Cambridge Learning Limited

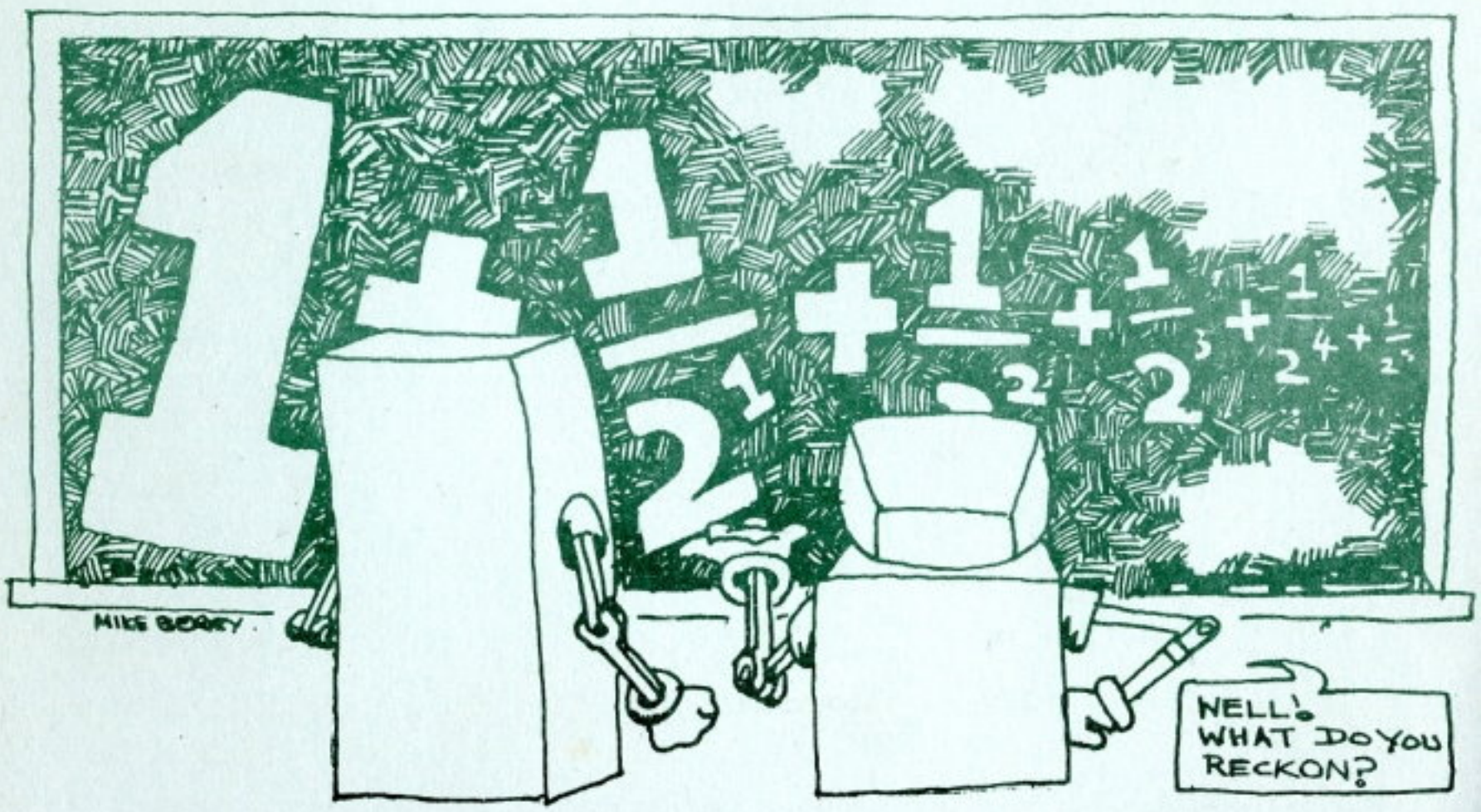
Computer Programming in BASIC

A SELF INSTRUCTION COURSE in 4 Volumes

PART 4

Advanced BASIC

By:
Ian Williamson Rodney Dale Tim Eiloart



COMPUTER PROGRAMMING in BASIC

PART 1 - BASIC BASICS

PART 2 - INTRODUCING BASIC

PART 3 - APPLYING BASIC

PART 4 - ADVANCED BASIC

Computer Programming in BASIC

A unique, teach-yourself course

Part 4: Advanced BASIC

by

Ian Williamson, Rodney Dale and **Tim Eiloart**

Copyright © Ian Williamson, Rodney Dale and Tim Eiloart 1979
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the copyright owner.

First published 1979.
Reprinted 1980 (twice), 1981, 1982

ISBN 0 905946 05 7

published by
Cambridge Learning Limited
Rivermill Lodge
St. Ives
Huntingdon
Cambridgeshire
U.K.

printed in England by
Cambridge Learning Limited

CONTENTS with Summaries

LESSON 49

ADVANCED BASIC

As a computer language, BASIC has been developed considerably since it was born at Dartmouth College, New Hampshire in 1965. The popularity of BASIC in time-sharing computer systems has led to a large number of dialects of the language, offering quite diverse facilities. *Advanced BASIC* (or *extended BASIC* as it is sometimes called) has in many respects grown far beyond its original objective of providing a simple and easy-to-learn computer language, and has become a mature and powerful language in its own right. Advanced BASIC is quite capable of replacing FORTRAN and COBOL in many applications. However, whereas FORTRAN and COBOL are 'standard languages', controlled by national bodies such as the American Standards Institute (USASI) for ten years or more, a standard for advanced BASIC has only recently been published - and has yet to take effect.

You have already learned sufficient BASIC programming to solve difficult and complex problems. If you have studied Parts 1, 2 and 3 with care and attention, you will have acquired a solid core of programming skills, and you will be able to deal competently with most of the problems which you may wish to solve using a computer.

The further techniques introduced in this part of the course enhance these basic skills. Some of the lessons introduce features of BASIC which are common to many computer languages - in particular, the lessons dealing with *subroutines* and *functions* - you should study these lessons carefully. Other lessons concern features which are special to Advanced BASIC, which are often not found in standard BASIC, and which are not common practice in other languages such as FORTRAN and COBOL. The lessons on *strings* and *matrices* fall into this category - and you may opt to skip these lessons if your computer system does not possess these features.

Finally, we have included a brief lesson describing the file-handling operations which are available on some Advanced BASIC systems. File-handling is essential to the business-user of BASIC, but, unfortunately, file-handling facilities and instructions vary widely between computer systems. This last lesson in the course is simply an introduction to a subject which merits a book in its own right.

LESSON 50

SUBROUTINES

A subroutine is a set of program statements which may be used several times in a program but which need be written only once.

BASIC programs are lists of statements, each statement being executed in turn. You have learnt how to alter and direct the execution of the program using conditional and unconditional jump statements - IF...THEN and GO TO.

Often we find that the same calculating procedure is required more than once in different parts of a program, and, with your current knowledge of BASIC programming, you would have no choice but to repeat sections of code which are functionally similar to other parts of the program. This is not only inconvenient, it is poor programming style.

BASIC provides a feature whereby sections of the program called subroutines can be called up as and when required in the program. When a subroutine is called, the program execution is directed to the start of the subroutine. But when the subroutine is completed, program execution *returns* to the place in the main program where the subroutine was originally called.

Below we show a program to calculate the factorial of a number using a subroutine for the calculation.

```
The factorial of N is written as N! and is equal to:

N!  =  N x (N-1) x (N-2) ..... x 2 x 1

Hence: 5! = 5 x 4 x 3 x 2 x 1  = 12Ø
```

LIST

```
1Ø REM  PROGRAM TO CALCULATE FACTORIAL OF N
2Ø PRINT "INPUT NUMBER (NEGATIVE NUMBERS TERMINATE PROGRAM)";
3Ø INPUT N
4Ø IF N>Ø THEN 6Ø
5Ø STOP
6Ø GOSUB 1ØØØ
7Ø PRINT "FACTORIAL";N;"IS";F
8Ø GO TO 2Ø
1ØØØ LET F=1
1Ø1Ø FOR X=1 TO N
1Ø2Ø   LET F=X*F
1Ø3Ø NEXT X
1Ø4Ø RETURN
1Ø5Ø END
READY
```

This is the subroutine

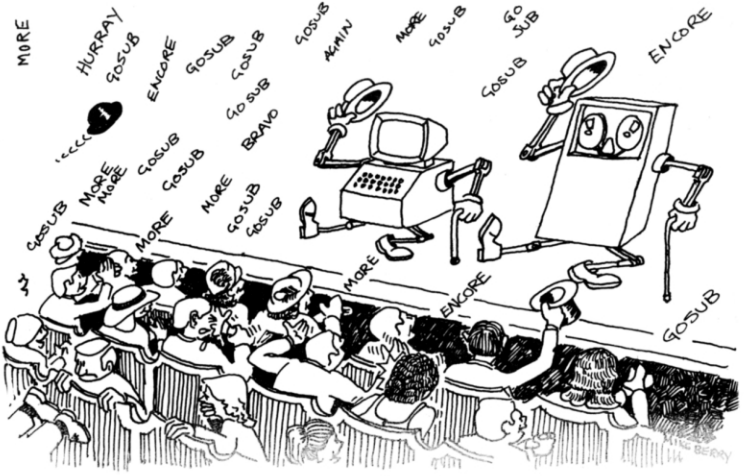
```
RUN

INPUT NUMBER (NEGATIVE NUMBERS TERMINATE PROGRAM) ? 5
FACTORIAL 5 IS 12Ø
INPUT NUMBER (NEGATIVE NUMBERS TERMINATE PROGRAM) ? 8
FACTORIAL 8 IS 4Ø32Ø
INPUT NUMBER (NEGATIVE NUMBERS TERMINATE PROGRAM) ? -9

STOP AT LINE 5Ø
READY
```

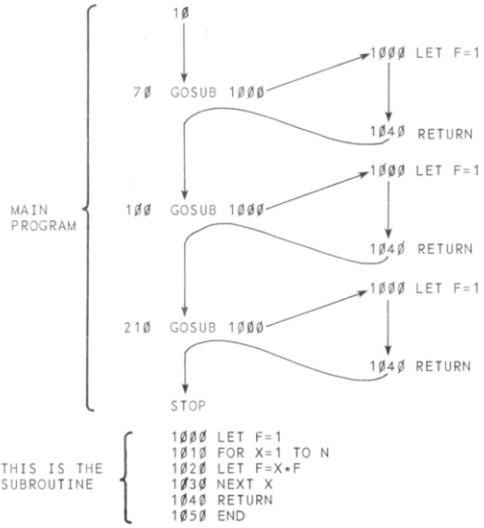
The subroutine is formed from statements 1ØØØ to 1Ø4Ø which are called from the main program by the statement GOSUB 1ØØØ. When the subroutine has been executed the computer returns to the main program using the RETURN statement. Execution of the main program continues from the statement immediately following the GOSUB statement. The subroutine uses the same variables as the main program so that N is automatically available for use in line 1Ø1Ø. Care must be taken not to corrupt variables in the main program by using the same variable inadvertently for another purpose in the subroutine.

Note that the END statement is always the last statement in a BASIC program.

You may be thinking 'Big deal, the program would be shorter without the GOSUB 1ØØØ and RETURN statements!'. True, but this factorial routine can be called many times by the main program.

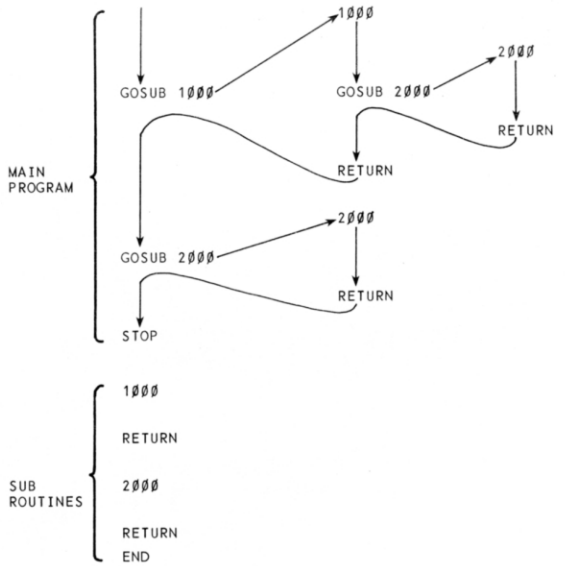
This is illustrated in the diagram on the next page.





In effect the subroutine from 1ØØØ to 1Ø4Ø is inserted within the program wherever GOSUB 1ØØØ is encountered.

One subroutine can call another subroutine and so on, giving rise to *nested subroutines*. The power of the subroutine lies in its ability to return to the point in the program whence it was called, as illustrated below.



Apart from eliminating repetitive sections of code in a program, subroutines also permit the program to be coded and tested in sections formed by the subroutines themselves - very much reducing the development time and cost of a complicated suite of programs. For example, the factorial subroutine in our example could be used in many other programs - the programmer knowing that the routine is functional and has been properly tested.

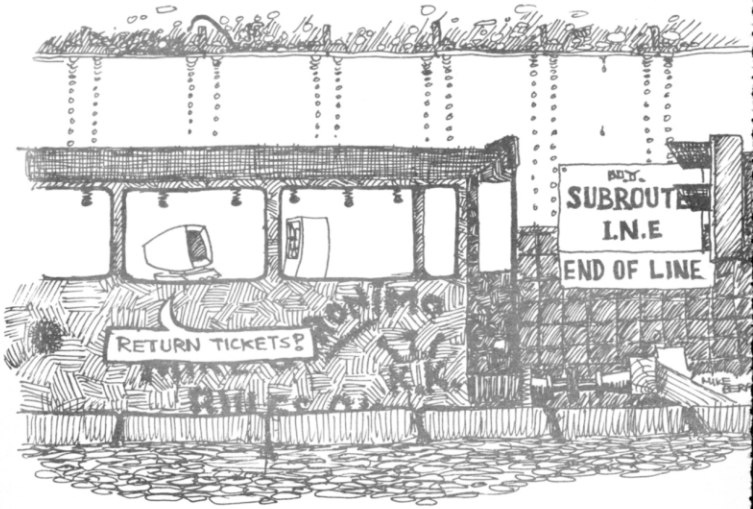
Using subroutines in BASIC is straightforward, and very much to be commended as it is good programming style. However, many programmers persistently fall into common traps when using subroutines, so it is worthwhile using the following checklist.

1. The variable-names in the main program and in the subroutines are common in BASIC (unlike other programming languages such as FORTRAN); have you created a variable-name in a subroutine which alters the value of a variable-name in the main program, or vice versa?

2. Have you remembered the RETURN statement at the end of each subroutine?

3 It is common practice to code the subroutines at the end of the main program. Have you included a STOP statement in the main program before the subroutine? If not the subroutines will be incorrectly executed after the main program execution is complete, which is not what you want.

4. Don't forget that the last statement in all BASIC programs is the END statement.



LESSON 51

PERMUTATIONS AND COMBINATIONS

You practice using subroutines.

Gambling is based on the probability that certain specific events will or will not occur out of a range of possible events. Thus poker players are betting that their hands have a higher value than those of their fellow players. Similarly, a backgammon player is concerned with the odds that certain combinations of dice will be thrown - and so on for all games involving chance.

We can calculate the probability that certain events will occur, by taking account of the total number of events which could occur. We need to calculate the *permutations* and *combinations* of events which can occur.

In the UK, millions of people each week ('punters') gamble on the football pools. The idea behind a football pool is simple - a coupon is provided listing approximately 55 football matches which are played each week on Saturday afternoon. The punter has to guess which 8 games will result in a "score draw" as opposed to a "goal-less draw". Dependent upon the football pool, the punter will select 1Ø,11, or 12 matches per bet. The punter's hope is that there will be at least 8 score draws in the list of matches she or he has picked.

The number of combinations of n matches taken r at a time (e.g. 1Ø matches taken 8 at a time) is given by the expression:

$$ {}^{n}C_{r} = \frac{n!}{r!\,(n-r)!} $$

Thus the number of chances of picking 8 correct matches from a selection of 1Ø matches is:

$$ {}^{1Ø}C_{8} = \frac{1Ø!}{8!\,(1Ø-8)!} = \frac{1Ø \times 9 \times 8!}{8! \times (2 \times 1)} = \frac{1Ø \times 9}{2 \times 1} $$

$$ = 45 $$

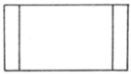
A selection of 1Ø matches on the coupon is equivalent to 45 different selections of 8 matches. However, the problem of which 1Ø matches to select from the list of 55 is somewhat more formidable.

$$ {}^{55}C_{1Ø} = \frac{55!}{1Ø!\,(55-1Ø)!} = 2.92486E1Ø $$

PROBLEM

Write a program using the factorial subroutine given in the last example to calculate ${}_nC_r$. Draw three flowcharts for the main program, a subroutine for ${}_nC_r$, and a subroutine for n! In your flowchart either of the 'pre-defined process' symbols given below may be used to represent a subroutine:

or

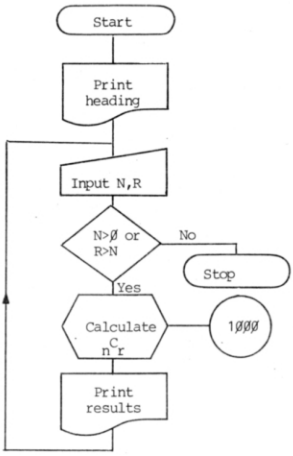


Use a negative input number to stop the program.

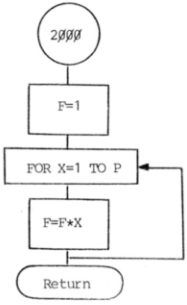
Test your program with inputs 1Ø and 8, 12 and 8, and -9.

SOLUTION

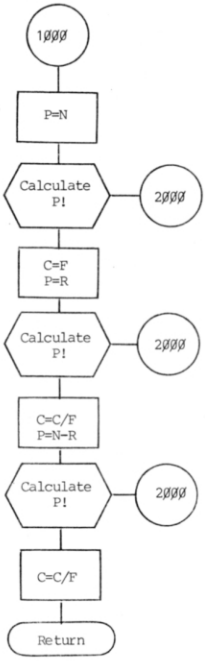
Main Program



Subroutine to calculate n!



Subroutine to calculate $_nC_r$



```
SOLUTION

LIST

1Ø REM CALCULATION OF COMBINATIONS  R. DALE   31-MAY-99
2Ø PRINT "PROGRAM TO CALCULATE COMBINATIONS :";
3Ø PRINT "NEG. NUMBER STOPS PROGRAM"
35 PRINT
4Ø PRINT "INPUT NUMBER OF ITEMS ";
5Ø INPUT N
6Ø IF N>Ø THEN 8Ø
7Ø STOP
8Ø PRINT "INPUT NUMBER TO BE SELECTED ";
9Ø INPUT R
91 REM CHECK DATA IS VALID
92 IF N>R THEN 1ØØ
93 PRINT "MUST BE LESS THAN NO. OF ITEMS"
94 GO TO 9Ø
1ØØ GOSUB 1ØØØ
11Ø PRINT
12Ø PRINT "NUMBER OF COMBINATIONS IS";C
13Ø PRINT
14Ø GO TO 4Ø
1ØØØ REM SUBROUTINE TO COMPUTE COMBINATIONS
1Ø1Ø REM NUMBER OF ITEMS IS N, GROUP IS F, COMBINATION IS C
1Ø2Ø LET P=N
1Ø4Ø GOSUB 2ØØØ
1Ø5Ø LET C=F
1Ø6Ø LET P=R
1Ø7Ø GOSUB 2ØØØ
1Ø8Ø LET C=C/F
1Ø9Ø LET P=N-R
11ØØ GOSUB 2ØØØ
111Ø LET C=C/F
112Ø RETURN
2ØØØ REM SUBROUTINE TO CALCULATE FACTORIAL
2Ø1Ø REM  NUMBER = P     FACTORIAL = F
2Ø2Ø LET F=1
2Ø3Ø FOR X=1 TO P
2Ø4Ø  LET F=F*X
2Ø5Ø NEXT X
2Ø6Ø RETURN
3ØØØ END
READY
```

```
RUN
PROGRAM TO CALCULATE COMBINATIONS :NEG. NUMBER STOPS PROGRAM

INPUT NUMBER OF ITEMS  ? 1Ø
INPUT NUMBER TO BE SELECTED  ? 8

NUMBER OF COMBINATIONS IS 45


INPUT NUMBER OF ITEMS  ? 12
INPUT NUMBER TO BE SELECTED  ? 8

NUMBER OF COMBINATIONS IS 495

INPUT NUMBER OF ITEMS  ? -9

STOP AT LINE 7Ø
READY
```

Note that the first statement in the subroutine can be a REM statement - which is non-executable and is useful for labelling the function of the subroutine and the variables used by the subroutine.

Returning to our punter. One week there are 12 score draws out of 55 matches. The punter has bet on a single 8 out of 1Ø selection. What are the odds that he has a winning line?

1. There are $_{55}C_8$ ways of selecting 8 matches from 55 matches.
2. $_{12}C_8$ of these combinations are winning lines.
3. The punter has selected $_{1Ø}C_8$ combinations of 8 matches.

The odds against a win are:

$$\frac{{}_{55}C_8}{{}_{12}C_8 \; {}_{1Ø}C_8} : 1 = 5466Ø.6 : 1$$

total number of possible combinations of 8 lines

the number of winning lines of 8

the number of lines of 8 the punter has chosen

If you are a regular punter, you might like to write your own program to calculate the odds against winning!

LESSON 52

FUNCTIONS

You start to define your own functions.

We introduced the 10 standard BASIC functions in Part 2, and in Part 3 we added to the list with the RND function and the TAB function. You can also define your own functions in BASIC using the DEF statement.

Here is a simple example:

```
10 REM CONVERSION OF DOLLARS TO STERLING
20 DEF FNR(X)=INT(100*X+0.5)/100
30 PRINT
40 PRINT "INPUT CONVERSION RATE, 1 DOLLAR = C POUNDS";
50 INPUT C
60 PRINT
70 PRINT "DOLLARS";
80 INPUT D
85 IF D<0 THEN 120
90 LET P=D*C
100 PRINT "EQUALS POUNDS STERLING";FNR(P)
110 GO TO 60
120 END
```

QUESTION

What do you think line 20 does?

ANSWER

In the DEF statement we define a function FNR which is subsequently called up in the PRINT statement at line 100. FNR is the rounding function, which was introduced in Part 3; the effect being to round P to the nearest penny.

A defined function may be called up many times during a program. The defined function is given the name FN followed by a single letter from A to Z. Thus 26 defined functions are permitted. In the example above we selected FNR, the R reminding us that it is the Rounding Function.

Each defined function requires an argument which is enclosed in parentheses, as do the ten standard BASIC functions.

The DEF statement must occur before the defined function is called within the program.

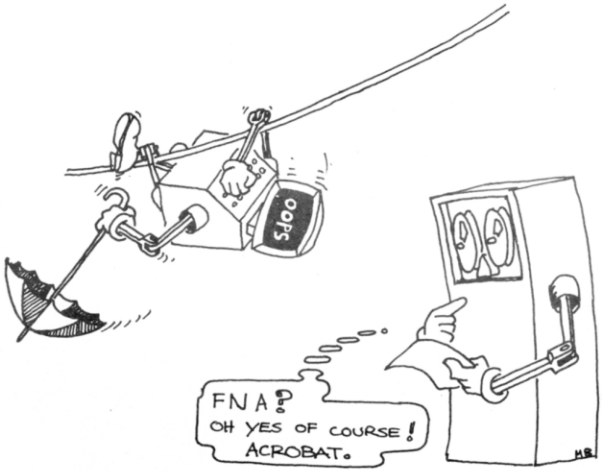
Some dialects of advanced BASIC permit defined functions with multiple arguments as in FNC(X,N) or multiple line definitions of functions using a special statement, FNEND, to close the definition. However, whilst these added facilities are useful in certain applications, more complex defined functions can be programmed using BASIC subroutines.

There is an important distinction between functions and subroutines in BASIC; their use of variable-names. Variable-names are shared between subroutines and the main program, and some care must be exercised in programming with subroutines to ensure that variable-names used in subroutines do not corrupt the values of variable-names in the main program.

The defined function, on the other hand, can be any variable-name as will be seen from the example above. The DEF statement uses the variable-name X - this is a *dummy variable-name* since it has no use in the program proper. The defined function, FNR, is used in line 1ØØ, and the variable-name in parentheses is P. The computer takes the value of P, and evaluates the defined function with X *replaced by* the value of P. The variable-name X is not made equal to P, and it may be used elsewhere in the program. Note that line 1ØØ could equally be:

```
1ØØ PRINT " EQUALS POUNDS STERLING";FNR(D*C)
```

where FNR is evaluated with the value D*C in the expression in the DEF statement.



LESSON 53

COMPUTER GAMES III

More practice using subroutines.

Here is a game for you to program - it is similar to the popular boardgame Mastermind, which was originally played in ancient Egypt, but our game uses numbers where Mastermind uses colours.

PROBLEM

Write a program to set up and play a game to the following rules:

i The computer selects four numbers in the range 1 to 6 and places them in a row, for example 3122. Numbers may be repeated.

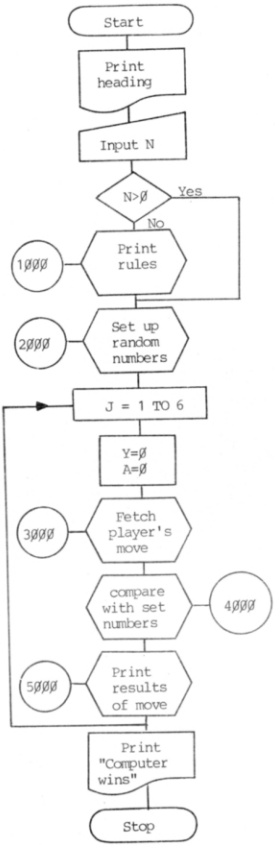
ii The player tries to guess the numbers in the row. If one of the four numbers is correctly guessed and placed, the computer responds with a 'Y'. If a number is correctly guessed but in the wrong position the computer responds with an 'A' for almost; if a number is incorrect the computer does not respond. Thus a typical response might be Y A A indicating one number correctly guessed and placed, and two correctly guessed but incorrectly placed.

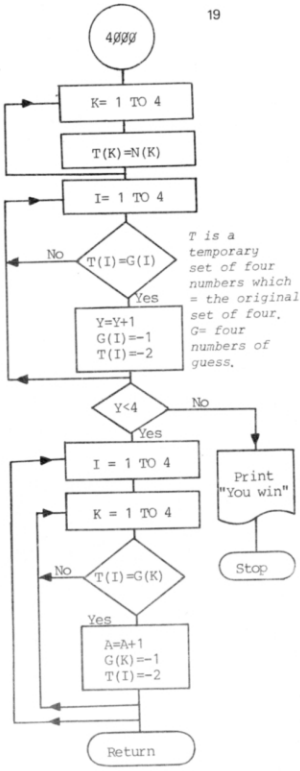
iii The player is allowed six guesses at the row of numbers before the computer wins the game.

Suppose the computer has selected the number 3122.

3122 HERE IT IS AGAIN FOR EASY REFERENCE.

If the guess is	Then the computer says	
4156	Y	The 1 is correct and in the right place
1111	Y	Note only one number 1 is scored correct
1611	A	Note that only one number 1 is scored 'almost'
2266	AA	Both the 2s score 'almost'
2222	YY	The two 2s in the end score 'Yes'; so the ones at the start do not score.
2262	YA	Only one of the 2s at the start scores 'almost', and the final 2 scores a 'Yes'.
2213	AAAA	All four numbers are 'almost', but all in the wrong place.
4456		There are no guesses right.
1322	YYAA	The computer *always* prints the 'Y's before the 'A's so there is no clue that in fact the first two guesses are 'A's and the last two guesses are 'Y's.





Subroutine to compare results

QUESTION

Why do we set T(I) to -2 when we find

that T(I) = G(I) and when

T(I) = G(K) ?

HINT

If 3 1 7 7 is correct and 1 1 1 1 is the guess, what would happen if we do NOT change 3 1 7 7 to 3 -2 7 7?

ANSWER

LIST

```
10 REM PROGRAM TO PLAY SIMPLE NUMBERS GAME
20 REM            I.WILLIAMSON        31-MAY-99
30 PRINT "HELLO.  INPUT A NEGATIVE NUMBER IF YOU NEED THE RULES"
40 PRINT "EXPLAINED";
50 INPUT M
60 PRINT
70 IF M>=0 THEN 100
80 REM SUBROUTINE TO PRINT RULES
90 GOSUB 1000
100 REM SUBROUTINE TO SET UP NUMBERS
110 GOSUB 2000
120 FOR J=1 TO 6  - J is a counter which tallies the number of times
130   LET Y=0       the player has guessed
140   LET A=0
150   REM SUBROUTINE TO FETCH PLAYER'S MOVE
160   GOSUB 3000
170   REM SUBROUTINE TO COMPARE RESULTS
180   GOSUB 4000
190   REM SUBROUTINE TO PRINT RESULTS
200   GOSUB 5000
210 NEXT J
215 PRINT
220 PRINT "THE COMPUTER WINS AFTER 6 MOVES - THE CORRECT NUMBERS"
230 PRINT "ARE AS FOLLOWS"
240 PRINT
250 PRINT "          ";"A  B  C  D"
260 PRINT "         ";N(1);N(2);N(3);N(4)
265 GOSUB 6000
270 STOP
1000 PRINT "THE COMPUTER SETS UP FOUR NUMBERS FROM 1 TO 6 IN"
1010 PRINT "POSITIONS A,B,C AND D."
1020 PRINT "YOU ARE ASKED TO GUESS THE NUMBERS."
1030 PRINT "IF YOU HAVE THE CORRECT NUMBER IN THE CORRECT"
1040 PRINT "LOCATION, THE COMPUTER PRINTS 'Y' FOR YES;"
1050 PRINT "IF YOU HAVE A CORRECT NUMBER IN THE WRONG LOCATION"
1060 PRINT "THE COMPUTER PRINTS 'A' FOR ALMOST"
1070 PRINT "YOU HAVE SIX MOVES TO WIN...GOOD LUCK!"
1080 RETURN
2000 REM SUBROUTINE TO SET UP NUMBERS
2010 RANDOMIZE
2020 FOR I=1 TO 4
2030   LET N(I)=INT(6*RND(0)+1)
2040 NEXT I
2050 RETURN
3000 REM SUBROUTINE TO FETCH PLAYER'S MOVE
3010 PRINT
3020 PRINT "MOVE";J;"A,B,C,D"
3030 PRINT "    ";
3040 INPUT G(1),G(2),G(3),G(4)  -  These are guesses - multiple
3060 RETURN                        inputs are separated by commas
```

```
4000 REM SUBROUTINE TO COMPARE RESULTS
4010 FOR K=1 TO 4
4020   LET T(K)=N(K)
4030 NEXT K
4040 FOR I=1 TO 4
4050   IF G(I)<>T(I) THEN 4090
4060   LET Y=Y+1
4070   LET G(I)=-1
4080   LET T(I)=-2
4090 NEXT I
4100 IF Y<>4 THEN 4140
4120 PRINT "YOU WIN IN ONLY";J;"MOVES"
4125 GOSUB 6000
4130 STOP
4140 FOR I=1 TO 4
4150   FOR K=1 TO 4
4160     IF T(I)<>G(K) THEN 4200
4170     LET A=A+1
4180     LET G(K)=-1
4190     LET T(I)=-2
4200   NEXT K
4210 NEXT I
4220 RETURN
5000 REM SUBROUTINE TO PRINT RESULTS
5010 PRINT "                    ";
5020 IF Y=0 THEN 5060
5030 FOR I=1 TO Y
5040   PRINT " Y ";
5050 NEXT I
5060 IF A=0 THEN 5100
5070 FOR I=1 TO A
5080   PRINT " A ";
5090 NEXT I
5100 PRINT
5110 RETURN
6000 PRINT "ANOTHER GAME? INPUT 1 FOR 'YES',0 FOR 'NO';
6010 INPUT Q
6020 IF Q=1 THEN 10
6030 RETURN
6040 END
READY
```

Here is a sample run from this program, showing how the computer appears to 'talk'.

```
RUN

HELLO.  INPUT A NEGATIVE NUMBER IF YOU NEED THE RULES
EXPLAINED ? -9

THE COMPUTER SETS UP FOUR NUMBERS FROM 1 TO 6 IN
POSITIONS A,B,C AND D.
YOU ARE ASKED TO GUESS THE NUMBERS.
IF YOU HAVE THE CORRECT NUMBER IN THE CORRECT
LOCATION, THE COMPUTER PRINTS 'Y' FOR YES;
IF YOU HAVE A CORRECT NUMBER IN THE WRONG LOCATION
THE COMPUTER PRINTS 'A' FOR ALMOST
YOU HAVE SIX MOVES TO WIN...GOOD LUCK!

MOVE 1 A,B,C,D
     ? 3,4,5,6
                          A
MOVE 2 A,B,C,D
     ? 1,2,3,3
                          Y  A  A
MOVE 3 A,B,C,D
     ? 2,1,3,2
                          Y  Y  Y
MOVE 4 A,B,C,D
     ? 2,1,3,1
                          Y  Y
MOVE 5 A,B,C,D
     ? 2,1,2,2
                          Y  Y
MOVE 6 A,B,C,D
     ? 1,1,3,2
                          Y  Y

THE COMPUTER WINS AFTER 6 MOVES - THE CORRECT NUMBERS
ARE AS FOLLOWS

         A  B  C  D
         2  3  3  2
ANOTHER GAME?  INPUT 1 FOR 'YES',0 FOR 'NO' ? 0

END AT 270
READY
```

Note the extensive use of subroutines in this program - each subroutine representing a separate processing function in the overall problem. Proper and careful use of subroutines is the key to large and complicated programs. Each of the subroutines used above can be tested apart from the main program itself.

LESSON 54

STRING VARIABLES

String variables are used to handle words and letters in the program, just as numerical data is processed with BASIC variables.

In the main, our interface with the computer is by way of a terminal consisting of a "typewriter-like" keyboard, and a visual display unit or printer. At the terminal we communicate with the computer using English-like commands and statements and much of our programming is directed towards easing the communication between the computer and ourselves - for example, by ensuring that print-outs are properly labelled and self-explanatory.

The manipulation and processing of alphabetic information has an important role in computer programming.

Within the computer, alphabetic characters are represented by a numerical code and they can therefore be processed in a manner similar to that used for numerical information for which the computer is designed. The most common code for alphabetic characters is referred to as ASCII (pronounced 'askey') = American Standard Code for Information Interchange, and the complete code is given in the table overleaf. ASCII code is mainly used by the computer to communicate with peripheral devices such as teletype terminals, VDUs and paper tape readers and punches. Many computers also use ASCII to process information internally.

An arbitrary combination of alphabetic and numeric characters, which may be a single letter, word or sentence, is referred to as a *string*. A good example of a string is the familiar text label used in the PRINT statement.

For example, the statement -

```
PRINT "THIS IS A STRING "
```

will directly generate the print-out

```
THIS IS A STRING.
```

Now within the computer, the characters between the quotation marks are stored in the ASCII code.

QUESTION

In ASCII what numeric code represents "THIS IS A STRING"?

Code	ASCII Character	Paper Tape
1	Special	▯ o 0
2	Special	▯ o 0
3	Special	o 00
4	Special	▯ o0
5	Special	o0 0
6	Special	o00
7	Special	▯ o000
8	Special	▯ 0o
9	Special	0o 0
10	Special	0o 0
11	Special	0o 00
12	Special	0o0
13	Special	▯ 0o0 0
14	Special	▯ 0o00
15	Special	0o000
16	Special	▯ 0 o
17	Special	0 o 0
18	Special	0 o 0
19	Special	▯ 0 o 00
20	Special	0 o0
21	Special	▯ 0 o0 0
22	Special	▯ 0 o00
23	Special	0 o000
24	Special	00o
25	Special	▯ 00o 0
26	Special	▯ 00o 0
27	Special	00o 00
28	Special	▯ 00o0
29	Special	00o0 0
30	Special	00o00
31	Special	▯ 00o000
32	SPACE	▯ 0 o

Code	ASCII Character	Paper Tape
33	!	0 o 0
34	"	0 o 0
35	#	▯ 0 o 00
36	$	0 o0
37	%	▯ 0 o0 0
38	&	▯ 0 o00
39	'	0 o000
40	(	0 0o
41	)	▯ 0 0o 0
42	*	▯ 0 0o 0
43	+	0 0o 00
44	,	▯ 0 0o0
45	-	0 0o0 0
46	.	0 0o00
47	/	▯ 0 0o000
48	Ø	00 o
49	1	▯ 00 o 0
50	2	▯ 00 o 0
51	3	00 o 00
52	4	▯ 00 o0
53	5	00 o0 0
54	6	00 o00
55	7	▯ 00 o000
56	8	▯ 000o
57	9	000o 0
58	:	000o 0
59	;	▯ 000o 00
60	<	000o0
61	=	▯ 000o0 0
62	>	▯ 000o00
63	?	000o000
64	@	▯▯ o

Note

▯ This hole is cut for an ASCII code with parity.
It is there so that there is always an even number of holes.

Code	ASCII Character	Paper Tape
65	A	O o O
66	B	O o O
67	C	OO o OO
68	D	O oO
69	E	OO oO O
70	F	OO oOO
71	G	O oOOO
72	H	O Oo
73	I	OO Oo O
74	J	OO Oo O
75	K	O Oo OO
76	L	OO OoO
77	M	O OoO O
78	N	O OoOO
79	O	OO OoOOO
80	P	O O o
81	Q	OO O o O
82	R	OO O o O
83	S	O O o OO
84	T	OO O oO
85	U	O O oO O
86	V	O O oOO
87	W	OO O oOOO
88	X	OO OOo
89	Y	O OOo O
90	Z	O OOo O
91	[	OO OOo OO
92	\	O OOoO
93	]	OO OOoO O
94	↑	OO OOoOO
95	←	O OOoOOO
96	-	OO o

Code	ASCII Character	Paper Tape
97	a	OOO o O
98	b	OOO o O
99	c	OO o OO
100	d	OOO oO
101	e	OO oO O
102	f	OO oOO
103	g	OOO oOOO
104	h	OOO Oo
105	i	OO Oo O
106	j	OO Oo O
107	k	OOO Oo OO
108	l	OO OoO
109	m	OOO OoO O
110	n	OOO OoOO
111	o	OO OoOOO
112	p	OOOO o
113	q	OOO o O
114	r	OOO o O
115	s	OOOO o OO
116	t	OOO oO
117	u	OOOO oO O
118	v	OOOO oOO
119	w	OOO oOOO
120	x	OOOOo
121	y	OOOOOo O
122	z	OOOOOo O
123	Special	OOOOo OO
124	Special	OOOOOoO
125	Special	OOOOoO O
126	Special	OOOOoOO
127	Special	OOOOOoOOO
128	Special	-

```
ANSWER

84 72 73 83 32 73 83 32 65 32 83 84 82 73 78 71
T  H  I  S     I  S     A     S  T  R  I  N  G
```

The label "THIS IS A STRING" is called a string constant. In BASIC we can also have string variables, as in the following program example.

```
LIST

10 LET X$="THIS IS A STRING"
20 LET Y$=" OK?"
30 PRINT X$,Y$
40 END

READY
RUN

THIS IS A STRING              OK?

END AT 40
READY
```

A string variable is distinguished by the $ sign after the variable-name. There is great variation in the way different BASIC systems treat string variables. Some BASIC systems do not permit string variables! All BASIC systems limit the number of characters in the string variable. At a minimum, (if string variables are included in the system) the string variable may be limited to 18 characters. At the other extreme some large BASIC systems permit 4096 characters in a single string variable. If you keep to 18 characters per string variable, your program can be run on any BASIC system which permits string variables.

How many string variables are allowed? Again, there are variations in the rules for different BASIC systems. All systems with strings permit 26 string variables A$, B$ ..... to Z$. Many systems also allow A0$, A1$, A2$ .... A9$ etc., and some systems permit string variable arrays and tables A$(N), B$(M,N)... etc.

String variables can consume a lot of computer memory, and readers using small computer systems may find that the number of string variables is limited by the computer memory.

String variables can also be used in INPUT and READ....DATA statements, as in the two programs shown below. (Note that the comma separates the strings in the data, and is not included in the string variables.)

```
1Ø INPUT X$,Y$
2Ø PRINT X$,Y$
3Ø END

RUN

 ? CATS,DOGS

CATS            DOGS

END AT 3Ø
READY
```

```
1Ø READ X$,Y$
2Ø PRINT X$,Y$
3Ø DATA "CATS","DOGS"
4Ø END

RUN

CATS            DOGS

END AT 4Ø
READY
```

String variables can also be compared using the IF...THEN... statement.

IF A$ = B$ THEN.. tests true if strings A and B are identical.

IF A$<BS THEN.. tests true if string A is alphabetically before string B, and vice versa for A$>B$.

QUESTION

In this section of program, what print-out will be produced by the computer, and why?

```
1Ø READ A$,B$
2Ø DATA "BEAT","BATE"
3Ø IF A$<B$ THEN 6Ø
4Ø PRINT B$,A$
5Ø GO TO 7Ø
6Ø PRINT A$,B$
7Ø END
```

HINT

If you find the test A$<B$ difficult, then try converting each letter to its numeric ASCII value and compare first letters, second letters and so on until the test A$<B$ passes or fails.

ANSWER

A$ = "BEAT" 66 69 65 84

B$ = "BATE" 66 65 84 69

Step 1: 66 < 66? Equal, therefore no decision.

Step 2: 69 < 65? No, therefore test fails.

Alphabetically, BEAT follows BATE. Therefore, A$ is not less than B$, and the print-out is:

BATE BEAT

LESSON 55

PROGRAM EXAMPLE - TALKING TO THE COMPUTER

Using words rather than numbers in INPUTS can make your computer programs seem much more human.

Equipment is available to talk to the computer - it's called voice recognition equipment. And the computer will talk to you if it has a voice synthesiser. But that's not what this lesson is about!

In formatting the print-out, we go to great lengths to ensure that the results are presented in an understandable form. Similarly, with INPUT data, we don't present the user with a bare ? - the data to be entered is labelled in a relevant and informative manner.

Using string variables, we can now enter written commands into the program, and, as a result, the computer can be made more acceptable to the unskilled user.

A good example is the computer game described in Lesson 53.

Here are the first ten lines of the program:

```
1Ø REM PROGRAM TO PLAY SIMPLE NUMBERS GAME
2Ø REM           I.WILLIAMSON        31-MAY-99
3Ø PRINT "HELLO.  INPUT A NEGATIVE NUMBER IF YOU NEED THE RULES"
4Ø PRINT "EXPLAINED";
5Ø INPUT M
6Ø PRINT
7Ø IF M>=Ø THEN 1ØØ
8Ø REM SUBROUTINE TO PRINT RULES
9Ø GOSUB 1ØØØ
1ØØ REM SUBROUTINE TO SET UP NUMBERS
```

And the resulting first three lines of print-out are:

```
RUN

HELLO.  INPUT A NEGATIVE NUMBER IF YOU NEED THE RULES
EXPLAINED ? -9
```

The program would appear much more 'human' if the first three lines became:

```
RUN

HELLO. DO YOU NEED THE RULES EXPLAINED ? YES
```

PROBLEM

Modify the first ten lines of the program on the previous page, and the subroutine at line 1ØØØ, so that the program accepts a 'YES' or 'NO' answer to the question "DO YOU NEED THE RULES EXPLAINED".

SOLUTION

```
1Ø REM PROGRAM TO PLAY SIMPLE NUMBERS GAME
2Ø REM MODIFIED   I.WILLIAMSON    31-MAY-99
3Ø PRINT "HELLO.  DO YOU NEED THE RULES EXPLAINED";
4Ø INPUT M$
5Ø IF M$="NO" THEN 1ØØ
6Ø IF M$="YES" THEN 9Ø
7Ø PRINT "SORRY - YOU MUST ANSWER 'YES' OR 'NO'";
8Ø GO TO 4Ø
85 REM SUBROUTINE TO SET UP THE RULES
9Ø GOSUB 1ØØØ
1ØØ REM SUBROUTINE TO SET UP NUMBERS
11Ø GOSUB 2ØØØ


RUN
HELLO.  DO YOU NEED THE RULES EXPLAINED ? CERTAINLY

SORRY - YOU MUST ANSWER 'YES' OR 'NO' ? YES

THE COMPUTER SETS UP FOUR NUMBERS FROM 1 TO 6 IN
POSITIONS A,B,C AND D.
YOU ARE ASKED TO GUESS THE NUMBERS.
IF YOU HAVE THE CORRECT NUMBER IN THE CORRECT
LOCATION, THE COMPUTER PRINTS 'Y' FOR YES;
IF YOU HAVE A CORRECT NUMBER IN THE WRONG LOCATION
THE COMPUTER PRINTS 'A' FOR ALMOST
YOU HAVE SIX MOVES TO WIN...GOOD LUCK!
```

LESSON 56

PROGRAM EXAMPLE - A TELEPHONE DIRECTORY

Practice with strings.

In Lesson 45 we introduced bubble-sorting, and showed you how to sort 20 numbers efficiently into descending order.

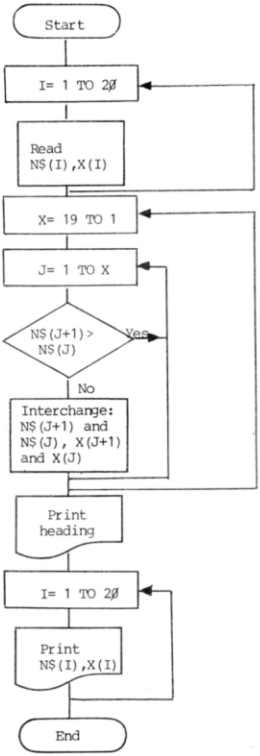
Using string variables, a list of names and associated numbers can be sorted into alphabetic order using exactly the same principle of bubble-sorting.

PROBLEM

Here is a list of names and telephone numbers. Write a BASIC program to sort the names into alphabetic order and print the ordered list at the terminal.

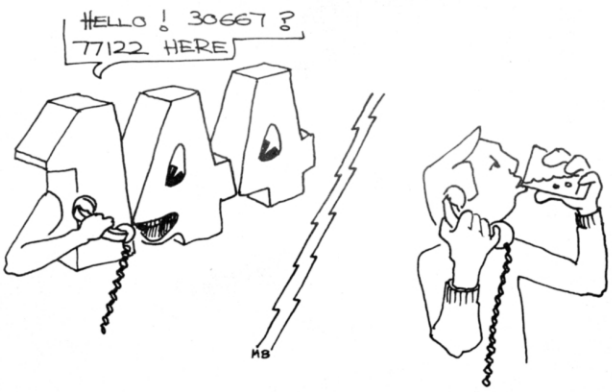
JONES N, 45210 ; DALE R, 23212; SIMPSON D, 70426;
CLARK J, 36092; SINCLAIR C, 70156; TAYLOR A, 30761;
ANDERSON D, 76581; VALE R, 39871; STRUBE M, 59872;
DRINKWATER L, 30667; EDGE G, 70891; SOUTHWARD D, 50386;
GIST M, 47921; SOUTHWELL C, 59871; STOKES M, 21213;
WRIGHT R, 30777; GROSS J, 77122; GARDEN A, 40302;
BRAITHWAITE C, 62051; HINDE H, 37712.

SOLUTION


Start
I= 1 TO 2Ø
Read
N$(I),X(I)
X= 19 TO 1
J= 1 TO X
N$(J+1) > N$(J)
Yes
No
Interchange:
N$(J+1) and
N$(J), X(J+1)
and X(J)
Print
heading
I= 1 TO 2Ø
Print
N$(I),X(I)
End

```
SOLUTION

LIST

10 REM PROGRAM TO SORT 20 STRINGS INTO ASCENDING ORDER
20 REM USING A BUBBLE SORT
30 REM I.WILLIAMSON          31-MAY-99
40 DIM N$(20),X$(20)
50 FOR I=1 TO 20
60   READ N$(I),X$(I)
70   NEXT I
80 FOR X=19 TO 1 STEP -1
90   FOR J=1 TO X
100      IF N$(J+1)>N$(J) THEN 170
110      LET T$=N$(J)
120      LET K$=X$(J)
130      LET N$(J)=N$(J+1)
140      LET X$(J)=X$(J+1)
150      LET N$(J+1)=T$
160      LET X$(J+1)=K$
170    NEXT J
180 NEXT X
190 PRINT "NAME",,,"TELEPHONE"
200 FOR I=1 TO 20
210   PRINT N$(I),TAB(45);X$(I)
220 NEXT I
230 DATA "JONES N","45210","DALE R","23212","SIMPSON D","70426"
240 DATA "CLARK J","36092","SINCLAIR C","70156","TAYLOR A","30761"
250 DATA "ANDERSON D","76581","VALE R","39871","STRUBE M","59872"
260 DATA "DRINKWATER L","30667","EDGE G","70891","SOUTHWARD D","50386"
270 DATA "GIST M","47921","SOUTHWELL C","59871","STOKES M","21213"
280 DATA "WRIGHT R","30777","GROSS J","77122","GARDEN A","40302"
290 DATA "BRAITHWAITE C","62051","HINDE H","37712"
300 END

READY
```

```
RUN

NAME                    TELEPHONE
ANDERSON D                 76581
BRAITHWAITE C              62Ø51
CLARK J                    36Ø92
DALE R                     23212
DRINKWATER L               3Ø667
EDGE G                     7Ø891
GARDEN A                   4Ø3Ø2
GIST M                     47921
GROSS J                    77122
HINDE H                    37712
JONES N                    4521Ø
SIMPSON D                  7Ø426
SINCLAIR C                 7Ø156
SOUTHWARD D                5Ø386
SOUTHWELL C                59871
STOKES M                   21213
STRUBE M                   59872
TAYLOR A                   3Ø761
VALE R                     39871
WRIGHT R                   3Ø777

END AT 5ØØ

READY
```



LESSON 57

OTHER ADVANCED BASIC FEATURES - Matrices, multiple branching, print formatting

In this lesson we briefly introduce some other features of advanced BASIC which you may find useful in your own computer programming. In reading this lesson, remember that the features described are not available on all advanced BASIC systems.

Matrix Operations

Lists and tables are the equivalent of vectors and matrices. In certain disciplines most processing is expressed in a vector or matrix notation, and therefore advanced BASIC provides a convenient set of instructions specifically for matrix operations.

A matrix is defined using the DIM statement, which is interpreted as follows:

1Ø DIM X(2,4)

defines a matrix:

X(1,1)	X(1,2)	X(1,3)	X(1,4)
X(2,1)	X(2,2)	X(2,3)	X(2,4)

1Ø DIM Y(3) or DIM Y(3,1)

defines a column vector:

Y(1)
Y(2)
Y(3)

and 1Ø DIM Z(1,3)

defines a row vector:

Z(1,1)	Z(1,2)	Z(1,3)

Advanced BASIC provides three instructions for inputting and outputting matrices; MAT READ, MAT INPUT, and MAT PRINT. A number of special instructions are used to manipulate matrices within the program. For example:

```
MAT(X) = Y + Z
```

causes the addition of matrices Y and Z.

Matrices of all zeros, all ones, or an identity matrix (a diagonal column of 1s) can be defined using MAT ZER, MAT CON, and MAT IDN, respectively.

These matrix instructions are a distinguishing characteristic of advanced BASIC and represent a powerful facility not available in many other high-level languages. For example, whilst FORTRAN will manipulate one- and two-dimensional arrays, matrix operations such as addition and multiplication have to be specified with many instructions as opposed to the single MAT instruction in BASIC.

On the other hand, the scope for programming errors when using the MAT instructions is considerable, and you should satisfy yourself that you are fully able to understand the matrix dimensioning and redimensioning processes before using these instructions.

Multiple Branches: the ON...GO TO... statement.

As we have seen, the IF...THE... statement is used to make conditional jumps within the program. A conditional jump causes a branch in the program flow - that is, the processing will continue on one of two paths dependent upon the processed data.

BASIC also provides a multiple branch facility which is useful in some applications. The segment of a program below illustrates one application of the ON...GO TO... statement. When the ON....GO TO... statement is encountered, the computer takes the value of the variable N, and truncates it to an integer. It then jumps to the appropriate line number, for example, in the program below, if N = 4 the computer jumps to line 15Ø.

```
1Ø REM EVALUATION OF SEMICONDUCTOR DEVICE RELIABILITY
2Ø REM T.EILOART     31-MAY-99
3Ø PRINT "WHICH DEVICE TECHNOLOGY IS TO BE USED?"
4Ø PRINT " 1 = BIPOLAR"    These are kinds of semiconductors
5Ø PRINT " 2 = NMOS"       used in computers and other electronic
6Ø PRINT " 3 = PMOS"       equipment. The user of the program
7Ø PRINT " 4 = CMOS"       has to specify the kind that is to be used.
8Ø PRINT " 5 = STOP"
9Ø INPUT N
1ØØ ON N GO TO 11Ø, 13Ø, 13Ø, 15Ø, 18Ø
11Ø GOSUB 1ØØØ
12Ø GO TO 16Ø
13Ø GOSUB 2ØØØ
14Ø GO TO 16Ø
15Ø GOSUB 3ØØØ
16Ø PRINT "FAILURE RATE IS";F;"PERCENT PER 1ØØØ HOURS"
165 PRINT
17Ø GO TO 3Ø
18Ø END

READY
```

The subroutines are omitted as they are irrelevant here. This is an extract from a real program and some readers may feel disturbed if they have not previously heard of 'Bipolar' or 'CMOS' etc. However, as a computer programmer, you may be required to deal with programs like this that you do not fully understand. It is important to learn to take such programs in your stride.

```
RUN

WHICH DEVICE TECHNOLOGY IS TO BE USED?
 1 = BIPOLAR
 2 = NMOS
 3 = PMOS
 4 = CMOS
 5 = STOP
 ? 3
FAILURE RATE IS 4.56721 PERCENT PER 1ØØØ HOURS

WHICH DEVICE TECHNOLOGY IS TO BE USED?
 1 = BIPOLAR
 2 = NMOS
 3 = PMOS
 4 = CMOS
 5 = STOP
 ? 5

END AT 18Ø
READY
```

If, in this example, the user had typed a number greater than 5, then the computer would have continued with line 11Ø. If required, you can use this feature to print out an error message. You would insert lines such as:

```
1Ø5 PRINT "PLEASE TYPE OUT A NUMBER FROM 1 TO 5"
1Ø7 GO TO 3Ø
```

The ON...GO TO... statement in conjunction with the INPUT statement is particularly useful where multiple options are available - as in the program example shown above.

QUESTION
How does the user stop this program?
ANSWER
By typing in 5.

Print Format - the PRINT USING statement.

A weakness of standard BASIC, when compared with languages such as FORTRAN and COBOL, is the limited control of the print-out format available. In some versions of advanced BASIC, this deficiency is corrected by the PRINT USING statement. The facilities offered by the PRINT USING statement vary widely, dependent upon the design of the advanced BASIC system. Accordingly, we have limited our coverage in this course to an introduction of the principle underlying the PRINT USING statement.

Here is a simple program.

```
LIST

1Ø REM EXAMPLE OF A PRINT USING STATEMENT
2Ø READ X,Y
3Ø PRINT USING 4Ø,X,Y
4Ø: VALUE OF X IS ###.## AND Y IS ###.###
5Ø DATA 4.212,-5.31313
6Ø END

READY
RUN

 VALUE OF X IS    4.21 AND Y IS - 5.313

END AT 6Ø

READY
```

Line 3Ø is interpreted as PRINT variables X and Y according to the specification contained in line 4Ø, as indicated by the colon, and is an exact description of the format for the print-out. Line 4Ø is referred to as an image statement. Labels and alphanumeric data in the image statement are printed directly. The values of the variables X and Y are printed in the fields defined by ###.## and ###.### respectively.

The symbol # indicates that a number may be placed at that point in the print-out, the . indicates that the decimal point must be placed at that point. Thus, the value of X is printed as 4.21, with only two decimal places printed. Similarly, Y is printed as - 5.313 with three decimal places and the negative sign placed at the start of the print field.

The PRINT USING statement can be used to tabulate data with the columns of data lined up by the decimal points; this can be very useful. Also, with the PRINT USING statement we are no longer limited to the five print zones of standard BASIC.

The format of the print-out can be varied using the image statement alone. For example, the statement:

```
4Ø: X IS ### AND Y IS -###
```

produces a print-out:

```
 X IS   4 AND Y IS  -5
```

The absence of a decimal point in the image field forces the printed number to appear as an integer. A negative sign prior to the image field moves the negative sign in the print-out next to the most significant digit of the number being printed.

There are many other variations on the PRINT USING statement; fixed point, floating point (exponential) and integer formats are possible, and some dialects of advanced BASIC allow string variables to be printed by the PRINT USING statement. If you are fortunate enough to have access to a computer system with advanced BASIC, try a few experiments with the PRINT USING statement. You will be surprised how useful a facility it can be!

PROBLEM

Write a program to print twelve numbers in a 4 x 3 table (with 4 columns and 3 rows). All the decimal points are to be aligned on the table.
Here are the twelve numbers, Ø.Ø4, Ø.1, Ø.13, Ø.967, 12.ØØ, 1Ø.Ø1, 99, 99.99, 1.24, 24.9586, 8.8, ØØØ.ØØØ, and they are to be printed thus:

```
 Ø.Ø4      Ø.1       Ø.13      Ø.967
12.ØØ      1Ø.Ø1     99.ØØ     99.99
 1.24      24.9586    8.8Ø      Ø.ØØØ
```

HINT

You will need a PRINT USING statement that allows for numbers such as 12.Ø1 to be printed and followed by 99.

ANSWER

```
1Ø REM PROGRAM TO PRINT A REGULAR TABLE
2Ø REM WITH DECIMAL POINTS ALIGNED
3Ø REM TIM EILOART      31-MAY-99
4Ø DIM V(12)
5Ø FOR I=1 TO 12     - read the I value from the data block
6Ø    READ V(I)
7Ø NEXT I
8Ø FOR I=1 TO 12 STEP 4
9Ø    PRINT USING 1ØØ, V(I), V(I+1), V(I+2), V(I+3)
1ØØ:  ##.##    ##.##    ##.##    ##.##
11Ø NEXT I
12Ø DATA Ø.Ø4, Ø.1, Ø.13, Ø.967, 12.ØØ, 1Ø.Ø1, 99, 99.99, 1.24
13Ø DATA 24.9586, 8.8, ØØØ.ØØØ
14Ø END

READY
RUN

   Ø.Ø4     Ø.1Ø     Ø.13     Ø.97
  12.ØØ     1Ø.Ø1    99.ØØ    99.99
   1.24     24.96     8.8Ø     Ø.ØØ

END AT 14Ø
READY
```

LESSON 58

NUMERICAL METHODS I - RECURSION

Often, with mathematical, scientific, or engineering work, we use numerical methods that calculate an approximation to the exact value.

No text on computer programming would be complete without an introduction to numerical methods in computer programming. 'Numerical methods' refers to the class of techniques used to evaluate complex functions using a computer - they are most often applied by engineers (for example in major structural analysis), by scientists in the analysis of experimental data, and by business analysts and economists in the modelling of complex business situations.

If you are unlikely to need numerical methods in your own personal computer programming, then you can skip this lesson, and the next, without loss of understanding of the BASIC language and computer programming.

In two short lessons, we cannot hope to cover the range of numerical methods used in computer programming. There are many specialised text books on this subject packed with examples - some practical, some esoteric. Instead we concentrate on two fundamental techniques, evaluation of series using recursive formulae, and numerical integration using Simpson's Rule.

BASIC provides the 1Ø standard functions introduced in Lesson 24 and used many times throughout this course. How does the computer calculate functions of this nature? One way is to evaluate the series expansion of the function. Thus, EXP would be calculated using the series expansion:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!}$$

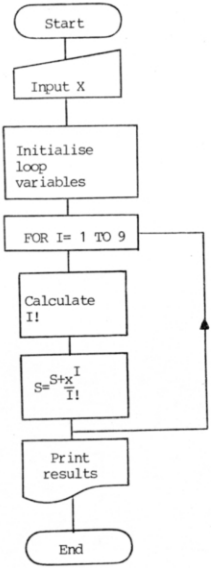
[Remember: 4! = 4 x 3 x 2 x 1 etc]

QUESTION

Draw a flowchart for a program to sum the first 1Ø terms of the series expansion e^x.

ANSWER

Here is a very inefficient solution to the problem.



A much more efficient approach is to use a *recurrence relationship*. Examine the fourth and fifth terms of the series:

$$T_3 = \frac{x^3}{3!} \qquad T_4 = \frac{x^4}{4!}$$

But

$$T_4 = \frac{x}{4} * T_3$$

Hence

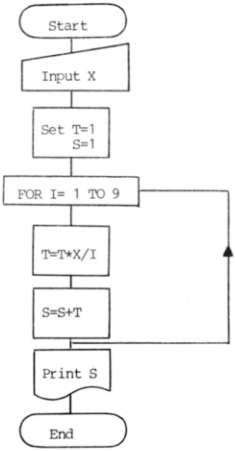
$$T_n = \frac{x}{n} * T_{(n-1)}$$

Using this terminology, the first term $T_\emptyset$ is equal to 1, and therefore the terms of the series can be computed sequentially from one another.

QUESTION

Now draw an improved flowchart to sum the first ten terms of the series expansion e^x.

ANSWER



Now write the BASIC program.

```
1Ø REM PROGRAM TO COMPUTE EXP(X) USING 1Ø TERMS OF THE
2Ø REM SERIES EXPANSION
3Ø REM I.WILLIAMSON  31-MAY-99
4Ø PRINT "INPUT VALUE OF X";
5Ø INPUT X
6Ø LET T=1
7Ø LET S=1
8Ø FOR I=1 TO 9
9Ø   LET T=T*X/I
1ØØ   LET S=S+T
11Ø NEXT I
12Ø PRINT "EXP(";X;") EQUALS";S
13Ø END
READY

RUN

INPUT VALUE OF X ? 4.78
EXP( 4.78 ) EQUALS 116.182

END AT 13Ø
READY
```

Recurrence relationships are of great importance in computer programming - you should always examine a series calculation for the possible existence of a recurrence relationship.

QUESTION

Here are two series expansions - can you see the recurrence relationships?

[a] $SIN(X) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} - \frac{x^9}{9!} \ldots\ldots$

[b] $ATAN(X) = \frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} \ldots\ldots \quad (x>1)$

Note: $\frac{1}{x}$ is $T_\emptyset$ in this example.

ANSWER

If you are not familiar with series expansions, then the question is rather difficult. Don't worry - here are the answers.

[a] $T_\emptyset = x$

$T_1 = \frac{-x^3}{3!}$

$T_n = (-1)^n * \frac{x^{(2n+1)}}{(2n+1)!}$

$T_{n+1} = \frac{-1*x^2*T_n}{(2n+2)*(2n+3)}$

[b] $T_\emptyset = -\frac{1}{x}$

$T_1 = +\frac{1}{3x^3}$

$T_2 = -\frac{1}{5x^5}$

$T_n = \frac{-(-1)^n}{(2n+1)*x^{(2n+1)}}$

$T_{(n+1)} = \frac{-(2n+1)*T_n}{(2n+3)*x^2}$

QUESTION

Look back to Lesson 51. Can you see an opportunity to use a recurrence relationship to improve the efficiency of your computer program?

HINT

Look at the formula

$$ {}^{n}C_{r} = \frac{n!}{r!\,(n-r)!} $$

ANSWER

$$ {}^{n}C_{r} = \frac{(n-r+1)}{r} * {}^{n}C_{(r-1)} \quad \text{where } {}^{n}C_{0} = 1 $$

As an exercise on your own, you should demonstrate that this recursive form leads to a more efficient computation of ${}^{n}C_{r}$.

In evaluating EXP(X), we calculated the first 10 terms of the series expansion. On the ninth execution of the FOR...NEXT loop, the variable T should be very, very small so that the cumulative sum of S is equal to e^{x}. Of course, for some values of x, the series will converge quickly, but for other values of x many more than 9 passes through the loop may be required to provide an accurate answer.

The error in the final result can be estimated by subtracting the result of successive passes through the loop - in this example, the variable T is a measure of the error. A quantity called the *normalised error* is often used to measure the accuracy of the result.

$$ E = \frac{I_{n+1} - I_{n}}{I_{(n+1)}} $$

In this example

$$ E = T/S. $$

PROBLEM

Modify the program on the next page to calculate EXP(X) so that the computation continues until E<0.01 per cent.

```
1Ø REM PROGRAM TO CALCULATE EXP(X) TO ACCURACY BETTER THAN
2Ø REM Ø.Ø1%
3Ø REM I.WILLIAMSON     31-MAY-99
4Ø PRINT "INPUT VALUE OF X"
5Ø INPUT X
6Ø LET T=1
7Ø LET S=1
8Ø FOR I=1 TO 2Ø
9Ø    LET T=T*X/I
1ØØ    LET S=S+T
1Ø5    IF T/S<.ØØØ1 THEN 12Ø
11Ø NEXT I
115 PRINT "AFTER 2Ø ITERATIONS T=";T
12Ø PRINT "EXP(";X;") EQUALS";S
13Ø END
READY

RUN
INPUT VALUE OF X ? 2
EXP( 2 ) EQUALS 7.389

END AT 13Ø
READY

RUN
INPUT VALUE OF X ? 1Ø
AFTER 2Ø ITERATIONS T= 41.1Ø32
EXP( 1Ø ) EQUALS 21991.5

END AT 13Ø
READY
```

Note that with large values of X the series does not converge quickly, and we must prevent the computer from carrying on forever and presenting us with a massive bill!

QUESTION

Is the program above free of error?

HINT

Here is the print-out with different input data

```
INPUT VALUE OF X ? 2
EXP( 2 ) EQUALS 7.389

END AT 13Ø
READY

RUN
INPUT VALUE OF X ? -2
EXP(-2 ) EQUALS 6.66669E-Ø2

END AT 13Ø
READY
```

Remember that $e^{-2} = 1/e^2$.

ANSWER

It is clear that the program is in error - we have proved this by testing the program with data which produces known results.

But the error is not obvious since e^2 is in fact 7.388995. To locate the error, we insert a trace statement before the IF statement and run the program once again.

```
1Ø2 PRINT T,S,T/S
RUN
INPUT VALUE OF X ? -2
-2              -1              2
 2               1              2
-1.33333        -.333333        4
 .666667         .333333        2
-.266667         6.66669E-Ø2   -3.99998
EXP(-2 ) EQUALS 6.66669E-Ø2

END AT 13Ø
READY
```

Now the error is clear, since the computer exits from the loop with the normalised error equal to -4. The error is in Line 1Ø5 where ABS(T/S) should be used instead of T/S.

We correct the error and delete the trace statement, and re-run the program as a final check.

```
1Ø2
1Ø5 IF ABS(T/S)<Ø.ØØØ1 THEN 12Ø
RUN
INPUT VALUE OF X ? -2
EXP(-2 ) EQUALS .135337
END AT 13Ø
READY
```

Always test programs which use complex numerical methods with data producing known results.

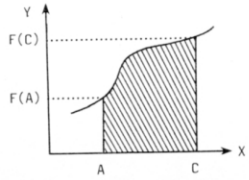
LESSON 59

NUMERICAL METHODS II - INTEGRATION

More methods of calculating approximate solutions with great accuracy.

Whilst much of a school mathematics course is concerned with the evaluation of the definite integrals, in many practical applications an exact solution is often tedious and difficult, if not downright impossible. In such cases, numerical integration will usually suffice, and the computer is particularly suited to this task. The "integral" is the area under a curve. If you have not previously encountered Simpson's Rule you will find it helpful to read Supplementary Lesson 7.

Two common techniques for numerical integration are the *trapezoid rule* and *Simpson's Rule*. Suppose we wish to integrate a function Y = F(X) between X = A and X = C. The function is represented graphically below.



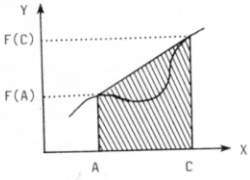
$$I = \int_a^c F(X).dX$$

Now an approximation to the integral $I = \int_a^c F(X).dX$ is given by:

```
I = [F(A)+F(C)]*(C-A)/2
```

This is the trapezoid rule; we have taken the average of F(A) and F(C) and evaluated the area of the rectangle with sides [F(A) and F(C)] and width C-A.

This is equivalent to drawing a trapezoid as shown below.

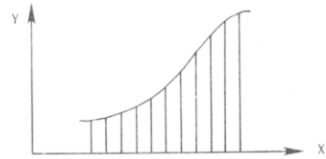


Simpson's Rule is a more complex approximation which proceeds as follows. A mid-point, B, is taken between A and C, and the integral is approximated by the equation:

$$I = (C-A)*[F(A)+4F(B)+F(C)]/6$$

In effect, Simpson's Rule is applied by fitting a portion of a quadratic equation. In general, therefore, Simpson's Rule is more accurate than the trapezoid rule, since most well-behaved functions are in fact smooth curves which are locally approximated by quadratic equations. Satisfy yourself that Simpson's Rule and the trapezoid rule are equivalent when used to integrate equations of the form Y = AX + B.

How is numerical integration accomplished using a computer? The function must first be divided into a number of small segments so that the integration rules can be applied without gross error. For example, the area might be divided into ten segments as shown below.



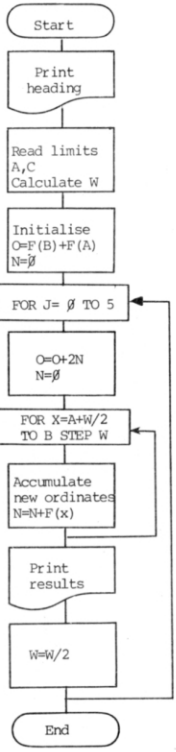
This approach gives us an estimate of the integral, but it provides no information on the accuracy of this estimate. If, however, we now compute the integral using twenty segments we can observe the change in the estimate of the integral, which will give us some indication of the accuracy of the estimate.

Thus a common approach to numerical integration is to start with just one segment, i.e. W = C - A, and to halve W at each iteration until the estimate has reached the required accuracy, or at least until the difference between successive iterations has fallen below a certain level.

PROBLEM

Write a program to integrate the function $Y = e^X \sin X$ between X = Ø and X = 1. Use both the trapezoid rule and Simpson's Rule, and starting with W = 1 progressively halve it until W = 1/32.

This is a difficult exercise, and it is wise to compare your flowchart with that overleaf before presenting your program to a computer. You may well find that the flowchart seems impossibly simple. If so, you will find it explained in the Supplementary Lesson on numerical integration.



O is originally set to the sum of the initial and final values of the ordinates.

Note that this looks only at the new ordinates and sums them ready for the next trapezoid calculation

*N is the sum of the new ordinate i.e. N=N+EXP(X)*SIN(X)*

This loop ensures that the segment width is halved each time i.e. reduces to 1/32nd of its original value in 5 passes round the loop.

SOLUTION: PROGRAM

```
1Ø REM PROGRAM TO INTEGRATE FUNCTION EXP(X)*SIN(X)
2Ø REM BETWEEN LIMITS OF X=Ø AND X=1
3Ø REM USES SIMPSON'S RULE AND TRAPEZOID RULE
4Ø REM     I.WILLIAMSON  31-MAY-99
5Ø PRINT "SEGMENTS","TRAPEZOID","SIMPSON'S"
6Ø READ A,C
7Ø LET W=C-A
8Ø LET O=EXP(A)*SIN(A)+EXP(C)*SIN(C) - see note A
9Ø LET N=Ø  - see note B
1ØØ FOR J=Ø TO 5
11Ø    LET O=O+2*N  - see note C
115    LET N=Ø
12Ø    FOR X=A+W/2 TO C STEP W
13Ø      LET N=N+EXP(X)*SIN(X)
14Ø    NEXT X
15Ø    PRINT 2↑J,O*W/2,(O+4*N)*W/6  - see note D
16Ø    LET W=W/2
17Ø NEXT J
18Ø DATA Ø,1
19Ø END
READY

RUN
SEGMENTS        TRAPEZOID       SIMPSON'S
 1               1.14368         .9Ø8185
 2               .967Ø58         .9Ø9253
 4               .9237Ø4         .9Ø9325
 8               .91292          .9Ø933
 16              .91Ø227         .9Ø933
 32              .9Ø9555         .9Ø9331

END AT 19Ø
READY
```

A This sets' O' to the sum of the initial and final ordinate
B It is always good practice to ensure that N=Ø before the program starts even though standard BASIC sets the value of all variable names to Ø before any program.
C This will increase the value of 'O' except the first when we look at the enclosing trapezium only.
D 2↑J is the number of segments - this PRINT instruction ensures that the first printing of area by Simpson's Rule takes account of the ordinate at the mid point of the initial and final ordinates.

In this program repetitive calculation of ordinates is unnecessary since the trapezoid rule requires only half the number of ordinates used in Simpson's Rule (for large J) and the ordinates are accumulated during each iteration. For effective computer programming you must always be alert to the possibility of minimising the computer load, particularly in calculations such as numerical integration which are naturally iterative.

It is clear from the print-out that integration by Simpson's Rule converges more rapidly to a steady-state value than integration by the trapezoid rule. The error can be estimated by subtracting the results of successive iterations; a quantity E, the *normalised error*, is often used as a measure of the accuracy of the result.

$$(I_{(n+1)} - I_n) \; / \; I_{(n+1)}$$

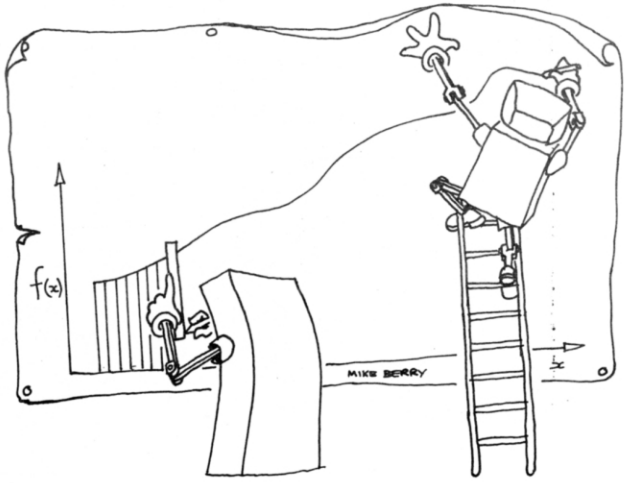
We can automate the integration by successively halving h until E is less than a fixed threshold.

PROBLEM

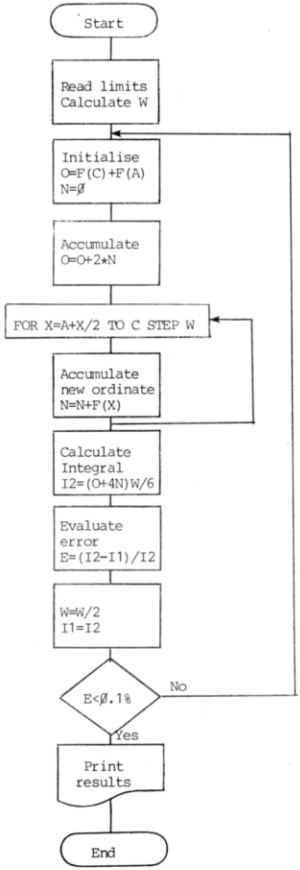
Construct a program which evaluates the integral X.

$$I = \int_0^1 e^X . \sin X . \, dX$$

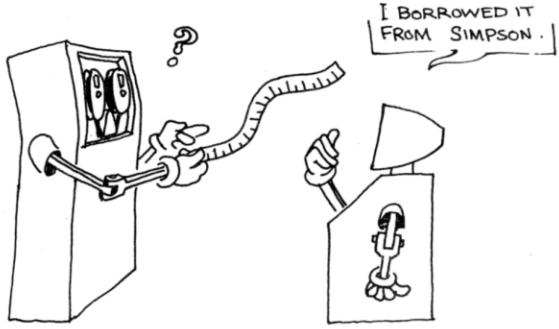
using Simpson's Rule. Stop the iteration when E is less than Ø.1 per cent. Approach this problem with caution as it is particularly difficult. Referring to the last example will help you to draw the flowchart.



SOLUTION : FLOWCHART



```
SOLUTION: PROGRAM

10 REM PROGRAM TO INTEGRATE FUNCTION EXP(X)*SIN(X) BETWEEN
20 REM LIMITS OF X=0 AND X=1 USING SIMPSON'S RULE
30 REM AUTOMATICALLY ITERATES UNTIL ERROR < 0.1%
40 REM    I.WILLIAMSON    31-MAY-99
50 READ A,C
60 LET W=C-A
70 LET I1=0
80 LET O=EXP(A)*SIN(A)+EXP(C)*SIN(C)
90 LET N=0
100 REM O IS ACCUMULATED ORDINATES;  N IS NEW ORDINATES
110 LET O=O+2*N
120 FOR X=A+W/2 TO C STEP W
130    LET N=N+EXP(X)*SIN(X)
140 NEXT X
150 LET I2=(O+4*N)*W/6
160 LET E=(I2-I1)/I2
170 LET W=W/2
180 LET I1=I2
190 IF ABS(E)>.001 THEN 110
200 PRINT "VALUE OF INTEGRAL IS";I2
210 DATA 0,1
220 END
READY

RUN

VALUE OF INTEGRAL IS  .0093258

END AT 220
READY
```



LESSON 60

FILES

Files are needed whenever a variety of programs are used with a single data base.

In this final lesson we have chosen to introduce files, and the use of file instructions in BASIC.

All the program examples in this course have used limited amounts of data, which have been entered using either the INPUT statement, or, for longer data sets, the READ and DATA statements.

In the majority of business applications of computers, large amounts of data are being processed. Data is also being continually updated, and daily, weekly, and monthly summaries of transactions are often required. For example, a stock accounting package for a small business would maintain a file describing the current stock levels against the active parts list. This file would be updated with data taken from the Goods Received Notes at Goods Inwards, with Stock Control Sheets from Production Control, with Invoices from the Parts and Service Group, and so on. The Purchase Director wishes to receive information from the computer telling him when to re-order and in what quantity. The Production Director is interested in stock levels against the current order book. The Finance Director, not surprisingly, usually wishes to minimise the stock level to reduce the company overdraft.

Thus, into one central file, we are entering data derived from a wide range of sources. From the one central file, we wish to present data in many different formats according to the interests of the person on the receiving end. Now we could conceivably write a single program to cope with all these conflicting demands - but it would be enormous, and would be forever being updated and modified to adapt to the changing business environment.

Accordingly, business system packages are usually file-structured, and the files are processed by many different programs to produce the desired final data. The data on Goods Received Notes may be entered into a file using one program. The data on this file could be sorted into part number order by another program and yet another program would take the sorted data file and use it to update the master file of stock inventory. Separate and distinct programs would then operate on the master file to produce the summaries demanded by the Purchase Director, Production Director, and so on.

Files are processed by programs which produce files as an output, such output files then serving as inputs for programs which further process the data, and so on. The system can be modified and developed as necessary to suit the demands of the business, PROVIDED that the FORMAT of the files is fixed and unchanged. Only if the format of the files needs to be modified need we re-think the basic system design.

Files are usually stored on the magnetic storage media of the computer - typically the magnetic disc or the magnetic tape (cassette or reel-to-reel). Files can also be stored on paper-tape, punched cards or printed at the terminal or line-printer - facilities which are used for back-up storage of valuable data and program files.

In all computer systems, the input and output devices such as magnetic tapes, printers, and so on, are controlled and managed by an executive (sometimes called monitor) program which controls the whole of the computer system. It follows that if a BASIC program is to access files stored on the system memory, it must call for assistance from the executive (you will remember from Lesson 35 that the BASIC interpreter and the system executive are resident in the computer memory whilst the BASIC program is being run). Unfortunately, the form of the call to the executive depends not only on the design of the BASIC interpreter but also upon the design of the executive itself.

In this course, therefore, we can only give you general guidance on the type of files you may be able to access, and the form of instructions likely to be included in the BASIC on your computer system. Indeed, BASIC was not conceived as a file-handling language, and for the business user COBOL, despite its age, offers many advantages.

For the BASIC user there are two main classes of file:

Sequential-Access ASCII file

Random-access file.

A sequential-access ASCII file is simply a stream of ASCII-coded characters exactly as typed on the terminal keyboard, line-feeds and carriage returns included. The file is called sequential-access because if we wish to read the 1Øth line we must sequence through the first 9 lines to locate the start of the 1Øth line. The computer cannot locate the 1Øth line without starting at the beginning of the file. For convenience, we will drop the sequential-access term and call these files simply ASCII files. An ASCII file can be listed directly on the terminal, and if line-numbers are included can be edited using the BASIC edit facilities. A BASIC program is stored within the computer as an ASCII file.

Random-access files are so called because they can be read from or written to at any point in the file - i.e. at random. To the user, random-access files appear to be a large array (or list) of numbers or string-variables. We can read the 5Oth entry in a random-access file in precisely the same way as we can call up A(5Ø), the 5Oth element of an array, in a LET statement. Whereas, in an ASCII file we must read all elements prior to number 5Ø, before reading number 5Ø itself. In this respect the ASCII file is analogous to the DATA block in a BASIC program, and the pointer must be moved down the block past each element in turn.

Random-access files are stored in a machine-coded format and cannot be directly listed at the terminal. As a result, random-access files are much more economical in their use of computer memory, and can store large amounts of numerical data very efficiently. However, since they can only be generated and printed via a BASIC program they are more difficult to use. We have not included an example of this type of file in this course and you should read the system manual of the computer you are using for an in-depth description of random-access files.

In the example on the following page, we describe a program using an ASCII file. Beware! You should check your computer system manual as the file-handling instructions described may not be available on your computer system.

In Lesson 55 we described a program to sort 2Ø names and associated telephone numbers into ascending order, and print out a directory. Here is the program modified to accept its input data from an ASCII file.

```
1Ø REM PROGRAM TO COMPILE A TELEPHONE DIRECTORY FROM ASCII FILE
2Ø FILES NAMES
3Ø DIM N$(1ØØ),X$(1ØØ)
4Ø FOR I=1 TO 1Ø1
5Ø   INPUT #1,L,N$(I),X$(I)
6Ø   IF END #1 GO TO 9Ø
7Ø NEXT I
8Ø PRINT "FILE HAS MORE THAN 1ØØ ITEMS."
9Ø FOR X=I-1 TO 1 STEP -1
1ØØ   FOR J=1 TO X
11Ø     IF N$(J+1)>N$(J) THEN 18Ø
12Ø     LET T$=N$(J)
13Ø     LET K$=X$(J)
14Ø     LET N$(J)=N$(J+1)
15Ø     LET X$(J)=X$(J+1)
16Ø     LET N$(J+1)=T$
17Ø     LET X$(J+1)=K$
18Ø   NEXT J
19Ø NEXT X
2ØØ PRINT "NAME",,,"TELEPHONE"
21Ø FOR A=1 TO I
22Ø   PRINT N$(A),TAB(45);X$(A)
23Ø NEXT A
24Ø END

READY
```

Let us look at the new statements in this program:

`2Ø FILES NAMES`

This tells the computer that file number 1 is an ASCII file called NAMES. If the statement had been FILES NAMES, TELE then file number 2 would be called TELE, and so on.

`5Ø   INPUT #1,L,N$(I),X$(I)`

This statement tells the computer to read the next data line from file number 1. The three elements of data on the line are the line number (variable L), the name (string variable N$(I)), and the telephone number (string variable X$(I)).

`6Ø   IF END #1 GO TO 9Ø`

When we reach the end of file number 1 we jump to line 9Ø.

`8Ø PRINT "FILE HAS MORE THAN 1ØØ ITEMS."`

This line is normally skipped by execution of line 6Ø at the end of the file. If the end of the file is not found, the warning message is printed.

Now having defined the new elements in the program, what does our ASCII file look like? Here is a typical session at the teletype:

```
OLD NAMES                  - we ask the computer to fetch the file NAMES
                             already stored in the computer
READY
LIST                       - the familiar list command

NAMES          31-MAY-99


10,JONES;A,45212
20,PEANUT;L,21555
30,JENKINS;P,00212
40,HALLAT;J,91919
50,GLOVER;R.J,62126
60,LISTER;R,89756
70,DALE;R,71122
80,DALES;S,09543
90,HOOK;CAPT,35442
100,KNIGHT;F,23342
110,STARK;W,51841
120,BROWN;D,01950
130,DICKINSON;M,10660

READY
140,WILLIAMSON;I,30198- we can add a line very simply

REPLACE NAMES              - Replaces old file with modified file stored
                             in temporary memory.
READY
OLD COMPLE                 Calls up the program COMPLE listed above.

READY
RUN

COMPLE         31-MAY-99

NAME                                TELEPHONE
BROWN;D                               01950
DALE;R                                71122
DALES;S                               09543
DICKINSON;M                           10660
GLOVER;R.J                            62126
HALLAT;J                              91919
HOOK;CAPT                             35442
JENKINS;P                             00212
JONES;A                               45212
KNIGHT;F                              23342
LISTER;R                              89756
PEANUT;L                              21555
STARK;W                               51841
WILLIAMSON;I                          30198

END AT 250
READY
```

A lot more can be done with ASCII files - for instance, REWIND or RESET will take the file back to the beginning, as the RESTORE statement does with the data block. Programs can also generate ASCII files using a PRINT statement of the form

```
PRINT #1,L,N$(I),X$(I).
```

However, as we have said, the detail of file handling is very much dependent upon the computer system you are using, and you must consult the BASIC manual for the system you are using.

In this lesson we have introduced the two main classes of file, ASCII and random-access files. We have demonstrated how the ASCII files can be simply used for both input and output from BASIC programs, and how they can be edited, listed and saved as if they were actually BASIC programs. From this brief introduction, you are already in a position to develop powerful and useful suites of computer programs - the rest is up to you!

SUPPLEMENTARY LESSON 6

SERIES EXPANSIONS

A 'series', mathematically, is a regular sequence of numbers or expressions.
A series of numbers might be 2, 4, 6, 8, 10, etc., or 1/2, 1/4, 1/8, etc.
A series of expressions might be x, x^2, x^3, x^4, x^5, etc.

As a rule we use a series in order to help us to calculate a number whose value can never be calculated exactly. For example

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} \ldots\ldots \text{ etc}$$

here x is the first 'term' in the series, $\frac{x^2}{2!}$ is the 2nd term, and so forth. 1 is the 'zeroth' term.

QUESTION

Using this expression for e^x write down the value of e itself.

HINT

Remember that $A^1 = A$.

ANSWER

$$e^1 = e = 1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} \quad \text{etc}$$

$$= 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \frac{1}{360} \quad \text{etc}$$

It might seem that this is no answer to the problem. However, we have already determined e to better than one part in 360 when we have calculated this sum. Later terms in the series will never add up to as much as 1/360, which was our last term. A computer can readily calculate such a series, however, there are dangers.

A series might not 'converge' to a finite sum. The terms of such a series add up to an infinitely large number if you take enough of them - the series is said to be divergent. Some series are obviously divergent - but some series are quite deceptive.

QUESTION

Three of the four series below are divergent - can you spot them?

a) $1 + 2 + 3 + 4 + 5 \ldots\ldots$

b) $1 - 1 + 1 - 1 + 1 - 1 \ldots\ldots$

c) $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \ldots\ldots$

d) $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \ldots\ldots$

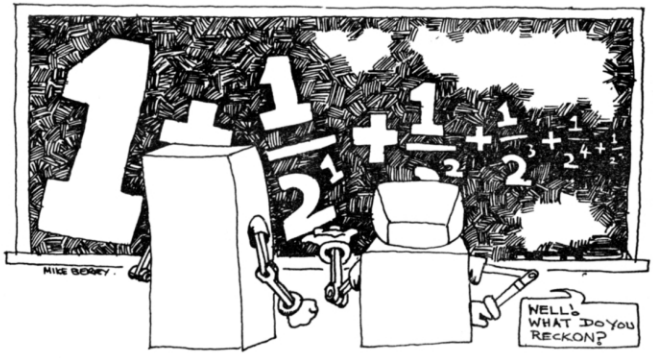
ANSWER

a) This series is obviously divergent.

b) The series adds up to 1 or Ø depending how many terms are counted - it is oscillating and never converges to a single sum. A divergent series need not diverge to infinity.

c) Surprisingly, this series is also divergent. Although the terms get progressively smaller, the sum of the series tends to infinity as more and more terms are added.

d) The series converges to a sum of 2.



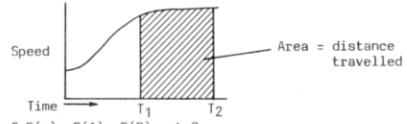
SUPPLEMENTARY LESSON 7

NUMERICAL INTEGRATION

This is a subject which you may find difficult to understand if you cannot do integral calculus. However, it can be explained even to people who have never heard of calculus.

What is 'integration'? In this lesson 'integration' simply means 'finding the area under a curve'. 'Numerical integration' means 'dividing the area into tiny parallel segments, finding the approximate area of each segment and adding the areas of all the segments'. This will only give us the approximate area, but a computer will very quickly find such an area to one part in a thousand, or ten thousand, or even one million, which will be accurate enough for every practical use.

Why bother - who needs such areas? The area may be needed literally, for example, to determine the amount of steel plate in an ellipse. More often the axes of the graph will represent, for example, speed and time, and the area will then represent distance travelled.



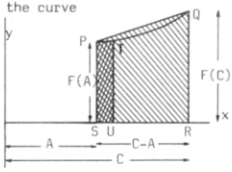
What is the meaning of F(x), F(A), F(B), etc?

y = F(x) means that y varies as x varies. The mathematical way of saying it is 'y is a function of x'.

F(A) is the value of y when x is A.
F(B) is the value of y when x is B.

What is the trapezoid rule?

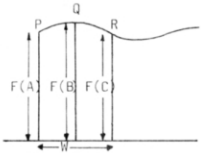
I = [F(A) + F(C)] * (C - A) / 2 means,
'I' the area under the curve



Area of trapezium = $\frac{[F(A) + F(C)]}{2} * (C - A)$

is assumed to be equal to the area of the trapezium PQRS. We can easily see that this is not so for such a wide trapezium. But if we look at the left hand edge only it is now impossible to see the difference with the naked eye between the area under the curve, PT, and the trapezium, PTUS.

How about Simpson's Rule? Now, instead of assuming that a series of side-by-side trapeziums would be a good fit to the curve, we assume that it is itself composed of many curves. Each segment is a part of a simple parabola which fits the two end points P and R and the mid point Q.



Now the exact area of a segment of such a parabola is

$$\frac{[F(A) + F(C) + 4*F(B)]}{6} * W$$

This can be proved but to do so now is not essential.

How can a computer best deal with this type of integration? The temptation with a computer is to divide a curve into, say, a million tiny parallel segments, work out the area of each segment, then add up all these areas, and then print the result.

QUESTION

This approach has four disadvantages. Can you think of any of them?

HINT

Think about: storage locations, time for computation, the use of tiny numbers, and how you will judge the accuracy of your results.

ANSWER

a) A million memory locations could be needed, one for the area of each segment. However, we could get round this snag by keeping a running-total, to which we add the area of each new segment as we find it.

b) The calculation will take a long time. Even if each area can be calculated in 1ØØ microseconds it will still need 2 minutes to compute a million areas. Time costs money on computers.

c) Each of the areas is itself tiny. There may be difficulty in computing numbers such as .ØØØØØØØ1Ø1Ø2. So the accuracy may prove to be lower than you might expect.

d) You will have no idea how accurate your calculation is. You could spend another minute calculating the area with 5ØØ,ØØØ segments. The difference in the two areas would tell you roughly the order of accuracy that you had achieved. To be pedantic the inaccuracy of your answer will probably be the same order of magnitude as the difference between the two answers.

QUESTION

Can you think of a better approach to this problem, by which you could calculate the area under a curve -

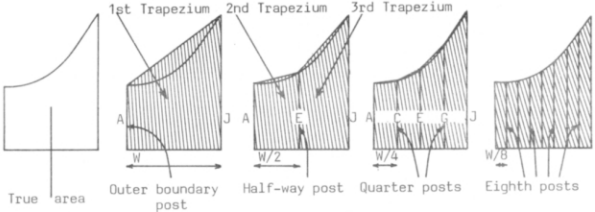
a) with increasing accuracy at each attempt
b) in a reasonable amount of computing time
c) without tiny segment-widths requiring that your computer works to many decimal places
d) which will give you an idea of how inaccurate your estimate might be?

HINT

Think of finer and finer subdivision.

ANSWER

You can use narrower and narrower trapeziums to estimate the area under a curve as shown in the diagram.



True area	Outer boundary post	Half-way post	Quarter posts	Eighth posts
	1st approximation to area. Trapezium within the outer boundary posts. Clearly very poor, with a big error.	2nd approximation to area. 2 trapeziums defined by halfway post and boundary posts. Already far more accurate.	3rd approximation to area. 4 trapeziums defined by linking quarter posts to previous boundary points. It is now hard to see the error.	4th approximation to area. 8 trapeziums defined by linking eighth posts to previous boundary points. By now it is impossible to see any error on a small-scale diagram.

Each approximation is more accurate than the last. A computer can work out each approximation by adding the height of the new posts to the height of the posts already measured.

QUESTION

Show how each approximation is made up of previously known post-heights plus the latest posts to be considered. (N.B. We use the word 'post' which is self-explanatory in preference to the correct mathematical term 'ordinate'.)

ANSWER

1) Area of 1st Trapezium = $(A+J)*W/2 = P_1*W/2$
   Where P_1 = sum of outer boundary posts.

2) Area of 2nd Trapezium = $(A+E)*W/4$
   Area of 3rd Trapezium = $(E+J)*W/4$
   So 2nd approximation = areas of 2nd and 3rd Trapeziums
   $= (A+E)*W/4+(E+J)*W/4$
   $= (A+J+2E)*W/4$
   $= (P_1+2E)*W/4$
   So, in computing terms, we could add twice the height of the half-way post to the sum of the boundary posts in order to reach our second approximation. And $P_1 + 2E$ can be called P_2.

3) Area of 4th Trapezium = $(A+C)*W/8$
   Area of 5th Trapezium = $(C+E)*W/8$
   Area of 6th Trapezium = $(E+G)*W/8$
   Area of 7th Trapezium = $(G+J)*W/8$
   So 3rd approximation = area of 4th, 5th, 6th and 7th Trapeziums
   $= (A+2C+2E+2G+J)*W/8$
   $= [P_2+2(C+G)]*W/8$
   here C+G is the sum of the height of the quarter posts.
   Again, in computing terms, we could add twice the sum of the height of the quarter posts to P_2, which we have already stored. We can now call the new sum P_3.

So the general rule for the next approximation is to add to the previous 'post sum' twice the sum of the heights of the next series of dividing posts and to multiply the new sum by half the width between the posts.

With Simpson's Rule the approach is slightly more complicated, but the same principle applies. We take the sum of the previous posts (ordinates) and add four times the new sum of posts (the mid-point of each integration area) and multiply the whole lot by the area width divided by twelve.

GLOSSARY OF TERMS

This glossary of computer terms is not fully comprehensive - it would require a whole book to be that - but it sets out to define intelligibly many of the terms you are likely to come across whilst using this course, and whilst working with computers.

Alphanumeric characters - the complete character set: numbers, figures and symbols.

Applications software - programs developed for specific tasks or applications: payroll systems, mathematical routines, graph-plotting, etc.

Argument - a variable upon whose value the value of a function depends (e.g. the argument of SIN(X) is X).

ASCII - pronounced 'askey'. Stands for American Standard Code for Information Interchange. The code by which alphanumeric characters are encoded for transmission in computer systems; by telephone line, paper-tape, or magnetic-tape.

Assembler - program which translates symbolic instructions into the machine code required by the computer.

Assembly language - sometimes also called assembler. A simple language, which is used to encode into machine code. Often there is a one-to-one relationship between instructions in assembly language and instructions in machine code. Assembly language includes mnemonics to help the user (e.g. the three-character group which adds two variables may be ADD).

Batch - has many meanings. Usually used to differentiate between on-line working and off-line working. In batch processing the computer processes the jobs in strict sequence, dedicating its full facilities to each batch job in turn. All instructions, programs and data for each job are pre-loaded onto the computer and stored on tape, disc, or punched cards.

Binary code - the method used by digital computers for encoding numbers. At its simplest level, the digits Ø-9 may be represented thus in binary code:

Ø	ØØØØ	5	Ø1Ø1
1	ØØØ1	6	Ø11Ø
2	ØØ1Ø	7	Ø111
3	ØØ11	8	1ØØØ
4	Ø1ØØ	9	1ØØ1

Bit - this is a Ø or a 1 in binary code. The word is derived from BInary digiT.

Branch - a departure from the straight down-the-page sequence in the execution of a program. Branchings can be divided into two sorts - conditional and unconditional. In BASIC, unconditional branching is represented by GO TO statements and the NEXT of a FOR...NEXT loop. The computer has no alternative but to follow these instructions, which have no conditions attached to them. In conditional branching, the branching is dependent upon the value of some variable, examples of this in BASIC being IF...THEN and ON...GO TO.

Bureau - a computer bureau is a company which leases computer time to clients, either on a time-sharing basis, by means of telephone-linked terminals installed in the client's place of business, or else by the batch method, where programs are coded onto tape or cards, and fed into the computer in sequence. Often computer bureaux will combine the two - using a time-sharing system during the day, and at night, when the demand for time-sharing is not so great, switch to batch working.

Byte - a set of bits considered as a unit, usually (but not always) 8. The memory of a computer is measured by the number of bytes it can store.

Central processing unit - in some senses, the 'heart' of the computer. It co-ordinates the activities of the computer's units, and performs the arithmetical and logical operations on data.

Command - can be synonymous with instruction, but not always so. In the sense we have been using, this word refers to those words typed by the user which are not part of the program, but nevertheless affect the computer's working, e.g. LIST, RUN, and SAVE.

Compiler - a program within the computer, usually supplied by the manufacturers, which 'translates' operating instructions (e.g. the BASIC instructions) into machine-code, and calls any library routine (applications software) which are required by the program. In this, it is similar to the assembler, but is different in that an assembler usually (but not always) produces one machine-code instruction for each source instruction, whilst a compiler is more sophisticated, and can generate many more machine-code instructions per source instruction. The compiler translates an entire program at one fell swoop.

Conditional branch - also known as conditional transfer or conditional jump. A branch which occurs if, and only if, some specified condition is met.

Control statement - synonymous with command.

Data - all values, conditions or states operated on by a program but not forming part of the program itself.

De-bug - the process and technique of detecting, diagnosing and eliminating errors in programs or systems (software or hardware).

Diagnostics - the result of a diagnostic routine (usually part of the software supplied with the computer), designed to trace (or diagnose) faults in a program or system. For instance, a sample diagnostic following the entry 1Ø LET C=BA might run: ERROR Ø2 - SYNTAX - thus giving the user a clue as to the nature of the error.

Enter - (of program) to feed in. Sometimes used in place of load.

ESC - the abbreviation in this course for the escape key on a terminal. Some terminals may not have an escape key, and the same result may be obtained by pressing CONTROL and Q simultaneously, or similar. The function of this is to halt the execution of the program.

Executable statement - a statement in a program which causes the computer to do something, e.g. PRINT, READ, LET (*cf* non-executable statement).

Execute - to perform the operations defined by a routine, program or instruction.

Executive program - software built into the computer which controls various functions of the system.

Expression - a mathematical or logical relationship expressed in symbolic form.

File - an organised collection of data or records. This definition applies, of course, as much to non-computer files as to computerised ones, but much jargon has grown up around computerised files. File handling is the technique of producing and manipulating files, and the file identification is the code used to identify and sometimes call the file for use. The file label is part of the file itself, and serves the same function as does the REM statement in the program - i.e. a non-executable datum serving as a reminder to the user of the purpose and function of the file. A file name may be the same as a file identification, and file processing the same as file handling. Two files may be merged, collated, sorted, or compared. Files, of course, may be on tape (magnetic or punched), cards, or in the main storage of the computer.

Flowchart - the system of symbolic representation of sequences of events. A program flowchart is the one used throughout this course, where different shaped boxes (each representing a different computer operation), are linked by flowlines to each other. Flowcharts are used to interpret the requirements for a program, to define the processes in the programming which will be involved, and to make the coded program more intelligible to others. The other form of flowchart which is used in computer operations is the systems flowchart, which details the organisational methods involved in collecting data, running the computer, and presenting the information obtained. A flowchart may also be called a flow diagram.

Hardware - the physical pieces of a computer system, as opposed to software.

High-level language - a set of coding instructions in which each statement or instruction in the language corresponds to more than one machine code instruction. They have the advantage of being used on different machines, irrespective of the machine code of different systems.

Input devices - devices used to input data and programs to the memory of the computer. Examples of such devices are the teletypewriter and VDU, paper tape readers, punched-card readers, etc.

Instruction - a statement within a program defining a specific action required from the computer.

Instruction mnemonics - abbreviations used in assembly language to define computer operations. For instance, in assembly language, the mnemonic representing addition may be ADD.

Interactive mode - also known as conversational mode. As the second name would imply, this is a method of using a computer in which the communication between the user and the computer is two-way, i.e. the computer is able to 'reply' to commands and instructions given it by the user, and the user is able to affect the workings of the computer and obtain immediate repsonse to input.

Interactive program - a program which is influenced by the user as it progresses by means of input messages and data.

Interpreter - a program within the computer which translates high-level language instructions into machine-code, *one at a time*. Every time the computer runs through a program, the interpreter translates the instructions. (*cf* compiler).

Job - work (one unit) given to a computer. It may well consist of several runs. Jobs given to a computer are controlled by the job control program, written in job control language, which enables the operator (or the user) to control the flow of jobs through the system.

Language - the method of communication between users and computers. From near-meaningless jumbles of symbols, languages have evolved to near-English. The breakthrough in language development came when it was realised that a computer could do its own 'translating' and the laborious and time-consuming checking which constituted much of programming could be done away with. It is regrettable that there is not yet a universal computer language, but various languages are designed to meet different needs - COBOL, for instance, being a business language, which is very 'English' in form, and FORTRAN, primarily a scientific language. A language called PL/1 has been designed to cover aspects of both these languages, and has gained some popularity.

Line printer - an output device capable of printing a complete line at a time (as opposed to a teletypewriter, which prints one character at a time).

Load - to input data or program instructions into the memory of a computer.

Local working - using a terminal without being connected to a computer. This may be for the purpose of producing paper-tape or similar data preparation.

Logging on - the act of gaining access to the computer from the terminal. Usually a secret password is used, to prevent access to personal files by unauthorised users.

Logic error - an error in a program resulting from faulty logic (e.g. dividing by Ø, or similar).

Low-level language - a language in which each instruction has a corresponding machine-code instruction as opposed to a high-level language, where many machine-code instructions may be generated by a single instruction. Sometimes referred to as a basic language (not to be confused with BASIC, which is a high-level language).

Machine-code - the binary code which defines the set of instructions available within the computer. A very elementary coding system, also known as computer code, instruction set, order code, or instruction code.

Magnetic disc - a memory device, consisting of a number of metal plates coated with a magnetic medium (similar to audio tapes). Discs may store millions of bytes of data which can be accessed (read or written) rapidly. A floppy disc (as opposed to a hard disc) is a thin plastic disc storing typically a quarter of a million bytes of data.

Magnetic drum - similar in principle to magnetic discs but in the form of a cylinder, coated with magnetic material, revolving past a line of read-write heads.

Magnetic tape - used for storage, and similar to audio tape, but ½-inch wide (as opposed to the usual ¼-inch in use on domestic tape recorders), and moving much faster (up to 75 inches per second). Cassette-tape is now widely used instead of paper-tape for long-term storage of programs and data.

Memory - the internal storage of a computer, and strictly the term is only applicable to those storage locations which can be directly reached by the CPU.

Micro computer - small computer, with limited storage facilities. Only recently available, due to developments in microelectronics and rapidly gaining in popularity.

Mini computer - larger than a microcomputer, and with more storage facilities, and a more sophisticated CPU, allowing for easier programming.

Modem - an acronym for modulator/demodulator. The device used to transmit data over distances. With such a device it is possible to use a terminal connected by an ordinary commercial telephone line to a computer on the other side of the world. This device has made possible the growth of time-sharing computer bureaux.

Non-executable statement - statements in a program which do not cause the program to do anything, e.g. REM, DATA (*cf* executable statement).

On-line - when a part of a computer system (terminal, memory, etc.) is connected and directly under the control of the CPU, then it is said to be on-line.

Operator - controls the computer's operations, as well as those of the peripheral units. He will probably be trained in a special operator's language to control the computer and its workings (*cf* user).

Output devices - means by which humans can understand computers. These may take many forms, but the most familiar is writing, either by means of a teletypewriter, line printer, or VDU. Other output devices include graphic displays, graph plotters, data transmission lines and speech synthesisers (devices to simulate the human voice).

Paper tape or punched tape - is one of the earliest devices for inputting data into digital computers. The advantage of paper tape is that it is compact, cheap and easy to transport, being unaffected by magnetic fields in the same way that magnetic tape is affected. Reading speed is around 1000 characters a second, and paper tape can often be prepared as a by-product of other processes, e.g. an accounting machine or cash register.

Parity - the system of checking data against original source material. This is done by introducing a parity bit, in which the bits of a word are added together to determine whether they are odd or even, and the value of the parity bit depends on this result.

Peripheral units - those units of a computer system under computer control. This includes output devices and memory stores.

Print-out - or 'hard copy' is the paper on which the computer has printed results, listings, etc.

Program - a set of computer instructions for solving a problem.

Punched cards - a method of representing data. Each card has 80 columns, with 12 positions in each column and a card will usually represent one line of a program. They are prepared on a key punch or card punch and then fed into a card reader, which forms the link between the cards and the computer. Cards may also be sorted mechanically, collated, or mechanically reproduced. Though bulky and occasionally cumbersome, punched cards are versatile in that data is not represented serially (in order), as on tape of any kind. For instance, if cards are being used to check data on shoe sales, all cards referring to ladies' shoes, size 5, can be mechanically sorted, saving computer time.

Run - the execution of a program. Also used as a verb to describe such an action. Often programs are combined to form a job consisting of several runs.

Shift - the key on a teletypewriter or VDU terminal which shifts the usage of the keys from lower- to upper-case characters.

Software - can be used to refer to all programs usable on a computer system. An example of software might well be a statistics package which will convert data to a set of meaningful statistics. These may well be usable as subroutines. The cost of developing software is high, and is an important consideration in the design of a computer system.

Storage - see memory.

Subroutine - a part of a program forming a logical section of the program, addressable by a branch from the main stream of the program. The advantage of a subroutine is that the instructions do not need to be repeated every time the subroutine is called.

TAB - a command to control the position of print-out.

Teletypewriter (Teletype *, teleprinter) - A typewriter-like input and output terminal. (*Teletype is the trademark of the Teletype Corporation of the USA.)

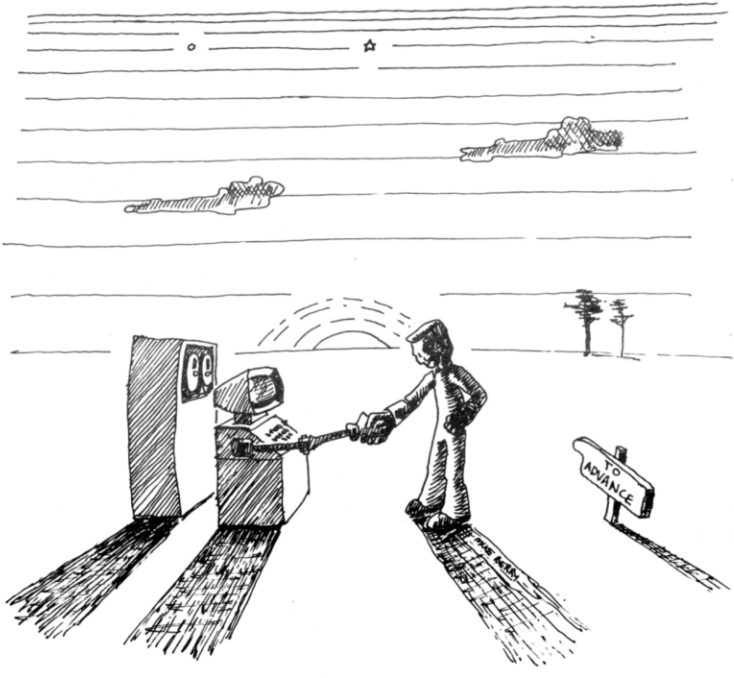
Test data - data prepared to test a program. Ideally, it should cover as many combinations of data, likely and unlikely, as possible.

Time-sharing - a method of operating a computer so that more than one user can be accomodated simultaneously, and all users have the illusion of constant use of the computer, as opposed to batch.

Tracing - the technique of debugging, using PRINT statements to print out intermediate values of variables throughout the program, with a view to locating the error precisely.

User - individual or group making use of the output from a computer system.

VDU - a form of terminal, consisting of a television-like display screen, onto which are projected the computer's responses. In addition there is a typewriter-like keyboard by means of which data and programs may be entered, or inquiries made to the computer. These user-entered characters are also displayed. The advatage of a VDU is primarily speed - it is much faster than any other method of printing, but does not leave a permanent record.



SUMMARY of BASIC LANGUAGE

Lesson numbers printed in italics

STATEMENTS

		Lessons
DATA	DATA 4,3.12,-24	*4,25*
DEF FN	DEF FNA(N)=(1+3.14159*SIN(N))	*52*
DIM	DIM A(2Ø),B(1ØØ),C(Y+2*Z)	*42*
END	END	*4*
FILES	FILES NAMES	*6Ø*
FOR...TO...STEP	FOR N=1 TO 1Ø STEP 2	*36*
GOSUB	GOSUB 1ØØØ or GOSUB (X+Y)	*5Ø*
GO TO	GO TO 8Ø or GO TO (X+Y)	*3Ø*
IF...THEN...	IF X>=Y THEN 9Ø	*3Ø*
IF END	IF END # 1 GO TO 9Ø	*6Ø*
INPUT	INPUT A,B,C	*28*
INPUT (of a file)	INPUT # 1,L,N$(I),X$(I)	*6Ø*
LET	LET A=B+1Ø	*6,13*
MAT	MAT X=Y+Z	*57*
NEXT	NEXT N	*36*
ON...GO TO...	ON N GO TO 1ØØ,12Ø,13Ø	*57*
PRINT	PRINT, "LABEL";N,A	*4,14,29*
PRINT USING	PRINT USING 4Ø,X,Y	*57*
RANDOMISE	RANDOMISE	*46*
READ	READ A,B,C	*4,25*
REM	REM THIS IS A REMARK	*27*
RESTORE	RESTORE	*39*
RETURN	RETURN	*5Ø*
STOP	STOP	*31*

NUMBERS

123	-456	7.981	-Ø.ØØ4	1E1Ø	-1.4E-4
(4)	*(4)*	*(4)*	*(4)*	*(11)*	*(11)*

VARIABLE NAMES

A	Z1	B(1)	X(12Ø)	Z(I+4*J)	T(4,4Ø)	C$	D1$	Y$(15)
(12)	*(12)*	*(41)*	*(42)*	*(42)*	*(42)*	*(54)*	*(54)*	*(54)*

FUNCTIONS

ABS	ATN	COS	EXP	INT	LOG	RND	SGN	SIN	SQR	TAB	TAN
(24)	*(23)*	*(23)*	*(24)*	*(24)*	*(24)*	*(46)*	*(24)*	*(23)*	*(9)*	*(48)*	*(23)*

OPERATORS

( )	↑	*	/	+	-	:	=	>	<	>=	<=	<>
(8)	*(6)*	*(4)*	*(5)*	*(4)*	*(6)*	*(SL3)*	*(4,54)*	*3Ø & 54*				

COMMANDS

DELETE	KILL	LIST	NEW	OLD	REPLACE	RUN	SAVE	UNSAVE	\	←
(SL2)	*(5)*	*(SL2)*	*(5)*	*(5)*	*(SL2)*	*(5)*	*(5)*	*(SL2)*	*(SL1)*	*(SL1)*

Cambridge Learning Limited,
Rivermill Lodge,
St. Ives,
Huntingdon,
Cambs PE17 4EP.

ISBN 0 905946 05 7