Chapter 3

STRUCTURED PROGRAMMING FACILITIES

3.1 INTRODUCTION

3.1.1 Structure charts (top-down design)

The BBC microcomputer has a wide range of facilities that offer the programmer the opportunity to write clear and easily followed programs. To make the best use of this potential it is worth spending some time in careful analysis of the problem that is to be programmed in order to organize the program in the most appropriate way. This analysis should be done before any of the BASIC code is written. It will be time well spent and if the decisions reached are recorded they will be useful in making modifications to the program at a later date. There are two main techniques for recording the decisions about the form of the program: the *flow chart*, and the *structure chart* produced in *top-down design*.

It is now accepted that top-down design is a method of structured programming that can lead to good programs. The name comes from the idea of considering a problem as a whole, then breaking it down into a small number of main sections, and then progressively breaking each section down into smaller sub-sections until each sub-section can readily be coded in the programming language to be used. The sections specified are recorded in a structure chart (not to be confused with a flow chart), which should show the overall structure of a program in diagrammatic form. The structure chart should be easily understood and so should not contain complex flow lines that belong in flow charts.

There are three main logic structures which may be represented in a structure chart: sequence, iteration and selection. It is appropriate to consider each of these in turn.

Sequence

The first stage in the analysis of a problem that is to be programmed is to break it down into a small number of identifiable sections. At this stage of analysis the natural sequence of different processes provides an obvious

division of the problem. Each of these is identified by a rectangular box called a sequence box. Figure 3.1 illustrates the use of sequence boxes.

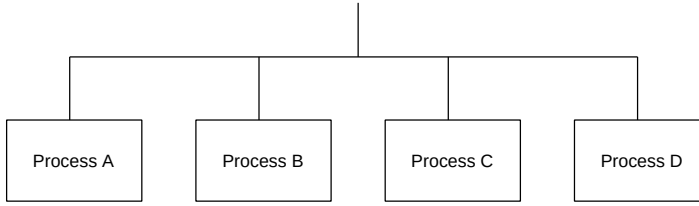

Figure 3.1 The use of sequence boxes in a structure chart.

The boxes should be read from left to right; the line links them to show they form the subdivided parts of a whole unit. Thus process A is followed by process B which in turn is followed by process C and last comes process D.

Iteration

An iteration is a process which involves repetition (possibly of a number of sequence events). A process which is repeated is represented in a structure chart by a rectangular box with an asterisk in the top right hand corner (see Figure 3.2). The number of times that the process is repeated is not necessarily indicated.

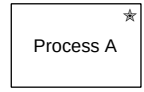

Figure 3.2 Sequence box for a repeated process.

Selection

If only one process is to be chosen from two or more possible processes a selection construction is used. The processes may be regarded as a number of options. The selection construction is represented by a series of rectangular boxes each with a small circle in the top right hand corner. The group from amongst which the choice is made is connected by a line but in this case only one process is executed. Figure 3.3 illustrates such a selection group.

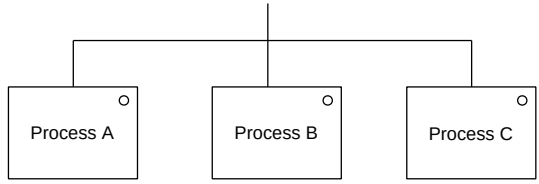

Figure 3.3 Representation of a selection group.

A structure chart is built up using the three key structures. It starts with a single box indicating the whole program. Examine the program to sum the first 10 prime numbers (Example 2.6) and its structure chart in Figure 3.4.

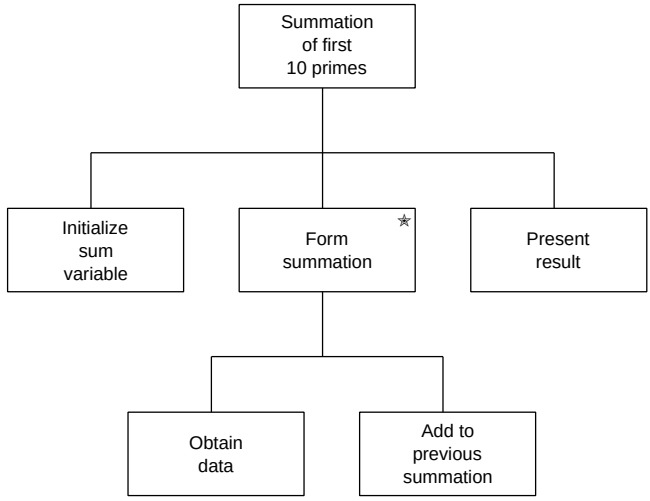

Figure 3.4 Structure chart for summation of primes.

The structure chart in Figure 3.5 is for Example 2.11 in which an array is used to solve the same program requirement. Note how much easier it is to follow and consequently the case for the use structure charts can be seen. It is worth pointing out that these programs are really too simple to need a structure chart.

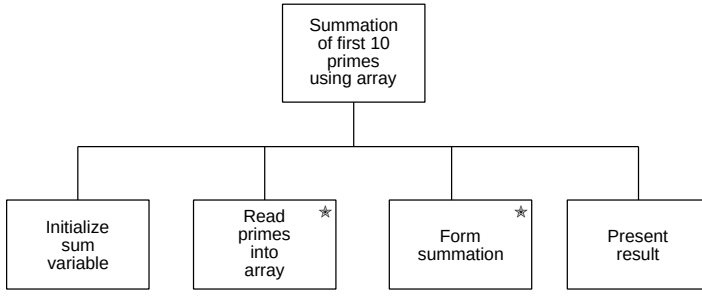

Figure 3.5 Structure chart for summation of primes using an array.

3.1.2 Good programming style

It is important that even the relative newcomer to programming seeks to adopt a systematic, coherent and professional approach to the design and implementation of computer programs.

There are general rules which can be applied by the BASIC programmer which will reduce the chances of error and confusion. Their importance will become more apparent as the sophistication of the problems tackled increases.

1. First analyse the problem systematically (before you start typing!) Decide on the best strategy to solve the problem.
2. Break the problem down into a series of simpler parts (modules). Each module might well correspond to one sequence box of the structure chart. Wherever possible, particularly in an extensive program, write each substantial module as a procedure. One of the great advantages of BBC BASIC is the provision of proper procedures and functions, in addition to the standard subroutines. Each procedure, function and subroutine should be separately tested and verified.
3. As an aid to clarity, use variable names which will remind anyone reading the program of the actual quantities to which they refer. Lower case can be used for variable, procedure and function names and this helps in distinguishing them from keywords. However, it makes typing more difficult, with the need to keep operating the CAPS LOCK key. (In this book, a convention has been used of lower case for procedure and function names, but upper case for variable names).
4. For the same reason, use REM statements to explain the function of any potentially unclear lines or parts of the program.
5. Try to avoid using GOTO instructions unnecessarily. This is one of the more unfortunate features of BASIC where some thought will often reveal a better way to write the program without them. Extensive use of GOTO leads to programs which are very difficult to check and underStand. Consider it as a challenge to keep GOTOs out of your programs.
6. In the case of extensive programs, the analysis of stages (1) and (2) should be separately documented.

3.2 FUNCTIONS

BASIC has many built-in functions, such as SIN(X), COS(X), RND(X), INT(X) and LOG(X). In addition, it is possible for the user to define his own functions, each consisting of a separate section of program which returns a single value to the calling statement in the main program. Functions created

by the user are used in exactly the same way as the built-in functions, but they are preceded (without any intervening spaces) by the letters FN. For example, function ‘round2’ would be called by

```
10 MONEY=FN_round(X)
```

The underline character is used here, and throughout this book, to give a visiual impression of a space between FN and its name since the space character cannot be used. (The same convention is followed for procedures.)

Functions may be defined outside the main body of the program and it is conventional to collect all the functions (and procedures) at the end of a program, after the final END statement. In any case they must not be in a part of the program where the lines comprising the function would be executed other than due to a function call.

A function is defined by a section of program that starts with a line such

```
100 DEF FN_round2(R)
```

and ends with a line such as

```
140 =X
```

In BBC BASIC a function may contain several statements and can be as long as necessary. The variable R used in the definition of a function is called the *formal parameter*. The value or expression used when the function is used is called the *actual parameter* and has no relationship to the formal parameter. The value of a variable used as an actual parameter is unchanged after use of the function even if the value of the formal parameter is reassigned within the function. Functions may have more than one parameter, separated by commas.

Example 3.1

This program uses a function FN_round2 which rounds a given number to two decimal places. This might be a useful function in a program involving monetary calculations or for rounding off average or scaled exam marks. The function is called at line 50 and the rounding to two decimal places is done by multiplying by 100 and then taking the integer part of the result with the INT(X) function. The integer result is then divided by 100 to get the required two decimal place form of the number.

```
10 REM TO ILLUSTRATE THE USE OF DEF FN
20 MODE 7
30 REPEAT
40 INPUT "Give a number to more than 2 decimal places ",R
50 MONEY=FN_round?(R)
60 PRINT "To the nearest penny £";MONEY
70 PRINT "Do you wish to do a nw rounding?" '"Answer Y or N."
```

```
 80 UNTIL NOT (GET$="Y")
 90 END
100 DEF FN round2(R)
110 REM ROUND A GIVEN NUMBER TO 2 DECIMAL PLACES
120 X=INT(R*100+0.5)/100
130 =X
```

The function definition is spread over several lines for clarity, but if necessary this particular definition could be in one line as

```
100 DEF FN_round2(R)=INT (R*100+0.5)/100
```

(There is another way of achieving this presentation with @%, which is the print format control parameter - see Chapter 11.)

In BBC BASIC a function can have several parameters so that the distinction between procedures and functions is blurred. However, only the function can return a value without the need for a common variable. (To return more than a single variable you would clearly have to use a procedure.)

Example 3.2

The following program uses a function closely related to FN_round2, together with two of our other structures, IF...THEN...ELSE and REPEAT...UNTIL.

```
 10 REM TO ILLUSTRATE VARIOUS PROGRAMMING STRUCTURES
 20 MODE7
 30 REPEAT
 40 FLAG=0: REM USED TO INDICATE ROUNDING CAN BE DONE (FLAG=0)
 50 INPUT "Give a number to more than 2 decimal pLaces ",R
 60 INPUT "Hem many decimal places ",NDP
 70 IF (NDP=2) THEN MONEY=FN_round(R,2) ELSE IF NDP=3 THEN
    MONEY=FN_round ( R , 3 ) ELSE FLAG=1
 80 IF FLAG=0 THEN PRINT "To ";NDP;" decimal places "; MONEY
    ELSE IF NDP=INT(NDP) THEN PRINT "Out of range of available
    functions." ELSE PRINT "The number of decimal places
    required"'"must be given as a positive integer."
 90 PRINT "Do you wish to do a new rounding?"'"Answer Y or N."
100 UNTIL NOT(GET$="Y")
110 END
120 DEF FN_round(R,N)
130 REM ROUND A GIVEN NUMBER TO N DECIMAL PLACES
140 X=INT(R*10^N+0.5)/10^N
150 =X
```

In this program a variable FLAG is used to indicate whether rounding can be done. A value of zero for FLAG means that the function can be used, and at line 70 FLAG is set to 1 if FN_round can not be used. Try running the program with a range of values for R and NDP including non-integer values

for NDP. The program should cope with this unreasonable data. Note that FN_round allows rounding to any number of decimal places specified by NDP. The IF...THEN statement at line 70 limits the use FN round in this program. This restriction has been introduced deliberately to illustrate the use of a function within an IF...THEN statement.

Exercise 3.1

Write a program to define functions for the hyperbolic sine and cosine and test, for a range of values of x, the quantity

```
FN_hcs(X)*FN_hcs(X) - FN_hsn(X)*FN_hsn(X)
```

where functions FN_hcs and FN_hsn are defined by

```
FN_hcs(X) = (EXP(X)+EXP(-X))/2
FN_hsn(X) = (EXP(X)-EXP(-X))/2
```

Exercise 3.2

Given that 0°C corresponds to 32°F and that 100°C corresponds to 212°F, devise a function to convert centigrade to Fahrenheit and another one to convert Fahrenheit into centigrade.

3.3 LOCAL PARAMETERS

It is possible to define variables used in a function to be LOCAL to that function so that variables in the main part of the program are not corrupted by use of the same variable name in a function. Such variables must be declared as local by a line such as

```
110 LOCAL IPART,DPART
```

Any variable not declared as LOCAL is ‘common’ or ‘global’; that is, it takes the value assigned in the function when next used in the main program; or when used on the right hand side of an assignment in the function it retains any previously set value.

The use of functions is not restricted solely to where a value is to be returned to the calling program. The value returned can be ignored or given to a variable such as DUMMY which is not used elsewhere in the program.

Example 3.3

This program uses two functions FN_delay and FN_response. FN_delay produces a random delay up to a maximum value by using the standard RND function and the pseudo-variable TIME. The function FN response is used to measure the time between the random display of a random character and the

user response by pressing the selected key. The FN response also uses the RND function. Functions can be nested 26 deep. Examine the program and note that the variable START is used in both functions but is declared to be LOCAL. Try printing START in immediate mode after the program has been run.

```
 10 REM FN response
 15 MODE 7
 20 REPEAT
 30 CLS
 40 PRINT "Press the letter given below as quickty as possible
    after it is displayed"
 50 REACTION-TIME=FN_response(A$)
 60 PRINT: PRINT "You took ";REACTION-TIME;" seconds to press
    ";A$
 70 PRINT "For another trial press Y "
 80 A$=GET$
 90 UNTIL A$<>"Y"
100 END
110 DEF FN_delay(CENTISEC)
120 LOCAL START
130 START=TIME
140 REPEAT
150 UNTIL (CENTISEC+START)<TIME
160 =CENTISIC
170 DEF FN_response(A$)
180 LOCAL D,START,B$,X,Y
190 DUMMY=FN_delay(RND(150))
200 A$=CHR$(64+RND(26))
210 START = TIME
220 X=RND(39)
230 Y=3+RND(15)
240 REPEAT
250 PRINT TAB(X,Y);A$
260 B$=GET$
270 UNTIL B$=A$
280 =(TIME-START)/100
```

Try the effect of adding the line

```
215 FN_delay(100)
```

This has the effect of using the LOCAL variable START in each function at the same time so that it has different values in each function. This is permitted since START is LOCAL in both functions. This alteration will mean that the minimum response time will now be one second.

3.4 PROCEDURES (DEF PROC)

Procedures are similar in some respects to functions. They do not, however, appear at the right hand side of expressions and do not return values to the calling program as functions can do. There are two important restrictions on the use of procedures in BBC BASIC. Firstly, only common variables can be used to transfer values back to the main program. Secondly, whole arrays can not be used as parameters, although array elements may be used as actual parameters when a procedure is called. Procedures are defined by lines such as

```
150 END
200 DEF PROC_nice_one(FRED,JIM,ETC)
    ...
    ...
    ...
240 ENDPROC
```

They are used within a program by a 'call' of the form

```
70 PROC_nice_one(ALICE,JOAN,ANON)
```

Procedure definitions should be outside the main program and are usually placed after END along with the function definitions. As with functions, variables used in procedures are held to be the common variables in the main program unless they are specifically declared as LOCAL. The procedure may have any number of formal parameters or none at all. The actual parameters are unchanged after the procedure has been called.

Exercise 3.3

The program in Example 3.3 uses a function, FN_delay, which does not return a value to the calling program. It is thus appropriate to rewrite this program to use a procedure to give the required delay. After you have made and tested this modification alter the procedure to give a one second delay without the need for a formal parameter.

Example 3.4

```
10 REM DEF PROC DEMONSTRATION PROGRAM
20 CLS: PRINT
30 READ FRED,JIM,ETC
40 INPUT "Three ages in years < 70 ",ALICE,JOAN,ANON
50 SUM=FRED+JIM+ETC
60 PRINT "Sum of men's ages = ";SUM
70 PROC_nice_one(ALICE,JOAN,ANON)
80 PRINT "Return to main program "
90 PRINT "FRED = ";FRED
```

```
100 PRINT "JIM = ":JIM
110 PRINT "ETC = ";ETC
120 PRINT "Sum of women's ages = ";SUM
130 PROC nice_one(RND(70),RND(70),RND(70))
140 PRINT "Sum of 3 random ages = ";SUM
150 END 200 DEF PROC_nice_one(FRED,JIM,ETC)
210 PRINT "Procedure NICE_ONE entered"
220 PRINT "INITIAL VALUES ";FRED,JIM,ETC
230 SUM=FRED+JIM+ETC
240 ENDPROC
250 DATA 27,23,30
```

In this example FRED, JIM and ETC are the formal parameters of the procedure, and all formal parameters of a procedure are LOCAL to it even if FRED, JIM and ETC are also in the main program. One rather serious restriction, as already mentioned, is that it is not possible to return values through the formal parameters as is the case in languages such as Pascal and FORTRAN. Values to be used subsequently in the main program must be passed through common variables such as SUM. These features of procedures are illustrated by this trivial example. At line 30 FRED, JIM and ETC are given values in the READ statement. Lines 90 to 110 demonstrate that these values are retained even when the procedure is called, as the formal parameters are automatically LOCAL. The variable SUM is calculated at line 50 and printed at line 60. The variable SUM is common to the main program and the procedure so that at line 120 when it is printed again it will have a value determined by the procedure. The formal arguments of the procedure may be variables, as at line 70, or arithmetic or other valid statements as at line 130 where the RND function is used.

Example 3.5

The procedure in this example is used to solve the quadratic equation $ax^2+bx+c=0$. It shows a more useful application of a procedure within an IF...THEN...ELSE, where GOTOs would otherwise have been needed. It would of course be possible to include the IF...THEN statement within the PROCedure, but the point here is to illustrate the use of a procedure call in an IF...THEN statement.

```
10 REM SOLUTION OF QUADRATIC EQUATIONS
20 MODE 7
30 PRINT
40 PRINT "SoLution of quadratic of the form:"'"A*X*X + B*X +
   C = 0"
50 REPEAT
60 PRINT
70 INPUT "Coefficients of quadratic A,B and C separated by
   ','.",A,B,C
80 REM USE OF CONDITIONAL TEST TO PREVENT PROGRAM FAILURE ON
```

```
      EVALUATION OF NEGATIVE ROOT
   90 IF B*B<4*A*C THEN PRINT "No simple roots": PRINT ELSE
      PR0C_quad(A,B,C)
  100 PRINT "Do you wish to give new data? ";
  110 A$=GET$
  120 PRINT
  130 UNTIL A$<>"Y"
  140 END
  150 DEF PR0C_quad(AQ,BQ,CQ)
  160 ROOT=SQR (BQ*BQ-4*AQ*CQ)
  170 PRINT "Roots are ";(-BQ+ROOT)/(2*AQ);" AND ";
      (-BQ-ROOT)/(2*AQ)
  180 ENDPROC
```

Exercise 3.4

Write a procedure to examine a list of 10 numbers stored in an array, find the largest and the smallest, and return them in common variables LOW and HIGH. Test the procedure in a simple program.

Exercise 3.5

Write a procedure to evaluate the first N prime numbers and store them in an array PRIME(J) of dimension N. Incorporate the procedure in a program to calculate the sum of the first N prime numbers, where N is an integer supplied by the user. (Hint: write and test the procedure for the prime numbers then look at Chapter 2 for the summation program using an array.)

3.5 OTHER PROGRAM STRUCTURES

3.5.1 Subroutines (GOSUB)

It is usually preferable to use procedures rather than GOSUB in BBC BASIC, since they are available. However, the BBC BASIC supports subroutines so that programs taken from listings for other computers can still be run.

The GOSUB facility has some attractions in that the routine can be entered at any point. However, subroutines do not have the possibility of LOCAL variables or formal parameters.

3.5.2 Computed GOTO and GOSUB (ON...GOTO and ON...GOSUB)

The usual numerical sequence in which the BASIC program statements are executed may be modified not only by REPEAT...UNTIL and GOTO, but by the statement

```
10 ON <value> GOTO XX,YY,ZZ
```

where XX, YY and ZZ are line numbers. If the <value> is 1 then the effect is tn GOTO XX. If the <value> is 2 then the program goes to YY. Finally if the

<value> is 3 then the effect is to GOTO ZZ. If the <value> is less than 1 or exceeds 3 then an error occurs. Strictly the values for 1, 2 and 3 can lie in the range

```
0.9999999999  to 1.99999999975
1.99999999976 to 2.99999999953
2.99999999954 to 3.99999999952
```

respectively on input of data. It is left as an exercise for the reader to determine the range for higher switches.

Example 3.6

The program structure chart in Figure 3.6 illustrates the program below, which demonstrates a simple ON...GOTO statement. The user is asked to supply a number at line 20. This should be 1, 2 or 3. The program then prints ONE, TWO or THREE as appropriate by use of the ON...GOTO statement at line 30. Examine the program and run it.

```
 10 REPEAT
 20 INPUT "Choose a value for I ",I
 30 ON I GOTO 40,60,80
 40 PRINT "Value of I = ONE"
 50 GOTO 90
 60 PRINT "Value of I = TWO"
 70 GOTO 90
 80 PRINT "Value of I = THREE"
 90 UNTIL FALSE
100 END
```

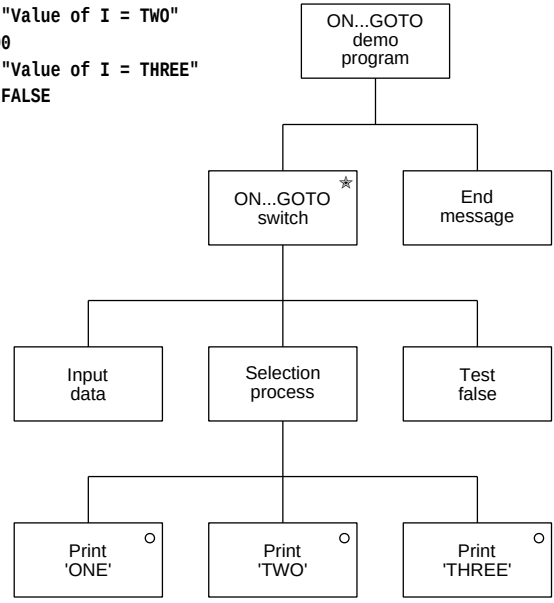

Figure 3.6 Structure chart for the ON...GOTO demonstration program.

Try values for I of 1, 2, 3, non-integer values less than 4 and then a value greater than 4. In the latter case you will get the error message

```
ON range at Line 30
```

This can be overcome by modifying line 30 to be

```
30 ON I GOTO 40,60,80 ELSE 90
```

Note: the use of ELSE with ON.. .GOTO upsets the return from functions and procedures in the first version of BASIC (BASIC I), and so must not be used with a function or procedure. An alternative which overcomes the problem on BASIC I is to use the ON ERROR GOTO statement by adding the line

```
25 ON ERROR GOTO 20
```

and leaving line 30 as

```
30 ON I GOTO 40,60,80
```

Remember to turn the error trapping off with ON ERROR OFF when the range of this REPEAT sequence is ended. Note that ON ERROR GOTO 90 will not work since you cannot return into a FOR...NEXT or REPEAT...UNTIL loop, (in fact this generates error 43, No REPEAT) but note that it is permissible to add the line

```
5 ON ERROR GOTO 10
```

Exercise 3.6

Write a program that asks the user the time (on a 24 hour clock) and then responds with the appropriate message from the list GOOD MORNING, GOOD DAY, GOOD AFTERNOON, GOOD EVENING or GOOD NIGHT (in sequence through the 24 hours).

Exercise 3.7

Write a program to calculate the time in hours, minutes and seconds from the psuedo-variable TIME.

Exercise 3.8

Write a program to produce a calendar for any year since the year 1800 given that the first of January in that year was on a Wednesday.

3.6 RECURSIVE USE OF PROCEDURES AND FUNCTIONS

The BBC microcomputer permits the recursive calling of procedures and functions. This is valuable for the evaluation of a mathematical recursion formula in which successive terms of a series can be expressed in terms of

previous terms, and for other problems of this type. For example, the set of natural numbers, the integers, may be defined recursively in the following way.

1. 1 is a natural number.
2. The successor of a natural number is a natural number, where the successor of a number is obtained by adding one to that number.

The power of the recursive definition is its ability to define a set consisting of an infinite number of elements using only two statements. In a similar way a recursive procedure or function can specify an infinite number of computations using a few program statements and without looping. Using recursion in solving a problem often leads to a new problem which is similar to the original but smaller in scope.

Example 3.7

A simple example of a recursive solution is the computation of the factorial of a non-negative integer N. The non-recursive solution is given in the program below. This program uses a FOR...NEXT loop to evaluate N!

```
10 REM FACTORIAL EVALUATION
20 MODE 7
30 PRINT "Evaluation of the factorial for integers Give a
   positive number less than 33 and a NEGATIVE number to finish."
40 PRINT: INPUT "Remember to give a POSITIVE No. < 33 ";X
50 IF X>0 THEN FAC=1: FOR I=1 TO X:FAC=I*FAC: NEXT ELSE END
60 PRINT "FACTORIAL ";X;" IS ";FAC
70 GOTO 40
80 END
```

Now consider the recursive solution. The stopping or end state is considered first.

1. If N is zero, the factorial is 1.

Next, the recursion step is considered.

2. For N>1, the solution is N times the factorial of (N— 1).

Thus the recursion step has reduced the problem to that of finding the factorial of the next smaller number and multiplying that result by N. Obviously, N-1 is closer to the stopping state of zero than N is. Repeated applications of the recursion step will eventually lead to the stopping state.

Example 3.8

The implementation of this in BBC BASIC is shown in the following program.

```
10 REM FACTORIAL EVALUATION
20 MODE 7
```

```
 30 PRINT "Evaluation of the Factorial. for integers"'"Give
    positive number less than 33 and a NEGATIVE number to
    finish."
 40 PRINT: INPUT "Remember to give a POSITIVE No. < 33 ";X
 50 IF X>0 THEN FAC=FN_fac(X) ELSE END
 60 PRINT "FACTORIAL ";X;" IS ";FAC
 70 GOTO 40
 80 END
 90 DEF FN_fac(I)
100 LOCAL F
110 IF I<>0 THEN F=I*FN_fac(I-1) ELSE F=1
120 =F
```

The function FN_fac returns a value of 1 if its argument is zero. If the argument is non-zero then the value returned is determined by the expression N*FN_fac(N-1). This calling of FN_fac from within itself is what is termed recursion.

In evaluating factorial N each new call of FN fac increases the level of recursion. The BASIC interpreter saves the values of the function's parameter N and of any local variables just before each new call. At the time of the call, the N-1 is calculated and becomes the parameter N for the next call of FN_fac. Clearly the value of N will be zero for the last call of the function. As the interpreter returns through each level of recursion, it evaluates the value returned by multiplying the parameter saved at this level, by the result just returned from the level below. Each return reduces the level of recursion by one. When the level of recursion is back to zero the value returned represents the value of the initial function call.

Figure 3.7 shows the case of 6! (factorial 6). There are 7 levels of recursion (0 to 6) and 6 recursive calls of FN_fac plus the initial call. The arrows indicate the flow of the calculation.

Level of recursion	*What is required*		*What is actually calculated*
0	6!		720
	↓		↑
1	6×5!		6*120
	↓		↑
2	5×4!		5*24
	↓		↑
3	4×3!		4*6
	↓		↑
4	3×2!		3*2
	↓		↑
5	2×1!		2*1
	↓		↑
6	1×0!	→	|

Figure 3.7 Action of a recursive function

Example 3.9

Consider the problem of obtaining the digits of a number printed in reverse order. The recursive solution is as follows

1. Output the last digit of the number.
2. If further digits remain then ‘reverse’ remaining digits, otherwise print the remaining digit.

The recursive step reduces the number of digits that must be reversed by one each time until just one remains which needs no reversal. The procedure is used to generate the PRINT statements which give the reversed number. This example has been written as a recursive procedure since the number of levels of recursion is not of great interest and thus there is no need to return a result to the calling program.

```
 10 REM Number reversal example
 20 REPEAT
 30 PRINT: PRINT "To finish type a negative number"
 40 INPUT "Number to be reversed "N
 50 PRINT ;N;" reversed is ";
 60 PROC_reverse(N)
 70 PRINT
 80 UNTIL N<=0
 90 END
100 DEF PR0C_reverse(N)
110 IF N DIV 10 = 0 THEN PRINT ;N;:F=1 ELSE PRINT ; N MOD 10;:
    PROC_reverse(N DIV 10)
120 ENDPROC
```

Example 3.10

Suppose that it is desired to obtain a number of permutations of the integers 1 to 4. The function RND(N) can be used to select the order but each time a fitimber is chosen there is one less to choose from next time, until there is no Choice. The program below shows a non-recursive solution.

```
 10 REM for Linda Leech
 20 PRINT"RANDOM PERMUTATIONS OF NUMBERS 1 TO 4"
 30 DIM N(4),M(4)
 40 FOR L=1 TO 20
 50 FOR 1=1 TO 4:N(I)=I: NEXT
 60 FOR I=4 TO 2 STEP -1
 70 J=RND(I)
 80 M(I)=N(J )
 90 N(J)=N(I)
100 NEXT I
110 M(1)=N(1)
120 PRINT M(1);"  ";M(2);"  ";M(3);"  ";M(4)
130 NEXT L
```

Two arrays M and N are used in this program. The first FOR...NEXT loop allows 20 permutations to be selected. The FOR...NEXT loop at line 30 fills the N (I) array with the numbers 1 to 4 in initial order. The permutation loop begins at line 40 and at line 50 a random number between the count variable I and 1 is chosen by the RND(N) function. This number is used to choose the first number of the permutation which is stored in the Ith element of the M array. At line 70 the N array is changed to take account of the number that has been selected in the first permutation, and then I is decremented and the next number selected. This process continues until just one number remains. The last number is transferred to the M array at line 90 and at line 100 the array is printed for the permutation. This process in lines 30 to 100 is repeated 20 times.

Example 3.11

The recursive solution is shown in the solution below. In this example of recursion a procedure is used rather than a function. The arrays M and N are common to the main program and the procedure.

The recursive solution is as follows

1. Find the next integer for the permutation.
2. If only one integer remains then this is the last integer for the permutation.

This procedure is called at line 40. The two arrays are used in the same way as before.

```
10 REM RECURSIVE PERMUTATIONS
20 PRINT"RANDOM PERMUTATIONS OF NUMBERS 1 TO 4"
30 DIM N(4),M(4)
40 FOR L=1 TO 20
50 FOR I=1 TO 4:N(I)=I: NEXT
60 PROC_comb(4)
70 PRINT L;"  ";M(1);"  ";M(2);"  ";M(3);"  ";M(4)
80 NEXT L
90 END
100 DEF PROC_comb(I)
110 IF I<>1 THEN J=RND(I):M(I)=N(J):N(J)=N(I): PROC_comb(I-1)
    ELSE M(I)=N(1)
120 ENDPROC
```

Exercise 3.9

Write a program using a recursive procedure to select combinations of 3 integers from 6.