AAEC/M105

AUSTRALIAN ATOMIC ENERGY COMMISSION

RESEARCH ESTABLISHMENT

LUCAS HEIGHTS RESEARCH LABORATORIES

AN INTRODUCTION TO PASCAL PROGRAMMING FOR NUMERICAL COMPUTATIONS

by

J.M. BARRY

MAY 1986

ISBN 0 642 59829 0

PREFACE

These notes arose out of a series of Summer Schools conducted by the AAEC for Higher School Certificate students who were about to enter their final school year. The approach adopted avoids formalism and introduces quickly to the students sufficient programming concepts to enable them to undertake scientific problem solving with the help of tutorial sessions.

The notes have been modified extensively so that the reader can work alone through this introduction, attempting exercises designed to build up programming and mathematical skills. The reader needs a knowledge of calculus to work through all the practice exercises; however, School Certificate mathematics is more than sufficient for understanding the expository sections.

The examples and the orientation of the presentation are very much mathematical problems. There is no attempt to develop computer games writing skills since, in the author's opinion, more than enough material is available elsewhere.

The Pascal programming language is very extensive and contains many more facets than are considered here. These features involve more intricate and interesting aspects of data structures, most of which are not necessary for numerical scientific problem solving. Their omission is completely intentional to keep the volume of material presented suitable for a basic first course in computational computing. The students who work successfully through these notes should be able to come to terms with the other concepts by extending their reading (for this the student is referred to Welsh and Elder [1979]).

The material presented here could also be programmed in other scientific computer languages such as FORTRAN or BASIC. The programming methodology with languages such as Pascal needed to achieve the same goal can be very different. Hopefully, the student who works through this presentation will be capable of thinking in a modern structured sense and not merely rewriting FORTRAN or BASIC concepts in Pascal.

National Library of Australia card number and ISBN 0 642 59829 0

CONTENTS

1. INTRODUCTION

Each digital computer is capable of obeying a number of basic instructions. These instructions vary for different computers, but they have several common attributes:

(a) The ability to perform the four arithmetic operations (addition, subtraction, multiplication and division).

(b) The ability to perform logical operations (is a $\geq$ b?).

(c) The ability to perform 'housekeeping' instructions (*e.g.* moving numbers from addressable memory to registers, where arithmetic and logical operations may be performed on them.

For a programmer to communicate with the computer at this most fundamental level, it is necessary to develop programs in the basic machine language of the specific computer. In the early days of computing, scientists and mathematicians had to concern themselves with the intricacies of binary coding. The long delays and inconvenience of this form of user-machine communication accelerated the introduction and growth of programming languages that the problem solver could use more readily. Many languages (FORTRAN, BASIC, COBOL, ALGOL, PL1, APL, ACL, Pascal, *etc.*) have been developed for scientific, commercial and other applications.

Pascal is a powerful modern computing language which is suited to most computing tasks. These notes consider a restricted subset of Pascal suitable for handling numerical computations arising in scientific calculations. In fact, they may well be thought of as a programming conversion guide for the scientific FORTRAN user.

There are no computers that obey programs written directly in Pascal. It is usually necessary for programs in 'high level' languages, such as Pascal, to be translated into an appropriate set of machine language instructions. This process is known as *compilation*.

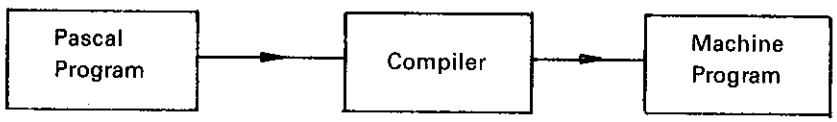


Figure 1 - Compilation of a Pascal program

The Pascal source statements are translated into a set of machine language instructions by a Pascal compiler (Figure 1). The compiler is itself a program usually supplied by the machine manufacturer. In these notes, the Pascal language described is appropriate for a compiler developed partially at Lucas Heights. (Cox *et al.* 1980). The compiler (Pascal 8000) first checks to ensure that the Pascal statements obey the 'rules' of the language (syntax analysis), then supplies a set of machine instructions that will implement the specifications of the programmer. When the compilation process is completed, the machine instructions generated may be executed. The finer details of this process are not of immediate concern, since the purpose here is primarily to use the computer as a tool for mathematical problem solving.

2. OVERVIEW OF PASCAL PROGRAMMING

Let us first consider the steps involved in solving a sample problem, and the Pascal program that could be developed to carry them out. When this is done, we shall examine the various Pascal statements in closer detail.

Problem If $30,000 is borrowed at a rate of 13% (monthly reducible) and repayments of $400 each month are made, then how many years will it take to repay the loan?

Before we can program a digital computer to solve a problem, it is necessary for us to be able to detail logically the steps that are needed to solve the problem, in much the same way as we would if we were going to tackle the problem with a desk calculating machine, slide rule, or pen and paper. Some people find it helpful to draw a flow chart (Figure 2) showing the steps involved, whereas others prefer to visualise all the steps in their mind.

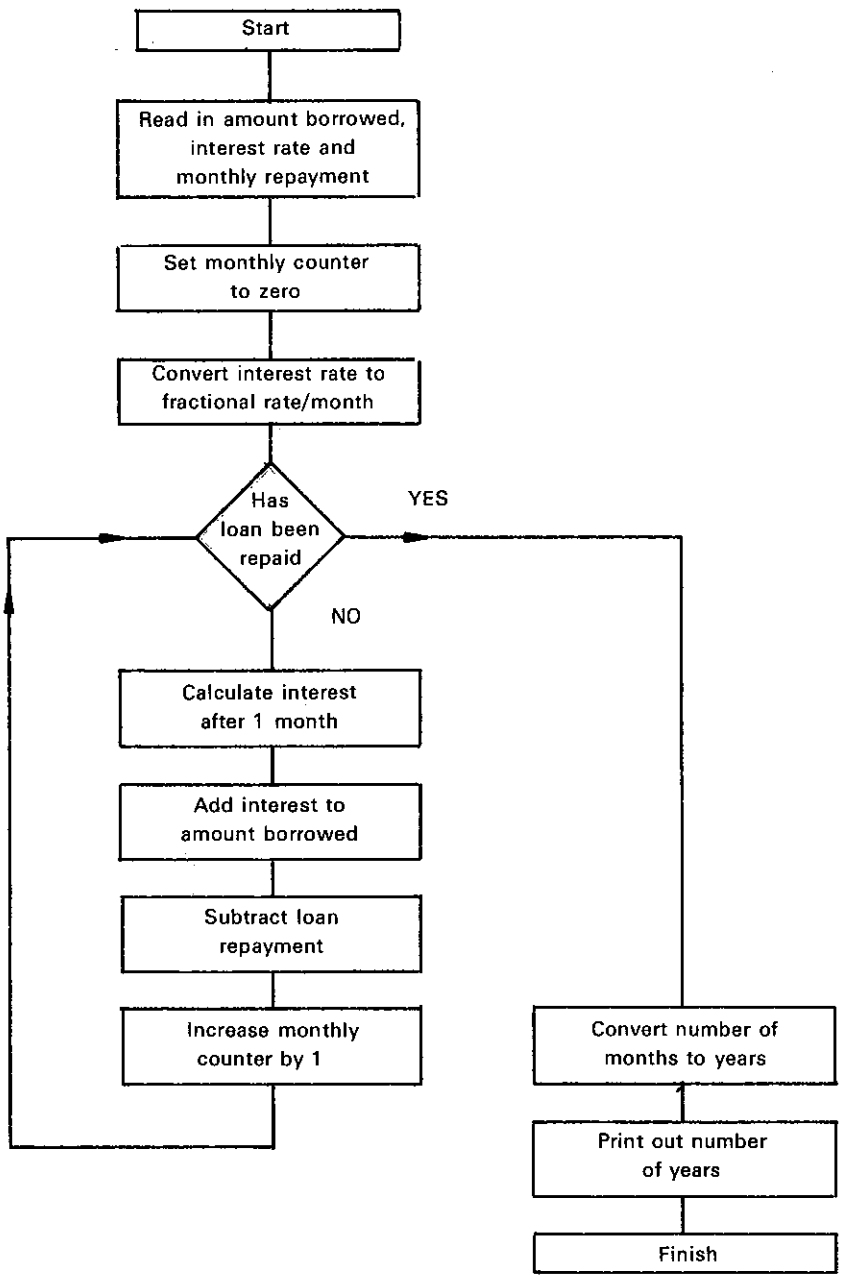


Figure 2 Flow chart for compound interest problem

In common with most flow diagrams, Figure 2 has a preliminary section in which certain items are *initialised*. The three data items, amount borrowed, interest rate and monthly repayment, are read from a data input record. The monthly counter is set to zero. The repetitive section follows and is executed, provided that the loan has not been repaid. The monthly interest is calculated and added to the amount borrowed. The loan repayment is subtracted and the monthly counter is increased by one. When the loan has been repaid, the number of years is calculated and the result printed out. The result most probably will involve a decimal portion.

From this flow chart, the following program can be coded. At this point we will not concern ourselves with the formal rules for coding but just look at the end product (Figure 3), so do not worry too much about the details at this stage - just look at the overall structure and see how it relates to the flow chart.

```
{ program to determine number of years to repay a loan }

program  repayment(input, output);

var  principal, rate, payment
     no_of_years,
     interest_rate, monthly_interest  : real;
     month_counter                    : integer;

begin

  readln (principal,rate,payment);
  month_counter := 0;
  interest_rate := rate / (100 * 12);

  while  principal > 0  do

  begin

    monthly_interest := principal * interest_rate;
    principal := principal + monthly_interest;
    principal := principal - payment;
    month_counter := month_counter + 1
  end;

  no_of_years := month_counter / 12;
  writeln (' number of years = ',no_of_years)

end.
```

The data input record sufficient for this problem would be
30000.0 13.0 400.0

Figure 3 Sample program and data for compound interest problem

In this example, the principal, interest rate and monthly repayment were read from a separate data input record. The separation of program code from prepared data input is frequently a good programming practice. In this way, someone else can use the program simply by preparing data for it. They do not have to understand programming or know anything about the internal workings of the program to modify the data.

The alternative way of specifying input data is to replace the

```
readln (principal,rate,payment);
```

statement with three assignment statements:

```
principal := 30000.0;
rate := 13.0;
```

payment := 400.0;

Now that some idea of how Pascal programming statements fit together has been acquired, we can investigate the rules for the use of the Pascal language. Rather than give a formal definition of each Pascal concept, the approach adopted is to demonstrate its use through an appropriate example.

3. ENTRY OF PASCAL STATEMENTS TO THE COMPUTER

Pascal programs may be entered into a computer by typing them in, line by line, through a terminal connected to the computer or by punching them on data cards which are read through a card reader. The keyboards of all input devices vary; for example the card punch machine does not support lower case letters and some special symbols. As visual display terminals (VDT) are these days the most common mode of entry, the programming examples use lower case characters. To enter the same program with punched cards, it is sufficient to substitute upper case letters and special symbols where appropriate. Should your terminal not possess all the special characters, it will be necessary for you to enter alternative symbols.

The following characters are available in the Pascal 8000 language.

(i)	26 capital letters	A, B, C, ..., Z ;
(ii)	26 lower case letters	a, b, c, ..., z ;
(iii)	10 numerals	0, 1, 2, ..., 9;
(iv)	20 special characters	+, -, *, /, ., ', (,), ;, :, =, <, >, _, [,], {, }, @ , , , ;
(v)	blank	

Pascal statements may be typed anywhere on the input line, with more than one statement appearing on each line if appropriate. The free form of the layout permits statements to be indented. This assists the development of readable code, a programming practice which is to be encouraged.

To aid program documentation, comment statements may be incorporated in the code. Comments may span several lines and are basically ignored by the compiler. All statements enclosed within the symbols { and } are treated as comments:

{this is a comment in Pascal}

Should the symbols { and } be unavailable, an alternative set (* and *) may be employed. Pascal 8000 permits the use of all symbols other than (*, { and ; within comments.

4. ARITHMETIC DATA

Let us consider two different arithmetic quantities that are sufficient for handling data (numbers) in most scientific problems. It is essential that the beginner clearly understands the difference between the two types :

(a) *INTEGER* (or fixed point) constants:
a whole number without a decimal point whose absolute value is $\leq 2^{31} - 1 = (2147483647)$.

Valid integer constants: 0 -5 +357 7005192

Invalid integer constants: 27. 5,132 9812735997 27.0

(b) *REAL* (or floating point) constants:
up to sixteen decimal digits with a whole number part, a decimal point, a fractional part and an optional exponent. The absolute magnitude is approximately 10^{-78} to 10^{75}.

Valid real constants:
5.0 0.6 +0.61 7.91 5.3E+2(=5.3×10^2)
5.3E2(5.3×10^2) -0.051E-03 (-.051 x 10^{-3})

Invalid real constants:
1 3,471.2 1.E 6. .7

Distinctions are carefully drawn between the two types of constant for electronic rather than mathematical reasons. The electronic 'hardware' necessary for *INTEGER* arithmetic operations is less complex and consequently,

for most machines, it is faster than that used for *REAL* arithmetic. By performing those operations that require no decimal point in the integer mode, considerable time savings can be made.

In addition to arithmetic data, Pascal supports other forms, for example character type and even more complex structures and types of your own making. Support for such data is of great advantage for non-numerical computing, but for this introduction to Pascal, these data constructs are avoided (except for a brief mention of character variables in Section 17).

5. VARIABLES

A variable is a symbolic name used to identify a data item that will occupy a location of directly addressable storage. The actual address of this location is assigned by the compilation proce . If we move a number into a variable, it will replace the previous contents of that location:

```
time := 0.0
```

This places zero in the location reserved for time. When a transfer is made from a location, the previous contents remain unaltered:

```
x := time
```

This assigns the contents of the location reserved for time to that reserved for x without altering the contents of the location associated with time.

The := operation is interpreted as the assignment of the result of the right hand expression to the left hand location. Consequently, an expression such as

```
a := a + 1.0
```

does not yield an algebraic result but rather is interpreted as increasing the old value associated with a by 1.0 to give a new result also called a. The old value of a is, of course, lost.

Variable names may be of any length, however, only the first eight characters are significant in Pascal 8000. Consequently, variable identifiers with the same first eight characters are considered to be the same identifier in this Pascal implementation. Variable identifiers may be composed of letters or digits but the first must be a letter. Valid variable identifiers are

```
time , x3b , i5 , t .
```

For ease of identification, very long variable names such as

```
thisisalongvariable
```

may be entered as

```
this__is__a__long__variable
```

The underscore '__' is not part of the variable name itself, it is ignored by the Pascal compiler and merely serves to assist in understanding the program.

In Pascal the use of reserved words as variable identifiers is not permitted. The reserved words (such as **for**, **while** *etc.*) serve special functions and are introduced in later sections. These words are shown in **bold** type for convenience, but there is no such distinction made when they are entered at the input terminal.

For immediate purposes, we restrict the use of variables to *REAL* or *INTEGER* type. All variable names and their type must be defined by use of the declaration command **var**:

```
var     x, y, z     : real;
        i, counter  : integer;
```

Unlike some other languages there are no default options for the type of variable. All variables used must be declared and, although this may seem annoying at first sight, it reduces possible confusion when the body of code is written. For example, it avoids hours of wasted effort spent in debugging a program when time1, for example, is incorrectly entered as timel (a not infrequent type of error).

Variable names may also be used with subscripts in Pascal. Such variables can be used to represent vectors and matrices which the user may have encountered in mathematics courses:

v[3] or v(.3.) is the Pascal representation of the vector component v_3,

a[3,4] or a(.3,4.) is the Pascal representation of the matrix element a_{34}.

Unfortunately, the symbols [,] are unavailable on the card punch machines, so the alternative notation is necessary should there be a need to use subscripted quantities. (Further discussion of SUBSCRIPTED variables is delayed until Section 12.)

6. SIMPLE INPUT AND OUTPUT (I/O)

One way of assigning values to variables is by the direct use of an arithmetic expression:

```
x := 6.3
```

Should we wish to alter the data on which a program is to operate without changing the program (a most frequent requirement, particularly when non-programmers are going to prepare data to run someone else's program), then a readln statement may be used.

The Pascal procedure readln may be invoked to read values from any specified input file into a set of listed variables. The input file (along with an output file) is identified through the program declaration, *e.g.*

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

The declaration 'input' refers to the ordinary input stream of data when a Pascal program is submitted in batch mode, whereas 'output' refers to printout of data on the line printer. In batch mode the user has no contact with the job after it is submitted; the compilation and execution phases are carried out without user intervention. The output will be printed eventually on the line printer and the user can collect it when ready. Any data required by the program must be prepared in advance for batch submission of the job. The data for the input stream must be separated from the Pascal program, which is always positioned first.

The input data consist of a series of data records. These are either separate lines of input data for a terminal or separate punched cards. Each input record may contain more than one item.

An alternative mode for running the job is available; this is the interactive mode which gives the user the ability to communicate with the job while it is executing. In this mode, the input data need not be prepared in advance, since the program can prompt the user for appropriate data. For the present we shall ignore this mode of program operation. (Details of interactive operation are given in Section 21.)

The readln procedure initiates the reading of a list of data items. In the sample example of Section 2

```
readln (principal, rate, payment)
```

causes three numbers to be read and stored in the variables principal, rate and payment. After the third item has been transferred, a subsequent read will commence with a new input record, so any additional information which may have been typed on the last record read is skipped.

For the loan repayment problem in Section 2, the three data items are on the same input record; the items are separated by at least one blank character. It is possible, however, to have the data on a number of input records. In this case, sufficient input records are read so that three data items are transferred. It is important that each data

item corresponds to the type of variable for which it is intended.

Messages and output from your program may be directed to the line printer by the writeln procedure. For example the statement

```
writeln (' number of years = ', years)
```

might display

```
number of years = 1.1666666666666667E+01
```

It can be seen that this form of output, known as free-format output, has the advantage that the Pascal rules for its use are easily understood. There is not a great deal of control over the layout and it emerges frequently in an untidy form. This is unfortunate, particularly when printed output for publication is required. The free-format form of output is sufficient for the user to get under way, developing the first programs. For those who wish to know a little more about I/O control to obtain 'prettied up' output (and it is not all that difficult), see Section 15.

7. ARITHMETIC OPERATIONS AND EXPRESSIONS

There are six ordinary arithmetic operations available to Pascal users:

(i)	addition	+	*e.g.*	a + b
(ii)	subtraction	-	*e.g.*	a - b
(iii)	multiplication	*	*e.g.*	a * b
(iv)	division (integer)	**div**	*e.g.*	a **div** b
(v)	division (real)	/	*e.g.*	a / b
(vi)	exponentiation	**	*e.g.*	a ** b for a^b

Expressions may be enclosed within parentheses, as in normal algebra:

(a+b)(c+d)	(a+b)*(c+d)
$(a+b)^2$	(a+b)**2
$\frac{a}{bc}$	a/(b*c)

Parentheses are necessary to prevent two operators from appearing next to each other (should such a combination be possible):

```
x*-y     must be coded     x*(-y)
```

The sequence of operations in expressions is determined from the following hierarchy which is consistent with normal mathematics:

(i) **
(ii) * / **div** left to right precedence if they occur together
(iii) + - left to right precedence if they occur together

Consequently, the expression

```
x + (y / a) - (3.0 * u) + p * (s**4) / 3.0
```

could have been correctly abbreviated to

```
x + y / a - 3.0 * u + p * s**4 / 3.0
```

Pascal also features the **mod** (division remainder) function as an arithmetic operator rather than as a special function. It has equal precedence with multiplication and division. Consequently,

```
4 * 5 mod 3 div 2
```

returns the integer value 1 because it is interpreted on

```
(20 mod 3) div 2
```

The integer variables and constants deserve special attention. Integer division of one integer by another results in the truncation of any fractional remainder.

```
var i, k : integer ;
begin
  i := 9;
  k := i div 2
```

would result in k taking the value of 4. This property can often be exploited to the programmer's advantage when testing for even integers:

```
k := i - i div 2 * 2
```

would assign 1 to k if i is odd, and 0 if even. The expression i/2, however, would be performed as a real division even though in this case both operands are of integer type because / is a real division operator.

If an expression consists of different types of operand, the mode of the result is determined by the type of operand at various stages of the calculation. In the case of integer and real modes, the latter is considered to be the higher mode, *e.g.* for

```
var  x : real;
     y : integer;
```

the expression x*y would be evaluated in the higher mode and a real result computed. For

```
x * ( y + 3 )
```

the addition is done in the integer mode and the multiplication is performed in real mode. In assignment statements, the mode on the left should be the same as that on the right, with the exception that it is possible for an integer quantity to be assigned to a real variable.

```
x := y
```

is permitted, however

```
y := x
```

is not accepted by Pascal.

Should the user wish to assign a real quantity to an integer variable the trunc (or round) functions (Section 9),

```
y := trunc(x)   or   y := round(x)
```

should be used to remove the decimal component.

8. STATEMENT SEPARATOR

Statements in Pascal may be separated from each other by the use of a semi-colon (;). This permits more than one statement to appear on each line. The semi-colon does not form part of a statement and should not be interpreted as a statement terminator. Its function is to separate statements in a block of code or to separate one block from another, as shown in the following examples:

```
(i)  begin
       x := y + z;
       t := x + b        (separator not necessary)
     end.

(ii) begin
       .
       .
       .
       begin
         .
         .
         .
       end;              (separator necessary to separate blocks)
         .
         .
         .
       begin
         .
         .
         .
       end               (separator not necessary)
     end.
```

To someone new to programming, the subtlety of this distinction may seem puzzling to say the least! Should there be confusion about when a ; is required, remember that unnecessary separators are treated by the Pascal compiler as null statements and will not lead to any harm. Take great care, however, with use of inappropriate separation in the **if-then-else** construct (Section 11).

9. SUPPLIED MATHEMATICAL FUNCTIONS

As there are a number of special mathematical functions or operations that are common to many problems, the Pascal compiler provides these as part of the normal system. To calculate the exponential function $x=e^t$ for a particular value of t, all that needs to be done is to code

```
x := exp(t)
```

To use most supplied mathematical functions, it is only necessary to follow the function name by an argument enclosed in parentheses. The result will be returned as though the function name designated a variable in the program. The argument may be a variable, constant or arithmetic expression, *e.g.*

```
a := exp(a - c) + sqrt(15.0)
```

The following table lists frequently required functions supplied by the Pascal compiler:

Mathematical Function	Function Name (Argument)
square root, $\sqrt{x}$	sqrt(x)
square, x^2	sqr(x)
exponential, e^x	exp(x)
natural logarithm, $\log_e x$ (or ln x)	ln(x)
sine of an angle (in radians), sin x	sin(x)
cosine of an angle (in radians), cos x	cos(x)
arctangent (result in radians), $\tan^{-1} x$	arctan(x)
absolute value (real numbers), x	abs(x)
truncation	trunc(x)
rounding	round(x)

The way in which these functions work is clear enough except for the trunc and round functions. Trunc(3.7) returns 3, trunc(-3.7) returns -3; round(3.1) is 3, round(3.9) is 4 and round(-3.6) is -4.

In addition to the standard functions, we need special functions to undertake some of the preliminary exercises at the end of these notes. These functions, which are not part of the Pascal compiler, are supplied in special program libraries on the AAEC's computing facility at Lucas Heights. (Users at other facilities should find that there are equivalent functions). Because they are additional functions, it is necessary for them to be identified by an appropriate declaration at the start of the Pascal program (an explanation of such details is given in Appendix B).

Two special functions necessary for the preliminary exercises are the random number generators. The first, rand, generates a sequence of random numbers starting each time with the same number, so that the sequence generated can be repeated at a later time. It is a little like having purchased a printed book of random numbers (Yes! Such things are available.) and commencing each time at the start. This can be of use when developing and testing a new program, in which case a constant environment may be of assistance. The second function, rnd, produces a random sequence of random numbers. The user may find that this fits in more nicely with basic intuition.

Both generators can be invoked in a similar manner:

```
y := rand        y := rnd
```

Each results in a random number being assigned to the variable y, where the random number lies in the range

$$0 < y < 1.$$

(*n.b.* the values 0 or 1 are never returned by the random number generator.)

Unlike the previous functions there is no need to pass an argument to the random number generator.

10. CONTROL STATEMENTS

Execution of a program will commence at the first executable statement and, unless a transfer of control statement is encountered, proceed in order through subsequent instructions. The simplest means of transfer of control is through the **goto** statement.

```
program list (input, output);

label 1;
var   x : real;
begin
  1: readln (x);
     writeln (x);
     goto 1
end.
```

This program will cause input records with one number punched per record to be read and listed on the printer with no escape mechanism until the supply of data records was exhausted. When this happens, an error condition is raised by an attempt to read a non-existent record. At that point, the program will fail. Although the **goto** statement allows transfer of control, it is of little real use on its own.

Most early high-level computer languages, because their design reflected computer architecture, relied heavily on some form of **goto** statement. This seemed appropriate at the time because the idea was simple and easily implemented by compiler writers. Much modern thought in language design is entirely opposed to the use of **goto** statements, mainly because they make large programs difficult to follow and increase the possibility of incorrectly specifying the required operations. This is particularly true for program systems developed in a team environment. Considering that today, computer program development usually involves more cost than the machine itself, it is not surprising that well-structured, easy-to-follow and self-documenting programs are demanded. Pascal goes a long way towards satisfying these demands. It does, however, permit the use of a **goto** statement but its use is discouraged should another construct be available (which is nearly always the case). Should you find a need to put **goto** statements in your program it is most probable that you have not employed a modern structured approach to program development. Consequently, little more attention will be paid to the **goto** statement on its own or as an adjunct to the **if** construct.

The real value of a computer lies in its ability to repeat a set of statements until the task at hand is completed.

A typical task might be to print out all the names and addresses from a data file. The statements to do this must be executed repeatedly until the required task is completed. A group of such statements is commonly referred to as a loop. Three types of Pascal looping are considered:

(i) **while**
(ii) **repeat**
(iii) **for**

In the compound interest example, we saw an example of a **while** statement, which allows a block of statements to be repeated until a certain condition is satisfied. In this instance, the loop was repeated while the loan was not repaid (*i.e.* while the principal > 0). When the principal was repaid, instead of the loop being repeated, control passed immediately to the statement.

```
no_of_years := month_counter / 12;
```

following the end of the repeated block, which is identified through the **begin** ... **end** construct.

The **while** statement is used when the number of times the loop needs to be repeated is not known in advance. In more formal terms the **while** statement may be described

while *e* **do** *s*,

where *e* is a logical expression and *s* is a single or block of statements that are repeated while the logical expression *e* remains true.

Supppose we wish to evaluate and print out y given by

$$y = x^3 + x^2 + x + 1$$

for x = 1.,1.1,1.2,...,10. The following program would suffice:

```
program evaluate_y (output);
var   x, y : real;
begin
  x := 1.0;
  while x <= 10.0 do
  begin
    y := 1.0 + x * (1.0 + x * (1.0 + x));
    x := x + 0.1;
    writeln (' y= ',y)
  end
end.
```

The block form of a **while** statement is shown in Figure 4.

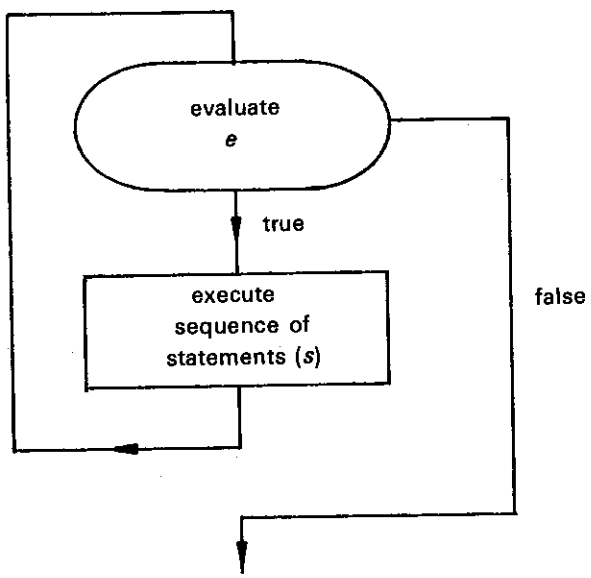


Figure 4 Schematic representation of while

The **repeat** statement (Figure 5) similarly offers a means of repeating a loop. The **repeat** statement has the form

repeat *s* **until** *e*

Here *s* represents a single statement or set of statements that are executed repeatedly until the logical expression *e* is satisfied. In schematic form, the **repeat** statement appears as

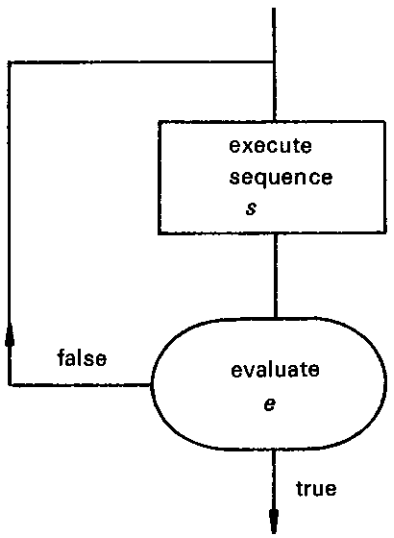


Figure 5 Schematic representation of repeat

Like the **while** statement, the **repeat** statement can be used when the number of times a loop will be repeated is not known in advance. It has been argued that any task which can be performed with a **while** statement may also be done with a **repeat**, hence a language such as Pascal need consider only one form or the other. There is an essential difference, however, between the **while** and **repeat** statements. The body of code must be executed at least once for the **repeat** because the test comes at the end of the loop. It is possible for the body of the **while** to be avoided completely because the test is at its head.

Now consider the previous example, performed this time with the use of the **repeat** statement.

```
program evaluate_y (output);
var   x, y  :  real;
begin
  x := 1.0;
  repeat
    y := 1.0 + x * (1.0 + x * (1.0 + x));
    x := x + 0.1;
    writeln (' y= ', y)
  until x > 10.0
end.
```

The final type of looping mechanism considered here is the **for** statement. Unlike the **while** and **repeat** statements, **for** is used only when the number of times the loop is to be repeated is known in advance. One form of **for** statement is

for control variable:=initial expression **to**
or
down to final expression **do** *s*

For the computing tasks considered in these notes, the control variable is of the integer type and the initial and final expressions must yield values of the same type. The control variable must not be altered in the subsequent block. Again *s* represents a single statement or sequence of statements (a block) that are executed repeatedly, with the control variable first taking the value of the initial expression and then being increased (or decreased) in

single steps until the block is repeated for the final value of the expression. The schematic form of the **for** statement is given in Figure 6.

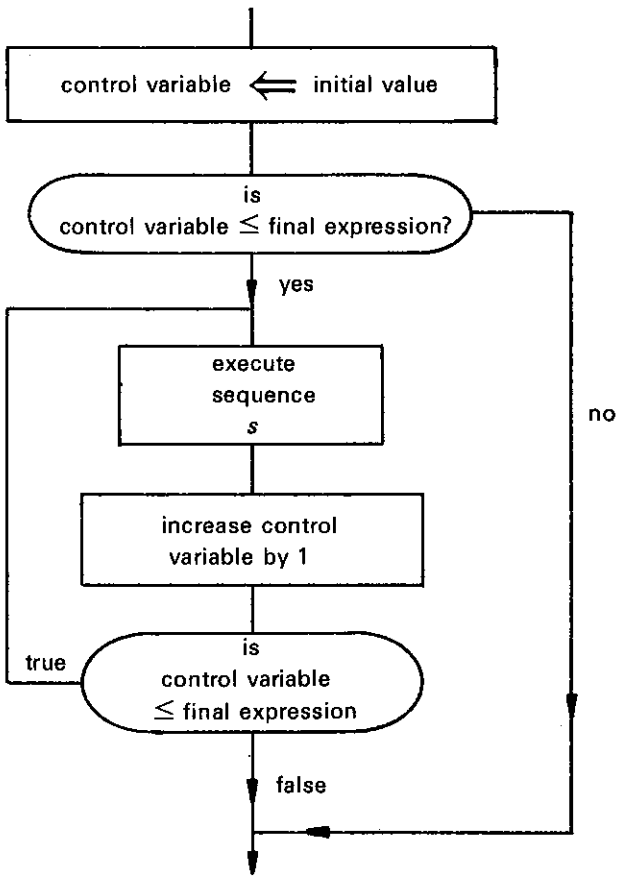


Figure 6 Schematic representation of for-to-do

Should the value of the initial expression exceed that of the final expression, the **do** block is avoided in the **for-to-do** construct. (It is avoided similarly in the **for-downto-do** form if the initial expression is already less than the final value.)

The initial and final values for the **for** construct are evaluated once only — on entry to the **for** statement. No further evaluations are carried out on each cycle and changes during execution of the **do** block to any of the variables that comprise these expressions do not affect the values assigned to the control variable.

The control variable is changed implicitly on repetition of the **do** block and no attempt should be made to alter its value within the body of the block. When the **for** statement has been completed, the control variable becomes undefined; this means that the programmer cannot make any assumption about its value. Before the control variable can be used again it must have a value assigned to it.

Consider now an example showing the use of the **for** loop. Suppose our problem is to find the sum of the first 20 integers, *i.e.* 1 + 2 + 3 + ... + 20, and to print out a running sum as we proceed. Ignoring any appeal to mathematical analysis, the following code would be sufficient:

```
program sum20 (output);
var  sum, i  :  integer;

begin
  sum := 0;
  for i := I to 20 do
  begin
    sum := sum + i;
    writeln (' sum of ',i, ' integers is ',sum)
  end
end.
```

Of course, it is possible to have coded the above example in terms of the while or repeat structures. The code for the **while** construct would be

```
program sum20 (output);
var   sum,i  :  integer;

begin
  sum := 0;
  i := l;
  while i < 21 do

  begin
    sum := sum + i;
    writeln (' sum of ',i,' integers is ',sum);
    i := i + l
  end
end.
```

and for the **repeat** construct it would be

```
program sum20 (output);
var   sum, i  :  integer;

begin
  sum := 0;
  i := l;

  repeat
    sum := sum + i;
    writeln (' sum of ',i,' integers is ',sum);
    i := i + l
  until  i > 20
end.
```

It is obvious that the code for the **while** and **repeat** constructs is a little longer than that of **for**. This is because the automatic steps of the **for** loops (*i.e.* the incrementing and testing of the control variable) must now be undertaken in more specific terms.

As a further example, consider the problem of finding the mean of ten numbers. Let us assume that the numbers are real and that they are typed one number per input line. Because we know, in advance, the number of times the loop must be repeated, we shall use the **for** construct.

```
program average (input, output);
var mean, sum, x  :  real;
                i  :  integer;

begin
  sum := 0.0;
  for i := l to 10 do

  begin
    readln (x);
    sum := sum + x
  end;

  mean := sum/10.0;
  writeln (' average = ',mean)
end.
```

Sometimes it is necessary to nest one loop inside another. Suppose we have 100 input data records, with one number typed per record, and our aim is to find the mean of each group of ten and print out that value. The following listing is a complete program capable of doing this - **for** loops are used because the number of times repetition is required is known in advance.

```
{ program to find the means for 10 groups of 10 numbers }

program avg (input, output);
var x, sum, mean :  real;
    i, j         :  integer;

begin
  for i := 1 to 10 do

  begin

    sum := 0.0;
    for j := I to 10 do

    begin
      readln (x);
      sum := sum + x
    end;

    mean := sum/10.0;
    writeln (' average of group  ',i,' is ',mean)

  end
end.
```

The loops function so that the inner counter will vary the most rapidly, *i.e.*

i= 1 1 1 ... 1 2 2 2 ... 2 ... 10 10

---

j= 1 2 3 ... 10 1 2 3 ... 10 ... 9 10

Let us consider further the logical expression *e* that is used with the **while** and **repeat** statements. A logical expression returns the value of TRUE or FALSE. For the present, we restrict use of the logical expression to testing the equality or inequality of arithmetic expressions. The simplest logical expression can be constructed with the following relational operators:

```
<    less than
>    greater than
<=   less than or equal to
>=   greater than or equal to
<>   not equal to.
```

The expressions

```
principal > 0.0
x <= y
i <> j
x + sqrt (z) <= sin (y)
```

are simple forms of logical expressions returning a TRUE or FALSE (Boolean) value.

Frequently we wish to carry out more than one logical test at a time. This can be done by combining logical expressions with one of the following logical operators:

and both expressions must be TRUE to return a TRUE result
or result is TRUE if either expression is TRUE.

Suppose we have a series of input data records of indeterminate number, and that three real numbers are typed on each record. Does each trio of numbers possibly constitute the sides of a triangle? The following code will read each data record in turn, test the possibility that each triplet forms a triangle and stop when a record is encountered with numbers that do not correspond to the sides of a triangle.

```
program triangle (input, output);

var a, b, c : real;

begin
  readln (a,b,c);
  while (a+b>c) and (a+c>b) and (b+c>a) do
  begin
    writeln (' triangle possible for ',a,b,c);
    readln (a,b,c)
  end;

  writeln (' finished test ')
end.
```

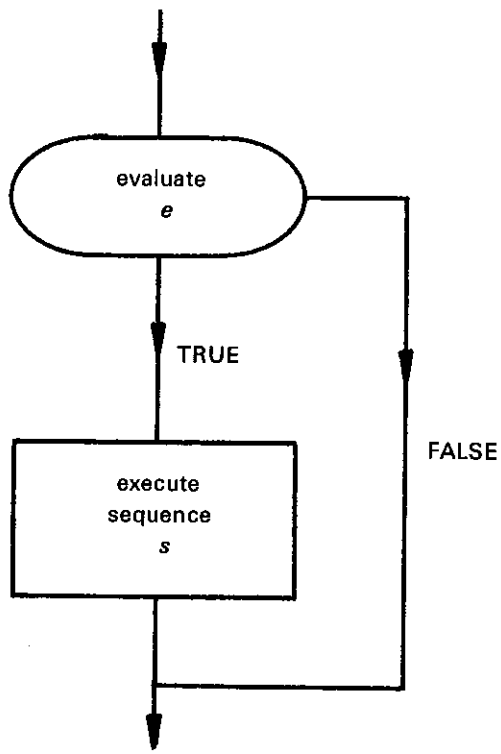


Figure 7 Diagrammatic representation of if-then

In this example, a **for** loop is not appropriate because the number of repetitions is unknown. The **repeat** structure would have been inappropriate because it is necessary to execute the body of the loop before the logical test is encountered.

Should we have a complex logical expression involving both **and** and **or** operators, it would be necessary to have rules for precedence of evaluation. Like the rules for arithmetic operators, expressions enclosed in parentheses are evaluated first; **and** has a higher precedence than **or**, and evaluation proceeds from left to right when operators are of equal precedence.

11. CONDITIONAL STATEMENTS

It is frequently necessary to execute a single statement dependent upon some condition, or at some point to execute one of a number of possible statements dependent upon some condition. The **if** statement of Pascal (which comes in two basic forms) may be used to achieve this end:

(a) **if** *e* **then** *s*
(b) **if** *e* **then** *s* **else** *t*

where *e* represents a logical expression, and *s* and *t* represent single statements or blocks of code to be executed,

depending on the result of the logical expression. The diagramatic representation of the simpler form (a) is shown in Figure 7.

Imagine for a moment that there was no absolute value function (**abs**) and that we wished to implement our own code to take the absolute value of a number when read from an input record; the following lines would then suffice:

```
    .
    .
    .
readln (x);
if x < 0.0  then  x := -x;
writeln (x);
    .
    .
    .
```

When a value of x is read that is negative, the value of x is negated. When a positive value is read for x, the negation operation is avoided altogether and control passes directly to the writeln statement.

The more involved form of the **if** statement (b) is represented in schematic form in Figure 8: where a two-way selection is now possible, depending upon the value of the logical expression. Suppose we wish to read ten real numbers and find separately the sum of those that are negative and positive. The **if-then-else** construct enables us to do this very neatly.

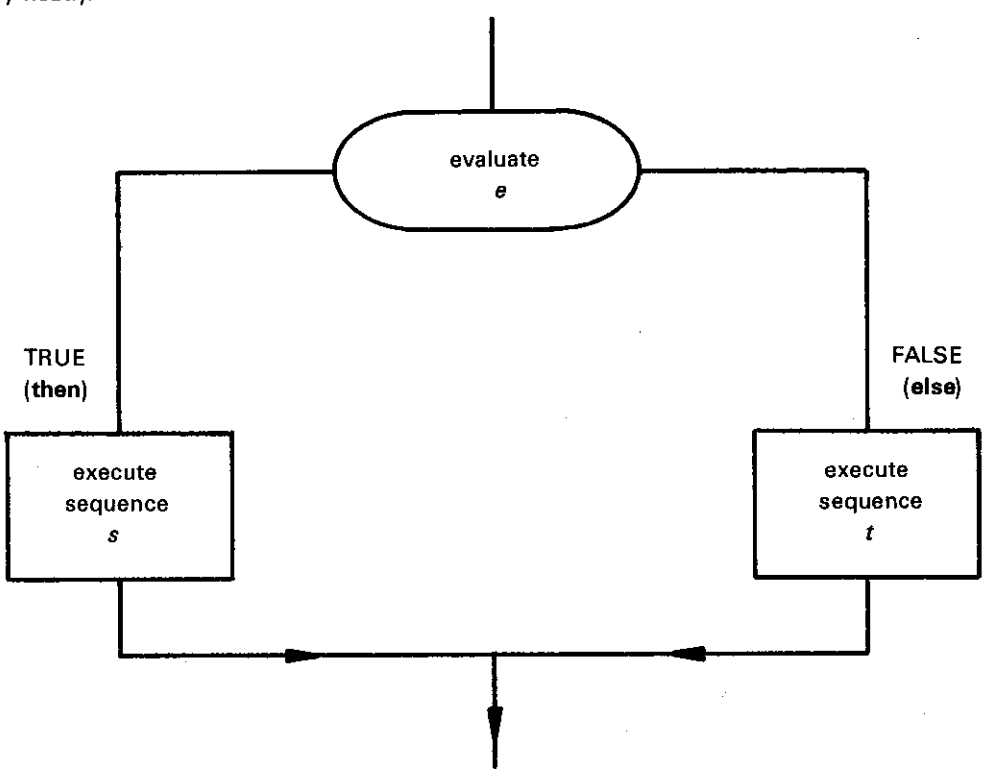


Figure 8 Diagrammatic representation of if-then-else

```
program posneg (input, output);
var x, sum_positive, sum_negative  :  real;
    i                              :  integer;
begin
  sum_positive := 0.0;
  sum_negative := 0.0;
  for  i := i to 10 do
```

```
  begin
    readln (x);
    if x > 0.0 then sum_positive := sum_positive + x
               else sum_negative := sum_negative + x
  end;

  writeln (' positive ',sum_positive,'negative ',sum_negative)
end.
```

Note that the **else** is not a separate statement, consequently there is no separator (;) between the **then** and the **else** clause.

The **if-then-else** construct can be made to do more, as the following example indicates. Suppose we wish to test that three numbers on the input record correspond to the lengths of possible sides of a triangle, and to print a message to this effect along with the perimeter of the triangle. The following code shows how a block of instructions can be executed conditionally as part of the **then** clause of an **if** statement.
Naturally blocks may be used for the **else** clause as well.

```
program triangle (input, output);
var a, b, c, perimeter : real;

begin
  readln (a,b,c);
  if (a+b>c) and (b+c>a) and (a+c>b) then

  begin
    writeln (' triangle possible for ',a,b,c);
    perimeter := a + b + c;
    writeln (' perimeter is ',perimeter)
  end
  else writeln (a,b,c,' do not form triangle')
end.
```

Suppose we wish to decide whether a, b and c correspond to the sides of a triangle, then determine if the triangle is equilateral, isosceles or scalene. To do this, the **if** statements may be nested, as indicated in the following code:

```
program triangle (input, output);
var  a, b, c  :  real;

begin
  readln (a,b,c);
  if (a+b>c) and (b+c>a) and (a+c>b) then
    if (a=b) and (b=c) then
    writeln (' triangle equilateral ')
    else   if (a=b) or (b=c) or (a=c) then
      writeln (' triangle isosceles')
      else writeln (' triangle scalene')
  else writeln (' a,b,and c do not form a triangle')
end.
```

At first sight, the idea of a nested **if** may appear to be ambiguous. For example

if e_1 **then if** e_2 **then** s_1 **else** s_2

needs special rules. In Pascal, the above construct is interpreted as though it were written

```
if e1 then
begin
  if e2 then s1
        else s2
end
```

Consequently, the execution of s_2 is dependent upon e_2 (as well as e_1). If we wished to have s_2 independent of e_2 (*i.e.* for s_2 to depend directly on e_1 it would be necessary to code the expression as

```
if e1 then
begin
  if e2 then s1
end
else s2
```

It is not necessary to lay out the code as shown above; it could all have been typed on one line, however, it is far easier to understand in the form shown.

12. SUBSCRIPTED VARIABLES

Many mathematical operations require the use of vectors and matrices. Pascal supplies a means of handling one, two or higher dimensional arrays. For the simplest array (the one-dimensional vector), the i^{th} element of the vector v (v_i) is represented in Pascal as v[i]. (Should your input device not feature the symbols [ ], then v(.i.) is an alternative form of input.) Elements of an array or vector can be used in Pascal in the same way as ordinary variables:

```
v[i] := 0      the i^th element of v is set to zero
a := v[i] + c[j] - d[3]
v[i-1] := v[3*kp-7]
```

For the range of problems considered here, the subscripts used to refer to vector or array elements are restricted to integer type. They may be constants, variables or expressions. The Pascal compiler reserves one location to store non-subscripted variables. As subscripted variables take one location for each array element, it is necessary for the programmer to specify for the compiler the maximum number of elements associated with each array. This is done through the non-executable **var** statement. For example, suppose we wished to associate 15 locations with the vector v (*i.e.* $v_1, v_2, \ldots, v_{15}$), then a declaration could be

```
v: array [1..15] of real,
```

if v were to contain 15 real elements. The loop

```
for i := 1 to 8 do v[2*i-1] :=0.0
```

would set the odd components of v to zero. The subscripts in Pascal may be non-positive; for example the declaration

```
x: array [0..16] of real;
```

would associate 17 elements with the vector x, the first being identified by x[0].

It is usually easiest for students of mathematics to associate subscripted variables in a computer language with the mathematical concept of an array or vector. This approach is used for this presentation. In the commercial programming world, where mathematical applications take second place to business needs, it is worth noting that subscripted variables still play an essential role. They no longer have the obvious mathematical meanings, and frequently the programmer has little (if any) idea of what constitutes a vector or matrix. Pascal supports many other array concepts, however, these possibilities are ignored here in favour of the ordinary mathematical outlook.

The next example demonstrates how a vector may be used to calculate the mean and standard deviation of a set of ten numbers. These numbers are read from ten input records (*i.e.* one number per record):

$$\bar{x} = \frac{\sum_{i=1}^{10} x_i}{10}$$

$$\textit{Standard deviation} = \sqrt{\frac{\sum_{i=1}^{10} (x_i - \bar{x})^2}{9}}.$$

```
{ calculates the mean and standard deviation of 10 numbers }

program stats (input, output);
var x                       : array [ 1..10 ] of real;
    sum, avg, sumsq, sdev   : real
    i                       : integer;
begin
  for i := 1 to 10 do readln (x[i]);
  sum  := 0.0;
  for i := 1 to 10 do sum := sum + x[i];
  avg := sum / 10.0;
  sumsq := 0.0;
  for i := 1 to 10 do sumsq := sumsq + sqr(x[i] - avg);
  sdev := sqrt(sumsq/9.0);
  writeln (' mean  and standard deviation are ',avg,sdev)
end.
```

Here we use the array x to store ten numbers before finding the mean and standard deviation. Before employing vectors in a program, *make sure they are really necessary.* In a previous example (Section 10) the mean of a set of numbers was required. There was no need in that case to retain the ten numbers because the sum accumulated when each number was read from an input record. When the standard deviation is calculated by the above formula (without rearrangement of terms), however, it is necessary to retain all the numbers in memory, so a vector is required.

The next example demonstrates a program that computes the vector sum s of two vectors u and v. For those unfamiliar with vector summation the following definition should suffice

$$s = u + v,$$

for $u = (3,5,2)$
and $v = (4,2,7)$
then $s = (3+4,\ 5+2,\ 2+7)$
$= (7,7,9)$

Mathematically we say that the i^{th} component of s is formed as

$$s_i = u_i + v_i \qquad 1 \leq i \leq 3$$

The program will read the three pairs of data from separate input records as shown

u	v
3.0	4.0
5.0	2.0
2.0	7.0

into two vector arrays u and v, compute the vector sum in s and print out each component of s on a separate line.

```
program vector (input,output);
var u, v, s  :  array [ 1..3 ] of real;
        i    :  integer;
begin
  { first read in the data }
  for i := 1 to 3 do readln ( u [i],v [i] );
  { form vector sum }
  for i := 1 to 3 do s [ i ] := u [ i ] + v [ i ];
```

```
  { write out results }
  writeln (' vector s ');
  for i := 1 to 3 do writeln (s[i])
end.
```

In this example, it would have been possible to perform the vector addition operation without the use of subscripted variables. Such an operation, however, is frequently a small part of a much larger program in which it is necessary to store the data in subscripted variables for later use. Students who are not familiar with matrices should move on to Section 13.

When arrays of higher order than the one-dimensional vector are needed, the array declaration informs the compiler of the number of dimensions (*i.e.* the number of subscripts and the total storage required for the array):

```
var  a : array  [ 1..5,1..5 ]  of real;
```

This informs the compiler that a is a matrix (two-dimensional array) requiring 25 locations for storage.

```
var a, b, c  :  array [1..5,1..5] of real;
     .
     .
     .
for i := 1 to 5 do
for j := 1  to 5 do  c [i,j] := a [i,j] + b[i,j] ;
     .
     .
     .
```

In this case, two matrices a and b are summed and the result is stored in a new matrix c.

13. PROCEDURES AND FUNCTIONS

In Section 9 were introduced the special mathematical functions supplied through the Pascal compiler. The users are able to supply two types of routines of their own where necessary:

- **function**
- **procedure**

Need for these special routines arises:

(i) when the same mathematical function or procedure is required at many points in a program;

(ii) in larger programs, where it pays to write and test sections of the code independently; and

(iii) when more than one person is responsible for developing the code.

The **function** returns a single value as its result and is usually used to perform mathematical operations similar to $\surd$ or other function evaluation. The user-supplied function is best demonstrated by an example. Suppose we wish to evaluate a given cubic polynomial for various real values of x:

$$f(x) = 1 + 1.5x + 3.2x^2 + 6x^3$$

which, for speed of computation, is best written in the Horner form as

$$f(x) = 1 + x(1.5 + x(3.2 + 6x))$$

When we have eventually constructed the code for this function it may be invoked through such Pascal statements as

```
y := f(x)
y := f(x) + 6.0
y := f(x-3.0) + 7.0
```

The following example demonstrates the use of a **function** and the main program code necessary to invoke it so that the above function f(x) is evaluated for x=0.,1.,...,10.

```
program evaluate_polynomial (output);

var x, y : real;
    i : integer;

  function f (z : real) : real;

  begin                                          function code
    f := 1.0 + z * (1.5 + z* (3.2 + 6.0 * z))
  end;

  { main program }

begin
  x := 0.0;
  while x <= 10.0 do
  begin                                          main program code
    y := f (x);
    writeln (y);
    x := x + 1.0
  end
end.
```

We notice, in the simple example, that the code for **function** f has been separated from and must precede the main body of code.

Because the variables x,y and i were declared at the head of the program, they are available if required in all portions of the program. These are known as global variables.

The variable z used to define **function** f is known as a local variable and may be invoked only through the scope of the **function**. Any reference to it outside the **function** would not be permitted by the Pascal compiler. The scope of a **function** is identified easily through the appropriate **begin** ... **end** delimiters within the **function**.

The **function** must be declared to be of real type to avoid truncation of the fractional portion of the result. The **function** parameter z is a dummy but it must be of assignment compatible type to the actual parameter in the calling program (x). When the **function** is invoked through

```
y := f (x)
```

the value of x is transferred effectively to z for evaluation. The **function** operates on a copy of x in z and must transfer the calculated result back by assigning it to f, the name of the **function**. In this case, the **function** cannot alter the value of z.

Should you wish the **function** to alter the actual parameter, the dummy parameter must be declared to be changeable. This is done by the alternative **function** declaration

```
function f (var z : real) : real;
```

The actual parameter and the **function** parameter must be of the same type in this form of **function** call. Consider now a second example involving a **function**. Ignoring an appeal to mathematical analysis, suppose we create a new mathematical function

$$g(x) = 1 + \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + \cdots$$

and that a sufficient number of terms are included so that $\frac{1}{x^n} < 0.00001$. The following program defines the function g(x) and evaluates it for x = 2.0,2.1, ... , 3.0.

```
program evaluate_g (output);

var x, y : real;
  function g (z : real) : real;
  var  sum, term : real;
  begin
    sum := 0.0;
    term := 1.0;
    repeat sum := sum + term;
         term := term / z
    until  abs(term) < 0.00001;
    g := sum
  end;

{ main program }

begin
  x := 2.0;
  while x <= 3.0 do
  begin
    y := g (x);
    writeln (x,y);
    x := x + 0.1
  end
end.
```

In this example, additional variables sum and term are introduced in the formulation of **function** g(z). These variables are defined within the scope of g, hence are local to it and unavailable for use outside.

The **procedure** is the more powerful version of a sub-program and usually performs more involved operations than those for which the **function** is designed. Typical tasks for which **procedures** are used would include finding the roots of equations, multiplications or other operations on matrices, and solving sets of linear equations. Unlike the **function**, the **procedure** is not restricted to returning one result as part of an arithmetic expression.

The following code shows the use of **procedure** quad to determine real roots of a quadratic equation $ax^2 + bx + c = 0$. The coefficients of this equation are supplied as parameters for the **procedure**. The **procedure** is responsible for returning the two roots and an indication that real roots are possible. It is assumed the coefficients (a,b,c) are real numbers supplied on input data records (three numbers per record). The number of data records is indeterminate, however, all coefficients are known to be less than 10^6 in absolute value. Consequently, the end of the data is signalled by a record with the first coefficient outside this range.

```
program solver (input, output);

var c1, c2, c3, r1, r2  :  real;
                  ier  :  integer;

procedure quad (a, b, c : real; var x1, x2 : real; var k : integer);
var discriminant : real;

begin
  discriminant := b * b - 4.0 * a * c;
  if discriminant > 0.0 then

  begin
    discriminant := sqrt (discriminant);
    r1 := (-b + discriminant) / (2.0 * a);
    r2 := (-b - discriminant) / (2.0 * a);
    k := 0
  end

  else  k := 1

end
```

```
{ main program }
begin
  readln (c1,c2,c3);
  while abs(c1) < 1.0e6 do
  begin
    quad (c1, c2, c3, r1, r2, ier);
    if ier=0 then writeln(' real roots are ',r1,r2)
    else writeln(' no real roots');
    readln (c1, c2, c3)
  end
end.
```

Like the previous **functions** the **procedure** quad is positioned after the initial declarations but before the body of the main program. Unlike the **function**, which is used to return a single value as part of an arithmetic assignment, the **procedure** quad is invoked by a single statement.

```
quad (c1, c2, c3, r1, r2, ier);
```

The main program passes three coefficients of the quadratic, whereas the **procedure** returns the roots in x1 and x2 and an indication (k=0 or 1) as to the sign of the discriminant. The three values are retumed in r1, r2 and ier. Because these variables are assigned values by the **procedure** it is necessary to indicate that the appropriate dummy parameters may be assigned values by use of the **var** declaration.

14. CONSTANTS

Frequently, programmers wish to associate particular values with certain names and use these identifiers throughout the program. These values are not to be altered within the program. A common requirement is to associate an approximation for π with the identifier pi. This is done through a **const** statement, which precedes the **var** declaration. In the following example, the areas of circles of radii 1.0,1.1,1.2,...,2.0 cm are calculated:

```
program area (output);
const pi = 3.1415926;
var r, area : real;
begin
  r := 1.0;
  while r <= 2.0 do
  begin
    area := pi * r ** 2;
    writeln (' area of circle with radius ',r,' cms is ',area);
    r := r + 0.1
  end
end.
```

The constant pi is not mentioned in the **var** declaration because it is not a variable; however, it has a type associated with it from its definition, here it is real. The declaration

```
const t = 3;
```

causes t to be of the integer type. Use of the **const** declaration provides some protection in the program which might otherwise accidentally alter the value of a fundamental constant.

15. MORE ABOUT CONTROL OVER OUTPUT

The output statements treated so far are sufficient to enable most simple problems to be solved. The tidiness of such output, however, frequently leaves much to be desired. This section is included for those who wish to know

more about control over output. Rather than state formal rules, the explanation is performed through examples. For

```
x := -1.36427815E+02
```

the statement

```
writeln ('bx =b',x)
```

causes

```
x=  -1.3642781499999998E+02
```

to be printed on the line printer. It will be noticed that the '$_b x_b =_b$' starts with the x in column 1. The leading blank in '$_b x_b =_b$' is stripped from the output and used to control line skipping on the printer. The blank causes a single line to be skipped. The following symbols have special significance if they are used as the first character of the output destined for the line printer:

b	single space
0	double space
-	triple space
1	new page
+	no advance of the paper (useful for over-printing)

The writeln procedure after writing the named variables or items causes a new output record to be created (*i.e.* writeln adds to an existing output record but afterwards appends an end-of-line). The write procedure functions similarly to writeln in that it also adds on to an existing record, but does not lead to the creation of a new record when the items have been added. For example, the sequence

```
writeln ('bA','bBb');
write ('bCD');
writeln ('bE');
```

would produce

```
AbBb
CDbE
```

The default options for *real* variables allow 24 columns of output to display the number in floating point format, whereas *integer* variables take 12 columns (leading blanks are inserted where necessary).

Further control may be obtained over field positions, as indicated by the following examples. Assume that x is a real variable set to -1.36427815e+02 and i is an integer that has been assigned the value 5392:

		printer control printed record
writeln (x:11, I:6)	produces	-1.364e+02$_{bb}$5392
writeln (x:9:2)	produces	$_{bb}$136.43
writeln ('$_b x =_b$', x:9:2, '$_b$I=', I:7)	produces	x=$_{bbbb}$136.43$_b$I=$_{bbb}$5392

16. MORE ABOUT INPUT

On input, the procedure readln causes the data items associated with the input list to be read and transferred to the appropriate variables. On completion of the transfer, the file is positioned at the start of the next input record (a line for terminal-oriented input or a new card on punched card input). On the other hand, the procedure read simply causes the next data items on the current record to be transferred without the input file being advanced to the next record. For example with

```
readln (x,y);
read (z);
```

the first two numbers are transferred to the variables x and y and the input file is advanced to the beginning of the next line. The subsequent read procedure extracts a third number from the next record and places it in z.

Pascal provides two special functions that are of assistance with input control:

(i) eof
(ii) eoln

Each is a Boolean function (*i.e.* it returns a value of TRUE or FALSE). eof (end of file) returns a value of FALSE when data input records are still to be processed. It returns a TRUE value only after the last input record has been reached. Any attempt to read additional data after the end of input file would cause the program to 'crash' (terminate with an error diagnostic printed) at that point of the execution phase. To prevent this, particularly when the number of input data records is not known in advance, the eof function can be used. The following program reads and lists a set of input records (of unspecified number) containing one real number per record and finds the mean value of the set:

```
program mean (input, output);

var sum, x, mean :  real;
    number       :  integer;

begin
  sum := 0.0;
  number := 0;
  while not eof do
  begin
    readln (x);
    sum := sum + x;
    number := number + 1
  end;
  mean := sum / number;
  writeln (' mean = ',mean)
end.
```

The block of statements to read and accumulate the sum of the numbers punched on a set of data records is executed until the expression

not eof

is false. The **not** is a logical operator which negates the value of eof. When the last card has been read, eof is TRUE and is negated to FALSE by the **not** operator. Hence the block of code is avoided and the mean is calculated and printed.

The eoln (end of line) function works similarly, returning a value of TRUE when the last actual character of the current input record has been read.

17. NON-NUMERIC DATA AND VARIABLES

Pascal provides very complex structures for handling non-numeric data. In fact, the language is so flexible that it permits users to define individual data structures. This section is introductory, being concerned primarily with numerical computation, so no attempt is made to present the bulk of this material. However it is necessary, even with numerical computations, to perform a little character processing at times.

The permissible set of non-numeric characters depends upon which version of Pascal is used and the computer hardware facilities available. For simplicity, we restrict consideration to the alphabetic letters, arithmetic numerals and a few special symbols. In a Pascal program a particular value of type char (character) is enclosed within single quotes

'a', '?', '4', 'B'

It is important at this stage to distinguish the character '4' from the integer 4. The character '4' cannot be used in arithmetic operations but the integer 4 can. To represent the single quote as a character, it is written twice ''''.

Variables used to hold character information must be specifically defined

```
var x, y, z : char;
```

defines three variables x, y and z, each of which may hold one character of information. The following example shows how all the items in a current input record are read and the number of times the symbol '+' occurs is counted:

```
program character (input,output);
var x    : char;
    sum  : integer;
begin
  sum := 0;
  while not eoln do
  begin
    read (x);
    if x = '+' then sum := sum + 1
  end
  writeln (' number of times + occurs in record is ',sum)
end.
```

It is only possible to store one character data item with each of the non-subscripted character variables introduced above. To be more powerful, we need the ability to store longer items in a single entity. Pascal provides a number of ways of doing this. The first is though the use of an array:

```
var name : array[1..20] of char;
```

enables up to twenty characters to be stored in the vector array called name. Individual characters may be selected by the use of subscripts. For example, name[i] selects the i^{th} character from the 20 character vector.

The following code reads in twenty character names from input records and immediately prints them out on separate lines. The process continues until no more input records are available.

```
program list_names (input,output);
var name : array [1..20] of char;
    i    : integer;
begin
  while not eof do
  begin
    for i := 1 to 20 do read (name[i]);
    write(' ');
    for i := 1 to 20 do write (name[i]);
    writeln;
    readln
  end
end.
```

It is possible to transfer the information in two arrays directly provided that they are of the same length and type. For example, in

```
var card1, card2 : array [1..80] of char;
    i            : integer;

begin
  for i := 1 to 80 do read (card1[i]);
  card2 := card1;
```

the 80 characters read into the array card1 are assigned directly to card2. This assignment is equivalent to writing

```
for i := 1 to 80 do card2[i] := card1[i]
```

In Pascal, character information stored in an array construct may be held in either packed or unpacked form. The above examples show the use of the default form, the unpacked array. With packed arrays, considerable savings in storage space are possible, however, there may be a small price to pay because of the additional time used in packing and unpacking the array in some Pascal implementations. To use a packed array in the above example, the only change required is in the declaration statement, *i.e.*

```
var card1, card2 : packed array[1..80] of  char;
```

The packed array provides one additional powerful feature in Pascal, namely the use of a string of characters.

```
'John Smith'
```

is an example of a typical string. To use string variables in Pascal it is necessary to go a step further and define a ones own data type. This is done through a **type** statement which introduces the type and associates it with the chosen name. This name is not used directly as a variable, but rather to define variables. For example, in

```
type name = packed array [1..10] of char;
var  name1, name2 : name;
```

a ten character string type is associated with name and two variables name1 and name2 are declared to be of this type. Assignments of the form

```
name1 := 'John Smith';
name2 := name1
```

may be made. Individual elements of the string may be extracted with references such as name1[i]; this permits access to the i^{th} element of the string. A complete string may be written out in Pascal:

```
writeln ('name is', name1)
```

would display

```
name is John Smith
```

Note that the output capability applies to string variables only. Unfortunately it is not possible to read string variables in this fashion in standard Pascal; they must be processed character by character, as in

```
for i := 1 to 10 do read (name1[i])
```

Pascal 8000, however, permits string variables to be read directly.

18. CASE STATEMENT

When a two-way decision based on the value of a logical expression is required, the **if-then-else** construct is ideal. When the choice involves more options, the **case** statement can be of considerable value. A typical case structure is shown in figure 9.

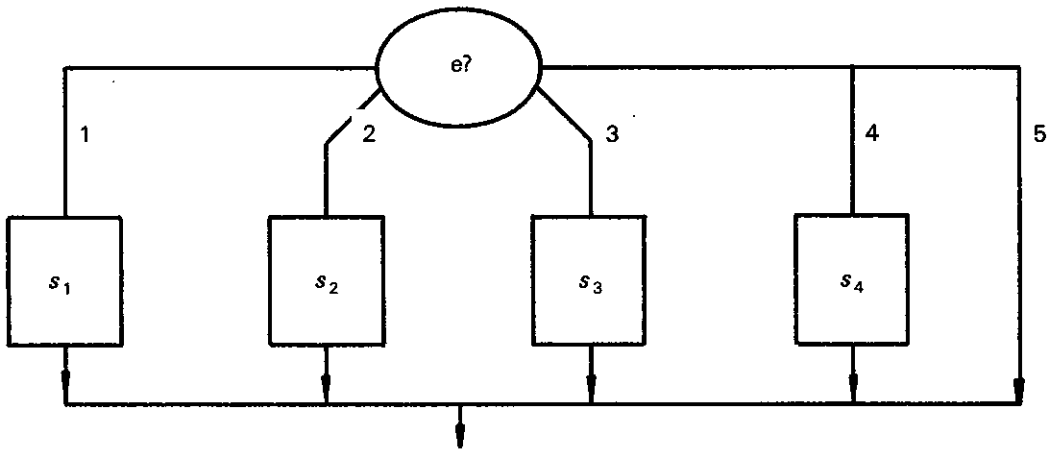


Figure 9 case - statement structure

In this case, there are five paths to be chosen depending upon the value of *e*. For the first four choices sequences of statements s_1, s_2, s_3 or s_4, are selected. The fifth choice, however, is to do nothing at all. The general form of the case statement is

```
case e of
  1 : s1;
  2 : s2;
  3 : s3;
  4 : s4;
  5 :   ;
end
```

where we notice the ; associated with case label 5 denotes a null statement. The order in which the different case labels appear is unimportant, however, *e* must return a defined value appropriate to the case statement labels unless an **otherwise** clause is provided as a suitable escape mechanism. (The **otherwise** option is available in Pascal 8000 and not in standard Pascal.)

If i and j are assumed to be integer variables, a typical example could be

```
case i+2*j of
  1:  x := x + 1.0;
  2:  x := sin(x);
  10: begin
        x := cos(x);
        y := sin(x)
      end
otherwise
  begin
    x := 0.0;
    y := 1.0
  end
end
```

For the structures considered here, the case selector (i+2*j) must be of the integer type. The expression is evaluated and its value determines the subsequent course of action defined by the case-statement-labels. For a value of 1, 2 or 10 respectively, the first three statements are chosen. Any other value directs execution to the

otherwise. It is not necessary to include an **otherwise** clause if the programmer does not require a 'catch-all'. Failure of the value of the case selector to correspond to a case-statement-label (in the absence of the **otherwise**, however, leads to an error at the execution phase of the program.

19. A COMPLETE PROGRAMMING EXAMPLE

The following example demonstrates the stages of development involved in determining an approximation to π using a computer to simulate a dart board. (The approach is illustrative only - it is not the way to determine π.) If we have a circular dart board mounted on a square background, as shown in Figure 10, then it would be possible to determine experimentally an approximation for π. When a dart is thrown randomly to land in the square, it may land within the circle or outside it. The probability of it landing within a certain gsection is proportional to the area of that section:

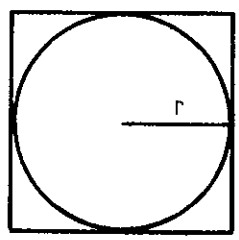


Figure 10 Dart board

Relative proportion of darts landing in the circle

$$= \frac{\textit{area of circle}}{\textit{area of square}}$$

$$= \frac{\pi r^2}{(2r)^2}$$

$$= \frac{\pi}{4}$$

$\pi = 4 \times$ relative proportion of darts landing in the circle.

Consequently, by randomly throwing darts and measuring the relative frequency of those falling within the circle, π can be determined approximately. Instead of throwing darts, a computer can be employed to do this by direct simulation. The process can be simplified, as shown in Figure 11, by taking a quadrant of a circle of unit radius. By selecting two random numbers, using our random number generator, we may let these two numbers (say x and y) represent the coordinates of the point where the dart lands. This point may be within the circle or outside it. If we measure the distance d of the point from the origin

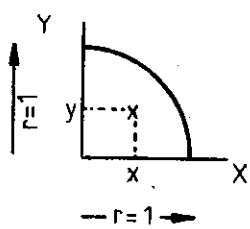


Figure 11 Quadrant simplification of dart board

$$d = \sqrt{x^2+y^2}$$

we can determine if it lies in the circle or not depending upon whether $d \leq 1$ or $d > 1$ (or, equivalently, whether $d^2 \leq 1$ or $d^2 > 1$, to save the square root operation).

A flow chart to describe the steps involved is shown in Figure 12 for a sample of 1000 darts.

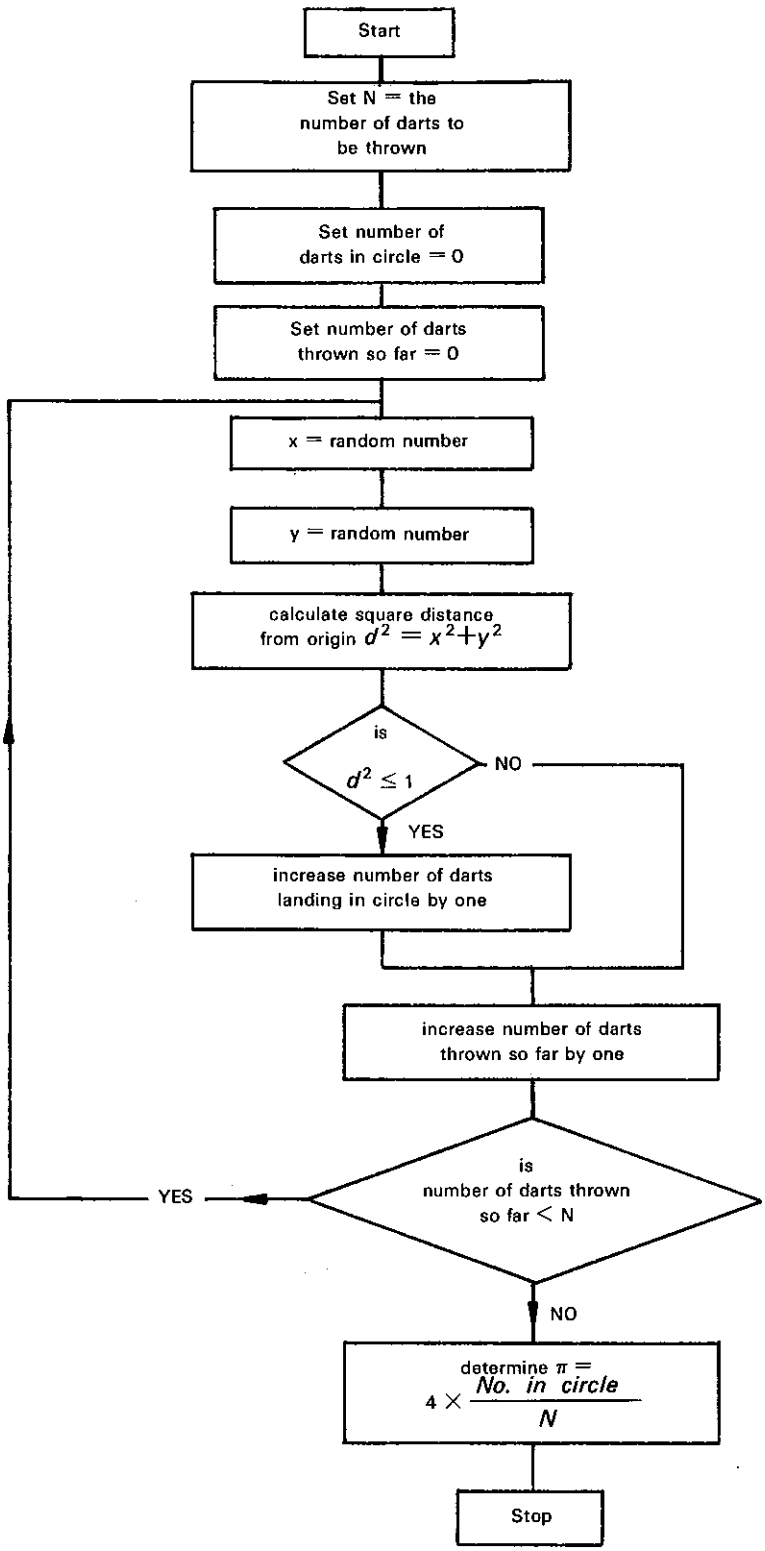


Figure 12 Flow chart for π problem

It remains only to generate random numbers to simulate the throw of a dart. The process of random number generation is not as trivial as it may at first appear. For the purpose of this exercise we shall develop our own random number function rand. A random integer number can be generated from another random integer by the operation

$$i_{rand} := i_{rand} * i_{const} \ ,$$

where i_{rand} initially is 1764435781 and i_{const} is 1220703125. The reason for selecting these numbers in particular, however, is beyond the scope of this presentation. The integer multiplications produce a result greater than the largest integer that the computer can offer. Some of the result is lost in the multiplication, a phenomenon called 'fixed point overflow' in computer jargon. Because we require a random number in the range 0 < random number < 1 the integer remaining after the overflow is scaled to that range by

$$\text{rand} = \frac{i_{rand}}{largest\ integer} * .9999999999999999 \ .$$

To set the initial value of i_{rand} = 1764435781 in the random number function rand it is necessary to invoke a separate procedure rand__initialise. This procedure also sets up i_{const} and the scaling factor necessary to convert the random integer to the appropriate range.

```
program pi (output);

var i__rand, i__const            : integer;

    no__thrown, no__in__circle : integer;
    no__to__be__thrown          : integer;
    rand__biginv                : real;
    x, y, dsq, pi               : real;

{ first define random number procedures }

procedure rand__initialise;

const
  igen = 1220703125;
  istart = 1764435781;
  maxint = 2147483647;

{ the value of maxint is determined by the maximum
  absolute integer value representable on your
  computer. For the IBM360/370 series maxint = 2**31-1
  while on a 36 bit machine you would use maxint = 34359738367 }

  begin
    rand__biginv := 0.99999999999999999/maxint;
    i__rand := istart;
    i__const := igen
  end;

function rand : real;

{   rand generates a random number in the range (0,1).
    This version won't run on a pascal system that traps
    fixed point overflow.  }

  begin
    i__rand := abs (i__rand * i__const);
    rand := i__rand * rand__biginv
  end;
```

```
begin

{  main program
   determination of pi by dart board simulation  }

{  initialise random number generator  }

  rand-initialise;

{  now perform dart board simulation  }

   no__to__be__thrown := 1000;
   no__in__circle := 0;

   for no-thrown := 1 to no__to__be__thrown do

     begin
     x := rand;
     y := rand;
     dsq := x * x + y * y;

     if dsq < 1.0 then no__in__circle := no__in__circle + 1
   end;

   pi := 4.0 * no__in__circle / no__to__be__thrown;
   writeln (' pi = ',pi)

end.
```

20. ERRORS IN PROGRAMMING

The Pascal compiler will inform us in no uncertain terms of any syntactical errors we make in coding a program. Such errors are easy to detect and correct. The computer is a totally obedient servant; provided that we ask it to perform a task in the language it understands, it will obey us without question. Therefore, the hardest errors to identify are the ones we make in specifying the logic or steps involved in solving our problem. In reverse to British justice, all programs should be considered guilty (of containing bugs) until proved innocent ('debugged').

Too often the poor computer is blamed for an error in the program that should have been found and removed by the programmer when debugging the code.

garbage in implies garbage out

This adage is certainly true but the programmer and, in particular, the scientific programmer may find it difficult to recognise the output of a program for what it is. It is advisable to test programs thoroughly before placing any confidence in their output. Too often this step is overlooked and the output of the program may be misleading to say the least. Verification of the correctness of the program is not a simple task. Testing is often commenced by comparing the computer solution with a known mathematical or physical solution. When agreement is satisfactory, we may then proceed to use our program for all the cases in which we are interested.

There are three ways in which the problems of the scientific programmer are different to those of the more commercially oriented programmer. As most commercial tasks are well defined, errors in the computer output are due directly to the program or incorrect data on which it operated. The scientific problem solver is solving a mathematical model of some real physical system. When this model was developed, many assumptions (and probably simplifications) were made. Just how valid were these and are they the source of errors? Were the errors caused by the type of numerical technique chosen to solve the model? Or were the errors due to the coding of these techniques?

21. INTERACTIVE MODE OF RUNNING

A Pascal program can be made to run in interactive mode so that data can be read from and/or written to the terminal (for example, under the IBM TSO operating mode). This provides the user with some dynamic control over the program. The way in which interactive communication is established is not defined in the Pascal standard and implementations vary between compilers. In Pascal 8000 the following approach is adopted.

Firstly the user must specify that either or both input and output are coming or going to the terminal in an interactive fashion. This is done by inserting a solidus (/) where appropriate in the program declaration, *e.g.*

```
program xyz (input/,output/);
```

indicates that both input and output are terminal oriented. This rule seems logical enough. The sequence of events actually necessary to read input data, however, will make little sense without a more detailed study of Pascal. Instead, the technique is best demonstrated by an example. Suppose we wish to read three coefficients of the quadratic polynomial $ax^2 + bx + c$ from the same input line and, before we enter the data, wish to be prompted by a special message at the terminal. The following code would suffice:

```
writeln (' enter three coefficients a, b and c ');
readln;
read (a, b, c);
```

On the other hand, if we wished to enter the three coefficients on separate lines (with appropriate prompting messages) the following code would suffice:

```
writeln (' you are now to enter 3 polynomial cofficients ');
writeln (' a :=');
readln;
read (a);
writeln (' b :=');
readln;
read (b);
writeln (' c :=');
readln;
read (c);
```

The readln statement must precede every read statement for interactive input in Pascal 8000. Other Pascal systems will have their own peculiarities for handling interactive input.

22. PRACTICE EXAMPLES

Here are a series of questions to test your understanding of the material presented. The answers to the questions are given in Appendix A, but don't be too hasty to seek these out until you have had a go yourself!

Q1

In the list below, some of the items are arithmetic constants, some are variables while others have no such status in Pascal.

(a) Which are variables and which are arithmetic constants?

(b) Are any invalid (if so, why?)

(c) If the item is an arithmetic constant, what is its mode (integer or real)?

List (1) 1. (2) ABC (3) South__America (4) 14 (5) .6 (6) five (7) 2ue (8) 0 (9) bos (10) a*b (11) 5,132.6 (12) Ire-land (13) water__2__drink (14) -0.001E-10 (15) coca__cola

Q2

Write each of the following algebraic formulae as a Pascal statement to calculate y. Use any convenient names for the variables, which will be assumed to be of the appropriate type and to have been assigned values by previous steps of the program.

(1) $y = \frac{1}{2}(b+c)$

(2) $y = \frac{\sqrt{b^2-4ac}}{2a}$

(3) $y-x = a-\pi y \qquad (\pi=3.1415926)$

Q3

What values would be stored in the variable on the left of the following arithmetic statements? Are any of the statement invalid?

```
const a = 3.8;
var u    : real;
    k, i : integer;

(1) i := a;
(2) i := round(a);
(3) i := trunc(a);
(4) u := a / 2.;
(5) u := a / 2.0;
(6) k := trunc(a + 2.9);
(7) i := (k - 1) div 2;
(8) a := a / 2.0;
```

Q4

Write the necessary statements of portion of a program to calculate the variables given by the following expressions. Use any convenient names for the variables. You may assume that variables on the right have been assigned values by previous steps of the program and that the values do not require special consideration in calculating the expressions. There is no unique answer to each question, so even if your answer differs from that given it doesn't necessarily mean you are incorrect. Some of the answers given use temporary variables to avoid repeating the same calculation in each expression.

(1) $s = \sqrt{x^2 + y^2 + z^2}$

(2) $y = e^x$

(3) $u = \dfrac{\frac{1}{2}(e^x-e^{-x})}{\frac{1}{2}(e^x+e^{-x})}$

(the mathematical function tanh x)

(4) $v = tanx$

(5) $c = ln\ \frac{1}{1+a^3}$

(6) $y = (e^{ax} + e^{-\sqrt{6}x})/3$

Q5

What, if anything, is incorrect in the following section of Pascal code? Assume that the declarations and other statements that would normally precede each is appropriate.

```
(1) writeln (' result is ',a);
```

(2)
```
x := a + b * sqrt(d);
```

(3)
```
for i := 1 to n do x := x + 1.;
```

(4)
```
for i := 1.0 to 10.6 do x := x + 1.0;
```

(5)
```
for i = 1 to 5 do anything;
```

(6)
```
x := 0.0;
i := 0;
while i < 100 do
begin
  x := x + 1.0;
  i := i + 1
end;
```

(7)
```
x := 0.0;
i := 20;
repeat
  x := x + i;
  i := i - 2
until  i < 1
```

(8)
```
x := 0.0;
i := 20.0;
repeat
  x := x + i;
  i := i - 2.0
until  i < 1.0
```

(9)
```
x := 0.0;
i := 0;
while i < 100 do
begin
  x := x + 1.0;
  writeln (x)
end
```

(10)
```
if x >= y then writeln (x);
else begin
       x := x + 1.0;
       writeln (x)
     end;
```

(11)
```
x := x if a < b
```

(12)
```
var x : real;
    i : integer;
begin
  .
  .
  .
x := i;
```

(13)
```
var x : real;
    i : integer;
begin
  .
  .
  .
i := x;
```

(14)
```
var x : real ;
    i  : integer;
begin
  .
  .
  .
i := trunc(x);
```

(15)
```
if a=b=c then x := a + b + c;
```

(16)
```
var t, x : array [0..100] of real;
```

(17)
```
var tt, xx : array [0..100,0...100] of integer;
```

(18)
```
program calculate (output);
var x, i : real;

function cost (z : real) : real;

begin
  cost := 2.0 * z + 6.0
end;

{ main program }

begin

  i := 1.0;
  while i < 10.0 do
  begin
    x := cost(i);
    writeln (i,x);
    i := i + 1.0
  end

end.
```

(19)
```
program calculate (output);
var x : real;
    i  : integer;

function cost (z : real) : real;

begin
  result := 2.0 * z + 6.0
end;

{ main program }

begin
  i := 1
  while i < 10 do
  begin
    x := cost (i)
    writeln (i,x);
    i := i + 1
  end

end.
```

(20)
```
if x < 0 or y < 0 then s := x + y
```

(21)
```
while x < 0.0 do x := x * x;
```

(22) **while** x < 0 **then**

(23)
```
program vector (input, output);
var x : array [1..10] of real;
    i : integer;

begin

  { reads input from 10 separate input records }

  for i := 1 to 10 do readln (x[i]);
  for i := i to 10 do x[i] := x[i] + x [i+1];
  for i := 1 to 10 do writeln (x[i])
end.
```

Q6

Write a Pascal program to calculate and print the perimeters of a set of circles whose radii are 1., 1.5, 2., 2.5, ..., 10. cm, respectively
($\pi \cong 3.1415926$).

Q7

Write a Pascal program to sum the first 20 terms of the integer series

$$1 + 4 + 7 + 10 + \ldots$$

Q8

Write a Pascal program to sum the first 20 terms of the integer series

$$1 + 2 + 4 + 7 + 11 + 16 + \ldots$$

Q9

Write a Pascal program to sum the first 20 terms of the integer series

$$1 + 3 + 7 + 13 + 21 + \ldots$$

Q10

(For advanced students only.)

A mathematical quantity S defined by the infinite series

$$S = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} \quad \ldots$$

needs to be computed and tabulated for x = 0, .1, .2, .3, ..., 1. Write a Pascal program to do this using sufficient terms so that $S_{n+1} - S_n < 0.00001$. (Remember n! = 1 × 2 × 3 × ... × n.)

The next three questions demonstrate how a simple mathematical idea may be built up into a general purpose subprogram that others may use without them ever having any idea of the underlying mathematical technique. The aim is to find the area under a mathematically defined curve. You will probably find the first example (*Q11*) easy to code, however, the other two may appear beyond you. Should this be the case, be content at this stage just to work through the given answers and see that they make sense. The building of a final programming package (such as *Q13*) is no trivial matter, it is something that only comes with experience. It is included to indicate where you should be heading as you develop general purpose programs.

Q11

If you are asked to evaluate $\int_0^1 e^{-x}dx$ (*i.e.* find the area under the curve) then you could with a knowledge of

high school mathematics, quickly give the correct answer (I hope). At times we can be given very difficult functions to integrate and the only recourse is to a numerical approximation. One way of approximating $\int_0^1 e^{-x}dx$ numerically is to sum the area of the histogram sections approximating e^{-x} in Figure 12. This is done easily, but we must take particular care when treating the segments at x=0 and x=1. With the approximation shown above, we write

$$\int_0^1 e^{-x}\,dx \sim .1\left(\frac{e^0}{2} + \sum_{i=1}^{9} e^{-i*.1} + \frac{e^{-1}}{2}\right) .$$

(i) Write a Pascal program to evalue $\int_0^1 e^{-x}dx$ with the above discrete approximation.

(ii) How good is your answer? Do you notice any improvement if you run your program a second time with double the number of subdivisions?

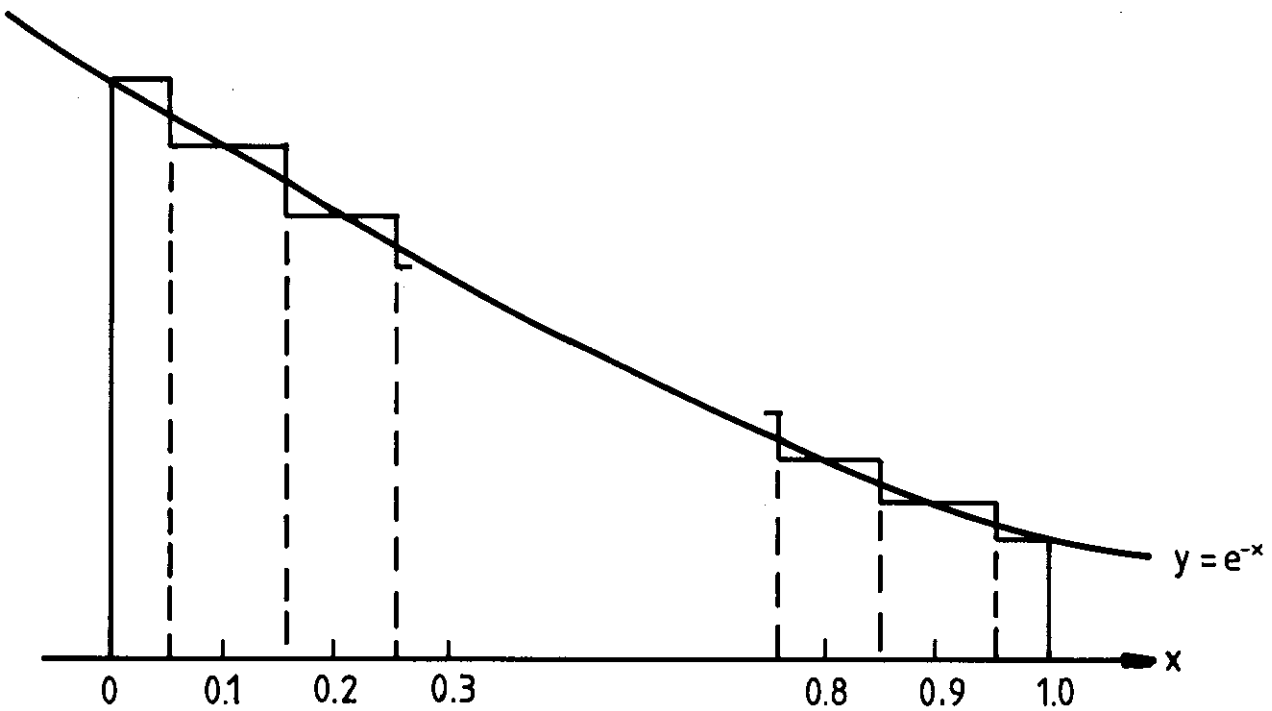


Figure 12 A numerical approximation to $\int_0^1 e^{-x}dx$

Q12

Let us generalise a little on the previous question. Mathematicians would write the above technique of integration (known as the histogram method) for a general function f(x) as

$$\int_a^b f(x)dx = h\left[\frac{f(a)}{2} + \sum_{i=1}^{n-1} f(x_i) + \frac{f(b)}{2}\right] ,$$

where $h = \frac{b-a}{n}$, and n+1 is the number of points x_i at which the function $f(x_i)$ is evaluated.

Should you wish to integrate an arbitrary function, say,

$$\int_0^{\pi} \sin 2x\ dx,$$

then the code written in the previous question could be changed, however, there are a number of places where the function being integrate must be altered. This leaves plenty of scope for error. A better method is to write a more general program where the function f(x) to be integrated is specified only once. To do this in Pascal f(x) is defined through a function definition. Write a more general integration program to invoke the supplied function, where the limits of integration a, b and the level of grid refinement are specified in the main program. Test your program by coding the function f(x) = sin2x and integrating between 0 and π with n=8. (Before you attempt this example you may care to reread Section 13.)

Q13

The previous exercises have not really considered the all important mathematical question of how good an approximation to the actual integral is our numerical procedure? In the limit as h=0 we know the two are identical, however, the practical question is how small need h become before the result is accurate enough. One approach commonly adopted in numerical computing is: let S_{n+1} denote the summation approximation to the integral with (n+1) grid points. If we start with n=2, the grid may be refined as we recompute S_{n+1} for n=4,8,16,...; effectively halving the mesh each time. We can continue halving the grid until the sequence S_{n+1} converges. A frequently used test is

$$\frac{S_{2n+1} - S_{n+1}}{S_{2n+1}} < \varepsilon .$$

This time try to generalise the program even further so that you have created a general purpose integration procedure that others could use. All the user of the package should have to do is

(i) write a function f(x) to specify the function being integrated; and

(ii) write a main program where the limits of integration a, b and a required accuracy factor ε are specified.

The main program can then invoke your integration procedure by a call such as

integrate (a,b, accuracy__required, intgrl, err)

where a, b and accuracy__required are real variables previously defined. The integral result is returned in the real variable intgrl, but err provides a safety valve should something go wrong. If the integral has been approximated successfully err returns an integer result of 0, otherwise 1. The integrate procedure should continue to refine the mesh for n $\leq$ 1024 if convergence is not obtained earlier. If convergence is not achieved err should be set to 1. Test your package on

$$\int_0^{\pi} sin2\ x\ dx$$

The following code is a possible section of the main program to invoke the package:

```
begin
  a := 0.0;
  b := 3.1415926;
  accuracy__required := 0.001;
  integrate(a, b, accuracy__required, intgrl, err);

  if err=0 then writeln (' integral ',intgrl)
       else writeln (' integral not found ')
end
```

Q14

(For advanced students only.)

The techniques of numerical integration outlined in *Q6* are fairly basic and obvious. What is perhaps not so obvious is that such 'crude' techniques can at times be highly effective. They are particularly valuable for integrating data that has already been discretised (*e.g.* economic data where sales statistics may have been averaged over certain time periods). When this is done, the use of more complex techniques would be meaningless anyhow!

An integration scheme for continuous functions that is more advanced than the histogram method of Q5 is Simpson's rule. You may have encountered this in a mathematics course. Simpson's rule approximates the integral of a function f(x) by

$$\int_a^b f(x)dx = \frac{h}{3}[f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + ... + 4f_{n-1} + f_n]$$

where $h = \frac{b-a}{n}$ and f_i = f(a + *i*h) for $i = 0,1,...,n$.

It is obvious that n must be an even number for Simpson's rule to be applied. (This is equivalent to saying there must be an odd number of grid points.) Try to generalise the package from the previous question so that n

takes the values 4,8,16,...; and then test it on the previous integral.

As an exercise in mathematics, try to calculate the following integrals with Simpson's rule, restricting n to 4 in each case.

$$(i) \quad \int_1^2 (5x^4 + 4x^3 + 6x^2 + 4x + 1)dx$$

$$(ii) \quad \int_0^1 (4x^3 + 3x^2 + 2x + 4)dx$$

Compare your numerical results with the analytic integral for each case. What comment could you make concerning the accuracy of this method of integration?

We now change our orientation in problem solving away from integration and consider the use of a computer in simulation studies for the next five exercises. These will require the use of a random number generator. You may assume that one exists, as was indicated in Section 9 (rand or rnd). Should one not be available on your computer, you could include the code shown in the determination of the π example in Section 19.

Q15

Write a Pascal statement that will generate a random real number in the range

$$0 < y < 10 \ .$$

Q16

Write a Pascal statement that will generate a random real number in the range

$$0.1 < y < 0.6 \ .$$

Q17

Write a Pascal statement that will generate a random integer from the set

1, 2, 3, ..., 15

Q18

Bill Smith travels to work five days each week by bus. Bill is an extremely methodical person who arrives at the bus stop at precisely 8.00 *a.m.* The government bus service is also extremely punctual and its vehicles call at Bill's stop at 8.01, 8.06 and 8.11 *a.m.* Each of these is capable of getting Bill to work on time. His employer, although somewhat flexible and tolerant, will dismiss him should he arrive at work late more than once a month (one month = four working weeks) averaged over the period of employment. Can Bill reasonably expect to retain his job in the long term if the number of passengers each bus can pick up at his stop varies randomly between 0 and 15 inclusive, and if he could find any random number of people up to 10 in front of him in the queue? Ignore any appeal to probability theory and write a Pascal program using a random number generator to simulate the bus stop situation and help estimate Bill's employment security with his present firm. Run the daily simulation for 1000 such mornings and print out the number of days per month Bill is late. Before you write any Pascal code, you might like to draw a flow chart to make sure you clearly understand the steps involved.

The question arises with all simulation exercises — have we studied sufficient cases to obtain a useful result? The answer requires a careful use of statistical techniques that are beyond high school mathematics and will not be discussed further.

Q19 (A more involved simulation exercise for advanced students.)

Assume that the Sydney Harbour Bridge authorities wish to reduce the average time a motorist spends waiting to get through the toll gate section of the bridge during peak hours. At present, there are 11 toll collecting stations - seven automatic and four manual. Can the traffic situation be improved by exchanging an automatic station for a manual station of *vice versa*? The authorities are unable to alter the number of stations by more than one either way for various reasons that do not concern you here. Your task is to help the authorities make the correct decision. They have studied traffic flow across the bridge for some time and have come up with the statistics given below. You have been engaged as a programmer to enable the authorities to try out the various configurations and to estimate the average time that each car spends waiting to get through the toll area.

(1) The correct toll for all vehicles is 20¢ per vehicle.

(2) There are seven automatic and four manual stations.

(3) Every five seconds either seven, eight or nine cars enter the toll area.

(4) On average, 6.4 out of every 10 cars will select the automatic station.

(5) Of the drivers selecting an automatic station, eight out of ten will choose the station with the shortest queue. The other drivers will select a queue at random.

(6) Of the drivers selecting the manual station, seven out of ten will choose the station with with the shortest queue. The other drivers will select a queue at random.

(7) It takes five seconds to process the correct money at an automatic station, provided the coins are not dropped.

(8) If the coins are dropped at an automatic station, it takes 20 seconds to retrieve them.

(9) At a manual station the following times are involved, provided the toll money is not dropped:

20¢	5 seconds
$1	10 seconds
$2	15 seconds
≥ $5	20 seconds

(10) If the money is dropped at a manual station, it takes five seconds to retrieve.

(11) One out of every hundred motorists will drop the money.

(12) One out of every hundred motorists will stall the vehicle at an automatic station and take 10 seconds to re-start.

(13) Two out of every hundred motorists will stall at a manual station and take ten seconds to re-start.

Write a Pascal program to simulate the above bridge situation for a one-hour time span. Assume no motorists are at the toll gates when the hour commences. Determine the average time a motorist spends waiting, and the average length of the queue. Also determine the length of the longest queue that forms. Given the constraints mentioned previously what is the best action open to the authories?

Q20

Write a program that will

(1) accept the four coefficients of a cubic polynomial

$$f(x) = a + bx + cx^2 + dx^3 ;$$

(2) accept an estimate x_o of one root of the equation $f(x) = 0$;

(3) improve the estimate of the root by the Newton-Raphson method, i.e.

i.e. $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} ;$$

(4) the process can be considered to have converged if

$$\frac{x_{n+1} - x_n}{x_n} < 0.001 \quad ;$$

(5) print out the improved estimate of the root; and

(6) allow only nine iterations. If convergence has not been achieved, print a message warning of this.

Q21

Write a program that will read in a set of ten real numbers from one input record and then print them out in the reverse order. (Note — you need to use subscripted variables to do this effectively.)

Q22 (For advanced students only)

Read in a set of ten real numbers from one input record and write a program that will sort these into descending order.

23. REFERENCES

Cox, G., Tobias, J. and Perkins, H. [1980] - Pascal 8000 Reference Manual Version 2.0. AAEC/M

Welsh, J. and Elder, J.E. [1979] - Introduction to Pascal. Prentice Hall, New Jersey.

APPENDIX A
ANSWERS TO SELECTED QUESTIONS

Q1

(1)	invalid (should be 1.0)	(2)	variable
(3)	variable	(4)	integer number
(5)	invalid (should be 0.6)	(6)	variable
(7)	invalid (first must be alphabetic)	(8)	integer number
(9)	variable	(10)	invalid (expression)
(11)	invalid (comma not allowed)	(12)	invalid (expression)
(13)	variable	(14)	real number
(15)	variable		

Q2.

(1) y := 0.5 * (b + c)

(2) y := sqrt(b * b - 4.0 * a * c) / (2.0 * a)

(3) y := (x + a) / 4.1415926

Q3.

(1) invalid statement (cannot assign a real to an integer)

(2) i=4

(3) i=3

(4) invalid (should be 2.0)

(5) u=1.9

(6) k=6

(7) i=2

(8) invalid (attempt to alter a constant)

Q4.

(1) s := sqrt(x * x + y * y + z * z)

(2) y := exp(x)

(3)
```
w1 := exp(x);
w2 := 1.0 / w1;
u := (w1 - w2) / (w1 + w2)
```

(4) v : = tan(x)

(5) c := -ln(abs(1.0 + a * a * a))

(6)
```
w1 := a * x;
y := (exp(w1) + exp(-sqrt(w1))) * 0.33333333333333333
```

Q5.

(1) correct

(2) correct

(3) 1. should be 1.0

(4) limits for a **for** loop must be integer

(5) (remember anything is invoking a procedure, as yet undefined)

(6) correct

(7) correct

(8) correct

(9) probable error in logic; there is no escape from the **while** block because i is never incremented

(10) incorrect ; is not permitted before **else**

(11) incorrect (should be **if** a < b **then** x := -x)
(12) correct
(13) a real value may not be assigned to an integer variable
(14) correct
(15) logical expression should be (a = b) and (b = c)
(16) correct
(17) incorrect 0...100 should be 0..100
(18) correct
(19) the result of a function evaluation must be returned through the appropriate function name (cost)
(20) Should be **if** (x < 0) **or** (y < 0) **then** s := x + y
(21) correct
(22) should be **while** ... **do** ...
(23) Pascal syntax will be passed by the compiler, but an execution error will occur on the second **for** statement when x[11] is referenced.

Q6.

```
program perimeter (output);
const pi = 3.1415926;
var r, perimeter : real;
begin
  r := 1.0;
  while r <= 10.0 do
  begin
    perimeter := 2.0 * pi * r;
    writeln (' perimeter of circle ', r,' cm is ', perimeter);
    r := r + 0.5
  end

end.
```

Q7.

```
program sum (output);
var sum, j, item : integer;
begin
  sum := 0;
  item := 1;
  for j := 1 to 20 do
  begin
    sum := sum + item;
    item := item + 3
  end;
  writeln (' sum ',sum)
end.
```

Q8.

```
program sum (output);
var sum, j, item : integer;
begin
```

```
  sum := 0;
  item := 1;
  for j := 1 to 20 do
  begin
    sum := sum + item;
    item := item + j
  end;
  writeln (' sum ',sum)
end.
```

Q9.

```
program sum (output);
var sum, j, item :  integer;
begin
  sum := 0;
  item := 1;
  for j := 1 to 20 do
  begin
    sum := sum + item;
    item := item + 2 * j
  end;
  writeln (' sum ',sum)
end.
```

Q10.

```
program calculate__s (output);
var x, sum, denom, num, term : real;
i                            : integer;

begin
  x := 0.0;
  while x <= 1.0 do
  begin
    sum := 1.0;
    i := 1;
    num := -x;
    denom := 1.0;
    term := num / denom;

    while abs(term) > 0.00001 do
  begin
    sum : = sum + term;
    num := num * (-x);
    i := i + 1;
    denom := denom * i;
    term := num / denom
  end;
  writeln (x,sum);
  x := x + 0.1
  end
end.
```

Q11.

```
program integral (output);

var sum : real;
    i   : integer;

begin
  sum := 0.5;
  for i := 1 to 9 do sum := sum + exp( -i * 0.1);

  sum := 0.1 * (sum + exp( -1.0)/2.0);
  writeln (' integral = ',sum)

end.
```

Q12.

```
program integral (output);
var sum, a, b, h, x : real;
    i, n            : integer;

function f( z : real) : real;
begin
  f := sin(2.0 * z)
end;

begin
  a := 0.0;
  b := 3.1415926;
  n := 8;

  h := (b - a) / n;
  sum := f(a) / 2.0;
  x := a;

  for i := 1 to n-1 do
  begin
    x := x + h;
    sum := sum + f (x)
  end;

  sum := h * (sum + f (b)/2.0);
  writeln(' integral = ',sum)
end.
```

Q13.

```
program integral (output);
var intgrl, a, b, accuracy__required  : real;
    err                               : integer;

function f ( z : real) : real;
begin
  f := sin 2.0 * z
end;

procedure integrate(a, b, accuracy : real;
var result : real;
    err    : integer);
```

```
var h, x, sum, sumold : real;
    n, i              : integer;

begin
  n := 2;
  err := 1;

while ( n <= 1024 ) and (err <> 0) do
   begin
     h := (b - a) / n;
     sum := f (a) / 2.0;
     x := a;

     for i := 1 to n-1 do
     begin
       x := x + h;
       sum := sum + f (x)
     end;

     sum := h * (sum + f (b) / 2.0);
     if n >= 4 then
       if abs((sum - sumold) / sum) < accuracy  then
       begin
         err := 0;
         result := sum
       end;

       sumold : = sum;
       n := n * 2
  end
end;

{ Main program to define data and invoke the integration package }

begin
  a := 0.0;
  b := 3.1415926;
  accuracy__required := 0.001;

  integrate(a, b, accuracy__required, intgrl, err);

  if err=0 then writeln (' integral ',intgrl)
           else writeln (' integral not found ')

end.
```

Q14.

```
program integral(output);
var intgrl, a, b, accuracy__required  : real;
    err                               : integer;

function f ( z : real) : real;
begin
  f := sin(2.0 * z)
end;

procedure integrate(a, b, accuracy : real; var result : real;
                                           var err    : integer);

var h, x, sum, result__old  : real;
    n, i                    : integer;
```

```
begin
  n := 4;
  err := 1;
  while ( n <= 1024 ) and (err <> 0) do
  begin
    h := (b - a) / n;
    result := f (a) + f (b);
    sum := 0.0;
    i := 1;
    repeat
      sum := sum + f (a + i*h);
      i := i + 2
    until i > n - 1;
      result := result + 4.0 * sum;

      sum := 0.0;
      i := 2;
    repeat
      sum := sum + f (a + i * h);
      i := i + 2
    until i > n - 2;
    result := (h / 3.0) * (result + 2.0 * sum);

    if n >= 8 then
    if abs ((result - result_old) / result) < accuracy
    then  err := 0;

    result_old := result;

    n := n * 2
  end
end;

{ Main program to define data and invoke the integration package }

begin
  a := 0.0;
  b := 3.1415926;
  accuracy_required := 0.001;
  integrate (a, b, accuracy_required, intgrl, err);

  if err=0 then writeln (' integral ',intgrl)
           else writeln (' integral not found ')
end.
```

Q15.

```
y := 10.0 * rand
```

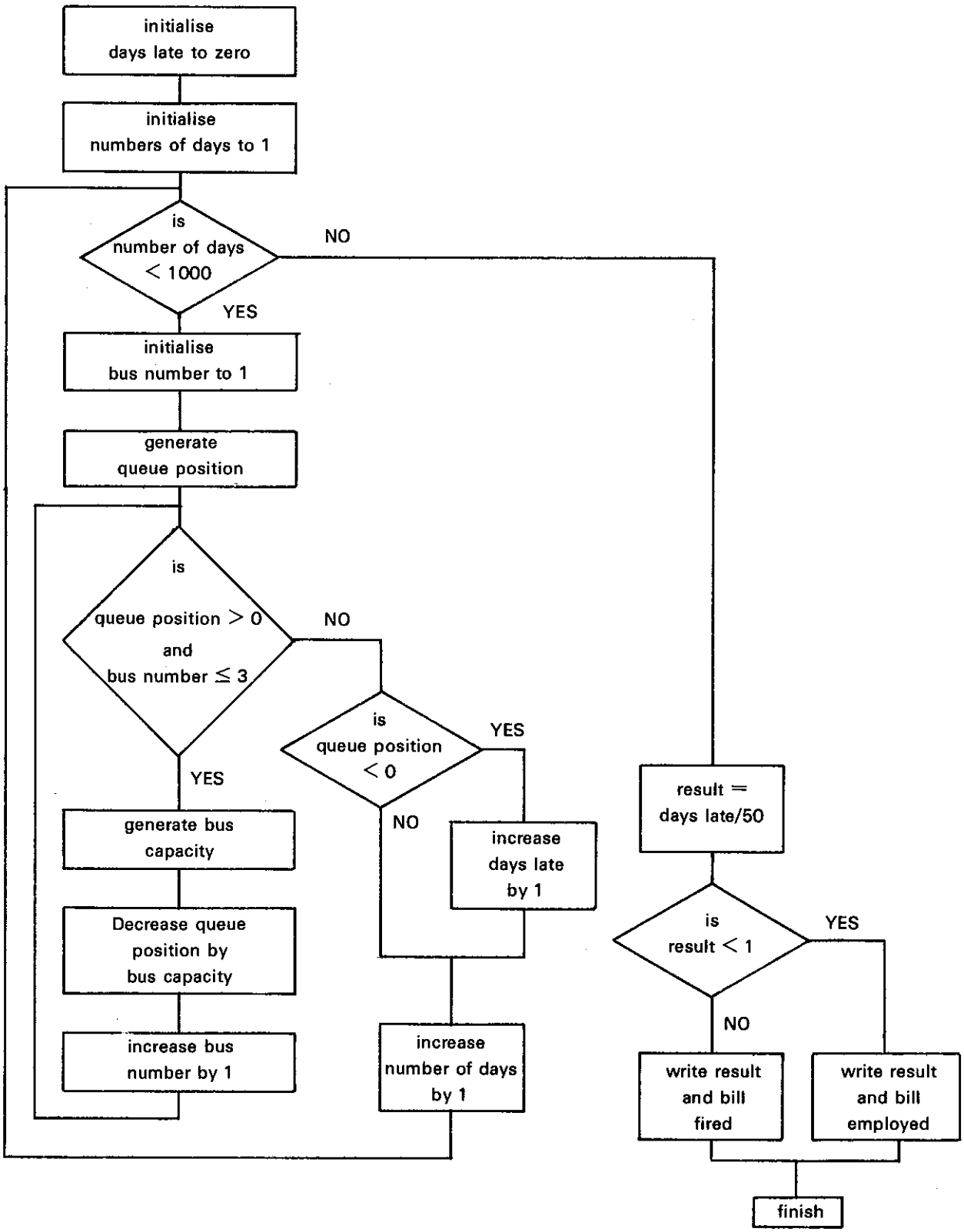
Q16.

```
y := 0.5 * rand + 0.1
```

Q17.

```
  i := trunc (rand * 15.0) + 1
or i := round (rand * 14.0) + 1
```

where i is declared integer



Q18. Figure Flow Chart

Q18.

```
program bus (output);

var no_of_days, no_days late, no_queue, bus_no,
    no_on_bus, bus_capacity  : integer;
    fraction_late            : real;

begin
  no_days_late := 0;

  for_no_of_days := 1 to 1000 do
  begin
    bus_no := 1;
    no_queue := trunc(rnd * 11 + 1);
    while (no-queue > 0) and (bus_no <= 3) do

    begin
      bus_capacity := trunc (md * 16);
      no_queue := no_queue - bus_capacity;
      bus_no := bus_no + 1
    end;

    if no_queue > 0 then no_days_late := no_days_late + 1
  end;

  fraction_late := no_days_late / 50.0;
  if fraction_late < 1.0 then writeln (' bill employed ')
  else writeln (' bill fired ')

end.
```

Q19.

No solution is provided to this question

Q20.

```
program cubic (input, output);
var a, b, c, d, xn, xnp1  : real;
    i                     : integer;

begin
{ program to find roots of a cubic
  coefficients of cubic read from input records }

readln (a, b, c, d);

{ read estimate from input record  }

readln (xnp1);

i := 1;

repeat

  xn := xnp1:
  xnp1 := xn - (a + xn * (b + xn * (c + d * xn)))/(b + xn * (2.0 * c + xn * 3.0 * d));
  i := i + 1;
until  (abs((xnp1 - xn) / xn) < 0.001)
```

```
or  (i > 9);

if abs ((xnpl - xn) / xn ) < 0.001
   then writeln (' root is ',xnpl)
   else writeln (' no roots possible ')

end.
```

Q21.

```
program reverse (input, output);

var x : array [1..10] of real;
    i  : integer;

begin
  for i := 1 to 10 do read (x[i]);
  for i := 10 downto 1 do write (x[i])
end.
```

Q22.

```
program sort(input,output);

var temp  : real;
    i, j  : integer;
    x     : array [1..10] of real;

begin

  for i := 1 to 10 do read ( x[i] );
  for i := 1 to 9 do

  begin

  for j := i + 1 to 10 do
  if x [j] >x [i] then
  begin
    temp := x [j] ;
    x [j] := x [i] ;
    x [i] := temp
  end;
 end;
 for i := 1 to 10 do write (x [i] )
end.
```

APPENDIX B

USE OF NON-PASCAL SPECIAL FUNCTIONS AND PROCEDURES

Special functions and procedures written in some languages other than Pascal may be invoked from a Pascal program. This allows us to take advantage of the large number of mathematical routines written in FORTRAN. The desired function or procedure must exist in an appropriate library. If we wish to invoke the normal random number generator on the Lucas Heights computing facility the following identification must be inserted in the Pascal code after the **var** declaration

```
function rand: real; fortran;
```