & Wirth's *User Manual & Report* for a purpose for which it was not designed, and for which it is clearly inadequate. Since then there has been a steady stream of books each apparently better than its predecessor. So why then did we persevere?

Apart from the obvious democratic motive of providing as broad a selection of books as possible for the potential users, we felt that our approach offered certain advantages.

The book is designed not as a reference book to be used to back up a lecture course, but as a textbook around which a course might be constructed. With a few exceptions, each chapter covers the material appropriate to one 45 to 50 minute lecture. [The exceptions are generally of nearly double the length.] As appropriate, and especially in the early chapters, we follow the text with a few small exercises based on the material of the chapter. Thus the designer of a course has a number of options: he can give a lecture on the text; he can give a lecture on material complementary to the text; he can ask the students to pre-read a chapter and hold a tutorial rather than a lecture; and, of course, he can simply use it as background reading.

We hope that with this approach the book will be useful for students studying on their own, with the short chapters and the consequently increased frequency of exercises.

A picture may not be worth a thousand words, but it certainly helps in crystallising important concepts and relationships. We use them quite frequently: structure diagrams, flow diagrams, store layout pictures and syntax diagrams. The last two deserve special mention.

We believe that a Pascal program is best understood in terms of the Pascal machine on which it runs. Consequently, throughout the book we give illustrations of the machine (in particular its store) to underline new facilities as they are introduced.

The most obvious difference between this book and others lies in the syntax diagrams. We believe that the function of syntax diagrams in a textbook is to communicate as much information about the syntax (and implicitly the semantics) as possible. This is rather different from their function in compiler-writing which is to provide a skeleton around which a compiler can be constructed. These two functions are not mutually exclusive and it is possible to construct syntax diagrams almost anywhere in the spectrum. It is our view that the syntax diagrams of the *User Manual and Report* are at the compiler-writing end of the spectrum: we have constructed a set which, at the price of an increase in size, is much nearer the textbook end. We are not completely satisfied with them (especially in the context sensitive parts of the language) and any suggestions for improvements would be greatly welcomed. During the production stage a proposed standard for Pascal was published for comment. Two of its points have found their way into this book. Firstly, the classification of the types of variable has been altered to reflect more accurately the distinction between the real types and the ordinal types. This has no effect on the language, only on its description. Secondly, and this does change the language, when a procedure is used as a parameter its whole heading must be given. Since this proposal has not, and may not, be accepted, we have noted the point in the text where this facility is described.

Another obvious difference is the reduced emphasis in this book on the top-down design technique. There are many reasons for this: partly it is because of the way the Pascal concepts are introduced; partly it is because we believe that top-down design is a basic human technique and that the current obsession with it in programming is due to our embarrassment at our poor performance in the past; partly it is because we feel that it is a technique to be learned by practice rather than by reading about it. One example is perhaps too few, and in our courses we give a second, much larger example after procedures have been introduced. [The current one is an interpreter at the bit level of a computer designed in an associated course.] It seemed excessive to add the 50 or so pages required to describe the top-down design and testing of this program: far better for the lecturer to do it dynamically for a program which is relevant to his class.

It has long since been realised that programming is not a subject that can be adequately covered in a single course. A first course has to be supplemented by a second, and possibly a third course. With this in mind we have planned two books of which the present one is the first. This decision has given us flexibility in the way material is presented. We have decided that the second book will concentrate on algorithms and data structures. This book, then, concentrates on the construction of programs in which the underlying algorithms are fairly simple. The problems presented are essentially data processing ones using that phrase in its most general sense.

Even so we made the (not easy) decision to cover all of Pascal, including pointers and sets whose full potential can be tapped only in the context of advanced data structures. Thus the book is a complete primer for Pascal.

The manuscript of this book has been read by a number of our colleagues who have made many invaluable comments, which have caused us to make many important improvements. We are particularly grateful to Andrew Lister of the University of Queensland, Rodney Topor of Monash University, Jan Hext of the University of Sydney and John Pollard of the Australian Atomic Energy Commission.

Nevertheless writing a book calls up a great deal of inner resources so that, although feedback from our current students has been very useful, our real debt is to those in Manchester and Stafford who were our teachers, our colleagues and our students. Great, too, has been the contribution of Chris Ward and Lyn McGrath who typed the manuscript, and our families who supported us.

Perth, November 1978

J.S. Rohl
H.J. Barrett

CONTENTS

1

INTRODUCTION

The general-purpose computer is a very powerful device, its power deriving from the fact that it can be turned into a machine for performing any particular specific calculating task at will. This transformation is effected by providing the computer with a *program*,† which is a set of instructions defining precisely the calculation to be performed. The language in which a program is expressed is called a *programming language*. There are a number of these in existence and the one this book uses is called *Pascal*.

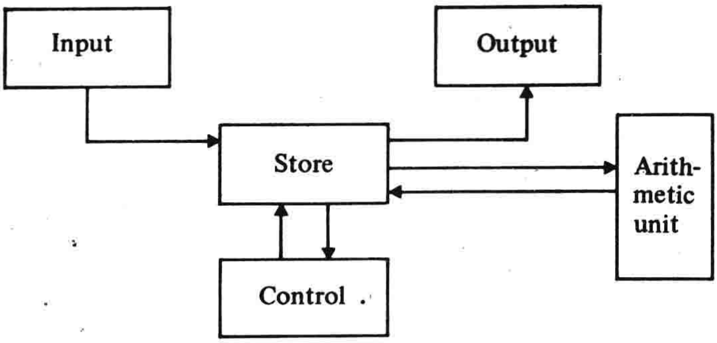


Fig. 1.1 A broad schematic of a computer

Let us look at the broad structure of a computer. It consists of five components as shown in Fig. 1.1. The program is held in part of the store and we assume that it is there initially. (How it gets there is something we will consider later in this chapter.) The machine is always under the control of the control unit which has a very simple operation: it takes an instruction from the store and causes it to be obeyed; then it takes the next instruction from the store and causes it to be obeyed; and so on. The instructions may:

(i) cause some data to be read into the (other part of the) store,
(ii) cause some appropriate operations to take place on data held in the store,

† As technical terms are introduced they will appear in *italics*.

(iii) cause some answers to be printed out from the store.

1.1 A simple example: payroll

Let us illustrate this by a simple example, a very simplified simple example. Fig. 1.2 gives a program in Pascal which will calculate the gross pay of one employee, given the number of hours he works and his rate of pay.

More precisely, the program will read in three numbers, an integer representing his personal code number, and two real numbers representing the number of hours worked (a real allows fractions of an hour to be recorded) and the rate of pay in dollars respectively; calculate from this the gross pay; and print out two numbers, the personal code number and the gross pay. The overtime pay provisions are quite simple: overtime is paid at time and a half after *37½* hours have been worked. Further, the program assumes that the employee works at least the regulation *37½* hours.

```
program payroll (input, output);
var code : integer; hours, rate, gross, overtime : real;
begin
read (code, hours, rate);
overtime := hours - 37.5 ;
gross := 37.5*rate + overtime* 1.5* rate;
write (code, gross)
end.
```

Fig. 1.2 A very simple payroll program

The program, as written, follows very precise rules (and much of this book is concerned with specifying these rules) but for the moment we are concerned only with explaining how the computer works.

Let us assume therefore that this program is in the machine and that the data:

17426   *42.75*   *4.40*

(which describes employee number *17426* who has worked for *42¾* hours and is paid at a rate of $*4.40*) is available at the input.

With the description of the action we will give some pictorial representation at certain points in time of the part of the store that contains the data. (Part of the store, of course, contains the program.) Initially the store contains the program and whatever was left there by the previous program.

The segment:

```
program payroll (input, output);
```

is called the *program heading.* This starts the program, gives it the name *payroll,* and says that the program has both *input* and *output.* It is possible and sensible to have a program with no input; but it is not really sensible to have one with no output. No action is taken by the computer. Note that **program** is in bold type and does not start with a capital letter. Note, too, that precise punctuation is important: the brackets, the comma and the semi-colon are all necessary. The spacing is rather more flexible, as we shall describe in Chapter 6, but until then we will put blanks in as we would when writing English.

The rest of the program has the form of a *block* (for reasons we will explain later) and is terminated by a full stop. The block consists of a variable declaration part followed by a statement part.

The *variable declaration part* :

var *code : integer; hours, rate, gross, overtime : real;*

defines the characteristics of the quantities that the program operates on, the so-called *variables.* It says that there are five variables, one called *code* which may take on integral values only, and four called *hours, rate, gross* and *overtime,* all of which may take on real values.

Clearly the variables of this program correspond to the quantities described above. For example, the variable *gross* corresponds to gross pay. There is a certain freedom of choice in this matter as we will see in Chapter 2.†

The effect is that five pieces (or *locations* as we generally say) of storage are allocated, one to each of the variables. Pictorially we can imagine that the name of the variable is attached to the location as shown in Fig. 1.3.

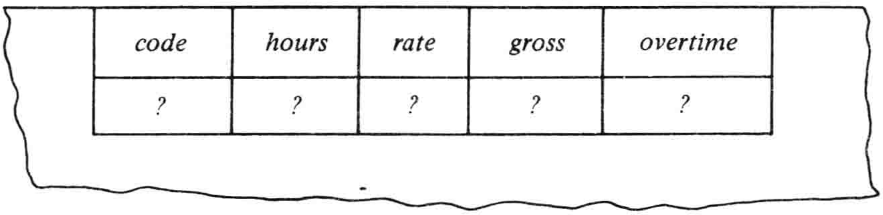


Fig. 1.3 The store after the declarations

No value is associated with any of the variables as yet; each location still contains whatever value happened to be in the store when the program was started. We indicate this by a question mark though later on we will merely leave the location blank.

The next segment:

begin

† There is no real reason for choosing *gross* rather than, say, *grosspay* or *gpay*.

simply heralds the beginning of the ***statement part*** which extends up to and including the

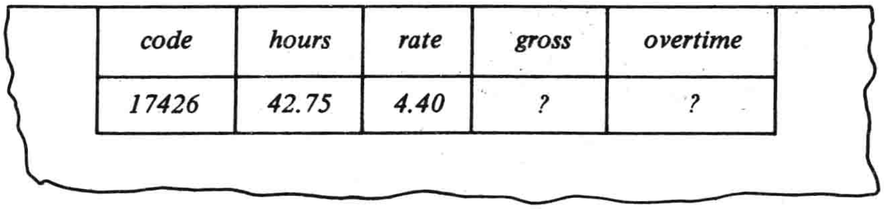
end

at the end of the program. The statement part consists of four *statements* separated by semi-colons. (Again the punctuation is important.)

The first statement (a form of *procedure statement*):

```
read(code, hours, rate)
```

causes the next three numbers to be read from the input, (there should, of course, be only the three) and their values assigned to *code, hours* and *rate* respectively. Fig. 1.4 shows the effect on the store.



code	hours	rate	gross	overtime
17426	42.75	4.40	?	?

Fig. 1.4 The store after the read statement

The next two statements:

```
overtime := hours - 37.5;
gross := 37.5*rate + overtime*1.5*rate
```

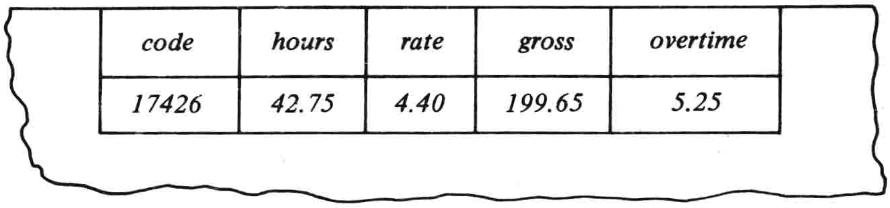
are *assignment statements.* Each causes appropriate (here arithmetic) operations to take place and the result assigned to a variable.

Thus, in the first, *37.5* is subtracted from the value of *hours* (which is *42.75*) giving *5.25*, this value being assigned to *overtime.*

Similarly, the value:

$$37.5 \times 4.40 + 5.25 \times 1.5 \times 4.40$$
$$= 199.65$$

is assigned to *gross.* Fig. 1.5 shows the state of the store after these two statements.



code	hours	rate	gross	overtime
17426	42.75	4.40	199.65	5.25

Fig. 1.5 The store after the assignments

The next statement (another procedure statement):

write (code, gross)

causes the values of *code* and *gross* to be printed out. This is the last statement of the statement part.

Note that, as we mentioned before, the whole program is terminated by a full-stop.

1.2 The compiler

We have assumed here that the program is initially in the store and that the data is available at input. The question is: how does the program get into the machine? The answer is it is read by another program which resides in the machine, called a *compiler*†. The compiler not only reads in the program but also converts it from Pascal (the so-called *source language*) into the language of the machine (the so-called *object language*). We can usually forget this conversion process and pretend, as we have here, that the machine obeys Pascal. It will be convenient, though, to refer to the compilation process throughout this book to explain some of the features of Pascal.

We ought, at this point, to distinguish carefully the roles of program and data. In real life a program once written will be used unchanged on different sets of data perhaps over an extended time. Thus we conceive of the program as the very general component and the data as the component which makes it specific. The program, once properly developed, will reside, like the compiler, within the system and data will be presented at the input whenever the program is to be run. In an environment where programs are being developed, however, we tend to use very simple data (for ease of testing) and present program and data together to the machine.

1.3 Running the program

A Pascal program contains certain words which, as written in manuscript or typescript, are underlined and as printed, appear in bold face.‡ This convention serves two purposes:

(i) Experience with similar languages shows that the underlining makes it easier to read programs.
(ii) It reminds us that the underlined words are *word delimiters* (or *reserved words*) which have a fixed meaning and can only be used in a fixed context.

† It may be an *interpreter* but the differences are not important at this point.

‡ It is unfortunate that there is no consensus amongst authors on this point. Our convention is essentially that of Wirth.

The *identifiers* (to be defined shortly) can, on the other hand, be changed at will. Thus Fig. 1.6 performs the same function as Fig. 1.2.

```
program p (input, output);
var c : integer; h, r, g, o : real;
begin
read (c, h, r);
o := h - 37.5;
g := 37.5*r + o*1.5*r;
write (c, g)
end.
```

Fig. 1.6 The simple payroll program of Fig. 1.2 with single-letter identifiers

Note that there are some identifiers such as:

integer real read write

which are *pre-defined.* That is, they *could* be used for another purpose but it would be difficult to find a good reason for doing this. As is the case with most other programming languages, we simply have to learn which words are pre-defined identifiers and which are word delimiters. Appendix 2 contains a list of both classes.

To run a Pascal program it must be presented to the machine on punched cards or punched paper tape, or from a terminal. This involves us in two problems:

(i) Most input devices have only upper case letters and so compilers are generally written to accept upper case and treat it as lower case. This allows us to use lower case letters in our identifiers which makes the writing of manuscript programs much easier.

(ii) The *operating system* of the computer demands that certain other information be supplied on control cards. Further, as noted earlier, the program and data are presented together in small development programs.

The payroll program of Fig. 1.2 might well be presented as in Fig. 1.7, where we have added some commentary between the braces.

```
{Some control cards as prescribed by your computer system.
The information provided will include at least a name which
identifies you to the computer system.}
PROGRAM PAYROLL (INPUT, OUTPUT);
VAR CODE : INTEGER; HOURS, RATE, GROSS,
                         OVERTIME : REAL;
BEGIN
READ (CODE, HOURS, RATE) ;
OVERTIME := HOURS - 37.5;
```

cont'd

```
GROSS := 37.5*RATE + OVERTIME*1.5*RATE;
WRITE(CODE, GROSS)
END.
{A control card to separate the program from the data.}
17426 42.75 4.40
{A control card to indicate the end of the data.}
```

Fig. 1.7 How the program of Fig. 1.2 might be presented to the machine

It is unfortunately true that different compilers produce their output in different ways so that it would be fruitless to spend much time on any one in particular, and we leave it to course tutors to explain the system to be used.

It may be, of course, that our program is not correct. What happens then when we present it to the computer for running? One of three things may happen:

(i) Suppose for example, we misspelt the word *rate* as *rat* in the read statement of the payroll program. The compiler will discover this error by noting that, although we have referred to the variable *rat*, no such variable has been declared. There could be many other such errors. The compiler will always give some indication of the *symptoms:* we then have to *diagnose* the problem. While this is difficult at first it becomes fairly easy after a little experience.

(ii) Now suppose that we had typed the program correctly but had allowed two of the data values to become merged together. Then when obeying the (translated version of) the statement:

read (code, hours, rate)

the machine would stop running the program, giving a message which indicates where in the program the error was detected and printing out the values of the variables as a guide to help in diagnosing the error.

(iii) Suppose, finally, we mispunched the + as a – in an assignment statement. The program would run to a conclusion; it just would give the wrong answer! This is always a difficult mistake to correct. We will discuss it in some detail in Chapter 12 but at this stage the only thing to do is to work through the program step by step (like the computer) using diagrams of the store like those used earlier (or at least stylised forms of them) until we find the cause. Note that we must have known that there was an error before we could have corrected it. That is, we must have known the correct answer in order to determine that the program-produced answer was wrong. The subject of testing programs we