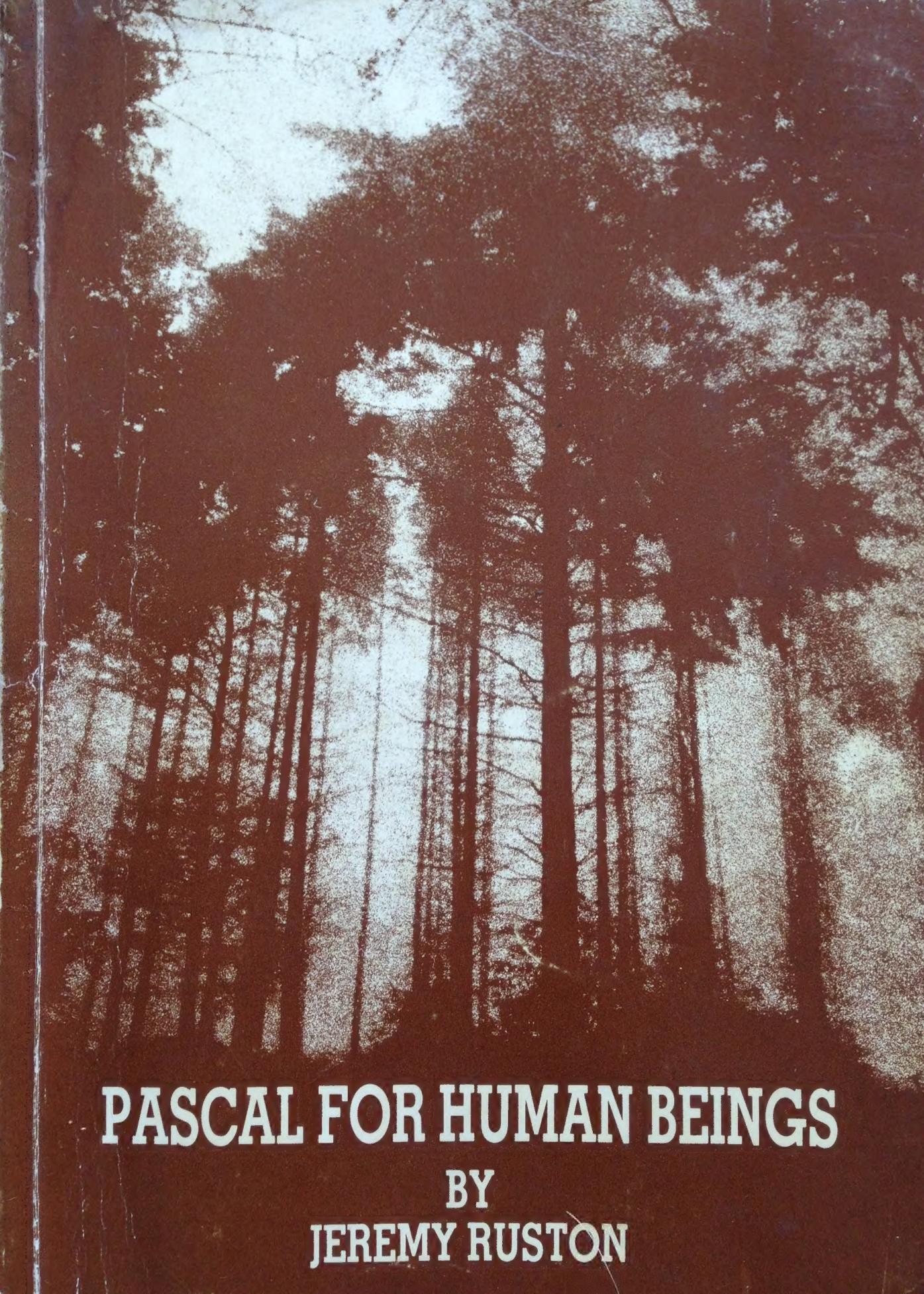

PASCAL FOR HUMAN BEINGS

BY

JEREMY RUSTON

PASCAL FOR HUMAN BEINGS

BY

JEREMY RUSTON

Published in Great Britain by:

INTERFACE
44-46 Earls Court Road,
LONDON W8 6EJ

ISBN 0 907563 09 0

This printing January, 1982

Any enquiries regarding contents of this book should be directed by mail to INTERFACE, 44-46 Earls Court Rd., LONDON, W8 6EJ

Contents

Chapter 1
Simple Pascal programs

Not everyone has a Pascal computer, so to run most of the simpler programs in this book you only need to be able to run the compiler, which is written in BASIC and will convert a Pascal program into a BASIC equivalent. It still can be no substitute for a real Pascal computer, and as Pascal is becoming more readily available for more computers the reader should consider purchasing such a package for his own computer.

If you do have access to a Pascal computer I am assuming you know how to enter and execute a program, as the procedure is different from one machine to another, and that you have your manual to hand, to reconcile differences between versions of Pascal.

The reader with no computer is at a slight disadvantage, but there is no reason why such a person should not become proficient in Pascal.

The book is scattered with example programs. In addition to each Pascal program I have included the required input to my computer.

To introduce Pascal, here is a simple BASIC program to add two numbers together.

```
10  PRINT "ENTER TWO NUMBERS";
20  INPUT A,B
30  C=A+B
```

```
40  PRINT "THE ANSWER IS",C
50  END
```

Now compare it to a Pascal solution to the same problem.

```
VAR
  A,B,C:REAL;
BEGIN
  WRITE ('ENTER TWO NUMBERS');
  READLN (A,B);
  C : =A+B;
  WRITELN ('THE ANSWER IS',C)
END.
```

It doesn't matter how the program works at this stage, I only include it to point out the differences between BASIC and Pascal.

Perhaps the most striking difference is the absence of line numbers in the Pascal program. In fact the BASIC version does not use the line numbers anyway! The main use of line numbers in BASIC is in editing programs, for referencing which line you wish to make changes in. Line numbers are used, of course in GOTO and GOSUB statements. There is a trend in BASIC interpreters to use labels in these statements in the same way as Pascal, which removes the need for line numbers in the logical program. That is, the program seen by the computer, as distinct from that seen by the user. Most Pascal implementations do provide line numbers for the purpose of editing, but published programs do not include line numbers.

The second difference is the way the variables have been declared to be of a certain type. The use of variables is covered more fully in later chapters.

The similarities should be easy to spot, the formula used is the same and the END statement is used in much the same way in both languages. Some of the Pascal statements have direct

similes in BASIC, for instance READLIN and INPUT. If you examine the compiler you can see that there are a lot more statements with direct similes.
Study this Pascal program:

```
BEGIN
  WRITE ('HELLO')
  (*THIS IS A COMMENT AND IS IGNORED*)
END.
```

It is easy to see what this program does, it just prints (or writes) the word 'hello' on the printer or TV screen. Now enter it onto your system and execute it.

A number of things could happen, first the computer could do what you asked of it and write 'hello' on the output device, or it could output a few error messages.

If your system does retort with an error message the two sources of possible error are:

1. The system requires " to go round the word 'hello', rather the more logical ' mark that other systems require. If that is the case, bear in mind that all the programs in this book use single quotes, and be prepared to alter them when you input the programs.
2. The system requires an extra 'program' statement at the beginning of the program. This takes the form 'PROGRAM (OUTPUT)' where you insert your descriptive name for the program before the brackets, but do not include spaces in the name and try to make the name reflect what the program does. The brackets should also contain the word 'INPUT' if the following program takes input from the user as well as outputting information to him. The statement is used to specify where the input and output is to come and go from and to, for the purpose of accessing files, which outside the scope of this book, not because it is difficult (it is) but because systems vary too much for the subject to be covered adequately in a general book such as this.

The solution to the 'program' problem is to look through your manual and see if the example programs in it use the word 'program' as the first line and if so, it is a fair bet you should too!

To cover more specific points about the program:

1. The full stop after the word END in the last line tells the computer that the program has finished. Most systems are peculiarly unhelpful in their advice if you miss the full stop out.
2. The WRITE statement outputs the words in brackets. We shall see later that all Pascal statements and functions operate on the data in brackets following them.
3. The words between (*and*) are comments and are ignored by the computer. Comments can be put anywhere in a program where spaces are allowed. Some systems use percentage signs to delimit comments and some others use 'curly' brackets. You can put anything in these comment sections, but many versions of Pascal interpret some sequences of characters between (*and*) as 'compiler directives'. These allow the use of advanced features of a compiler, and are not standardised in any way.

Most programs in this book are not commented in the listing but in the surrounding text. It is very important to comment programs properly. If you or someone else has to alter any of your programs after they were written it makes their task much easier if the program is as easy to understand as possible.

4. BEGIN and END delimit where the program starts and where it ends. The BEGIN-END construction does have other uses which we will come to later.

There are a few interesting alterations we can make to this program. Try altering the text in brackets in the WRITE statement. No surprises there. Try putting whatever kind of quote your system does not use in the middle of the text and you should find that it works. A lot of systems will print the kind of quote it uses to delimit the text if you include it twice in a row in a string of characters. All of this is useful if you need to print Irish names as O'Rorke, O'Callaghan etc. or if your

English is bad and you need to print things like "it's" in prose, or generally need to use apostrophes.

Try replacing WRITE wute with the same results as before. WRITE on its own just prints the contents of the brackets but the WRITELN statement moves to a new line after printing the contents of the brackets. In BASIC an equivalent would be "PRINT SOMETHING = WRITELN" and "PRINT SOMETHING; = WRITE." (A semi colon stops the carriage return, line feed sequence).

The next program prints something twice, on adjacent lines:

```
BEGIN
  WRITELN ('SOMETHING');
  WRITELN ('SOMETHING')
END.
```

The confusing point about it is the presence of the semi colon. The reason for this is actually quite simple.

The definition of a program only specifies one statement so if you want to use more than one statement in your program you must tell the machine where one ends and the next begins, so the statements are separated by semi colons. I underline separated because it is important that you do not put a semi colon before the word END or after the word BEGIN, just between statements. As an exercise there is one program in this book without any semi colons in it. Put them where you think that they ought to go.

And now onto more challenging stuff – Mathematics.

Chapter 2
How numbers work

Pascal allows you to perform arithmetic on two types of number. These are whole numbers ('integers') and decimals ('real numbers'). There is a simple rule for telling the two types apart: real numbers have decimal points and integers do not. Here are some examples of each kind of number:

Integers
123
−98765
345

Reals
3.14159265358979323846264338327 9
9.999999
3.333345678909
3.0
−9.0

And so on. Notice that some real numbers are whole numbers, but not integer due to the presence of a decimal point, therefore, all possible integers are equal to a corresponding real number.

The other immediate difference between the two types is the range of values which can be represented by a number. Usually integers must be between −32000 and +32000, with real numbers being very much bigger, depending on the computer. If a real number becomes too large, it is

printed by the computer in exponential format. For example:

9.987654E + 28
Ø.1E + 9
9.8E − 9

This means that the number after the letter 'E' is the power of ten that the first number must be multiplied by. If you do not follow this do not worry, as only a very mathematically minded person would come across this form of number in 'every day' computing.

Great, you know the difference between integers and reals, but what is the point of integers when all integers can be represented as real numbers? There are two reasons, speed and accuracy.

Integers are stored in most computers in just two parts, whereas a real number may have to be divided into five or six parts and it is clearly easier, and therefore faster, to manipulate two parts than six. As for accuracy, a fraction such as $\frac{1}{3}$ may not directly be represented in decimal – Ø.333 may be closer to the actual value of $\frac{1}{3}$ than Ø.33, but no finite series of numbers can ever *equal* $\frac{1}{3}$.

This is irrelevant to the majority of users, so the lesson to be learned is to use integers if you possibly can, and if you need to use fractions, store each fraction at its top and bottom in two separate integers. If you are not concerned about getting 10 places of decimals then use real numbers, but be prepared for errors in the last few places of decimals.

We will deal with real arithmetic first, as it is easier in relation to 'calculator' Maths.

As in most computer languages, Pascal uses plus and minus signs to indicate addition and subtraction of numbers respectively. Division is represented by a slash sign and multiplication by an asterisk. So 3.Ø*6.Ø is the same as three times six and 3.Ø/6.Ø is the same as three divided by six.

On the subject of numbers it is common practice to represent the number zero by Ø and the letter 'O' as 'O'. This avoids confusion, in theory, but as there is no accepted way of differentiating one and 'I' or '1' the system falls down after a

printed by the computer in exponential format. For example:

9.987654E + 28
Ø.1E + 9
9.8E − 9

This means that the number after the letter 'E' is the power of ten that the first number must be multiplied by. If you do not follow this do not worry, as only a very mathematically minded person would come across this form of number in 'every day' computing.

Great, you know the difference between integers and reals, but what is the point of integers when all integers can be represented as real numbers? There are two reasons, speed and accuracy.

Integers are stored in most computers in just two parts, whereas a real number may have to be divided into five or six parts and it is clearly easier, and therefore faster, to manipulate two parts than six. As for accuracy, a fraction such as $\frac{1}{3}$ may not directly be represented in decimal – Ø.333 may be closer to the actual value of $\frac{1}{3}$ than Ø.33, but no finite series of numbers can ever *equal* $\frac{1}{3}$.

This is irrelevant to the majority of users, so the lesson to be learned is to use integers if you possibly can, and if you need to use fractions, store each fraction at its top and bottom in two separate integers. If you are not concerned about getting 10 places of decimals then use real numbers, but be prepared for errors in the last few places of decimals.

We will deal with real arithmetic first, as it is easier in relation to 'calculator' Maths.

As in most computer languages, Pascal uses plus and minus signs to indicate addition and subtraction of numbers respectively. Division is represented by a slash sign and multiplication by an asterisk. So 3.Ø*6.Ø is the same as three times six and 3.Ø/6.Ø is the same as three divided by six.

On the subject of numbers it is common practice to represent the number zero by Ø and the letter 'O' as 'O'. This avoids confusion, in theory, but as there is no accepted way of differentiating one and 'I' or '1' the system falls down after a

good start. The context will generally show which is required.

If you combine two real numbers with any of the four arithmetic operations, the answer will automatically be a real number. This means that it is not possible to write an expression such as 3 + 8.Ø in Pascal. There are ways around this restriction, of course.

Integer arithmetic works in much the same way, except an integer result is always derived as the answer to a sum.

The symbols '+' and '−' are used as normal, as is the asterisk for multiplication, but the slash (division) gives a real number result for two integers. This is fair enough, you may say, because it is impossible to make an answer such as Ø.5 into an integer. If you wish to perform integer division and be left with an integer result, it is necessary to use the operators 'DIV' and 'MOD'. (X DIV Y) gives the whole number part of X divided by Y and (X MOD Y) gives the remainder of that division. For example:

2 MOD 3 = 2	2 DIV 3 = Ø
234 MOD 56 = 1Ø	234 DIV 56 = 4
− 1Ø MOD 23 = 1Ø	− 1Ø DIV 23 = Ø

The practical application of arithmetic will be covered in the next chapter, but it is important that you know the different operations, what they do, and what type of numbers they give and take.

Enter/examine this program. (This section is not applicable to the compiler at the back of the book.)

```
BEGIN
  WRITE     (3.14159265358979323846264338327Ø)
END.
```

When you execute it, Pi will be output to about 1Ø places of decimals. The computer only digests as many of the input places of decimals on which it can conveniently perform arithmetic.

Fair enough, but supposing you had been writing a program for your boss to show him what your next year's salary should be? The computer could output something like

'£5999.238739583', in the same way as it did above. (If you use ten places of Pi to calculate the circumference of the earth, the answer would be accurate to ten inches!)

Plainly then, the computer sometimes gives you an unreasonably accurate answer. There is a way to limit the accuracy of the computer's output (not the accuracy which it uses internally) and to get more control over all numerical output at the same time. (Did you notice the 'leading spaces' the computer printed before it printed Pi? We can get rid of those too.) Change the contents of the WRITELN statements brackets to 3.Ø:2Ø:3.

The number three is printed out with 3 decimal places (that is, with three zeros after the decimal point) the last zero is 2Ø spaces from the left hand side of the screen/paper.

You can insert any integers in the place of 2Ø and 3 but do not make the second larger than the first, otherwise you will get an error message. This is called formatting, and is also applicable to integers, except the number which specifies the number of decimal places to be output is not used in integer write statements (obviously!).

One other point, a WRITE statement can incorporate both numbers and text, separated by commas. For instance: WRITELN ('The answer is',3).

Chapter 3
Variables (simple types)

Imagine you wish to write a program to multiply two numbers together, where the user specifies the two numbers. You could, using what you know so far, produce something like this:

```
BEGIN
  WRITELN (34*45)
END.
```

If you want to alter the values of the two numbers you have to alter the program, which is not always convenient. Readers with previous computer experience will know that one uses 'variables' to represent values not known at the time of writing the program, for instance those numbers input by the user.

Here is a program using variables to achieve the same effect as the last one. There is much to be said about this program, so I have included line by line notes on it after the listing.

```
VAR
  (*THESE NAMES WILL PROBABLY HAVE TO BE
CHANGED FOR USE ON MY*)
  (* COMPILER*)
```

```
  ONENUMBER,TWONUMBER,THREENUMBER:
  REAL;
BEGIN
  WRITE ('ENTER TWO NUMBERS');
  READLN (ONENUMBER,TWONUMBER);
  THREENUMBER:= ONENUMBER* TWONUMBER;
  WRITELN (THREENUMBER)
END.
```

Lines 1 and 2: These tell the computer that the next few lines are 'variable declarations'. In other words, all the words that follow VAR will be given the attribute that follows the next colon, so in my example I have allowed for three real numbers which shall be called (as we do not know their actual value) ONENUMBER, TWONUMBER, THREENUMBER. From that point on in the program these three numbers may be manipulated like a normal number and if I try to print them, the computer inserts the values last assigned to those names, and prints them out instead. Variables can be thought of as little boxes, with names, which can hold a piece of paper with a number on it.

(Unlike BASIC, variable names in Pascal can be quite long, the exact length depends on the version of Pascal, but usually only the first 8 characters are significant, so variables such as 'ABCDEFGHIJKLN' and 'ABCDEFGHIJO' are thought of by the computer as being the same.)

Variables can be built up of numbers and letters, but must not start with a number, or contain spaces. It is good practice to make the names of your variables reflect what they represent, so a program to calculate VAT should contain a variable called VAT.

Lines 3 and 4: Work the same way as in earlier programs.

Line 5: When this statement is encountered, the computer asks the user to input two numbers, usually by printing a

```
  ONENUMBER,TWONUMBER,THREENUMBER:
  REAL;
BEGIN
  WRITE ('ENTER TWO NUMBERS');
  READLN (ONENUMBER,TWONUMBER);
  THREENUMBER: = ONENUMBER* TWONUMBER;
  WRITELN (THREENUMBER)
END.
```

Lines 1 and 2: These tell the computer that the next few lines are 'variable declarations'. In other words, all the words that follow VAR will be given the attribute that follows the next colon, so in my example I have allowed for three real numbers which shall be called (as we do not know their actual value) ONENUMBER, TWONUMBER, THREENUMBER. From that point on in the program these three numbers may be manipulated like a normal number and if I try to print them, the computer inserts the values last assigned to those names, and prints them out instead. Variables can be thought of as little boxes, with names, which can hold a piece of paper with a number on it.

(Unlike BASIC, variable names in Pascal can be quite long, the exact length depends on the version of Pascal, but usually only the first 8 characters are significant, so variables such as 'ABCDEFGHIJKLN' and 'ABCDEFGHIJO' are thought of by the computer as being the same.)

Variables can be built up of numbers and letters, but must not start with a number, or contain spaces. It is good practice to make the names of your variables reflect what they represent, so a program to calculate VAT should contain a variable called VAT.

Lines 3 and 4: Work the same way as in earlier programs.

Line 5: When this statement is encountered, the computer asks the user to input two numbers, usually by printing a

question mark. The prompt is supplied by the 'WRITE' statement.

The READLN statement that I used has a variant 'READ' which predictably stays on the same line after requesting input from the user.

The two numbers are called the names in brackets.

Line 6: This is an assignment statement, where the equation on the right of the ':=' is evaluated and the result is stored under the name on the left. The asterisk could be replaced with any other sign (i.e. +, − or /).

Line 7: WRITEs the contents of a variable (or variables), separated by commas to the screen or printer, note there are no quotes around the variable name.

Line 8: ENDs the program as before.

To make the program work with integer arithmetic, make the variables integer variables by replacing the word 'real' with 'integer', in the second line.

Notice that each type of variable (e.g. the word REAL) in a VAR section of a program is terminated by a semicolon.

This use of semicolon is, unfortunately, an exception to the rule that I gave earlier.

My own compiler will only accept those variables names that are acceptable by the version of BASIC that you are using, thus on the BBC machine, you have the same choice of variables as on a real Pascal computer, with the exception that variable names cannot begin with any of BASICs reserved words.

If you are using Microsoft BASIC, you will have to ensure that the first two characters of each variable name is different, and that none of them have imbedded reserved words.

If you are used to BASIC the presence of a colon before the equals sign may be perplexing. The reason is that in BASIC an assignment such as 'B = B + 3' could appear, which is probably fine from the point of view of the computer and you, but to a mathematician it is nonsensical. B does *not* equal B + 3, and never will. So using an equals sign is, in this case, a misuse of the sign. The man who invented Pascal was a purist and an optimist. He was a purist because he didn't like seeing the equals sign misused in this way, and so decreed that the colon

should be used. He was an optimist because he did this to aid the beginner to computer programming, not realising that just about everybody learning to program does so on a BASIC computer.

Chapter 4
CHAR type variables & CONSTants

So far I have not mentioned any provision in Pascal for processing words as opposed to numbers. In BASIC this is called 'string processing'. In real life (i.e. life outside computer books) standard Pascal is not at all suitable for large amounts of text processing. This is a limitation of Pascal as defined by Wirth, Pascal's inventor, but most extended versions of Pascal will, in addition to many less desirable 'improvements', allow for string processing in a more useful way.

CHAR type variables are variables in the same way that numeric variables are, except that rather than storing a number, they store a single letter or symbol. To show their use, here is a short program which allows the user to enter a single letter, and then it prints out the same letter.

```
VAR
  ALETTER : CHAR;
BEGIN
  READLN (ALETTER);
  WRITELN (ALETTER);
END.
(*FOR RUNNING ON MY COMPILER,
CHANGE THE VARIABLE NAME*)
```

There are similarities between this and a program to get a number from the user and print that out. In fact, if you replace the word 'CHAR' in the variable declaration section by the word 'REAL', you have just such a program.

If you try to enter more than one letter, the computer will probably either take the first or the last letter that you entered. In fact, my compiler will allow you to set up a CHAR variable with as many characters as your version of BASIC allows.

There is a simple way to manipulate a string of more than one character, under certain conditions, namely that all you do with it is print it.

The CONST declaration section of a program allows you to define an entity with a name and a value. This value can be REAL, INTEGER or CHAR, but in the case of CHAR type constants, more than one character may be used. These constants may not have their value altered in the program, and if they are CHAR type with more than one character may not be used in equations, in all other cases, CONSTants may appear on the right hand side of an assignment statement, but not on the left.

Here is an example of the CONST statement in use, showing that the CONST declaration section of a program comes before the variable declaration section.

```
CONST
  AMESSAGE = 'THIS IS A MESSAGE';
VAR
  (*DECLARE SOME VARIABLES HERE*)
  ...
BEGIN
  WRITELN (AMESSAGE);
  ...
END.
(*Change the name AMESSAGE to something
shorter for my compiler*)
```

Back to CHAR variables:

CHAR variables cannot have any recognizable arithmetic performed on them, but as we shall see in Chapter 6, there are equivalent functions to CHR$ and ASC in Pascal. They may however be assigned to in the normal way, for example this program assigns the letter 'T' to a CHARacter variable. Usually letters in this context are contained in single quote symbols. Note the use of ':=' to assign values.

```
VAR
  ALETTER : CHAR;
BEGIN
  ALETTER := 'T';
  WRITELN (ALETTER)
END.
(*Change the name of ALETTER to something
shorter for my compiler*)
```

The use of a variable which can only hold one character is rather limited, but CHAR variables are useful for extracting Yes/No responses from a user.

Most simple programs will not use CHAR type variables: there is very little point for them, except in file access.

Chapter 5
Rudyards bit (IF)

So far we have no way of introducing decision making into our programs and as a simple way of altering the course of a program is essential, I'll cover it now.

The 'IF' statement takes this form:

IF (condition) THEN (do something) ELSE (do something else).

The condition part is an expression which can be either true or false, for example, IF VAT = 34 THEN WRITE (''VAT is 34%'');. Notice that the equals sign is not preceded by a colon. The colon is only necessary in an assignment statement to make sure that VAT = VAT + 2 or some other nonsensical expression does not appear.

A condition section is more properly called a Boolean expression, after a man called Boole. ''Greater than'' signs and the like can also be used in Boolean expressions, even variables on their own, as long as the variable is of the type BOOLEAN, that is, can only take on the values 'true' or 'false'. The program below contains a number of 'IF' statements to illustrate their use:

```
VAR
  ANUMBER,BNUMBER: REAL:
  BOOL1,BOOL2,BOOL3 : BOOLEAN;
  BEGIN
  WRITE ('ENTER A NUMBER LESS THAN TEN');
```

```
  READLN (ANUMBER);
  IF  ANUMBER< 10.Ø  THEN  WRITELN  ('WELL
  DONE') ELSE WRITELN ('WRONG');
  BOOL1 : = TRUE;
  IF BOOL1  =  TRUE THEN
    BEGIN
    WRITELN ('PRINTED')
    END
END.
(*THIS WILL NOT RUN ON MY COMPILER*)
```

Notice the use of the words 'TRUE' and 'FALSE'.

Boolean variables, unlike CHAR variables, can have arithmetic performed on them, but Boolean operators are different from numerical ones. First of all it is possible to write an 'IF' statement of the form IF (condition 1) AND (condition 2) OR (condition 3) THEN ....

The Boolean operators and their effects are tabulated below:

```
  AND:  TRUE AND TRUE = TRUE
      :  TRUE AND FALSE = FALSE
      :  FALSE AND FALSE = FALSE
   OR:  TRUE OR FALSE = TRUE
      :  TRUE OR TRUE = TRUE
      :  FALSE OR FALSE = FALSE
  NOT:  NOT FALSE = TRUE
      :  NOT TRUE = FALSE
  EOR:  (Short for exclusive-or, not found on all systems.)
      :  TRUE EOR TRUE = FALSE
      :  TRUE EOR FALSE = TRUE
      :  FALSE EOR FALSE = FALSE
```

In words the definitions of the four operations are:

AND: If both operands are true, answer will be true, else false.

OR: If one or both of the operands are true, answer will be true, else false.

EOR: If one or other of the operands, but not both, are true, the answer will be true, else false.

NOT: If operand is true, answer will be false, else true.

For the moment it is only necessary to know how to use these connectives in 'IF' statements, we will come to other uses of Boolean expressions later.

If the two 'do something' sections are to contain more than one statement, they must be separated by semi colons, as explained in the first chapter, but the words 'BEGIN' and 'END' must be used before and after the list of statements.

Notice how the program has been 'indented'. This indentation is designed to make the program easier to read, by separating the levels of 'IF' statements, as the statement following say a 'THEN' section can be another 'IF' statement. In longer programs you may be confused as to where to indent and where to un-indent. There is another simple rule: 'Indent two spaces after every 'BEGIN' and go back two after every 'END''. Some systems will do the indentation for you, others expect you to do it.

It is also common practice to indent VAR sections, as I have done.

Here is a slightly more advanced program, taking in what we know so far, and the fact that a Boolean variable can be assigned to. The absence of semicolons is intentional, you must add them.

```
VAR
  A,B,C : BOOLEAN;
BEGIN
  WRITE ('THIS IS A DEMO PROGRAM');
  A := TRUE EOR (FALSE AND TRUE)
  B := FALSE OR (FALSE AND (TRUE EOR (NOT
  TRUE)));
  C := A AND (B EOR FALSE);
  IF C THEN WRITE ('C IS TRUE');
  IF B THEN WRITE (' B IS TRUE');
  IF A THEN WRITE ('A IS TRUE');
  IF NOT A THEN WRITELN (' A IS FALSE')
END.
```

Chapter 6
Standard functions

Functions are the next elements of a program that I will cover. If you wish to take square roots or compute with sines and cosines, you use functions. A function takes the form of the name of the function, then the operand of the function in brackets (the argument). For example;
ANGLE:=ARCTAN(1.Ø).

Here is a list of all of the standard Pascal functions, with brief notes on each:
ABS() gives the absolute value of the operand. For example, ABS(3) equals 3 but so does ABS(−3). In other words, ABS returns the number in brackets as it is, but if it is negative, makes it positive. ABS can take an integer or real argument, and gives a result of the same type.
ARCTAN() gives the arctangent of its argument. Its argument can be real or integer, and its result is real.
CHR() gives the character whose 'number' is the argument. Basically, each character which can be printed by your computer has a specific code, usually the American Standard Code for Information Interchange (ASCII), and it is this code that CHR takes and converts to a character. For example, CHR(65) gives, on a system which uses the ASCII code, the letter 'A'. The argument is an integer, and the result a character variable.
COS() returns the cosine of its argument (which must be in radians). Its argument can be real or integer and its result is real.

EXP() returns 'e' to the power of its argument, where 'e' is equal to 2.7182818. Its argument can be real or integer and its result is real.

LN() gives the natural logarithm, that is the logarithm to the base 'e' (see EXP()). Its arguments can be real or integer and its result is real.

ODD() returns 'true' if its argument is odd, 'false' otherwise. Its argument is integer and the result is Boolean.

ORD() returns the 'code' of the character which is its argument. For example, ORD('A') = 65. ORD does have other uses, which will be explained in due course. Suffice to say that it is the opposite of CHR(). Its argument is of type character and its result is of type integer.

PRED() gives the integer or character which is the predecessor of its argument. In the case of characters, it returns the character whose code is one less than the argument. The argument is either of type integer or of type character, and the result is of the same type as the argument.

ROUND() gives the nearest integer to a real argument.

SIN() gives the sine of its argument, which should be in radians. The argument can be real or integer and the result is real.

SQR() gives the square of its argument. If the argument is real, the result will be, if the argument is integer in type, so will the result be.

SQRT() gives the square root of its argument. Its argument can be real or integer and its result is real.

SUCC() gives the integer or character which is the successor of its argument. In the case of characters, it returns the character whose code is one more than the argument. The argument is either of type integer or character, and the result is of the same type.

TRUNC() gives the integer which is closest in value to a real argument, and is closer to zero than the argument. For example, both TRUNC(3.14) and TRUNC(3.9999666666) are equal to 3. The argument is real and the result is integer.

Notice TAN() is not provided; use the relationship $TAN(X) = \frac{SIN(X)}{COS(X)}$ instead.

Chapter 7
FOR loops

Consider the problem of printing out the integers from one to one hundred. So far the only way to do this is to have one hundred WRITELN statements, each with a different argument. Naturally there is an easier way.
For (variable): = (expression) TO (expression) DO (statement);

The FOR statement gives the variable following it all the values between the first expression and the second in turn and then executes the statement following the word DO. If more than one statement is needed the BEGIN-END construction must be used, with each of the statements being separated by semi colons.

Our program to print all the integers between one and one hundred becomes:

```
VAR
  H : INTEGER;
BEGIN
  FOR H : =1 TO 100 DO WRITELN (H)
END.
```

The variable used is called the control variable.

During the execution of a FOR loop you must never try and alter the value of the control variable, that is, it can be put on the right hand side of an assignment statement, but never on the left hand side.

If we wished to print out all the integers from one hundred down to one, we replace the word TO with the word DOWNTO, and put one hundred as the first variable and one as the second.

This program prints all the letters of the alphabet:

```
VAR
  LETTER : INTEGER;
BEGIN
  FOR LETTER := 65 TO 90 DO WRITELN (CHR(LETTER))
END.
```

This program prints out every other letter of the alphabet:

```
VAR
  LETTER : INTEGER;
BEGIN
  FOR LETTER := 1 TO 13 DO WRITELN (CHR(LETTER*2+63))
END.
```

Some systems will not allow CHAR type variables to be control variables in FOR statements, so it is better not to use them as such, to ensure that those programs that you write will be transferable to as many systems as possible.

Here are some more programs to demonstrate use of the FOR statement:

This program prints the squares of all integers between 1 and 10.

```
VAR
  NUMBER : INTEGER;
BEGIN
  FOR NUMBER := 1 TO 10 DO WRITELN (SQR(NUMBER))
END.
```

This one prints out all the integers between 1 and 100 which are perfect squares:

```
VAR
  NUMBER : INTEGER;
BEGIN
  FOR NUMBER : = 1 TO 100 DO
  BEGIN
    IF  SQRT(NUMBER) = TRUNC(SQRT(NUMBER))
THEN WRITELN (NUMBER)
  END
END.
```

This program prints out binary equivalent of any integer:

```
VAR
  A,B,C : INTEGER;
BEGIN
  WRITELN ('ENTER A NUMBER LESS THAN
256');
  READLN (A);
  C : = 128;
  FOR B: = 1 TO 8 DO
    BEGIN
    IF (A DIV C) 0 THEN WRITE('1') ELSE
WRITE ('0');
    IF (A DIV C) 0 THEN A: = A – C;
    C: = C DIV 2
    END;
  WRITELN () (*MOVES TO A NEW LINE*)
END.
(*THE ELSE PART MEANS THAT THIS WILL
NOT RUN ON MY COMPILER*)
```

Between all the example programs in this Chapter, you should be able to see exactly how the FOR statement is used.

So far, due to the limited number of Pascal statements that we have covered, I have not suggested any exercises for you to do. To remedy that, here are a couple of tasks you should find helpful to try.

1. Write a program to print all the sines of angles from 10 degrees to 90 degrees, in steps of 5 degrees. Remember that Pascal gives the sines of radians, not degrees. Use the fact that X degrees = X/57.29577951 radians.

2. Write a program to print out all the cubes between 1 and 1000. A cube is a number multiplied by itself three times. For example: 2 cubed is 2*2*2 = 8, 3 cubed is 3*3*3 = 27 and 4 cubed is 4*4*4 = 64 and 5 cubed is 5*5*5 = 125.

All these programs will have to use FOR loops.

Chapter 8
REPEAT UNTIL loops

Imagine that you wished to write a program to square numbers entered by the user, and when the user enters 99 as the number to be squared, the program prints out the sum of all the numbers entered and then ends.

There is no way, so far, to do this, because at the time of writing the program you don't know how many numbers the user is going to enter.

This is where the REPEAT – UNTIL construction comes in.

It takes the form:

REPEAT (dosomething) UNTIL (condition);

If you need to put more than one statement between the words REPEAT and UNTIL, there is no need for the BEGIN – END construction, since REPEAT – UNTIL is defined as operating on more than one statement, but only put the semi colons between the statements, not after the word REPEAT, nor before the word UNTIL.

The condition part is exactly the same as the condition part in an IF statement.

This construction executes the statement(s) between REPEAT and UNTIL, *until* the condition after UNTIL is evaluated true.

Two important points about this construction are:

1. The statement(s) are always executed at least once, even if the condition is false.

2. If none of the variables in the conditional expression, which is a Boolean expression, are altered in the body of the REPEAT – UNTIL construction, and the expression is false to begin with, the loop will execute for ever, which is a pity.

So here is the program outlined above, written using REPEAT – UNTIL:

```
VAR
  NUMBER,SUM : INTEGER;
BEGIN
  SUM := 0;
    REPEAT
    READLN(NUMBER);
    SUM := SUM + NUMBER;
    WRITELN (NUMBER*NUMBER)
    UNTIL NUMBER = 99;
  WRITELN ('SUM = ',SUM)
END.
(*THIS RUNS WITHOUT ALTERATION ON THE
COMPILER*)
```

As another example, here is a program which prints all the numbers between 1 and 50.

```
VAR
  NUMBER : INTEGER;
BEGIN
  NUMBER := 1;
    REPEAT
    WRITELN (NUMBER);
    NUMBER := NUMBER + 1
    UNTIL NUMBER = 51
END.
(*THIS RUNS ON MY COMPILER WITHOUT
ALTERATION*)
```

REPEAT – UNTIL is one of the most important constructions in Pascal, since it is this and the other 'loop' statements which are designed to do away with the BASIC statement GOTO, or its equivalent.

Chapter 9
WHILE loops

In the last chapter I discussed REPEAT – UNTIL, and mentioned that whatever the condition after the UNTIL statement, the statements between REPEAT and UNTIL are always executed. In a situation where the computer is monitoring some piece of equipment in a laboratory, it may be necessary to WRITE an error message if the machine is not functioning correctly, or more realistically, sound a buzzer. A small section of code to achieve this, only utilising the statements so far covered, may look like this:

```
REPEAT
  (* sound buzzer*)
  . . .
  . . .
  UNTIL NOERROR;
```

NO ERROR is a Boolean variable, which is 'true' if the machine is functioning, or 'false' otherwise.

If the machine were functioning, and the section of code is executed, the buzzer will sound once, until the UNTIL statement is reached. (Think about it.)

Obviously this is no good; the machine is reported as faulty even when it is not. The WHILE . . . DO construction takes over.

The syntax of a WHILE statement looks like this:

WHILE (condition) DO (statement);

If more than one statement is to be used in the statement

section, BEGIN – END must be used, together with semi colons to separate the statements.

The condition part is just a standard Boolean expression, but if it is evaluated to be *false*, the machine jumps to the end of the WHILE statement. If it is *true*, the statement(s) are executed until it is false.

This program will not print anything at all, because the condition for a WHILE is false to begin with.

```
VAR
  NUMBER : INTEGER;
BEGIN
  NUMBER : = 234;
  WHILE NUMBER = 233 DO
  BEGIN
    WRITELN (NUMBER)
  END
END.
(*THIS RUNS ON MY COMPILER WITHOUT
MODIFICATION*)
```

There is always difficulty in deciding whether to use a REPEAT or WHILE construction, but if you 'code' your program in English first, if it uses any kind of loop, you will find yourself saying either REPEAT or WHILE, so common sense is usually the best guide of which to use.

If you really cannot decide, the only difference between the two is basically what happens if the loop should not be executed, the WHILE statement is sensible and does not execute, but the REPEAT construction carries on regardless.

As an example this program prints out all the integers from one to ten, three times, once for each kind of loop.

```
VAR
  A,B,C : INTEGER;
  BEGIN
```

```
  FOR A:=1 TO 10 DO
  BEGIN
  WRITELN (A)
  END;
  B := 1;
REPEAT
  WRITELN (B);
  B := B+1
UNTIL B = 11;
C:= 1;
WHILE C<11 DO
  BEGIN
  WRITELN (C);
  C := C+1
END
END.
```

Chapter 10
Arrays

Most of the simple, and important, Pascal constructions have now been covered. This means that if you stopped reading this book now, you could write working programs to perform most tasks.

For this reason, the rest of the Pascal language is *inessential*, but still important, embellishment.

The reason why the 'essentials' are so few, and can be covered in such little space is that Pascal is a 'sparce' language. This means that only the bare minimum of framework is provided by the language itself; other statements/functions/variable types you may need have to be defined by the user. Thus the rest of this book only covers half as many 'reserved words' (i.e. words like REPEAT and IF) as the part which you have already read, but is much more important to the Pascal programmer.

Here is the first of those embellishments, arrays of variables, which is a concept which should be familiar to anyone who programs in BASIC.

An array is a series of variables, all with the same name, but with a number in brackets following the name, to differentiate which 'element' is being referred to. An array can be of any type, that is it can be composed of real or integer numbers, or even characters or Boolean variables. Arrays are declared in the VAR section of a program, like this:

```
VAR
TEMPERATURE,HOTNESS: ARRAY  [-100...100] OF REAL;
TIME: ARRAY [20...40] OF INTEGER
ALPHABET:ARRAY [1...26] OF CHAR;
etc.,
```

All elements of an array must be of the same type, and the 'index' (that is the number in brackets, sometimes called a 'subscript') must be an integer, or an integer variable.

The example above uses square brackets in the declaration, some computers use other symbols, check your manual.

The two numbers separated by three (sometimes two) periods (full stops) specify the maximum and minimum values of the subscript for the array.

For the above example, the third declaration reserves 26 variables of type CHAR, called ALPHABET [1] to ALPHABET [26].

All the subscripts must follow concurrently, that is you cannot define an array to have subscripts 2,4,6...20,22,24. In this example, it is easy to define the array 1...12 and then multiply any variables used to extract elements from the array by two.

Arrays have many uses:

1. If a series of numbers is to be input by a user of a program, it is not feasible to write a READ statement with over 10 variables, so a simple FOR statement linked to a single read statement can be used to input each element of the array.
2. In the same way, if a list of numbers (or letters) has to output, it is much more convenient to do so in a loop with an array.
3. If any task has to be performed on a series of numbers or any other data it is often neater and easier to use arrays. If you ever access an array in a section of program which is not a loop, beware, it would almost certainly be easier to use a normal variable.

Those who cannot see the point of arrays, it is sometimes helpful to consider an analogy such as a street of houses. The whole street has a name (the name of the array) but an individual house is known by both its number in the street and its street name. When the Avon lady goes down a street, she is, in effect, accessing our 'street array' by calling on each house, in some form of order, be it numerical order backwards or forwards.

The sort of array so far considered can be likened to a list, for instance, a list of numbers. Often a table of rows and columns is more convenient to manage, for instance a ladder of scores in a squash league between three people could be represented in a table like this:

	Player 1	Player 2	Player 3
Player 1	xxxxxxxx	3 – 4	5 – 2
Player 2	6 – 4	xxxxxxxx	3 – 9
Player 3	5 – 3	7 – 3	xxxxxxxx

The same table can be stored by a Pascal program in exactly that form, with each element (in this case the difference between the scores registered by each of the two players) being accessed by two numbers, separated by commas, the row numbers and the column number.

A two dimensional (as this sort of array is called, due to the presence of two indices) array can also be thought of as a matrix, for those that are mathematically minded. No commands are provided for multiplication with reference to complete matrices, although it would be relatively easy to design routines to carry out these operations.

A two dimensional array is declared in much the same way as a one dimensional array, except that maximum and minimum values have to be specified for both indices, as follows:

```
VAR
SQUASHSCORES:ARRAY [-99...99,-56...34] OF INTEGER;
BEGIN etc.
```

Then an element of the above array can be accessed with a command of the form 'WRITELN(SQUASHSCORES [99,23]);', breaking as it does the rule of not accessing array elements outside a loop.

As an extension of two and one dimensional arrays, Pascal provides 3, 4, 5 and sometimes greater dimensional arrays, in the same way as two dimensional arrays are themselves an extension of one dimensional arrays.

I stated that one cannot perform multiplication on matrices as such, nor any other kind of arithmetic operation. These are however a few ways of manipulating arrays without having to consider each element separately. These are:

1. One array of the same dimensions as another can be set equal to the other array by the use of a simple assignment statement, with just the names of the two arrays in it, and the subscripts and square brackets omitted. Note that to be able to do this you must ensure that the two arrays are of the same size, but the minimum and maximum values for each index can be different for the two matrices. For example:

```
VAR
A : ARRAY [1 ... 11,4 ... 14] OF INTEGER;
B : ARRAY [8 ... 18,99 ... 1Ø9] OF INTEGER;
etc.
A := B
```

is quite valid, but would not have been had the two matrices been of different size or made up of a different type of variable.

2. As an extension of this, and of normal assignments of the form A[9,9] := H[77,897], if not all the indices in a simple array assignment are specified, the computer extracts a 'sub-array' using those indices provided and works on that instead.

Thus the first row of our column of squash scores could be accessed by SQUASHSCORE [– 99], and could be set equal to

the second row with: SQUASHSCORE [-98] := SQUASHSCORES [-99]

This can be extended to the point that with SQUASHSCORES being a three dimensional array, a two dimensional sub-array is being transferred, from one part of the major array to another, with the simple statement above.

3. In Boolean expressions you can use arrays like any other variable, seeing if each element of one array is equal to the corresponding element of a different array just with the names of the two arrays and an equals sign.

AND, OR, NOT, EXOR and the inequality signs can be applied to arrays, as long as they are of the correct type for the manipulation. Thus the Boolean connectives (AND, OR etc.) can only be applied to Boolean arrays.

Chapter 11
The CASE statement

An IF-THEN statement allows you to decide on two possible outcomes to an event. The CASE statement allows you to branch to more than two places as the result of an event. It takes the form:

```
CASE NAME OF
   'A':do something;
   'B':do something;
   'C':do something;
END;
```

As before, the BEGIN-END construction must be used if more than one statement is required in the 'do something' sections of the above illustration.

The variable NAME should contain any of the values before colons in the next section of code, and when a match is found, that 'do something' segment is executed, than execution goes to the statement after the END; statement. If no match is found, the computer will halt with an error message.

The above program section is thus equivalent to:

```
IF NAME = 'A' THEN do something;
IF NAME = 'B' THEN do something;
IF NAME = 'C' THEN do something;
```

The difference between the two is that in the second case, after NAME has been given the value 'A', the first IF statement will be found to be true, and the statements

following it will be executed. They may alter the value of NAME to be 'B' however, in which case the second IF statement will be found to be true, and consequently the statements following it will be executed, which could lead to problems.

The CASE statement can be used with any type of variable, and the values before the colons should reflect the values you expect the variable to take. When using a real variable as the controlling variable, it is easy to misjudge the possible values, for this reason it is wise to TRUNCate the variable before it is used.

Here is a trivial program to illustrate the use of the CASE statement:

```
VAR
ETTER : CHAR;
BEGIN
   READLN (LETTER);
   CASE LETTER OF
   'A': WRITELN ('YOU TYPED A');
   'B': WRITELN ('YOU TYPED B');
   'C': WRITELN ('YOU TYPED C');
   'D': WRITELN ('YOU TYPED D')
END.
```

Chapter 12
The TYPE declaration

I have freely used the word 'type' so far to mean INTEGER, REAL CHAR or BOOLEAN. These are types of variable that were specified by the person who defined Pascal. In fact, by using the TYPE statement it is possible to define your own type of variable.

The TYPE statement takes the form of a new declaration section, before the VAR section of a program. Each TYPE takes the form of a name, with the usual rules of starting with a letter and only containing letters or digits, an equals sign and then the TYPE that that name represents.

The TYPE that appears after the equals sign is something like the type that appears after the colon in a VAR declaration. So the following are all valid TYPE declarations, with a VAR section included to show how the new types are used:

```
TYPE
  THINGY = REAL;
  MATRIX = ARRAY [Ø...1Ø , 4...78] OF INTEGER;
VAR
  HELLO : THINGY;
  GOODBYE : MATRIX;
  XMAS : THINGY;
  NEWYEAR : MATRIX;
BEGIN etc.
```

This example just saves some typing when you enter the

program, it achieves nothing we could not do before the TYPE statement came up.

The TYPE statement just specifies that every variable of type 'THINGY' is in fact of type REAL and every variable of type 'MATRIX' is an array of two dimensions.

Now, suppose you want to write a program which computes the number of hours in a week a worker is on strike. This particular striker strikes for a different time each day, depending on which day of the week it is. The program would have to look at each day of the week separately, and so some sort of loop from 1 to 7 would be required.

Pascal goes one step further than that, it allows you to set up a loop of the form 'FOR DAY: = MONDAY TO SUNDAY DO . . .'.

To do this you first have to define a new type which allows a variable to take on any of the values MONDAY,TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY or SUNDAY. A type declaration to do this would be:

```
TYPE DAY OF WEEK =
(MONDAY,TUESDAY,WEDNESDAY,THURSDAY,
FRIDAY,SATURDAY,SUNDAY);
```

The statement is of the form:

'TYPE' identifier ' = ' brackets containing all elements of the new type.

After this the VAR section of the program can include something like 'VAR STRUCK:DAYOFWEEK;', which will allow the variable STRUCK to take on the value of the name of any day of the week.

The variable STRUCK can be used in Boolean expressions (IF STRUCK = MONDAY THEN etc.).

Each of the possible values of STRUCK is given a number by the computer, either from Ø to 6 or from 1 to 7, depending on the compiler. This means that the three standard functions ORD,SUCC and PRED can be used. ORD gives the ORDinal value of a DAYOFWEEK type variable, that is ORD(MONDAY) returns Ø (or possibly 1) and ORD(SUNDAY) returns 6 (or possibly 7).

PRED and SUCC give the previous day and next day from the day they take as their operand, that is PRED(SUNDAY) gives SATURDAY and SUCC(MONDAY) gives TUESDAY. PRED(MONDAY) and SUCC(SUNDAY) are not defined and lead to an error.

In a single TYPE statement you may need to define both a type called WEEK which takes values MONDAY...SUNDAY and another type WEEKDAY which takes values MONDAY...FRIDAY. The compiler will not allow this because an element (more than one in this case) appears in two separate types. The way around this is to define WEEKDAY to be a sort of subset of WEEK, called a 'subrange'. (The sort of type statement we are considering at the moment is called a scalar type, since each of the possible values is listed implicitly.)

Using subrange types, the above example becomes:

```
TYPE
  WEEK = (MON,TUE,WED,THU,FRI,SAT,SUN);
  WEEKDAY = MON ... FRI;
```

Subranges are defined by two constants which must be of the same type. The constants may be of type CHAR,INTEGER, or a 'scalar' type. You cannot make a subrange of type REAL.

Any variable of a subrange type can only take on the values between the two constants, inclusive.

A subrange can only have those operations carried out on it which can be carried out on its constants.

One other point pertaining to user defined data types: If you define a scalar type, and then make a variable of that type and assign a value to it, for instance DAY:=MONDAY, when you print DAY, the computer will not print 'MONDAY' but it may print the ORDinal value of 'MONDAY', if not it grinds to a halt with an error message.

The other related form of user defined variable type is the 'set'.

A set is a group of related pieces of information, for example the set of letters of the alphabet is

```
[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z]
```

While the set of letters in the word 'ALPHABET' is
[A,B,E,H,L,P,T]

If all the 'members' of one set are also members of another, then the first set is included in the second. If we call the first set above 'SET A' and the second 'SET B', then set B is included in set A, in Pascal notation this is A> =B. (This is read as 'A contains B'). By the same token, B is contained by A, or, in Pascal notation, B< =A.

The union of two sets is the set that contains all of the elements of both sets. Take two sets, C and D, where C=[1,2,3,4] and D=[4,5,6,7]. Then C union D (or in Pascal notation, C+D) is [1,2,3,4,5,6,7].

The intersection of two sets is that set which contains all the elements in both sets. Then C intersection D (C*D) is [4].

The 'relative union' (opposite of union) of two sets is the set of all those elements of the first set which are not in the second. Thus C relative union D (C-D) is [1,2,3].

Sometimes you will come across a set which has no elements, the empty set. This is written as [].

The final set operation is inclusion, 'IN'. This operation gives a Boolean result. The form of this operation is element 'IN' name of set, thus [2] IN C is true but [2] IN D is false.

Every different set used in a program must be declared in the TYPE statement, like this:
name of set '=' 'SET OF' all possible elements, with square brackets around them

For example :

```
TYPE
   DAY = SET OF
(MONDAY,TUESDAY,WEDNESDAY,THURSDAY,
FRIDAY, SATURDAY,SUNDAY);
```

VAR etc.

Any variable later declared to be of type 'DAY' then becomes a set which can have up to 7 elements (MONDAY to SUNDAY).

A set variable can be assigned to in the following way:
variable name ':=' expression.

The expression part is a mixture of other variable names, operators and literals, such as [MONDAY].

Chapter 13
User defined functions

Earlier it was noted that the trigonometric functions found in all Pascal implementations took or gave all angles in radians, and not in degrees. It is easy to get around this by defining a new set of trig. functions which work in degrees, called, say, SIND,COSD,TAND and ATND, where D stands for 'degree' and ATN for arctangent 'etc.'

We accomplish this in the following way:

```
(*possible type declaration goes here*)
(*variable declaration section of program goes
here*)
FUNCTION SIND(X:REAL):REAL;
  VAR
  Z : REAL;
BEGIN
  Z := X / 57.29578;
  SIND := SIN (Z)
END;
FUNCTION COSD(X:REAL):REAL;
VAR
  Z : REAL;
BEGIN
  Z := X / 57.29578;
  COSD := COS (Z)
```

```
END;
FUNCTION TAND(X:REAL):REAL;
VAR
  Z : REAL;
BEGIN
  Z := X / 57.29578;
  TAND := SIN (Z) / COS (Z)
END;
FUNCTION ATND(X:REAL):REAL;
VAR
  Z : REAL;
BEGIN
  Z := ARCTAN (X);
  ATND := Z / 57.29578
END;
BEGIN
(*and so on with the rest of the program*)
```

The four function sections work in exactly the same way, so I will only consider the SIND function declaration.

The function SIND will be called by including a line like 'A := SIND(23.45)'. Notice that it used in the same way as any other function – a number in brackets giving a result dependent on that number.

Taking the function declaration line by line:
(where the line 'FUNCTION SIND(....' is Line 1)
Line 1: The word 'FUNCTION' tells the computer that the next few lines are a 'function declaration'.

The word after 'FUNCTION' is the name of the function. It follows the normal rules of identifiers, that is it can only contain letters and numbers, and must start with a letter. The brackets which come next contain the name of the variable that the function is acting on, and the type of that variable. When the function is later called, the computer substitutes the value in brackets from where it was called for the variable you

specify in the function declaration. The word REAL after the closing bracket defines the type of the result.
Note the presence of the semi colon.
Lines 2 & 3: declares a variable Z, which is a *local* variable. This means that Z cannot be used outside the function declaration, unless it is declared in the program's VAR section, in which case the variables are completely separate, and any changes to the value of Z outside, in the program do not affect the value of Z in a function. It's not really very sensible to use the same variable inside and outside, in functions, as it is rather confusing. Also *never* alter the value of a variable from the main program, inside the function.
Lines 4 to 7: This is the sub program which defines the function's operation. The variable SIND is the name of the function, and is the value returned by the function.

A function may, in fact, take more than one operand, that is more than one number is contained in the brackets of the function, separated by comma's.

The extra operands are specified by extra variable declarations in the brackets in the function heading. For example:

```
FUNCTION MAX(X:REAL;L,M:INTEGER):REAL;
```

Functions can also be 'recursive'. This is a term describing a vastly over-rated programming trick. Basically, it means defining a function to be itself, that is the function itself appears in the function declaration. For example consider a function to work out the factorial of any number, where the factorial of 'n' is defined as being n*(n – l)*(n – 2)*...2*1. In addition, the factorial of Ø is defined as being 1. Thus: (using the symbol ! to represent 'factorial').

0! = 1
1! = 1*0!
2! = 2*1!

.
.
.

n! = n*(n – 1)!

Here is a function declaration for FACTORIAL(N):

```
FUNCTION FACTORIAL(N:INTEGER):INTEGER;
BEGIN
IF N=Ø THEN FACTORIAL:=1 ELSE FACTORIAL:=N*FACTORIAL
END;
```

Note that even though the function calls itself, it does so a finite number of times, because of the presence of the 'IF' statement.

Most people will never use recursion, outside of contrived problems that force you to use it. As one of these contrived problems, write a program, or rather, a function declaration, using recursion, to add up all the integers from 1 to 1ØØ. Use the property that the sum of integers up to n will be n + (the sum of all the integers up to n – 1), and that the sum of all integers up to 1 is 1.

Another way of thinking of the process of recursion is to state that recursion is the 'splitting of a process into simpler and simpler parts, until the process is trivial to carry out'. Using the example of summing all the integers from 1 to 1ØØ, we are reducing the problem into the trivial task of summing all integers from 1 to 1.

It is always preferable to define functions to carry out complex operations, even if that operation is only carried out once, because even though you are not being saved any typing, when you come to look at the program later, it will mean much more to you to see a name than a string of hieroglyphics representing a formula.

For example, the calculation used above to calculate degrees from radians is a fairly trivial one, but if that step of calculation came up in a long involved formula, you would

almost certainly have difficulty picking it out from the rest of the equation.

Aim for the situation where every single assignment statement in your program is populated with function calls and variable names, so that not one figure or numerical operation occurs in it. This method of working will prevent you getting muddled, and will ease the process of finding and correcting errors in the program, since a complex equation occurring several times in a program would be a nuisance to correct in the event of an error, unless it was contained in a function declaration, where the equation would only have to be corrected once to remedy all occurrences of it in the entire program.

Chapter 14
User defined procedures

The last chapter was concerned with, effectively, constructing hibred functions to make up for those not supplied in the language, such as TAN().

Besides functions, the other main component of a program is the statement. Pascal also allows you to define hibred statements, called 'procedures'.

Procedures are defined in much the same way as functions, except they do not return a value as such, although they may alter the values of variables in the main program. In BASIC the equivalent of a procedure is the subroutine, the difference is that a subroutine in BASIC may only take as it's parameters (operands) the variables that the programmer has decided. For example there is a subroutine in my compiler which will take all the leading spaces from A$. This subroutine cannot be used for extracting the leading spaces off B$ without copying B$ to A$, but a procedure in Pascal will allow you to use a statement of the form 'STRIP(B$)', except strings are not valid in Pascal.

Here then is a sample procedure declaration, which will, whenever called, print the word 'FEMUR' the number of times that was the operand of the procedure.

```
(*CONST SECTION GOES HERE*)
(*TYPE DECLARATION GOES HERE*)
(*VAR DECLARATION GOES HERE*)
```

```
PROCEDURE FEMUR(TIMES:INTEGER);
VAR
  COUNTER : INTEGER;
BEGIN
  FOR COUNTER := 1 TO TIMES DO WRITELN ('FEMUR')
END;
(*FUNCTION DECLARATION GOES HERE*)
(*PROGRAM STARTS HERE*)
...
```

A call to this procedure takes the form of 'FEMUR(1Ø)', which will, in this case, write the word 1Ø times.

Again, a procedure may have any number of parameters, and if they are of different types, the procedure heading will look like this:

```
PROCEDURE WHAT(A:REAL;B:INTEGER);
```

The advice about making as many of your equations into functions that I gave earlier holds even more with procedures.

This time, aim for a situation where the main program only accounts for 1Ø% of the bulk of the whole program. In this way you will be able to write many, possibly trivial, procedures, which can be tested outside the environment in which they are finally to be used.

The other reason for this approach is that most systems allow you to save sections of a program separately on cassette or disk, which can then be loaded back into the computer without disturbing whatever is in the computer at the time. Thus you can build up a library of procedures to carry out operations you may need again in other programs.

That's all there is to Pascal, apart from records and file handling which is beyond the scope of this book, due to the varied methods various computer manufacturers, and others, have seen fit to implement file handling.

Chapter 15
The Pascal compiler

The Pascal compiler presented here will convert most Pascal programs into an equivalent BASIC program. There are some limitations however:

1. No user defined functions or procedures.
2. No user defined data types.
3. No sets.
4. No arrays/matrices.
5. No true integer arithmetic.
6. No numerical output formatting.
7. No file handling.
8. No nested 'IF-THEN' statements.
9. No more than one statement is allowed after the 'THEN' section of an 'IF' statement.
10. No spaces are allowed in expressions. Brackets can always be used to make up for this shortcoming.
11. The layout of VAR and CONST sections of a program is crucial, see example program.
12. The colon in a VAR section, the equals sign in a CONST section and the ':=' symbols in an assignment statement must all be flanked by spaces.
13. Very little syntax checking is carried out – you can write an incorrect Pascal program, and the compiler will not tell you, but the error routines in the BASIC interpreter which executes the compiled BASIC program should do.
14. The compiler outputs to the screen only, so you need a

screen editor to be able to execute the output BASIC. You do this by either pressing return over each line or by COPYing each line, as in the BBC computer.

15. No case statement.

16. Variable names are limited to those accepted by the version of BASIC you are using.

The program was written and tested on a BBC Model B micro-computer, but only standard microsoft BASIC has been used throughout.

Notice BEGIN and END are obligatory in WHILE and FOR loops.

The area from lines 1Ø to 999 is left free for the data statements defining the Pascal program and for the BASIC program to reside in. It is suggested that lines 5ØØ to 999 are used for the data statements, since the output BASIC program is renumbered to reside from line 1Ø onwards.

The compiler itself resides from lines 1ØØØ to 487Ø.

Each of the major routines is headed by at least three REM statements, mostly containing asterisks, detailing what that section does. The more obtuse routines are also REM'd line by line.

To go through the program routine by routine:

The routine starting at line 1Ø6Ø makes space for the variables used in the Pascal program by dimensioning two string arrays, one NA$ will contain all the variables names and the other, TY$, contains all the variables' types. NP is the counter which decides which element of NA$ and TY$ will be filled next.

C3 is a variable used to make certain subroutines behave differently depending on from where they are called. It either has the value Ø or 1. The routine then calls the subroutine at line 124Ø, which reads in the Pascal program to string array PA$. After printing a heading, space is set up in the string array BA$ to contain the BASIC program. Both the index of this DIM statement and the one at line 1Ø6Ø will have to be changed for larger programs. The array which holds the Pascal program is dimensioned in the subroutine at line 124Ø. LI and LP are initialized to be one. They are the pointers into the BASIC and Pascal arrays, respectively.

Next the first sentence in the Pascal array is acted upon. The subroutine at line 148Ø just extracts the first word of A$, up to the first space (or, if C3 = 1, up to the first opening bracket). Various routines are called in response to what that first word is, and if that word is illegal, an appropriate error message is output. All these routines will increment LP as much as necessary, and return to line 113Ø for the next word to be compiled.

The routine starting at line 124Ø reads the Pascal program into PA$() until the word 'END.' is found, which is gauged to be the end of the program. Variable LE is set to the length of the program, in lines, but this variable is not actually used in this revision of the program.

The routine at line 136Ø takes all the leading spaces off A$, returning the residue in A$.

The routine at line 142Ø takes all the trailing spaces off A$, leaving the residue in A$.

The routine at line 148Ø splits A$ into two parts, that part before the first space(s) and that part after, the first part is returned in A$ and the second in B$. In addition, if variable C3 is equal to one, an opening bracket is treated in the same way as a space. This feature is only used by the READ and WRITE routines, to make sure that it is not obligatory to leave a space before the opening brackets of their arguments.

The routine at 161Ø is the CONST routine, that is it is that section of code called by the section at line 1Ø6Ø when the keyword CONST is found in a program.

First it increments the Pascal pointer, past the word CONST (line 164Ø) then the next Pascal sentence is copied into A$. Subroutine 148Ø is called, which brings into A$ the line up to the first space, and the first space should be just before the equals sign, so A$ contains the name for that CONSTant, and B$ contains the equals sign onwards.

It may not be an identifier however, it could be the beginning of the VAR section, or something. If so the program goes back to line 113Ø to deal with it.

Otherwise the identifier is put into the current BASIC line, and into the list of variable names. The residue from the last

call to 148Ø is then copied in to A$ and the routine called again, which cuts off the equals sign, which is followed by a space so that B$ contains the value to be assigned to this constant. It is then copied into A$. The possible semi-colon is cut off from A$, and a call to 136Ø made, to cut off any spurious spaces at the beginning and a call to line 142Ø to cut off any spurious spaces at the end of A$. The program then determines whether the type of the constant is CHAR, by looking to see if there are any quote signs in the value, and if so, it puts the word CHAR into the array of variable types. Then, again if it is a CHAR type, quotes are changed to double and a dollar sign is added. The pointer into NA$ and TY$ is then incremented in readiness for the next encounter with a variable. Then an equals sign and the value of the constant is added to the BASIC array, before the BASIC pointer is incremented. The routine then goes back to the start, repeatedly until an illegal identifier (CONSTant name) is found.

The routine at line 186Ø acts whenever a VAR section is encountered. The operation of this section is almost identical to that of the CONST section, except that nothing is put into the BASIC program, only into the table of variable names and types. Also, variable names are separated from their types by a colon, and not by an equals sign.

The BEGIN section at line 207Ø marks the beginning of the Pascal program proper, and refers to the BEGIN found at the beginning of every program, not to the BEGIN found at the beginning of a FOR or WHILE loop.

The Pascal pointer is first incremented, and the flag E set to Ø. If E is found to be one, it means that the program has come to an end, because 'END.' was encountered. The first statement after the BEGIN is then extracted at line 212Ø and the three routines at 148Ø, 136Ø and 142Ø are called to extract the first word of this statement. This first word is copied into S$ and the routine at line 254Ø called. This routine compiles all the statements.

When this routine returns, the END flag is checked, and if the program has not ended, it goes back for a new statement to be compiled. If it has finished, the entire BASIC array is

printed out, and the compiler stops. This section will have to be changed by those who want to do more than just have the program printed out. For instance, APPLE or BBC owners can set up an EXEC file with the BASIC program in it, and get the computer to enter it into memory for them. If the output program is entered before the compiler, i.e. at line numbers less than 1000, it can be easily modified, and will run faster than if it was located above the compiler.

The expression evaluator at line 2230 will convert any Pascal expression in A$ to an equivalent BASIC expression in A$. This means that all CHAR type variable names are supplemented by a dollar sign, and functions are converted – see 2470 to 2510.

The section at line 2540 is the section that compiles any statement, where the statement is held in S$ and LP points to that statement in the Pascal array.

First any semi colons are removed and then tests are made to S$ to determine what statement it is. The relevant routines are gone to, and if no match to a statement is found, it must be an assignment. In this case, S$ contains the variable name that the assignment assigns to. The GOSUB 4810 at line 2710 adds a dollar sign to the variable name if it is of type CHAR.

The BASIC string is then updated to contain the variable name and an equals sign. The current Pascal line is then got into A$, and is manipulated until A$ only contains the expression to be evaluated, line 2800. The expression evaluator is then called, and the resulting expression added to the end of the current BASIC line. All pointers are then incremented, and the subroutine is returned from.

Most of the following routines are called by the statement routine in response to what each statement is.

The BEGIN routine at line 2860 refers to the BEGIN used inside programs, not the BEGIN used before every program, to show where it starts.

All it does is insert the words 'REM BEGIN' into the current BASIC line, and then increment all the pointers before returning.

The 'IF' routine at line 2930 is probably the most difficult

routine to understand in the entire program. It is however very well commented, so I suggest you study the REM statements.

The REPEAT segment at line 3120 inserts the words 'REM REPEAT' into the BASIC program, increments all pointers, and returns.

The 'WHILE' segment at line 3190 evaluates the expression in the while statement, and inserts *two* lines into the BASIC array. The first takes the form 'IF NOT ('expression') THEN GOTO', where the line number is inserted later on, and the second takes the form 'REM WHILE'. Then all pointers are incremented, the BASIC pointer by two, since two lines were inserted, and control returns to the calling segment.

The 'FOR' segment at line 3320 changes the upper and lower limits to BASIC by calling the expression evaluator, and if 'DOWNTO' is used adds 'STEP −1' to the BASIC string.

The 'END' segment at line 3560 does not mean the 'END.' found at the end of all Pascal programs, but the 'END' found at the end of FOR loops and WHILE loops.

The 'END' statement section first has to decide whether this 'END' statement belongs to a 'WHILE' loop or a FOR loop. It does this by looping backwards through the BASIC program and incrementing a variable (EN) whenever 'REM END' is encountered and decrementing it whenever 'REM BEGIN' is encountered. Thus if 'REM BEGIN' is ever encountered when EN is equal to it's initial value that is the BEGIN statement which matches the current END loop. If the statement before this BEGIN is a FOR statement, the program jumps to line 3730, and if it is 'REM WHILE' the program jumps to line 3780. If it is not either of these, an error message is output.

In the case of FOR, the current BASIC line is set to 'NEXT' and the next BASIC line is set to 'REM END', so that if another loop is come across, the mechanism for matching 'BEGIN's works. Then all pointers are incremented as necessary and the program RETURNs. In the case of a 'WHILE' loop, the BASIC line before the 'REM BEGIN' (which is an 'IF' statement) has the line number of the current line plus ten inserted after the word GOTO. The current BASIC line is set to 'GOTO (the line before the 'REM BEGIN')'. The next BASIC line is set to 'REM END' as in the 'FOR' section.

Then all pointers are incremented as necessary, and the program returns.

The 'UNTIL' segment at line 384Ø translates the expression following the word 'UNTIL' and incorporates it into the BASIC program as follows:

'IF NOT (expression) THEN GOTO'

Then the program searches back for a matching 'REPEAT' statement, in the same way as the 'END' statement looks back for a 'BEGIN'.

When it's found the 'REM REPEAT' statement, it sets the end of the current BASIC line to be the line number of that line. Then all pointers are incremented, and the routine returns.

The READ segment at line 4Ø7Ø gives an error message if 'READ' was used, since 'READ' does not have a corresponding BASIC statement, but READLN does.

This section replaces the word 'READLN' with 'INPUT' and checks all variables to see if they are CHAR type, and if so adds a dollar sign. It also sets C3 to 1, so that you don't need a space before the READLN statement's brackets.

The WRITE/LN part works in the same way as the READ section, except that it also calls the expression evaluator, so that WRITE can also print results directly.

The END. segment at line 475Ø sets the end flag (E) to 1, to signify the end of the program, and adds the word 'END' to the BASIC program.

The subroutine at line 481Ø adds a dollar sign to the end of the variable in S$ if it is necessary, and then returns.

The program is exceptionally well commented, and anyone with a working knowledge of BASIC should have absolutely no difficulty working out it's operation. Being able to understand this program also gives you an invaluable insight into Pascal as well.

Here is a full description of the facilities offered by the current version of the compiler:

RESERVED WORDS:

AND – Only allowed if your version of BASIC allows it.

BEGIN – Used to show where a program starts, and where WHILE and FOR loops start.

CONST – Used to define CONSTants. In fact the compiler allows you to change the values of CONSTants!

DIV – Integer division. Only allowed on the BBC version, where DIV is valid BASIC anyway.

DO – Used as part of FOR and WHILE loops.

DOWNTO – Used in FOR loops.

ELSE – Not allowed.

END – Shows end of program and end of FOR and WHILE loops.

FOR – Used as part of FOR loop.

IF – Part of IF statement.

MOD – Only allowed on the BBC version, where MOD is valid anyway.

NOT – Only allowed if your version of BASIC allows it.

OR – Only allowed if your version of BASIC allows it.

REPEAT – Part of REPEAT – UNTIL loop.

THEN – Part of IF statement.

TO – Part of FOR loop.

UNTIL – Part of REPEAT – UNTIL loop.

VAR – Defines start of variable declaration section of program.

WHILE – part of WHILE loop.

Standard constants:

FALSE

TRUE

Standard types:

INTEGER

BOOLEAN

REAL

CHAR

Standard functions:
ABS
ARCTAN
CHR
COS
EXP
LN
ODD
ORD
PRED
ROUND
SIN
SQR
SQRT
SUCC
TRUNC

Plus there are a few features of Pascal not implemented which won't be obvious from the above tables:
An IF statement may only be followed by one statement, and the rest of the points outlined at the start of the chapter.

```
>LIST
  500 DATA CONST
  510 DATA W = 'CONSTANT MESSAGE';
  520 DATA P = 3.1415926;
  530 DATA Q = 5;
  540 DATA TR = TRUE;
  550 DATA VAR
  560 DATA XR : REAL;
  570 DATA YR : REAL;
  580 DATA XI : INTEGER;
  590 DATA YI : INTEGER;
  600 DATA XC : CHAR;
  610 DATA YC : CHAR;
  620 DATA XB : BOOLEAN;
  630 DATA YB : BOOLEAN;
  640 DATA BEGIN
  650 DATA"WRITELN (W,P,Q);"
  660 DATA IF TR THEN WRITE ('YES ');
  670 DATA IF TR=TRUE THEN WRITELN ('ALS
O');
  680 DATA XR := 23.12;
  690 DATA YR := 12.34;
  700 DATA XB := TRUE;
  710 DATA YB := FALSE;
  720 DATA XC := '#';
  730 DATA YC := CHR(65);
  740 DATA XI := 12;
  750 DATA YI := 32000;
  760 DATA IF (XB)AND(YB)OR(3=4) THEN WR
ITELN ('TRUE');
  770 DATA FOR XR := 0.0 TO 10.0 DO
  780 DATA BEGIN
  790 DATA WRITE (XR)
  800 DATA END;
  810 DATA FOR XR := 10.0 DOWNTO 0.0 DO
  820 DATA BEGIN
  830 DATA WRITE (XR)
  840 DATA END;
  850 DATA REPEAT
  860 DATA WRITE (XI);
  870 DATA XI := SUCC(XI)
```

```
  880 DATA UNTIL XI=15;
  890 DATA WHILE YI>31990 DO
  900 DATA BEGIN
  910 DATA WRITE (YI);
  920 DATA YI := YI-2
  930 DATA END;
  940 DATA XR := ABS(-2.9);
  950 DATA IF ODD(3)=TRUE THEN WRITELN('
3 IS ODD!!!');
  960 DATA * PRINT "THIS IS BASIC"
  970 DATA END.
 1000 REM ***************************
 1010 REM This Pascal compiler is
 1020 REM COPYRIGHT (C) Jeremy Ruston,
 1030 REM and may not be reproduced
 1040 REM without express permission.
 1050 REM ***************************
 1051 REM *** IT IS NOT MENTIONED IN THE
 TEXT,***
 1052 REM *** BUT BASIC COMMANDS CAN BE
PUT ***
 1053 REM *** INTO THE PASCAL PROGRAM, I
F THEY ***
 1054 REM *** ARE PRECEDED BY AN ASTERIS
K AND A SPACE ***
 1055 REM *** THIS ALLOWS YOU TO USE FIL
E ***
 1056 REM *** HANDLING COMMANDS AND GRAP
HICS ***
 1057 REM *** COMMANDS IN PASCAL ***
 1058 REM *** SEE EXAMPLE PROGRAM ***
 1059 REM ***************************
 1060 DIM NA$(30),TY$(30)
 1070 NP=1
 1075 C3=0
 1080 GOSUB 1240 :  REM *** GET PASCAL *
**
 1090 PRINT:PRINT:PRINT:PRINT" -- BASIC
---":PRINT
 1100 DIM BA$(60) : REM *** MAKE SPACE F
OR BASIC ***
```

```
 1101 REM *** DEFINE FUNCTIONS ***
 1102 BA$(1)="DEF FNOD(X)=((X-INT(X/2)*2
)=1)"
 1103 BA$(2)="DEF FNPR(X)=X-1"
 1104 BA$(3)="DEF FNSU(X)=X+1"
 1105 BA$(4)="DEF FNRO(X)=INT(X+0.5)"
 1106 BA$(5)="DEF FNSQ(X)=X*X"
 1110 LI=6 :          REM *** CURRENT BASI
C LINE IN ARRAY ***
 1120 LP=1           : REM *** CURRENT PAS
CAL LINE ***
 1130 P$=PA$(LP) :  REM *** GET PASCAL L
INE ***
 1140 A$=P$
 1150 GOSUB 1480 :  REM *** GET FIRST WO
RD OF PASCAL LINE ***
 1160 IF A$="CONST" THEN GOTO 1610
 1170 IF A$="TYPE" THEN PRINT "ERROR - N
o structured types!":STOP
 1180 IF A$="VAR" THEN GOTO 1860
 1190 IF A$="PROCEDURE" THEN PRINT "ERRO
R - No procedures!":STOP
 1200 IF A$="FUNCTION" THEN PRINT "ERROR
 - No functions!":STOP
 1210 IF A$="BEGIN" THEN GOTO 2070
 1220 IF A$="PROGRAM" THEN PRINT "ERROR
- 'program' not needed!":STOP
 1230 PRINT "ERROR - I don't recognize "
;A$;"!!!!":STOP
 1240 REM **********************************
**********
 1250 REM **** READ IN PASCAL, TO PA$(X)
      ****
 1260 REM **********************************
**********
 1270 PRINT:PRINT:PRINT:PRINT" --- PASCAL
 ---":PRINT
 1280 DIM PA$(60)
 1290 LE=1
 1300 READ A$
 1310 PRINT,A$
```

```
 1320 PA$(LE)=A$
 1330 IF A$="END." THEN RETURN
 1340 LE=LE+1
 1350 GOTO 1300
 1360 REM ********************************
**********
 1370 REM **** TAKE LEADING SPACES OFF A
$     ****
 1380 REM ********************************
**********
 1390 IF LEN(A$)=0 OR A$=" " THEN RETURN
 1400 IF LEFT$(A$,1)=" " THEN A$=MID$(A$
,2):GOTO 1390
 1410 RETURN
 1420 REM ********************************
**********
 1430 REM **** TAKE TRAILING SPACES OFF
A$    ****
 1440 REM ********************************
**********
 1450 IF LEN(A$)=0 OR A$=" " THEN RETURN
 1460 IF RIGHT$(A$,1)=" " THEN A$=LEFT$(
A$,LEN(A$)-1):GOTO 1450
 1470 RETURN
 1480 REM ********************************
**********
 1490 REM **** GET FIRST WORD OF A$, IN
A$    ****
 1500 REM **** REST OF WORD IN B$
      ****
 1510 REM ********************************
**********
 1520 GOSUB 1360
 1530 WO$=""
 1540 IF A$="" THEN GOSUB 1360:B$=A$:A$=
WO$:RETURN
 1550 ST$=LEFT$(A$,1)
 1560 IF ST$=" " THEN GOSUB 1360:B$=A$:A
$=WO$:RETURN
 1561 IF C3=1 AND ST$="(" THEN GOSUB 136
0:B$=A$:A$=WO$:RETURN
```

```
 1570 WO$=WO$+ST$
 1580 IF LEN(A$)=1 THEN A$="":GOTO 1540
 1590 A$=MID$(A$,2)
 1600 GOTO 1550
 1610 REM ********************
 1620 REM *** CONST      ***
 1630 REM ********************
 1640 LP=LP+1:REM *** INCREMENT PASCAL P
OINTER ***
 1650 A$=PA$(LP):REM *** COLLECT NEXT PA
SCAL LINE ***
 1660 GOSUB 1480:REM *** GET IDENTIFIER
***
 1670 REM *** IF NOT AN IDENTIFIER, RETU
RN ***
 1680 IF A$="BEGIN" OR A$="CONST" OR A$=
"VAR" THEN GOTO 1130
 1690 IF A$="PROCEDURE" OR A$="FUNCTION"
 OR A$="TYPE" THEN GOTO 1130
 1700 BA$(LI)=A$:REM *** PUT IDENTIFIER
IN BASIC ***
 1710 NA$(NP)=A$:REM *** AND IN VAR. TAB
LE ***
 1720 A$=B$:REM *** GET EQUALS SIGN ***
 1730 GOSUB 1480
 1740 A$=B$:REM *** GET TYPE FOR IDENTIF
IER ***
 1750 IF RIGHT$(A$,1)=";" THEN A$=LEFT$(
A$,LEN(A$)-1):REM *** NO SEMI COLON ***
 1760 GOSUB 1360:REM *** REMOVE SPURIOUS
 SPACES ***
 1770 GOSUB 1420
 1780 IF LEFT$(A$,1)="'" THEN TY$(NP)="C
HAR"
 1790 REM *** IF A CHAR TYPE, ADD QUOTES
 AND DOLLAR SIGN ***
 1800 IF LEFT$(A$,1)="'" THEN BA$(LI)=BA
$(LI)+"$"
 1810 IF LEFT$(A$,1)="'" THEN A$=CHR$(34
)+MID$(A$,2,LEN(A$)-2)+CHR$(34)
 1820 NP=NP+1
```

```
 1825 GOSUB 2230
 1830 BA$(LI)=BA$(LI)+"="+A$:REM *** UPD
ATE BASIC ***
 1840 LI=LI+1:REM *** NEXT BASIC LINE **
*
 1850 GOTO 1640:REM *** GO BACK FOR MORE
 ***
 1860 REM ******************
 1870 REM *** VAR         ***
 1880 REM ******************
 1890 LP=LP+1: REM *** INCREMEMT PASCAL
POINTER ***
 1900 A$=PA$(LP):REM *** GET PASCAL LINE
 ***
 1910 GOSUB 1480:REM *** GET VARIABLE NA
ME ***
 1920 IF A$="PROCEDURE" OR A$="BEGIN" OR
 A$="TYPE" OR A$="PROGRAM" THEN GOTO 113
0
 1930 IF A$="FUNCTION" OR A$="CONST" THE
N GOTO 1130
 1940 GOSUB 1360:GOSUB 1420
 1950 NA$(NP)=A$:REM *** SAVE NAME ***
 1960 A$=B$
 1970 GOSUB 1480
 1980 A$=B$
 1990 GOSUB 1480
 2000 GOSUB 1360
 2010 GOSUB 1420
 2020 IF RIGHT$(A$,1)=";" THEN A$=LEFT$(
A$,LEN(A$)-1):REM *** NO SEMI COLON ***
 2030 GOSUB 1360:GOSUB 1420
 2040 TY$(NP)=A$
 2050 NP=NP+1
 2060 GOTO 1890
 2070 REM ******************
 2080 REM *** BEGIN     ***
 2090 REM ******************
 2100 LP=LP+1
 2110 E=0
 2120 A$=PA$(LP)
```

```
2130 GOSUB 1480
2140 GOSUB 1360
2150 GOSUB 1420
2160 S$=A$
2170 GOSUB 2540
2180 IF E<>1 THEN GOTO 2110
2190 FOR T=1 TO LI
2200 PRINT T*10;" ";BA$(T)
2210 NEXT T
2220 END
2230 REM **************************
2240 REM *** EXPRESSION       ***
2250 REM *** EVALUATOR        ***
2260 REM **************************
2270 W$=""
2280 ST=1
2290 S$=MID$(A$,ST,1)
2300 IF S$>="A" AND S$<="Z" THEN GOTO 2
360
2310 IF S$="'" THEN S$=CHR$(34)
2320 W$=W$+S$
2330 ST=ST+1
2340 IF ST>LEN(A$) THEN A$=W$:RETURN
2350 GOTO 2290
2360 IF ST>LEN(A$) THEN GOTO 2440
2370 X$=""
2380 S$=MID$(A$,ST,1)
2390 IF S$>"Z" OR S$<"A" THEN GOTO 2440
2400 X$=X$+S$
2410 ST=ST+1
2420 IF ST>LEN(A$) THEN GOTO 2440
2430 GOTO 2380
2440 S$=X$
2450 GOSUB 4810
2460 X$=S$
2470 IF X$="ARCTAN" THEN X$="ATN"
2471 IF X$="ODD" THEN X$="FNOD"
2472 IF X$="PRED" THEN X$="FNPR"
2473 IF X$="SUCC" THEN X$="FNSU"
2474 IF X$="ROUND" THEN X$="FNRO"
2475 IF X$="SQR" THEN X$="FNSQ"
```

```
 2480 IF X$="TRUNC" THEN X$="INT"
 2490 IF X$="CHR" THEN X$="CHR$"
 2500 IF X$="ORD" THEN X$="ASC"
 2510 IF X$="SQRT" THEN X$="SQR"
 2511 IF X$="FALSE" THEN X$="0":REM INSE
RT IN THE QUOTES WHATEVER YOUR
 2512 REM COMPUTER OUTPUTS TO 'PRINT (3=
2)'.
 2513 IF X$="TRUE" THEN X$="-1":REM INSE
RT IN THE QUOTES WHATEVER YOUR
 2514 REM COMPUTER OUTPUTS TO 'PRINT (2=
2)'.
 2520 W$=W$+X$
 2530 GOTO 2290
 2540 REM ********************
 2550 REM *** STATEMENT ***
 2560 REM ********************
 2570 REM *** STATEMENT - HELD IN S$ ***
 2580 IF RIGHT$(S$,1)=";" THEN S$=LEFT$(
S$,LEN(S$)-1)
 2590 IF S$="BEGIN" THEN GOTO 2860
 2600 IF S$="IF" THEN GOTO 2930
 2610 IF S$="REPEAT" THEN GOTO 3120
 2620 IF S$="WHILE" THEN GOTO 3190
 2630 IF S$="FOR" THEN GOTO 3320
 2640 IF S$="END" THEN GOTO 3560
 2650 IF S$="UNTIL" THEN GOTO 3840
 2660 IF LEFT$(S$,4)="READ" THEN GOTO 40
70
 2661 IF LEFT$(S$,5)="WRITE" THEN GOTO 4
400
 2662 IF S$<>"*" THEN GOTO 2680
 2663 A$=PA$(LP)
 2664 GOSUB 1480
 2665 BA$(LI)=B$
 2666 LI=LI+1
 2667 LP=LP+1
 2668 RETURN
 2680 IF S$="END." THEN GOTO 4750
 2690 REM *** S$ IS NOW NOT ANYTHING OBV
TOUS ***
```

```
 2700 REM *** CAN ONLY BE AN EXPRESSION
OR A PROCEDURE CALL ***
 2710 GOSUB 4810:REM *** GET POSSIBLE DO
LLAR
 2720 BA$(LI)=S$+" = "
 2730 A$=PA$(LP)
 2740 GOSUB 1480
 2750 A$=B$
 2760 GOSUB 1480
 2770 A$=B$
 2780 IF RIGHT$(A$,1)=";" THEN A$=LEFT$(
A$,LEN(A$)-1)
 2790 GOSUB 1360
 2800 GOSUB 1420
 2810 GOSUB 2230
 2820 BA$(LI)=BA$(LI)+A$
 2830 LI=LI+1
 2840 LP=LP+1
 2850 RETURN
 2860 REM **********************
 2870 REM ***  BEGIN          ***
 2880 REM **********************
 2890 BA$(LI)="REM BEGIN"
 2900 LI=LI+1
 2910 LP=LP+1
 2920 RETURN
 2930 REM **********************
 2940 REM ***  IF             ***
 2950 REM **********************
 2960 A$=PA$(LP):REM.....................
.COPY CURRENT PASCAL LINE
 2970 GOSUB 1480:REM.....................
.GET 'IF' PART
 2980 A$=B$:REM..........................
.COPY REST TO A$
 2990 GOSUB 1480:REM.....................
.STRIP OFF EXPRESSION
 3000 GOSUB 2230:REM.....................
.EVALUATE EXPRESSION
 3010 BA$(LI)="IF "+A$+" THEN ":REM.....
.PUT INTO BASIC
```

```
 3020 W3$=BA$(LI):REM.....................
.SAVE BASIC LINE SO FAR
 3030 A$=B$:REM...........................
.'THEN' ONWARDS IN A$
 3040 GOSUB 1480:REM......................
.STRIP OFF 'THEN'
 3050 PA$(LP)=B$
 3060 A$=B$
 3070 GOSUB 1480
 3080 S$=A$
 3090 GOSUB 2540
 3100 BA$(LI-1)=W3$+BA$(LI-1)
 3110 RETURN
 3120 REM ************************
 3130 REM ***  REPEAT          ***
 3140 REM ************************
 3150 BA$(LI)="REM REPEAT"
 3160 LI=LI+1
 3170 LP=LP+1
 3180 RETURN
 3190 REM ************************
 3200 REM ***  WHILE           ***
 3210 REM ************************
 3220 A$=PA$(LP):REM *** GET PASCAL LINE
 ***
 3230 GOSUB 1480:REM *** EXTRACT 'WHILE'
 ***
 3240 A$=B$
 3250 GOSUB 1480:REM *** GET EXPRESSION
IN A$ ***
 3260 GOSUB 2230:REM *** EVALUATE A$ ***
 3270 BA$(LI)="IF NOT("+A$+") THEN GOTO
"
 3280 BA$(LI+1)="REM WHILE"
 3290 LI=LI+2
 3300 LP=LP+1
 3310 RETURN
 3320 REM ************************
 3330 REM ***  FOR             ***
 3340 REM ************************
 3350 A$=PA$(LP):REM *** GET CURRENT LIN
```

```
E ***
 3360 GOSUB1480:REM *** EXTRACT 'FOR' **
*
 3370 A$=B$
 3380 GOSUB 1480:REM *** EXTRACT CONTROL
 VARIABLE ***
 3390 BA$(LI)="FOR "+A$+"=":REM *** UPDA
TE BASIC ***
 3400 A$=B$
 3410 GOSUB 1480:REM *** EXTRACT EQUALS
SIGN ***
 3420 A$=B$
 3430 GOSUB 1480:REM *** EXTRACT LOWER L
IMIT ***
 3440 GOSUB 2230:REM *** EVALUATE LOWER
LIMIT ***
 3450 BA$(LI)=BA$(LI)+A$+" TO "
 3460 A$=B$
 3470 GOSUB 1480:REM *** EXTRACT 'TO'/'D
OWNTO' ***
 3480 S8$=A$
 3490 A$=B$
 3500 GOSUB 1480:REM *** EXTRACT UPPER L
IMIT ***
 3510 GOSUB 2230:REM *** EVALUATE IT ***
 3520 BA$(LI)=BA$(LI)+A$
 3530 IF S8$="DOWNTO" THEN BA$(LI)=BA$(L
I)+" STEP -1"
 3540 LP=LP+1:LI=LI+1
 3550 RETURN
 3560 REM **********************
 3570 REM ***  END             ***
 3580 REM **********************
 3590 EN=0
 3600 BA=LI-1
 3610 REM *** FIND MATCHING BEGIN ***
 3620 IF BA$(BA)="REM BEGIN" AND EN=0 TH
EN GOTO 3680
 3630 IF BA$(BA)="REM BEGIN" THEN EN=EN-
1
 3640 IF BA$(BA)="REM END" THEN EN=EN+1
```

```
 3650 BA=BA-1
 3660 IF BA>0 THEN GOTO 3620
 3670 PRINT "ERROR 'BEGIN-END' GONE WRON
G !!!";:STOP
 3680 A$=BA$(BA-1)
 3690 GOSUB 1480:REM *** A$=FIRST WORD B
EFORE ***
 3700 IF A$="FOR" THEN GOTO 3730
 3710 IF BA$(BA-1)="REM WHILE" THEN GOTO
 3780
 3720 GOTO 3670
 3730 BA$(LI)="NEXT"
 3740 BA$(LI+1)="REM END"
 3750 LI=LI+2
 3760 LP=LP+1
 3770 RETURN
 3780 BA$(BA-2)=BA$(BA-2)+STR$(LI*10+10)
 3790 BA$(LI)="GOTO "+STR$((BA-2)*10)
 3800 BA$(LI+1)="REM END"
 3810 LI=LI+2
 3820 LP=LP+1
 3830 RETURN
 3840 REM **********************
 3850 REM ***  UNTIL          ***
 3860 REM **********************
 3870 A$=PA$(LP):REM *** GET PASCAL LINE
 ***
 3880 GOSUB 1480
 3890 REM ***B$ NOW CONTAINS EXPRESSION
***
 3900 A$=B$
 3905 IF RIGHT$(A$,1)=";" THEN A$=LEFT$(
A$,LEN(A$)-1)
 3910 GOSUB 2230:REM *** EVALUATE EXPRES
SION ***
 3920 BA$(LI)="IF NOT("+A$+") THEN GOTO
 ":REM *** UPDATE BASIC ***
 3930 REM *** NOW SEARCH BACK FOR 'REPEA
T' ***
 3940 BA=LI
 3950 RE=0
```

```
 3960 IF BA$(BA)="REM REPEAT" AND RE=0 T
HEN GOTO 4020
 3970 IF BA$(BA)="REM UNTIL" THEN RE=RE+
1
 3980 IF BA$(BA)="REM REPEAT" THEN RE=RE
-1
 3990 BA=BA-1
 4000 IF BA>0 THEN GOTO 3960
 4010 PRINT "ERROR - REPEAT LOOP FAILED!
!!":STOP
 4020 BA$(LI)=BA$(LI)+STR$(BA*10)
 4030 BA$(LI+1)="REM UNTIL"
 4040 LI=LI+2
 4050 LP=LP+1
 4060 RETURN
 4070 REM **********************
 4080 REM *** READ            ***
 4090 REM **********************
 4100 IF S$="READ" THEN PRINT "ERROR - N
O READ ALLOWED!!!!":STOP
 4110 A$=PA$(LP)            :REM *** GET P
ASCAL LINE ***
 4115 C3=1
 4120GOSUB 1480             :REM *** GET AR
GUMENT ***
 4125 C3=0
 4130A$=B$                  :REM *** GET RE
ADY FOR SPACES ***
 4140GOSUB 1360             :REM *** GET RI
D OF LEADING & TRAILING SPACES  ***
 4150GOSUB 1420
 4160IF RIGHT$(A$,1)=";" THEN A$=LEFT$(A
$,LEN(A$)-1):REM *** NO SEMICOLON ***
 4170GOSUB 1360
 4180GOSUB 1420
 4190IF LEFT$(A$,1)="(" THEN A$=MID$(A$,
2):REM *** NO LEFT BRACKET ***
 4200IF RIGHT$(A$,1)=")" THEN A$=LEFT$(A
$,LEN(A$)-1):REM *** NO RIGHT BRACKET
 4210GOSUB 1360
 4220GOSUB 1420
```

```
 4230REM *** A$ CONTAINS ARG., NO BRACKE
TS OR SEMI-COLON ***
 4240E$=S$
 4250BA$(LI)="INPUT "
 4260ST=1
 4270S$=""
 4280IF ST>LEN(A$) THEN GOTO 4340
 4290D$=MID$(A$,ST,1)
 4300IF D$="," THEN GOTO 4340
 4310S$=S$+D$
 4320ST=ST+1
 4330GOTO 4280
 4340GOSUB 4810
 4350ST=ST+1
 4360BA$(LI)=BA$(LI)+S$
 4370 IF ST>LEN(A$) THEN LP=LP+1:LI=LI+1
:RETURN
 4380BA$(LI)=BA$(LI)+","
 4390GOTO 4270
 4400 REM **********************
 4410 REM *** WRITE          ***
 4420 REM **********************
 4430 A$=PA$(LP)             :REM *** GET P
ASCAL LINE ***
 4435 C3=1
 4440 GOSUB 1480             :REM *** GET A
RGUMENT ***
 4445 C3=0
 4450 A$=B$                  :REM *** GET R
EADY FOR SPACES ***
 4460 GOSUB 1360             :REM *** GET R
ID OF LEADING & TRAILING SPACES  ***
 4470 GOSUB 1420
 4480 IF RIGHT$(A$,1)=";" THEN A$=LEFT$(
A$,LEN(A$)-1):REM *** NO SEMICOLON ***
 4490 GOSUB 1360
 4500 GOSUB 1420
 4510 IF LEFT$(A$,1)="(" THEN A$=MID$(A$
,2):REM *** NO LEFT BRACKET ***
 4520 IF RIGHT$(A$,1)=")" THEN A$=LEFT$(
A$,LEN(A$)-1):REM *** NO RIGHT BRACKET
```

```
 4530 GOSUB 1360
 4540 GOSUB 1420
 4550 REM *** A$ CONTAINS ARG., NO BRACK
ETS OR SEMI-COLON ***
 4560 E$=S$
 4570 BA$(LI)="PRINT "
 4580 ST=1
 4590 S$=""
 4600 IF ST>LEN(A$) AND MID$(A$,ST-1,1)=
"'" THEN GOTO 4680
 4601 IF ST>LEN(A$) THEN GOTO 4670
 4610 D$=MID$(A$,ST,1)
 4615 IF MID$(A$,ST-1,1)="'" AND D$=","
THEN GOTO 4680
 4620 IF D$="," THEN GOTO 4670
 4630 IF D$="'" THEN D$=CHR$(34)
 4640 S$=S$+D$
 4650 ST=ST+1
 4660 GOTO 4600
 4670 WE=ST:G$=A$:A$=S$:GOSUB 2230:S$=A$
:A$=G$:ST=WE
 4680 ST=ST+1
 4690 BA$(LI)=BA$(LI)+S$
 4700 IF ST>LEN(A$) THEN LP=LP+1:LI=LI+1
:GOTO 4730
 4710 BA$(LI)=BA$(LI)+","
 4720 GOTO 4590
 4730 IF E$="WRITE" THEN BA$(LI-1)=BA$(L
I-1)+";"
 4740 RETURN
 4750 REM **********************
 4760 REM *** END.          ***
 4770 REM **********************
 4780 BA$(LI)="END"
 4790 E=1 : REM *** SET END FLAG ***
 4800 RETURN
 4810 REM **********************
 4820 REM *** CHECK S$ FOR VAR *
 4830 REM **********************
 4840 FOR C=1 TO NF
 4850 IF S$=NA$(C) THEN IF TY$(C)="CHAR"
```

```
 THEN S$=S$+"$"
 4860 NEXT C
 4870 RETURN
>REM There now follows a sample run
>
>
>
>
>
>ruRUN
 -- PASCAL ---
CONST
W = 'CONSTANT MESSAGE';
P = 3.1415926;
Q = 5;
TR = TRUE;
VAR
XR : REAL;
YR : REAL;
XI : INTEGER;
YI : INTEGER;
XC : CHAR;
YC : CHAR;
XB : BOOLEAN;
YB : BOOLEAN;
BEGIN
WRITELN (W,P,Q);
IF TR THEN WRITE ('YES ');
IF TR=TRUE THEN WRITELN ('ALSO');
XR := 23.12;
YR := 12.34;
XB := TRUE;
YB := FALSE;
XC := '#';
YC := CHR(65);
XI := 12;
YI := 32000;
IF (XB)AND(YB)OR(3=4) THEN WRITELN ('TRU
E');
FOR XR := 0.0 TO 10.0 DO
BEGIN
```

```
WRITE (XR)
END;
FOR XR := 10.0 DOWNTO 0.0 DO
BEGIN
WRITE (XR)
END;
REPEAT
WRITE (XI);
XI := SUCC(XI)
UNTIL XI=15;
WHILE YI>31990 DO
BEGIN
WRITE (YI);
YI := YI-2
END;
XR := ABS(-2.9);
IF ODD(3)=TRUE THEN WRITELN('3 IS ODD!!!
');
* PRINT "THIS IS BASIC"
END.
 -- BASIC ---
        10 DEF FNOD(X)=((X-INT(X/2)*2)=1
)
        20 DEF FNPR(X)=X-1
        30 DEF FNSU(X)=X+1
        40 DEF FNRO(X)=INT(X+0.5)
        50 DEF FNSQ(X)=X*X
        60 W$="CONSTANT MESSAGE"
        70 P=3.1415926
        80 Q=5
        90 TR=-1
       100 PRINT W$,P,Q
       110 IF TR THEN PRINT "YES ";
       120 IF TR=-1 THEN PRINT "ALSO"
       130 XR = 23.12
       140 YR = 12.34
       150 XB = -1
       160 YB = 0
       170 XC$ = "#"
       180 YC$ = CHR$(65)
       190 XI = 12
```

```
        200 YI = 32000
        210 IF (XB)AND(YB)OR(3=4) THEN PR
INT "TRUE"
        220 FOR XR=0.0 TO 10.0
        230 REM BEGIN
        240 PRINT XR;
        250 NEXT
        260 REM END
        270 FOR XR=10.0 TO 0.0 STEP -1
        280 REM BEGIN
        290 PRINT XR;
        300 NEXT
        310 REM END
        320 REM REPEAT
        330 PRINT XI;
        340 XI = FNSU(XI)
        350 IF NOT(XI=15) THEN GOTO 320
        360 REM UNTIL
        370 IF NOT(YI>31990) THEN GOTO 43
0
        380 REM WHILE
        390 REM BEGIN
        400 PRINT YI;
        410 YI = YI-2
        420 GOTO 370
        430 REM END
        440 XR = ABS(-2.9)
        450 IF FNOD(3)=-1 THEN PRINT "3 I
S ODD!!!"
        460 PRINT "THIS IS BASIC"
        470 END
>
```

Typeset and Printed by Commercial Colour Press, London E.7.

Another great book from

INTERFACE