Cambridge Learning Limited

Computer Programming in BASIC

A SELF INSTRUCTION COURSE in 4 Volumes

PART 2 — Introducing BASIC

By:
Ian Williamson Rodney Dale Tim Eiloart

```
1Ø REM PROGRAM TO CALCULATE THE TOTAL WEIGHT OF
2Ø REM FOUR ITEMS AND PRINT TOTAL IN KILOGRAMS AND
3Ø REM NEAREST EQUIVALENT IN POUNDS AND OUNCES
4Ø REM    I WILLIAMSON     31-MAY-99
5Ø READ A,B,C,D
6Ø LET K=A+B+C+D
7Ø LET W=K*2.2Ø46223
8Ø LET P=INT(W)
9Ø LET O=INT((W-P)*16)
1ØØ REM OUNCES ARE TRUNCATED TO NEAREST WHOLE NUMBER
11Ø PRINT "TOTAL WEIGHT IS",K,"KILOGRAMS"
12Ø PRINT "WHICH IS APPROXIMATELY",P,"POUNDS",O,"OUNCES"
13Ø DATA 4.825,2.212,5.624,4.292
14Ø END
```

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 2: Introducing BASIC

by

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

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 18

HIGH-LEVEL AND LOW-LEVEL COMPUTER LANGUAGES

High-level languages, such as BASIC, are easy to learn. Low-level languages are much more like the computer's machine-code.

In Part 1 you learnt a few BASIC statements, how to write some simple BASIC programs, and how to enter these programs into the computer and execute them.

The programmer communicates with the computer using a language understandable both to him and his machine. BASIC is only one of many computer languages. In this lesson we describe the various types of computer language you are bound to meet if you become a programmer.

We start with *low-level languages*.

The operation of a computer is controlled by programs stored in it, which are indistinguishable from any other data or information stored in it. Programs, data, and other information are stored and manipulated in *binary code*, that is one composed of 1s and Øs only. A *binary digit* (Ø or 1) is called a *bit*, and a group of bits (usually 8) is called a *byte*. (Words in italics are defined in the glossary).

When computers were first developed in the early 1950s, all programs were written using *numeric codes*; such computer programs are said to be in *machine-code*. Programming in machine-code is tedious and difficult, as it needs great attention to detail, and, not surprisingly, the machine-code programmer is very much prone to error. Why? Because machine-code is unfamiliar, and therefore hard to remember; exactly the opposite of what is wanted in a language.

An improvement upon machine-code is one which allows a programmer to write his program using meaningful abbreviations and symbols. The computer itself is arranged so that it will translate this symbolic language into its machine-code. Abbreviations, when used to represent the action required of the computer, are called *instruction mnemonics*.

Symbols and mnemonics are easy to remember, which makes the task of programming much less prone to error. Let us suppose, for example, that to add two numbers the programmer writes ADD. Let us also suppose that the locations where those numbers are stored in our computer memory, have symbolic names such as CAT, or FRED.

In our symbolic language we write:

```
ADD CAT, FRED
```

meaning 'add the number stored in the memory location called 'CAT' to that in the memory location called 'FRED'.

Now the computer will have to turn this instruction into its binary machine-code, perhaps as follows:

ADD	1Ø1ØØØ1111
CAT	1Ø11Ø
FRED	11Ø1ØØ

It performs this transaction by means of an *assembler*, a program built into the computer by the manufacturer. Our simple symbolic language is called an *assembly language*. A distinguishing feature of assembly languages is that each symbolic instruction corresponds, except in special cases, to a single machine-coded instruction within the computer.

Different computers have completely different assembly languages, and programming in such a language needs a full understanding of the inner workings of a computer. However, a technique known as *microprogramming* can make one computer understand another computer's assembly language. Assembly-language programming is the province of the professional programmers, who receive training in the language of specific computer systems. Assembly language does have the advantage that great efficiency can be achieved, since all the computer operations are under the direct control of the programmer.

QUESTIONS

1) What is the distinction between machine-code and assembly language?

2) What is an assembler?

ANSWERS

1) Machine-code is the numeric code in which the instructions and data are stored within the computer.

   Assembly language is a set of mnemonics and symbols, meaningful and understandable to a human operator, which can be translated into machine-code by the computer.

2) An assembler is a program, supplied by the computer manufacturer, which, when executed by the computer, will translate an assembly language program into machine-code.

The terms *user-programs* and *system-programs* are often used to distinguish between the programs generated by programmers (users) for specific applications, and programs (such as the assembler) supplied by the computer manufacturer, as part of the computer system.

So low-level languages are computer languages where there is a one-to-one relation between the program instructions and the operations performed by the computer. A high-level language such as BASIC has a one-to-many relation - a single instruction may generate as many as 20 or more operations. (By "operation" I mean, for example, ADD 'a' to 'b' - strictly this may require several steps within the computer.)

As a general, though not rigid, rule, *system-programs* (those affecting the workings of the computer) are written in assembly-level or low-level languages, and *user-programs* are written in high-level languages.

In many high-level languages, the range of symbols and mnemonics is extended so much that the program is similar to, and almost as readable as, the English language itself. An important characteristic which distinguishes high-level languages from low-level languages is that the former are generally offered by several computer manufacturers, and are not limited to a single type of computer system as are the latter.

We can emphasise these points by examining a small program in a high-level language. Suppose we want to evaluate the expression AX/4+B and call the value Y, printing out this value.

PROBLEM

Write out a BASIC program to evaluate Y = AX/4+B

ANSWER

```
10 INPUT A,B,C
20 LET Y=A*X/4+B
30 PRINT Y
40 END
```

Now, whilst this small program replaces perhaps 100 machine instructions, its construction is simple, and, what is more important, it is easy to understand. High-level languages such as BASIC are sometimes called *problem-oriented languages* because they are designed to simplify the process of programming for specific types of problems; this example shows an arithmetic problem. Scores of problem-oriented languages have been developed over the last two decades, but only a few have achieved wide acceptance. Problem-oriented languages in common use are BASIC, FORTRAN, APL, and, to a lesser extent ALGOL for scientific problems. COBOL is used for business programs, and PL/1 (called 'P.L.one') for both business and scientific problems. BASIC is one of the most widely used high-level languages. It is also the high-level language which is used by almost all low-cost computers - the kind the man-in-the-street can buy.

PROBLEM

Give three distinctions between assembly languages and high-level languages.

ANSWER

1) A statement in high-level language may generate many machine-code instructions when translated, whereas there is usually a one-to-one relationship between an assembly-language statement and a computer's machine-code.

2) A high-level language is generally available on many different computer systems - whereas an assembly language is specific to one type of computer system.

3) A high-level language is easier to learn and understand.

A few more points on languages. When you are programming, various features of the computer make themselves more apparent when assembly language is used than if high-level languages are used. Physical constraints, such as the size of the memory available, make themselves evident.

Although many high-level languages are good, they are not as yet as efficient as a good human assembly-language programmer. For this reason good assembly-language programs are more efficient than high-level programs. However, the cost in time of preparing assembly-language programs is greater than that involved in high-level programming. For programs which are used over and over again (systems-programs), assembly language is often best, as maximum efficiency is obtained whilst running the program, in terms of storage-space and execution-time. This will often offset the extra time for preparation.

LESSON 19

BASIC AND COMPUTER PROGRAMMING

The reason that BASIC was developed, and the five steps in computer programming.

BASIC is a high-level computer language which is widely available on modern computer systems. It resembles other computer languages in many respects, but has been designed specifically to be simpler and easier to use. BASIC (Beginners All Purpose Symbolic Instruction Code) was developed at Dartmouth College, New Hampshire, USA in 1965 under the direction of J. Kemeny and T. Kurtz. It is a simple language to learn, but it is nevertheless powerful, and it also serves as a good stepping-stone for students who wish to master other languages such as FORTRAN or COBOL.

In most high-level languages, you must learn a lot before you can start to write simple programs. Some people find this quite difficult. But once the language has been mastered sufficiently to write even a simple program, relatively little extra knowledge is required to master the language completely. With BASIC this is not so; a wide variety of programs can be written using but a few simple instructions. Many computer systems offer an advanced version of BASIC called *extended BASIC* which has the power required for very sophisticated applications. Manufacturers of mini- and micro-computers often provide a limited version of BASIC, which may go under a trade-name such as "TINY-BASIC". In this course, we introduce the whole range of BASIC - from the most elementary statements, to the advanced instructions available in extended BASIC.

QUESTION

Why was the BASIC language developed?

ANSWER

To provide a simple and easy-to-learn (though nevertheless powerful) computer language, and to form a stepping-stone for students wishing to master other languages such as FORTRAN or COBOL.

You have already started to use BASIC as a programming language. However, the programming language or code is only one element in the programming process.

There are five steps in computer programming, namely:

1) - systematically defining the problem (you did this in an unsystematic way with the computer's estimates for making boxes in Part 1).

2) - drawing a flowchart (which you have not yet needed).

3) - coding the program, i.e. writing the BASIC program (you have started to do this).

4) - testing and correcting the program (which you have barely needed).

5) - documenting the program (which you have not yet done).

The *definition of the problem* must be completed before you think about writing the program. The elements of the problem and the objectives of solving it must be clearly defined at this stage. Four aspects of the problem should be considered:

- the computational methods to be used
- the source and form of the data input
- the type and form of the data output
- the means of program control

The next step is to draw a flowchart. A *flowchart* is a diagram constructed of standard symbols which shows the logical structure of the program.

Once the program has been defined, and a flowchart drawn, *coding the program* is straightforward and consists of translating the flowchart into a form which can be understood by the computer.

A program almost inevitably contains errors, so it must be tested and corrected, or *debugged*. Many of the errors will be trivial language errors (such as the incorrect use of verbs, or omission of essential punctuation). If such an error is detected by the computer, it issues an *error diagnostic message* which tells the programmer what and where the error is. However, there may be errors either in the logical structure of the program, or in the computational process applied. In Lesson 15 we distinguished between syntax errors - which are usually trivial - and run-time errors which are more serious. Although they cause the program to function incorrectly, they cannot be detected by the computer, and are therefore difficult to find.

When a program has been proven satisfactory, the next step is its *documentation* to a standard where it can be readily understood later. The quality of program documentation varies according to the purpose of the program. One of the marks of professional programmers is the standard of their documentation, which is such that any other competent programmer can step in, update, and modify the program, with a minimum of learning time. On the other hand, people who produce a program for their own use need not document to full professional standard. Nevertheless there is a minimum level of documentation, even when a program is for one's own eyes only, and it pays in the long run to aim at a high standard of documentation.

In the next lesson we show how to draw flowcharts; we have already taught you how to code simple problems in BASIC. Later you will learn how to define them.

However, we can only show you by example how to document the program, and how to test and debug it. You will best learn these aspects of programming by experience.

Equally difficult to teach, even in a programmed learning course, is the process of problem-definition, and how to become a good programmer. But it is the quality of your problem-definition which will determine how good a programmer you will eventually become. Remember this aspect of computer programming when you examine the examples discussed in the remainder of this course.

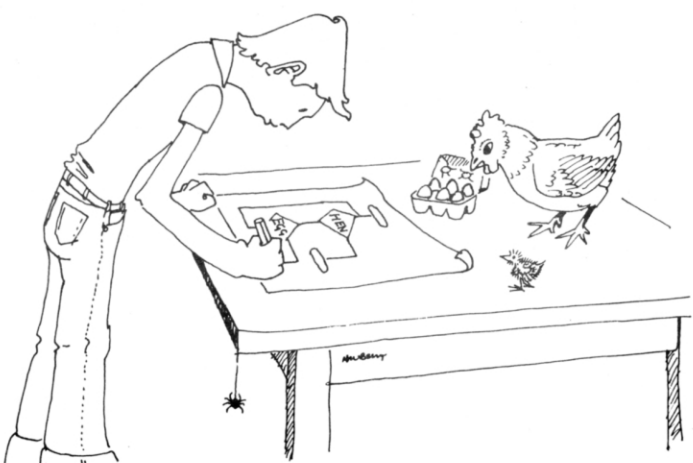
First, we introduce the terms and symbols used in program flowcharting.

QUESTION

Why is program documentation important?

ANSWER

Because a well documented program can easily be understood by another programmer, or by you if you come back to work with the program many months later.



LESSON 20

WHAT IS A FLOWCHART?

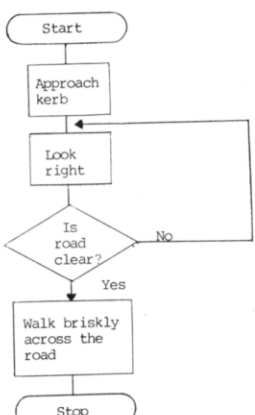
Flowcharts are step-by-step diagrams that lead to a solution by means of yes-no questions.

A computer comes into its own when it is applied to complex, repetitive or tedious processing problems. Whilst it is possible to translate a complex processing-problem into a computer program directly, the process is very much prone to error, and is avoided by good programmers. An essential intermediate stage is a program flowchart which outlines the procedure for solving the problem step-by-step. A diagram showing the *logical flow* of the problem helps you to define the computational techniques required, and to code the program. It is also an invaluable aid to debugging. In addition, the program flowchart is an essential part of the program's documentation. Below is a flowchart showing the solution of a simple, everyday problem. The computer is to cross a one-way street, with traffic approaching from the right. Think about this problem before examining the flowchart.

QUESTION

How would you draw a flowchart which shows how to cross a one-way street with traffic approaching from the right?

ANSWER



The logical sequence of the flowchart should be clear; but the symbols may need some explanation.

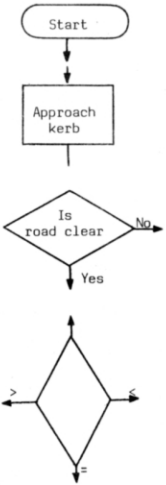
QUESTIONS

Look at this flowchart. Which shape of box is used:

1) as a *terminal* symbol (i.e. to begin and end the flowchart)?
2) as an *operation* (i.e. telling the person to do something)?
3) as a *decision-point* (i.e. when a yes-or-no question is asked)?

ANSWERS

Here are the *standard flowchart symbols* which are used in this program.



- This is the *terminal* symbol, which represents the beginning or end of a procedure or program.

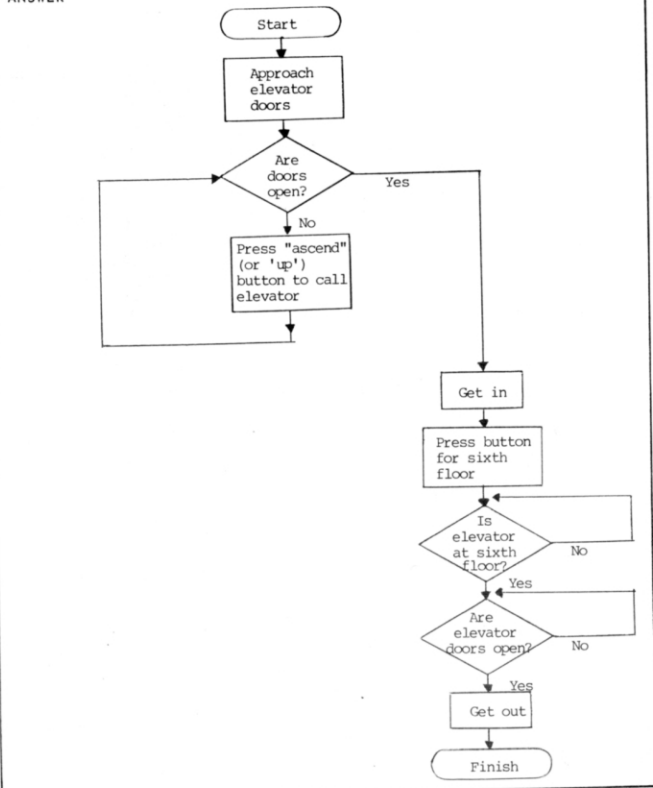
- This indicates an *operation*, which is specified in the box.

- This is a *decision-point*. A test is performed and the program flow continues on one of the outgoing paths. Note that in the road-crossing example the 'no' path loops back. Thus the user continues to ask the question 'Is the road clear?' until the road is clear. Frequently the decision point has only two outgoing paths which correspond to the true and false answers to the test. However, three or more outgoing paths are sometimes used, for example; greater than (>), equal to (=), less than (<).

PROBLEM

Using these symbols, draw a flowchart to describe the actions required to ascend in an automatic elevator from the ground floor to the sixth floor. Assume that the elevator is working properly.

ANSWER



When you have produced your flowchart, look carefully for redundancies - repeated questions and the like. When you come to code the program you will find it much simpler.

Your flowchart may look slightly different from this; compare the two and make sure that yours expresses a logical sequence which works.

In both these examples, we see that problems which are apparently easy for humans can be quite complex for computers. For example, we are crossing a road on foot. The question 'Is the road clear?' appears simple since we are aware of the criteria for judging whether or not a road is clear. The computer is not aware of these criteria, it must be painstakingly told of each decision process.

PROBLEM

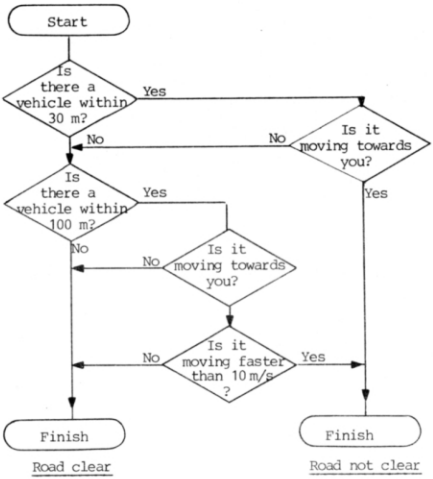
Draw a flowchart to illustrate the steps needed to decide the question 'Is the road clear?'

HINTS

1) Remember that fast-moving vehicles, even at a distance, mean that the road is not clear.

2) The 'Finish' boxes will be either "Road clear" or "Road not clear".

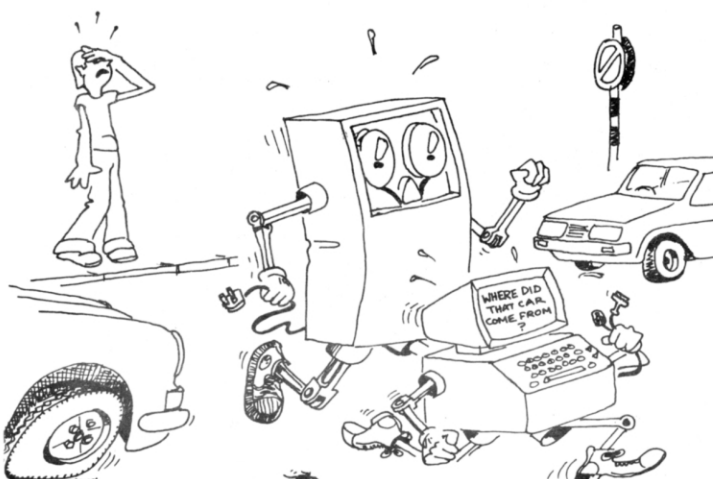
ANSWER



This problem is not easy to define. In thinking about it, and studying the above solution you will have appreciated some of the difficulties experienced in defining a problem in the pedantic detail needed by a computer.

You may have decided on a different approach to this problem. One question you might ask is "could the moving vehicle pass me within ten seconds?" If the answer is "Yes" then do not cross. You might even consider whether the vehicle is travelling on the same side of the road as you are standing. There is no "correct" answer to the drawing of this flowchart. You might even have considered a dozen other alternatives: using a subway or bridge, running across, crossing lane-by-lane, etc.

When writing a program, it is natural to write it considering only one set of circumstances. You may ignore various things which will occur when the program is used by others later. It is always a good idea to get other people to test your programs, as they will have a different outlook and a different approach to the program. Typists will often overlook their own typing errors, whilst outsiders will spot them immediately, and the same goes for programming.



LESSON 21

MORE ABOUT FLOWCHARTING

You start to draw flowcharts for BASIC

Of course computers don't cross roads as yet, although they do many other things. So what has flowcharting to do with efficient computer programming?

Let us look at a simple data-processing problem.

QUESTION

How would you arrange any three numbers into ascending order.....?

HINT

You may only compare two numbers at a time.

ANSWER

- Compare the first two numbers; if the first number is greater than the second, interchange them.
- Compare the second and third numbers; if the second is greater than the third, interchange them.
- Finally, compare the new first and second numbers; and if the first is greater than the second, interchange them.

This procedure is illustrated numerically below.

	Initial data		Here is the sequence after first step		Here is the sequence after second step		Here is the final data
A:	12	compare, interchange	7		7	compare, no change	7
B:	7		12	compare, interchange	1Ø		1Ø
C:	1Ø		1Ø		12		12

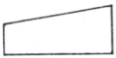
For convenience, we call the top number 'A', the next 'B', and the lowest one in the list 'C'. When we write a computer program to tackle this problem, A,B, and C will refer to the locations in the computer memory in which the numbers are stored. Thus, in this example, the largest number always ends up in location C.

PROBLEM

Using the symbols you have learnt so far, and the additional symbols introduced on the next page, draw a flowchart showing the steps which the computer follows in printing three numbers in ascending order. Assume the numbers are typed into the computer on a keyboard.

Three additional flowcharting symbols are useful for indicating input and output - the first is general, the second and third show particular input and output means.

- General input/output symbol; indicates any operation of *reading and writing*, and so may be used for any input or output function.
  Input: information for processing;
  Output: recording of processed information

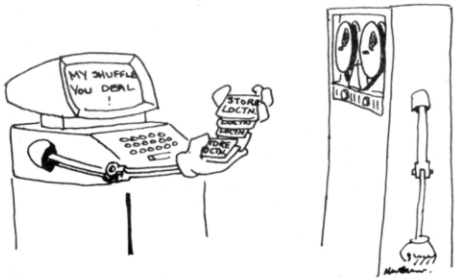


- Indicates data input from a *keyboard*

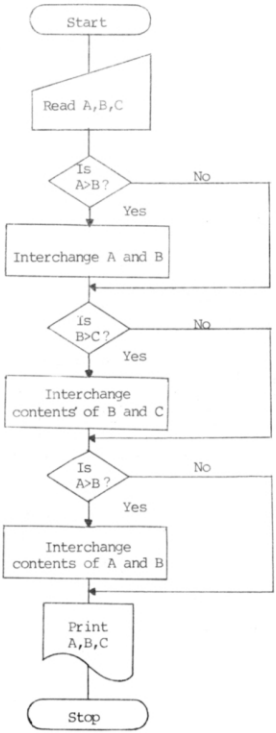
- Indicates input from or output to a *document*. An input document might be a paper tape. An example of an *input document* is typed text which is scanned by an Optical Character Reader (OCR) Machine. Document inputs are frequently used in the printing industry for computer typesetting of newspapers and magazines.
  Document output indicates output to the terminal printer or line printer (which can print a whole line at once).

Now draw the flowchart to sort three numbers, whose values are stored at A,B, and C, into ascending order. A,B, and C should be considered as storage locations. If you interchange A and B it means assigning the old contents of B to the location A, and vice versa.

This may be expressed in another way: we assign the former values of variable-name B to variable-name A and vice versa.



ANSWER



In a later lesson we will code this program in BASIC, but the flowchart above forms a complete definition of the logical structure of our program.

In Supplementary Lesson 4 on page 70 a complete set of standard flowchart symbols is given. You should read this lesson to familiarise yourself with the common flowchart symbols before going on to Lesson 21.

Problems in Part 3 require complex flowcharts. If you find flowcharts difficult, then read another text by this publisher - *The Algorithm Writer's Guide* by Doris Wheatley and Alan Unwin.

LESSON 22

USING WHAT YOU KNOW

You start to define programs, draw flowcharts, and program with BASIC - three of the five steps in programming.

Suppose you want to write a simple program to convert the following temperatures from degrees Centigrade to degrees Fahrenheit.

54°, -32°, 81°

To convert Centigrade (Celsius) to Fahrenheit, you can use the formula below:

F = 9 * C/5 + 32

where F = degrees Fahrenheit

C = degrees Centigrade

First you should write a "problem definition" and decide how you will present the results.

This is what a problem definition looks like for a program which computes the average price of a tin of beans from five different shops.

INPUT:	five prices 14p, 12p, 13p, 13p, 15p
PROCESS:	find average using formula A=(Sum of prices)/5
OUTPUT:	the average labelled 'AVERAGE PRICE OF BEANS'

Second, prepare a program flowchart.

Finally, code the program in BASIC.

PROBLEM

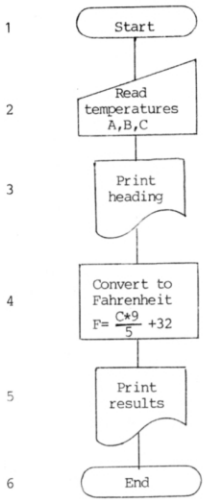
Now try the Centigrade to Fahrenheit problem for yourself with problem definition, followed by flowchart, followed by the program.

ANSWER

1) PROBLEM DEFINITION

INPUT:	three centigrade temperatures 54°, -32°, 81°
PROCESS:	conversion using formula
	F = 9*C/5 +32
OUTPUT:	tabulation of the three temperatures in degrees Centigrade and degrees Fahrenheit with each column labelled as such.

2) FLOWCHART



This is called a straight-line flowchart since there are no branches or decisions. However, the processes are not necessarily interchangeable; for example, process 3 above must precede process 5, and 2 must precede 4. Straight-line flowcharts are useful to define the sequence of processing.

3) PROGRAM

```
10 READ A,B,C
20 PRINT "CENTIGRADE","FAHRENHEIT"
30 LET X=A*9/5+32
40 LET Y=B*9/5+32
50 LET Z=C*9/5+32
60 PRINT A,X
70 PRINT B,Y
80 PRINT C,Z
90 DATA 54,-32,81
100 END

READY
RUN

CENTIGRADE     FAHRENHEIT
 54             129.2
-32            -25.6
 81             177.8

END AT 100
READY
```

This program illustrates three important elements of BASIC:

- processing using the LET statement and numbers, variables and formulae
- input using READ and DATA statements
- output using the PRINT statement for labels and results.

If you had any problems solving this problem, and particularly if you found the coding difficult, then you would find it helpful to revise Lessons 4,5,6 and 12 to 17 before you continue with Lesson 22.

This is not the way that such a program would normally be written, especially if there were more than three values to be converted. You will learn a more sophisticated method soon, using INPUT instead of READ.

LESSON 23

INTRODUCING FUNCTIONS: SIN, COS, TAN, ARCTAN, LOG, EXP

You may know how to 'solve' triangles trigonometrically and how to use natural logarithms. If you do not, this Lesson explains these functions which can be helpful in some people's work.

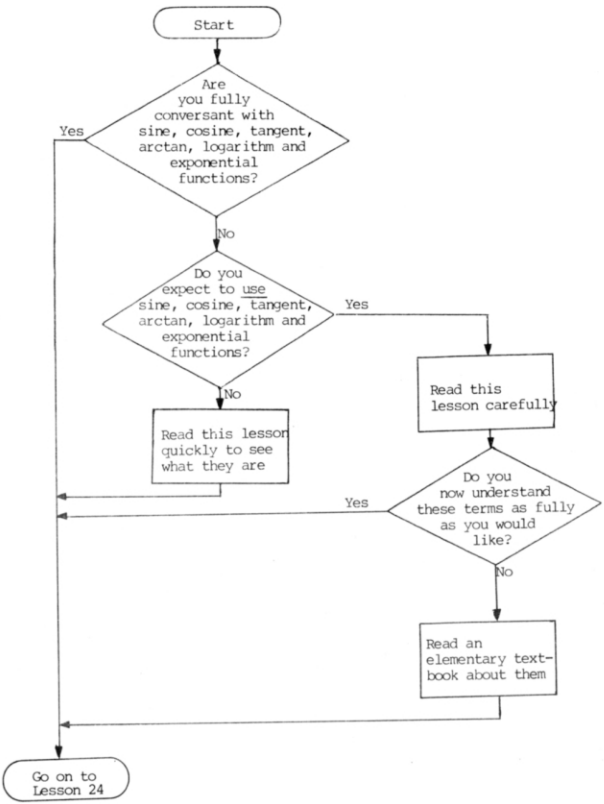
Can you understand the use of mathematical functions such as: sine, cosine, tangent, arctangent, and logarithm? Can you use arctangent and sine and cosine to solve arcsine and arcosine? If you can, then skip this Lesson.

Do you expect to use these functions? If not, read this Lesson quickly just to see what they are about.

If you expect to use these functions in your work, but you find them difficult, then read this chapter carefully. We hope you will then know all you need to know for an adequate understanding of the terms. If, however, you are still baffled, then you will need to read an elementary textbook about them - sorry, but to cover these functions in great detail we would need at least ten lessons.

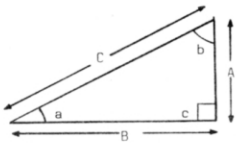
These functions are barely used in this book, and where they are used, they are explained again. So we do not pose questions on them in this Lesson.

On the next page is a flowchart to describe the advice in these paragraphs.


Start
Are you fully conversant with sine, cosine, tangent, arctan, logarithm and exponential functions?
Yes
No
Do you expect to use sine, cosine, tangent, arctan, logarithm and exponential functions?
Yes
No
Read this lesson carefully
Read this lesson quickly to see what they are
Do you now understand these terms as fully as you would like?
Yes
No
Read an elementary text-book about them
Go on to Lesson 24

SINES, COSINES AND TANGENTS

In trigonometry, sines, cosines and tangents are all defined with respect to right-angled triangles. Take the triangle shown below:



a,b,c, are angles
A,B,C, are the lengths of the sides opposite those angles.

The angle c is a right-angle, that is 90 degrees.
Sines, cosines, and tangents can be defined by the ratios of the lengths of the sides of the triangle.

For the angle a, which is in degrees:

sine; $\sin a = \frac{A}{C}$ and a = arcsine (A/C)

cosine; $\cos a = \frac{B}{C}$ a = arcosine (B/C)

tangent; $\tan a = \frac{A}{B}$ a = arctan (A/B)

You may have noticed that tan a = sin a/cos a

Since sines, cosines and tangents are ratios, they have no units such as metres, inches or kilograms. Their numerical values vary according to the angles.
Here are some examples:

Angle	sin	cos	tan
ø°	ø	1	ø
3ø°	ø.5	ø.866	ø.577
45°	ø.7ø7	ø.707	1
6ø°	ø.866	ø.5	1.732
9ø°	1	ø	infinite

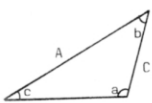
Why use the ratios sin, cos and tan?

The most straightforward use of the ratios is that they enable you to 'solve' triangles. If you know two sides of a right-angled triangle you can calculate the angles.

for example: tan a = 3/3.7 sin b = 2.6/4.1 cos c = 4.4/5.9

If you can calculate tan a, sin b, and cos c, then you can read tables of 'arctan', 'arcsin', and 'arccos' to find the values of a,b, and c.

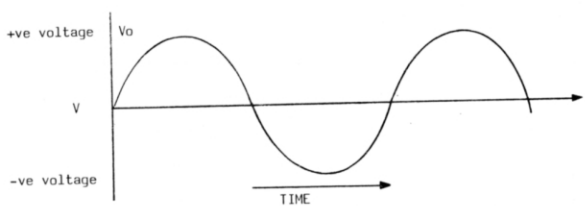
The use of sin and cos may be extended to 'solve' triangles which do not have a right-angle. Briefly there are two formulae for doing this:



$$\frac{A}{\sin a} = \frac{B}{\sin b} = \frac{C}{\sin c}$$

and $A^2 = B^2 + C^2 - 2BC \cos a$

In electrical engineering, sines and cosines are used to describe the waveform of alternating voltages and currents (AC). A sinewave varies between positive and negative:



$V = Vo \sin 2\pi ft$

where Vo = peak voltage

t = time in seconds

f = frequency in Hertz (the correct name for the unit of cycles per second). The higher the frequency the more rapidly the voltage alternates between positive and negative.

Degrees and radians

Most people are familiar with angles expressed in degrees. There are 90° in a right-angle, 180° in a semi circle, and 360° in a complete revolution. However, mathematicians, scientists and engineers often specify angles in *radians*. A radian is not a whole number of degrees, in fact, it is about 57°. There are π (or pi) radians in 180 degrees. So, because π is an irrational number, we cannot exactly specify the number of degrees in one radian.

Angles measured in degrees and radians

$45^{\circ}=\pi/4$ radians	57° app=1 radian	$90^{\circ}=\pi/2$ radians	$180^{\circ}=\pi$ radians	$270^{\circ}=3\pi/2$ radians	$360^{\circ}=2\pi$ radians

BASIC provides functions which calculate sine, cosine, and tangent from angles expressed in radians. It also provides a special function called arctangent to calculate the angle, in radians, from a known value of tangent.

For example:

if tan A = X	if sin B = Y	if cos C= Z
then arctangent X = A	then arcsin Y = B	then arcosine Z = C

Thus the arctangent of 1 is 45°, of 0 is 0° and so on. Arctangent is shortened to ATN.

We can calculate the arcsine, and arcosine from the arctangent using known mathematical relationships (which you need not remember).

B = arcsine Y = ATN $\left[\frac{Y}{\sqrt{1 - Y^2}} \right]$ where sin A = X

C = arcosine Z = ATN $\left[\frac{\sqrt{1 - Z^2}}{Z} \right]$ where cos A = Y

Remember in BASIC the result of the arctangent operation is in radians, *not degrees*.

Logarithms

In the days before calculators and computers, log-tables were taught to generations of school children. The principle behind logarithms is delightfully simple, and has already been introduced in this text, back in Part 1 in the Lesson on 'raising to a power'.

You may remember that numbers could be multiplied or divided simply by adding or subtracting the indices. Thus:

$$2^4 * 2^2 = 2^{(4+2)} = 2^6 = 64$$

$$2^8/_{2^3} = 2^{(8-3)} = 2^5 = 32$$

For this simple addition or subtraction to work, both numbers must be powers of the same number. It would not work if a different *base number* were used. Thus:

$2^2 * 3^2 = ?$ - there is no neat short cut to the answer to this

In fact the index (the power) in the first example is the logarithm of the number to the *base 2*, written $\log_2$.

Hence; to compute 256/8 we can say:

$2^3 = 8$ $\qquad$ $2^8 = 256$

$\log_2 8 = 3$ $\qquad$ $\log_2 256 = 8$

And: $\log_2 256 - \log_2 8 = 8 - 3$

$= 5$

But $\log_2 32 = 5$ (because $2^5 = 32$)

Therefore 256/8 = 32

In the last-but-one step, we have taken the antilog of 5, that is, we have simply said what is $2^?$ The answer is 32.

Log tables are usually to the base 10, so that all numbers are expressed as powers of 10. However, mathematicians use a type of logarithm called a *natural logarithm** which is to the base e. e is a number, which like π is irrational but has special properties in pure and applied mathematics. e is approximately 2.7182818284590452 .....

You need only remember two things about natural logarithms:

- firstly, if a = lnx where ln stands for natural logarithm, then

$e^a = x$. x is sometimes referred to as the exponential of a.

[* The graph of $y = e^x$ has a slope (y') of 1 when y = 1, 2 when y = 2, 200 when y = 200 - the value of e is defined from a curve having such 'exponential' growth, or, in terms of the calculus:

If $y = e^x$ then $\frac{dy}{dx} = e^x$.]

- secondly, you can convert natural logarithms to logarithms of any other base by a simple relationship.

For base 10: $\log_{10} x = \ln x * \ln 10$

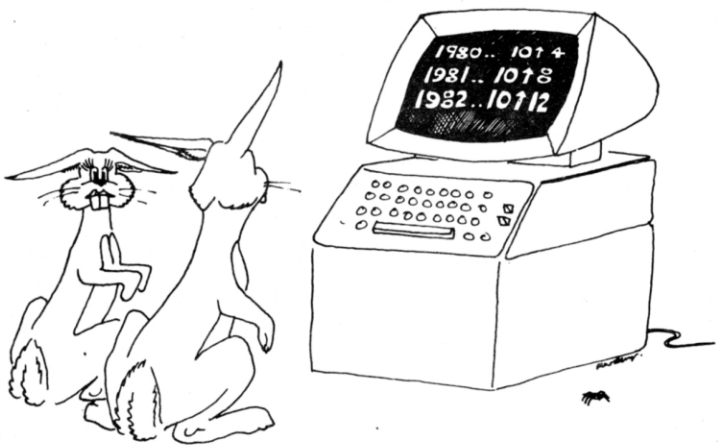
For base 2 : $\log_2 x = \ln x * \ln 2$

For base n : $\log_n x = \ln x * \ln n$, etc.

Why use log and exp?

You have probably heard jokes about breeding rabbits: 'We start with two, in six months there will be 100, in a year 10,000, and in 18 months a million rabbits - we're bound to be rich'. This type of growth is called exponential. e^m when m is the number of months, will give rise to a series of formulae like the growth of rabbits.

Other things fall logarithmically. So, if you need to do mathematical problems which have *mushrooming* or declining numbers you are likely to need exponential or log functions.



LESSON 24

FUNCTIONS IN BASIC

You learn to express the functions from Lesson 23 and some others in BASIC programs.

In many scientific problems, the algebraic expression to be evaluated contains one or more standard mathematical *functions* such as sine, cosine, logarithm and so on. An important example is the sine function which appears in a LET statement as

```
LET Y = SIN (X)
```

The function sine is defined by the three letters SIN which are recognised by the computer. These then generate the set of machine instructions required to evaluate sine.

The argument of a function can be a number, a variable, or a formula depending on the requirements of the program. Thus, the following statements are valid:

```
LET Y = SIN(2.4)
LET Y = SIN(2*S-A)
LET Y = SIN(2*SIN(X))
```

In BASIC, the argument of the function SIN must be expressed in radians.

Remember: there are π radians in 18Ø°; π = 3.14159265 (If you cannot remember π, there is a way to help you manage without it at the end of this Lesson.)

BASIC provides ten standard mathematical functions, and these are listed below:

FUNCTION	DESCRIPTION
SIN(X)	sine x, where x is measured in radians
COS(X)	cosine x, where x is measured in radians
TAN(X)	tangent x, where x is measured in radians
ATN(X)	arctangent x, where the value of ATN(X) is an angle, say Θ, expressed in radians. Where $-\pi/2<=\Theta<=\pi/2$
EXP(X)	exponential function, e^x
LOG(X)	natural logarithm of x; the sign of x is ignored
ABS(X)	absolute value of x (the value of x ignoring the sign)
SQR(X)	square root of x; the sign of x is ignored
INT(X)	integer part of x; the value of INT(X) is the largest integer not greater than x. For example, INT(5.9) = 5
SGN(X)	sign of x; +1 for x positive, Ø for x = Ø, -1 for x negativ

PROBLEMS

Using these standard functions, write BASIC statements to evaluate the following expressions:

1) $Y = \sqrt{x.\sin 2x}$

2) $z = e^{k} + 2k^{2}$

3) p is the cosine of the variable q which is in units of degrees.

SOLUTIONS

1) LET Y = SQR(X*SIN(2*X))

2) LET Z = EXP(K)+2*K+2

3) LET P = COS(Q*3.1415962/18Ø)

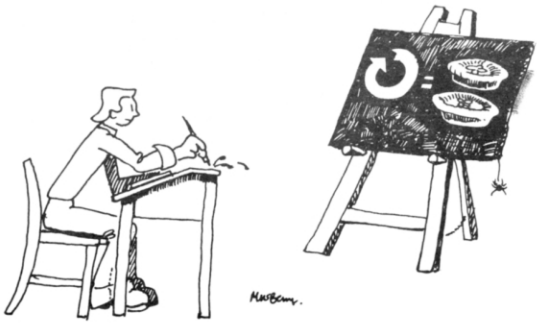
If you cannot remember the value of π you can use your knowledge of tangents and arctangents. The tangent of 45° is 1. But 45° is one quarter of 18Ø° and 18Ø° is π radians. So 45° is $\frac{\pi}{4}$ radians. So TAN(π/4)=1 or ATN(1)=$\frac{\pi}{4}$ and π=4*ATN(1).

So you could write for 3)

LET P = COS(Q*4*ATN(1)/18Ø)

or

LET P=COS(Q*ATN(1)/45).



LESSON 25

MORE ABOUT READ AND DATA STATEMENTS - THE DATA BLOCK

How the BASIC program treats READ, and DATA statements when they appear in several lines in the program.

Now let us turn to the problem of feeding data into the computer.
The following program illustrates the use of READ and DATA statements:

```
1Ø READ A,B,C
2Ø LET D=A*B*C
3Ø READ N
4Ø LET F=D/N
5Ø PRINT F
6Ø DATA 1,4,-6.2,7.235E4
7Ø END
```

Line 1Ø instructs the computer to read the first three numbers in the DATA statement and assign their values to the variable-names A,B, and C. Thus the computer sets the variable-names A=1, B=4 and C=6.2. In line 2Ø the computer multiplies the values assigned to A,B, and C, and calls the resulting value D. Line 3Ø instructs the computer to read the next number in the DATA statement and assign its value to the variable-name N, so that N=7.235E4. (If you are not sure about E-notation, look again at Lesson 11.) The result, F, is the output of the program.

It is important to understand the way in which the computer relates variables in the READ statements to numbers in the data statements.

Any number of DATA statements are permitted in a BASIC program, and they may occur anywhere in the program. The computer scans the complete program and arranges the numbers in each data statement into a *DATA block* in their order of occurence. The four DATA statements shown on the left below are arranged into the DATA block shown on the right. A DATA block is simply a grouping of consecutive numbers which appear in DATA statements.

DATA statements

```
14Ø DATA 4.2,3.1417,5.123E4
15Ø DATA -6.2,457,2.282
16Ø DATA -1,4.4
23Ø DATA 25,Ø.ØØ2,-3Ø
```

DATA block

```
4.2
3.1417
5.1234E4
-6.2
457
2.282
-1
4.4
25
Ø.ØØ2
-3Ø
```

PROBLEM

Compile, by hand, a DATA block from the following BASIC program, and assign values to the variables called up in the READ statements.

```
10 READ X1,X2,Y
20 DATA 2.4,-4.7,6.2E10
30 LET Z1=(X1+X2)*Y
40 READ N
50 LET Z2=N-Z1
60 READ K
70 LET Z2=Z2/K
80 PRINT Z2
90 DATA 4.6,7.2E-4
100 END
```

ANSWER

```
X1 = 2.4
X2 = -4.7
Y  = 6.2E10
N  = 4.6
K  = 7.2E-4
```

You will see, from the solution to the problem above, that when the computer encounters a READ statement, it assigns the next available numbers from the DATA block to the variables in the READ statement. The DATA block is stored as a list of numbers within the computer memory, and the computer reads numbers from this list, remembering after each READ statement the location of the next available number. We can imagine the computer moving a pointer down the DATA block, thus remembering how many of the numbers have so far been assigned to variables.

PROBLEM

Here is another BASIC program. Compile a DATA block, and assign values to each of the variables in the READ statements, following at each statement the movement of the pointer

```
10 READ B1,B2,B3
20 READ A,C,D
30 LET X=A*B1+C*B2+D*B3
40 READ N
50 LET Y=X*N
60 READ A,C,D,N
70 LET Z=A+C+D+N
80 LET W=Y/Z
90 PRINT W
100 DATA 0.247,0.341,0.565
110 DATA 3,2,5,10
120 DATA 16,12,2,18
130 END
```

HINT

Remember that a variable-name may have several different values assigned to it during the running of a program.

ANSWER

```
Ø.247
Ø.341
Ø.565
3          ← position of pointer after assigning a value to variable-name B3
2
5
1Ø         ← position of pointer after assigning a value to variable-name D
16         ← position of pointer after assigning a value to variable-name N
12
2
18         ← position of pointer after assigning new values to variable-names
               A,C,D, and N
1Ø READ B1,B2,B3     2Ø READ A,C,D      4Ø READ N        6Ø READ A,C,D,N
B1 =Ø.247            A =3               N =1Ø            A =16
B2 =Ø.341            C =2                                C =12
B3 =Ø.565            D =5                                D =2
                                                         N =18
```

You may have found this program confusing, as A,C,D and N all come up twice. Remember, though, that A,C,D and N are not values - they are variable-names, and any value we like can be assigned to them. If they could not change their values in the middle of a program, a line such as LET N = N+1 would not be valid; but we know it is. When the pointer is faced with the same variable-name twice, it merely moves down the data-block, assigning new values to this name.

It is good practice to collect the DATA statements and group them at the end of the program just before the end statement. This is convenient if the DATA statements need to be modified or changed - if the same program is used with different data-sets, especially in off-line working, when the program may well be in the form of punched tape or a deck of punched cards. It also allows the logical flow of the READ...DATA process to be checked thoroughly.

In some computers, attempts to execute a READ statement when the DATA-block is exhausted (i.e. the pointer has reached the bottom of the block) will generate a message such as OUT OF DATA IN XXXX where XXXX is the line number of the READ statement. This is called a *diagnostic* - it diagnoses the problem to assist the programmer in debugging the program. No diagnostic will be printed if the number of DATA values exceeds the number of READ statements, i.e. there will be some pieces of DATA unused. For this reason, it is important when writing the program to ensure that there is one - not more and not less - DATA number for each variable-name in the READ statements.

LESSON 26

ANOTHER PROGRAM EXAMPLE - WEIGHT CONVERSION

You learn to use the INTeger function in a simple example.

Here is a fairly difficult problem. Ensure that you fully understand the problem before drawing the flowchart and coding the solution. We have given you a few hints to help you along.

PROBLEM

Write a problem definition, flowchart, and a program to add the weight of four items which weigh

4.825 kg 2.212 kg 5.624 kg 4.292 kg

The total weight is required in both kilograms, and the nearest equivalent in whole numbers of pounds and ounces.

W = K*2.2Ø46223
when W = weight in pounds
K = weight in kilograms

HINT

The key difficulty in this problem is converting the total weight in kilograms to the equivalent *whole number* of pounds and ounces, *ignoring* fractions of an ounce. Suppose the total weight is K kilograms. The exact total weight in pounds is W = K*2.2Ø46223, and the whole number of pounds in this total is the INTeger part of W. We can compute this using the INTeger function INT(W). The remaining fractional part can now be converted to ounces, and the INTeger part similarly computed. So 6 lb 3.9 oz becomes 6 lb 3 oz.

ANSWER

1) Problem-definition

INPUT: Four weights, 4.825 kg, 2.212 kg, 5.624 kg, 4.292 kg

PROCESS: Add the four weights

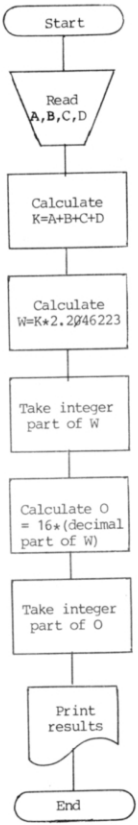
Convert the total to pounds and ounces, ignoring fractions of an ounce

using: W = K*2.2Ø46223 for converting kilograms to pounds and: 16 ounces = 1 lb.

OUTPUT: Print results

TOTAL WEIGHT IS K KILOGRAMS
WHICH IS APPROXIMATELY W POUNDS O OUNCES
where K = weight in kg
and W lb O oz is the weight in pounds and ounces.

2) Flowchart


Start
Read A,B,C,D
Calculate K=A+B+C+D
Calculate W=K*2.2Ø46223
Take integer part of W
Calculate O = 16*(decimal part of W)
Take integer part of O
Print results
End

3) Program

```
1Ø READ A,B,C,D
2Ø LET K=A+B+C+D
3Ø LET W=K*2.2Ø46223
4Ø LET P=INT(W)
5Ø LET O=INT((W-P)*16)
6Ø PRINT "TOTAL WEIGHT IS",K,"KILOGRAMS"
7Ø PRINT "OR",P,"POUNDS",O,"OUNCES"
8Ø DATA 4.825,2.212,5.624,4.292
9Ø END

READY
RUN

TOTAL WEIGHT IS              16.953        KILOGRAMS
OR              37           POUNDS         5              OUNCES

END AT 9Ø
READY
```

Note that once the program has been adequately defined, the coding is straightforward. In this program, the INT function is being used to *truncate* the total weight to an integral number of pounds and ounces: the result may be up to one ounce in error.

What happens is as follows:

2Ø K = 4.825 + 2.212 + 5.624 + 4.292 = 16.953 kilograms

3Ø W = 16.953 x 2.2Ø46223 = 37.374961 pounds

4Ø P(Whole number of pounds) = 37 [Fraction of pounds = Ø.374961]

5Ø O(Whole number of ounces) = 5 [Fraction of ounces = Ø.999376]

This shows the limitation of the computer at 'rounding off' by this means. We return to a more accurate method later, which would interpret 5.999, or even 5.51, ounces as 6 ounces.

PROBLEM

Now modify this solution, to provide an output which lists each item in kilograms, as well as the total weight in both metric and imperial units.

HINT

The OUTPUT should now be:

```
ITEM          WEIGHT IN KILOGRAMS
 A             4.825
 B             2.212
 C             5.624
 D             4.292

TOTAL WEIGHT IS              16.953        KILOGRAMS
OR              37           POUNDS         5              OUNCES
```

SOLUTION

```
10 READ A,B,C,D
12 PRINT "ITEM","WEIGHT IN KILOGRAMS"
14 PRINT " A",A
16 PRINT " B",B
17 PRINT " C",C
18 PRINT " D",D
19 PRINT
20 LET K=A+B+C+D
30 LET W=K*2.2046223
40 LET P=INT(W)
50 LET O=INT((W-P)*16)
60 PRINT "TOTAL WEIGHT IS",K,"KILOGRAMS"
70 PRINT "OR",P,"POUNDS",O,"OUNCES"
80 DATA 4.825,2.212,5.624,4.292
90 END

READY
RUN

ITEM            WEIGHT IN KILOGRAMS
 A               4.825
 B               2.212
 C               5.624
 D               4.292

TOTAL WEIGHT IS              16.953        KILOGRAMS
OR              37          POUNDS          5              OUNCES

END AT 90
READY
```

Note that the modification may be accomplished by inserting lines in the gaps left in the original line numbering. Spacing line numbers by 10 in the original program allows up to 9 statements to be inserted between processes without modification of the original program.

LESSON 27

THE REM STATEMENT FOR DOCUMENTATION OF PROGRAM

The fifth step in programming is to be sure both that other people can understand your program, and that it will be clear to you years from now, when you look at it again. REMarks in the program explain what it is all about in plain English.

In Lesson 19 we remarked upon the importance of program documentation in the programming process.

There are two principal items in the package of program documentation - the flowchart and the listing of the program. To be easily understood, the program must include comments describing the important sections of this program. Comments are added to a BASIC program using a REM statement. (REM comes from 'remark'.)

The solution to the problem in the last Lesson is repeated below, with the program properly documented.

```
10 REM PROGRAM TO CALCULATE THE TOTAL WEIGHT OF
20 REM FOUR ITEMS AND PRINT TOTAL IN KILOGRAMS AND
30 REM NEAREST EQUIVALENT IN POUNDS AND OUNCES
40 REM     I WILLIAMSON      31-MAY-99
50 READ A,B,C,D
60 LET K=A+B+C+D
70 LET W=K*2.2046223
80 LET P=INT(W)
90 LET O=INT((W-P)*16)
100 REM OUNCES ARE TRUNCATED TO NEAREST WHOLE NUMBER
110 PRINT "TOTAL WEIGHT IS",K,"KILOGRAMS"
120 PRINT "WHICH IS APPROXIMATELY",P,"POUNDS",O,"OUNCES"
130 DATA 4.825,2.212,5.624,4.292
140 END
```

At a minimum, a program should include the following comments:

- a brief description of its function - lines 10, 20 and 30
- the author - line 40
- the date - line 40
- a note of any special or difficult attributes of the program, as in line 100.

REM statements are very useful especially if you come back to a program after a few months. The one in line 100 could save you a lot of trouble in these circumstances. Put as many REM statements as you think you will need in your programs. It is better to be generous with REM statements than to be mean. They may make your program longer, but they can save lots of time later.

LESSON 28

THE INPUT STATEMENT

The INPUT statement allows you to write programs in which the computer asks you for information - data, or advice on the course the program should take. The information can be in the form of numbers or letters (alphanumeric). In this chapter we will be dealing solely with numbers, and later in the course we will deal with letters. The INPUT statement is far more powerful for on-line work than READ and DATA.

So far, we have used the READ and DATA statements to enter all data into the computer. For programs where we are entering large quantities of data, or possibly defining constants which are rarely changed, the READ and DATA statements are fine.

But BASIC offers a much more powerful statement, the INPUT statement, which allows us to write programs which make the computer come back and ask questions at the terminal, whilst the program is running. Using the INPUT statement, we can feed data into the computer *after* we have typed RUN, and started to execute the program.

You will remember that in Part 1, we described two classes of computer service available to the majority of users. We called these on-line and off-line processing. Off-line processing, often in the form of a batch-processing service, requires that the program and all data are prepared prior to the program execution. For batch-processing, you cannot use the INPUT statement, since nobody is present to provide the data. So you must therefore specify all your data using READ and DATA statements.

The difference between on-line and off-line processing is analogous to that between a Polaroid camera and a conventional one. With a Polaroid camera, if you don't like the picture, you can rearrange it and try again. With a conventional camera, you have to wait for the film to be processed, before you can judge your success or otherwise.

Programmers who can interact with their computers are fortunate. They have instant editing and debugging facilities ("debugging" a program is eliminating the bugs from it - that is, correcting all the mistakes in it). They can also modify the computer's action, by making decisions based on intermediate results which it presents to them. In BASIC, such interaction is achieved through the INPUT statement.

The INPUT statement is directly analogous to the READ statement, except that the numerical values of the variables called up by the INPUT statement must be typed in at the computer terminal, and cannot be read from a DATA statement forming part of the program. The simple program on the next page illustrates the use of the INPUT statement in an interactive BASIC program i.e. one which uses the INPUT statement.

```
1Ø REM CALCULATION OF VOLUME OF CYLINDER
2Ø REM I.WILLIAMSON      31-MAY-99
3Ø PRINT "INPUT RADIUS OF CYLINDER"
4Ø INPUT R
5Ø PRINT "INPUT HEIGHT OF CYLINDER"
6Ø INPUT H
7Ø LET V=3.14159*R*R*H
8Ø PRINT "VOLUME IS",V
9Ø END

READY
RUN
INPUT RADIUS OF CYLINDER
 ? 1     RET
INPUT HEIGHT OF CYLINDER
 ? 1     RET
VOLUME IS        3.14159

END AT 9Ø
READY
```

This line 4Ø causes the computer to ask for data at the terminal by printing a question mark. The computer will then wait until you type in the data and return control to the computer by pressing the carriage-return key, (RET)

In case you are confused as to who types what in this print-out - we have underlined the data typed by the computer, as opposed to yourself.

Note: We could, of course, use a more exact approximation for π. For most programs 3.14159 is as accurate as we need.

Many variables can be entered with a single INPUT statement, as for the combination of READ and DATA statements.

In response to the statement:

```
1Ø INPUT A,B,C
```

the computer will type a single question mark, and you enter the three numbers separated by commas.

```
? 1,2,3
```

Here is a program designed to simply PRINT three numbers; again we underline what the computer types.

```
10 PRINT "INPUT THREE NUMBERS"
20 INPUT A,B,C
30 PRINT A,B,C,
40 END
READY

RUN

INPUT THREE NUMBERS
? 1,2,3,4,5          (RET)

1             2             3

END AT 40
READY
```

the user has provided too much data but the computer ignores the additional input. (N.B. Some dialects of BASIC will signal an error if you input too much data.)

```
RUN

INPUT THREE NUMBERS
? 1                  (RET)
?? 2                 (RET)
?? 3                 (RET)

1             2             3

END AT 40
READY
```

the computer is insistent in asking for data until three numbers have been typed in.

```
RUN

INPUT THREE NUMBERS
? 1,2,A              (RET)
BAD DATA INTO LINE 20... DATA A
? 3                  (RET)
?

1             2             3

END AT 40
READY
```

'A' is not a number - the computer responds with a diagnostic error message and asks for new data.

PROBLEM

Using the INPUT statement, write a program to convert metres to the equivalent feet and inches, ignoring fractions of an inch. You may write your own problem definition using a flowchart if you wish.

25.4 mm = 1 inch

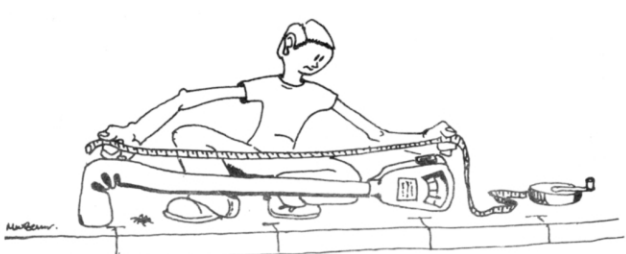
HINT

The problem is similar to the conversion of kilograms to pounds and ounces in Lesson 26.

```
ANSWER

10 PRINT "CONVERSION OF METRES TO FEET AND INCHES"
20 REM   I.WILLIAMSON          31-MAY-99
30 PRINT "LENGTH IN METRES"
40 INPUT L
50 LET D=L*1000/(25.4*12)
60 LET F=INT (D)
70 LET I=INT ((D-F)*12)
80 PRINT "CONVERTS TO",F,"FEET",I,"INCHES"
90 END

READY
RUN
CONVERSION OF METRES TO FEET AND INCHES
LENGTH IN METRES
 ? 2.47
CONVERTS TO     8              FEET              1              INCHES

END AT 90
READY

RUN
CONVERSION OF METRES TO FEET AND INCHES
LENGTH IN METRES
 ? 3.12
CONVERTS TO     10             FEET              2              INCHES

END AT 90
READY
```

NOTE: The use of PRINT to print a title at the start of each run. This helps any other people who might wish to use your program, and saves you having to include a title in a special REM statement.

The INPUT statement adds another dimension to the print-out, since the computer is now asking *you* questions. To avoid confusion in interpreting the results of your program, you should generally try to group together the questions and answers associated with the INPUT statement, and to separate these from the results of the computation statement.



LESSON 29

MORE ABOUT PRINT

You can already PRINT in 5 zones. You now learn to PRINT more closely than that, on a new line, leaving a line, or leaving gaps in the line.

By now, you will have realised that the print-out of your results is an essential part of the program - and much can be done to improve the clarity of the results by proper and careful formatting of the print-out.

In this lesson we revise the aspects of the PRINT statement which were introduced in Lesson 14, and go on to introduce some additional features of PRINT which you will find useful.

Below are two examples of the type of PRINT statement used so far.

1)

```
12Ø PRINT A,B,C,D,E
```

2)

```
14Ø PRINT "A=",A,"B=",B
```

To interpret a multiple-item PRINT statement, the computer divides the sweep of the teletypewriter carriage into five *zones*, of 14 or 15 spaces each. (The precise width of a zone varies in different computer systems; in our examples the computer uses a 14-character zone.) Each item in the PRINT statement is placed within a new zone on the printed line.

QUESTION

What print-out would the above statements produce? (in 12Ø A=123.4, B=-4. C=.ØØØØ232, D=1ØØØØØ, E=-2.456; in 14Ø A=.Ø12, B=45.)

HINT

Remember our computer uses 5 zones,each 14-characters wide.

ANSWER

1)

```
 123.4          -4             2.32E-Ø5       1ØØØØØ         -2.456
```

2)

```
A=              .Ø12           B=             45
```

There is no limit to the number of items in the PRINT statement. The excess items will be printed on a new line. This is shown in the following example.

```
15Ø PRINT "A=",12,"B=",42,"SUM=",A+B

A=            12            B=            42            SUM=
 54
```

(It is obviously necessary to remember this characteristic when arranging a format for one's results.)

The five zones are useful when we are printing data in columns, like the tabulated data in a ready reckoner or mathematical tables. Sometimes, the columns are a nuisance and we want to PRINT more closely. Semicolons can be used to pack data more closely.

Thus

```
6Ø PRINT A,B,C,D
```

will generate a print-out

```
 4            8            22.45        -13.4
```

whereas

```
6Ø PRINT A;B;C;D                Note the use of semicolons here
```

will produce

```
 4  8  22.45 -13.4
```

When numbers are separated by semicolons in the PRINT statement, a single space is inserted in the print-out after each number. If you look closely at the print-out above you will see that there are extra single spaces preceding 4, 8 and 22.4. This is to enable the sign (positive + or negative -) to be printed before the number - but the + sign is not printed in BASIC so a blank space is left. This space preceding the number is occupied by the - sign for the number 13.4.

Semicolons may also be used between labels, and between labels and numbers, but the additional space is not inserted after labels. Here are some examples:

1)

```
120 PRINT "A=";A,"B=";B
A= 2              B= 4
```

- *note the use of commas and semicolons here to give a very pleasing layout*

2)

```
140 PRINT "A=";
150 PRINT "ZERO"
A=ZERO
```

- *note that this semicolon ensures that the next item that is printed, even though it occurs in another line in the program, will be on the same line.*

3)

```
160 PRINT"A=";A;"METRES"
A= 21.2 METRES
```

In standard BASIC, the semicolon can be omitted between labels and variable-names, as the computer will assume a semicolon unless told otherwise. However, some BASIC dialects do not allow you to omit the semicolon. Here is the first of the examples above with the semicolon omitted. The print-out is identical

```
160 PRINT "A="A,"B="B
A= 2              B= 4
```

What else can we do with the PRINT statement?

We can insert blank lines into the print-out using the simplest of PRINT statements, e.g.

```
40 PRINT
```

inserts a single blank line in the print-out.

Alternatively, as we have seen, we can prevent a new line being started by terminating the PRINT statement with a comma or a semicolon. This is particularly useful in conjunction with the INPUT statement. Here is an example:

```
40 PRINT "HOW LONG IS A PIECE OF STRING";
50 INPUT L

HOW LONG IS A PIECE OF STRING? 22.4
```

Are you confused? The computer types the words: HOW LONG IS A PIECE OF STRING, and the teletypewriter carriage stops after the G of STRING. The next statement is an INPUT, so the computer types ? and waits for data. You type 22.4 and press carriage return - the teletypewriter carriage is returned to the margin and a new line is started.

Of course you could have carried on printing

```
40 PRINT "HOW LONG IS A PIECE OF STRING ";
50 PRINT "RUBBISH"
60 PRINT "OK"

HOW LONG IS A PIECE OF STRING RUBBISH
OK
```

Note the semicolon prevents a new line and also the space which is needed after STRING or we get STRINGRUBBISH

We can also control the horizontal spacing of the results by the free use of commas. In the PRINT statement which follows, each comma advances the teletypewriter carriage to a new zone - generating two blank zones in the centre of the print-out

```
100 PRINT "A=";A,"B=";B,,"A+B=";A+B
```

Note the two commas after B

```
A= 4            B= 8                    A+B= 12
```

Another example is:

```
100 PRINT,,"TOTAL";X1
```

Note the two commas after PRINT

```
                TOTAL 128.92
```

One thing that you cannot yet do in BASIC is to align columns, except from the left, e.g.

Thus	would appear as
Ø.ØØ3	.ØØ3
-3.6	-3.6
142.1916	142.1916
24.Ø	24.Ø

This may all seem a little confusing, so here is a brief summary of all that you need to remember.

a) There are five zones in the print-out, and a comma in the PRINT statement takes you to the next available zone.
b) If a PRINT statement is terminated with a comma or semicolon, then the next PRINT will be on the same line. Otherwise the next PRINT is on a new line.
c) Labels and data can be closely packed using semicolons in the PRINT statement. A space is inserted after numbers, but not after labels.
d) A space is available before each number for its sign. With positive numbers the + is not printed, it appears that such numbers have an extra space before them.

And that's all there is to it!

PROBLEM

Here is a print-out - reconstruct the program, showing the PRINT statements which produced it:

```
RUN

ITEM            WEIGHT
                 KGMS
 1               4.2
 2               2.1
 3               5.4

                          TOTAL WEIGHT IS 11.7 KILOGRAMS

                          AVERAGE WEIGHT IS 3.9 KILOGRAMS

END AT 14Ø
READY
```

ANSWER

```
10 PRINT "ITEMS","WEIGHT"
20 PRINT, "KGMS"
30 READ A,B,C
40 PRINT "1",A
50 PRINT "2",B
60 PRINT "3",C
70 LET T=A+B+C
80 PRINT
90 PRINT
100 PRINT,,"TOTAL WEIGHT IS";T;"KILOGRAMS" - these semicolons may
110 PRINT                                    be omitted
120 PRINT,,"AVERAGE WEIGHT IS";T/3;"KILOGRAMS"
130 DATA 4.2,2.1,5.4
140 END
```

At the end of Lesson 28, you wrote a program to convert metres to feet and inches. The print-out was not very neat.

QUESTION

Try again, using the features of PRINT introduced in this lesson.

ANSWER

Here is our solution:

```
10 PRINT "CONVERSION OF METRES TO FEET AND INCHES"
20 REM      I.WILLIAMSON      31-MAY-99
30 PRINT
40 PRINT "LENGTH IN METRES";
50 INPUT L
60 LET D=L*1000/(25.4*12)
70 LET F=INT(D)
80 LET I=INT((D-F)*12)
90 PRINT "CONVERTS TO";F;"FEET,";I;"INCHES"
100 PRINT
110 END

READY
RUN
CONVERSION OF METRES TO FEET AND INCHES

LENGTH IN METRES ? 4.5
CONVERTS TO 14 FEET, 9 INCHES

END AT 110
READY
```

Your answer may have been different to ours. That is not important. What matters is that it should have looked better than the previous print-out.

LESSON 30

IF.....THEN.....AND GO TO

If the IF....THEN statement is used, then a program will jump to a non-consecutive line-number when a certain condition is met. GO TO always makes it jump to a non-consecutive number.

In all the program examples so far in this course, we have thought out the problem in advance, and the computer has simply obeyed each statement in turn, stepping sequentially to the end of the program.

Two statements in BASIC allow the programmer to control the sequence of the computer's actions dependent upon the data actually being processed. These are the IF...THEN statement, and the GO TO statement; another way of describing these statements is as *conditional* and *unconditional* jump statements.

An example of an *unconditional jump statement* in BASIC is:

```
15Ø GO TO 4Ø
```

This GO TO statement causes the computer to jump to the statement at line number 4Ø. Thus the normal sequential process of the computer is interrupted, and the program is returned to line number 4Ø. GO TO statements are called "unconditional" jump statements since the computer must always obey them, regardless of the data being processed. GO TO statements are often used to force the computer to re-execute an earlier part of the program as in this example, or to skip round or avoid a section of the program.

Suppose you wanted to convert a long list of measurements from metres to feet and inches. You could modify the program given at the end of Lesson 29 as shown below.

```
1Ø PRINT "CONVERSION OF METRES TO FEET AND INCHES"
2Ø REM       I.WILLIAMSON     31-MAY-99
25 REM THIS PROGRAM CONTAINS A SERIOUS ERROR
3Ø PRINT
4Ø PRINT "LENGTH IN METRES";
5Ø INPUT L
6Ø LET D=L*1ØØØ/(25.4*12)
7Ø LET F=INT(D)
8Ø LET I=INT((D-F)*12)
9Ø PRINT "CONVERTS TO";F;"FEET,";I;"INCHES"
1ØØ PRINT
11Ø GO TO 3Ø  - this sends the program back to line 3Ø where it starts again
12Ø END
```

QUESTION

What is wrong with this program?

HINT

Try to see how the program ends.

ANSWER

Although the program has an END statement, the computer never reaches it; at line 11Ø it is always directed back to line 3Ø. The computer will never stop trying to run the program - and you will have to resort to one of the stop procedures described in Part 1, Supplementary Lesson 2.

In fact, to stop the program we need a "conditional" jump statement - where the action of the computer is dependent upon the data being processed. In the program above, we could use the data provided in response to the INPUT statement to determine whether the program should be stopped.

QUESTION

What input data could be provided to stop the computer?

HINT

Remember that we have to input numbers in response to the INPUT statement, or the computer will print an error message and ask for repeat data. So look for a number which you will *never* need to convert from metres to feet and inches.

ANSWER

Since the program is working with distance measurements, then either zero, or possibly a negative number could be used to terminate the program.

On the next page is the conversion program modified so that it will function correctly.

```
1Ø PRINT "CONVERSION OF METRES TO FEET"
15 REM     I.WILLIAMSON          31-MAY-99
2Ø PRINT
3Ø PRINT "INPUT ZERO TO TERMINATE PROGRAM RUN"
4Ø PRINT
5Ø PRINT "LENGTH IN METRES";
6Ø INPUT L
7Ø IF L=Ø THEN 13Ø  - Note that this takes us to the end IF we input zero.
8Ø LET D=L*1ØØØ/(25.4*12)
9Ø LET F=INT(D)
1ØØ LET I=INT((D-F)*12)
11Ø PRINT "CONVERTS TO";F;"FEET,";I;"INCHES"
12Ø GO TO 4Ø
13Ø END

READY
RUN
CONVERSION OF METRES TO FEET AND INCHES

INPUT ZERO TO TERMINATE PROGRAM RUN

LENGTH IN METRES ? 2.47
CONVERTS TO 8 FEET, 1 INCHES

LENGTH IN METRES ? 3.12
CONVERTS TO 1Ø FEET, 2 INCHES

LENGTH IN METRES ? 4.2E3
CONVERTS TO 13779 FEET, 6 INCHES

LENGTH IN METRES ? Ø

END AT 13Ø
READY
```

A *conditional jump statement* in BASIC is:

```
2Ø IF X=Ø.2 THEN 16Ø
```

which causes the computer to jump to line 16Ø if X equals Ø.2. If X is not equal to Ø.2, the next statement in the program is executed in the normal manner. Thus in executing an IF...THEN statement, the computer makes a *decision* based upon the data being processed; the result of an IF....THEN statement is *conditional* upon the input data.

In the IF....THEN statements, two expressions are evaluated and compared according to a *relational symbol*. There are six relational symbols, one of which is the '=' sign already introduced. The relational symbols used in the BASIC language are tabulated on the next page.

Relational symbol	Interpretation	Example	*Meaning of Example*
=	IS EQUAL TO	X=Y	
<	IS LESS THAN	X<Y	*X is less than Y*
>	IS GREATER THAN	X>Y	*X is greater than Y*
<> OR ><	IS NOT EQUAL TO	X<>Y,X><Y	*X is not equal to Y*
<= OR =<	IS LESS THAN OR EQUAL TO	X<=Y,X=<Y	*X is less than, or equal to, Y*
>= OR =>	IS GREATER THAN OR EQUAL TO	X>=Y,X=>Y	*X is greater than, or equal to, Y*

Note: If you have difficulty remembering what > and < mean, think of them as megaphones. The louder noise (the larger number) is at the wide end.

The expressions which are compared may vary in complexity from simple, straightforward variables or numbers to large formulae including BASIC functions.

Thus the following types of IF....THEN.... statements are legal in the BASIC language.

```
40 IF X+Y>1000 THEN 450
60 IF X<>4.2*SIN(X) THEN 220
80 IF SQR(Z)>1-Y+3 THEN 20
```

QUESTIONS

Which of the following IF...THEN... statements are not permitted in BASIC, and why?

```
10 IF X-Y/2<=220.4 THEN 1024
20 IF X >= 4E10 THEN Y=2
30 IF Z = 1 THEN -24
40 IF X<>Y THEN GO TO 220
```

ANSWERS

20 THEN Y=2 is not permitted: it is not a line number.

30 -24 is an illegal line-number.

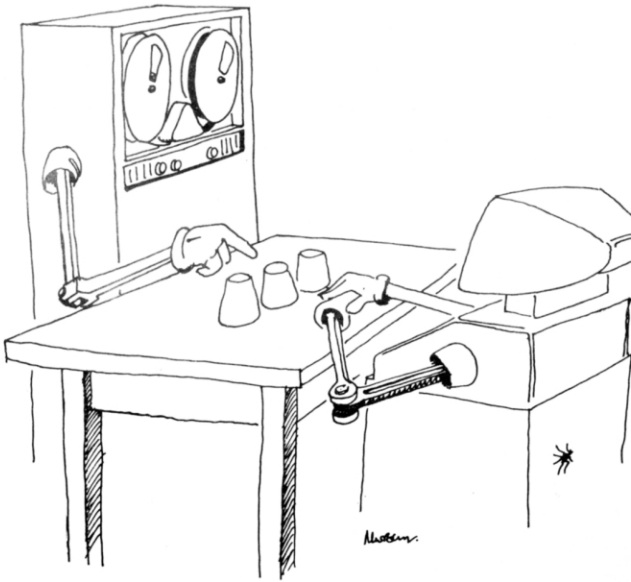
40 In *standard* BASIC you cannot combine an IF...THEN... statement with a GO TO statement. Some *dialects* of BASIC may permit this, however.

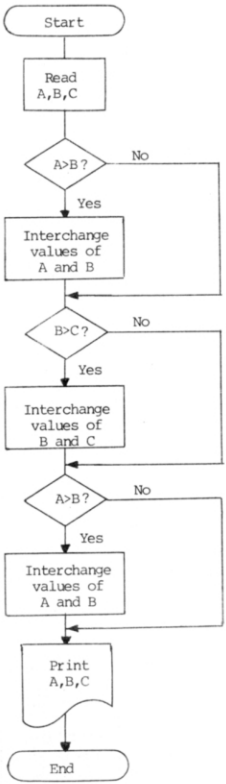
PROBLEM

Using the GO TO and IF...THEN... statements, you are now in a position to program the number-sorting problem discussed in Lesson 21.

HINT

The flowchart to sort three numbers into ascending order is repeated for convenience. When you wish to interchange the values of two variable-names you will need to use a third variable-name to store, temporarily, the value of one of the two. Just as if you want to interchange the contents of two pans you need a third pan to act as a temporary container.





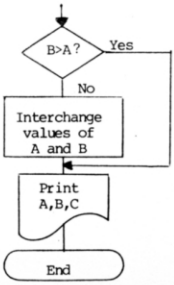
Now write the computer program.

SOLUTION

```
5 REM NUMBER SORTING PROGRAM
8 REM        I.WILLIAMSON        31-MAY-99
1Ø READ A,B,C
2Ø IF B>A THEN 6Ø
3Ø LET T=B
4Ø LET B=A
5Ø LET A=T
6Ø IF C>B THEN 1ØØ
7Ø LET T=C
8Ø LET C=B
9Ø LET B=T
1ØØ IF B>A THEN 14Ø
11Ø LET T=B
12Ø LET B=A
13Ø LET A=T
14Ø PRINT A,B,C
15Ø DATA 4.6,7.8,6.8
16Ø END

READY
RUN
 4.6            6.8            7.8

END AT 16Ø
READY
```

Notice the introduction of the variable T which was referred to in the hint. This is essential for the interchange to take place.

Notice also that in line 1ØØ the test is IF B>A THEN..., whilst in the flowchart, the test is the other way round. It is more convenient to code the test as B>A? So we redraw the flowchart at that point thus:

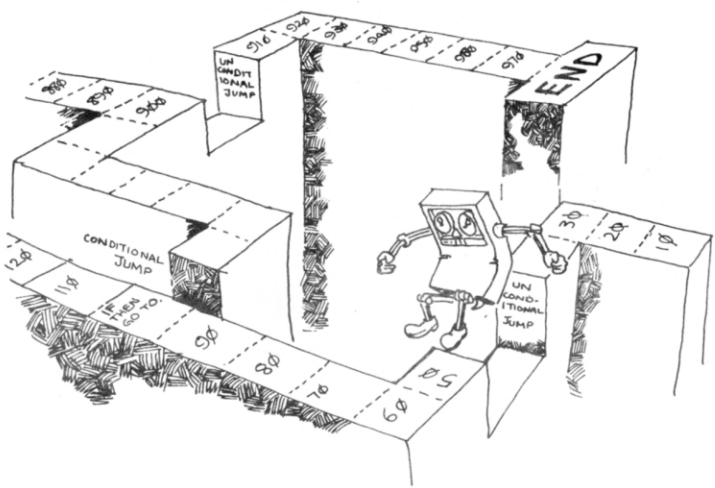


Had we blindly followed the original flowchart, we may have coded the program in the less efficient form shown below:

```
1Ø READ A,B,C
2Ø IF A>B THEN 4Ø
3Ø GO TO 7Ø
4Ø LET T=B
5Ø LET B=A
6Ø LET A=T
7Ø IF B>C THEN 11Ø
```

This program, whilst of poor programming style, illustrates the use of the GO TO statement to skip past sections of the program which do not require execution.

Note the importance of the flowchart in understanding and interpreting the program. Flowcharts are essential if a program contains jumps, conditional or unconditional.



LESSON 31

THE STOP STATEMENT

Programs can be written which will STOP under certain conditions.

BASIC programs will STOP if the END statement is reached, or if an error is detected which prevents the computer continuing with the program execution (a *fatal error*). Sometimes you may wish to stop execution at a number of different points in the program - the STOP statement is used for this. Here is an example.

```
1Ø REM PROGRAM TO COMPUTE THE SQUARE ROOT OF A POSITIVE NUMBER
2Ø REM STOPS IF NEGATIVE NUMBER IS ENTERED
3Ø PRINT "INPUT NUMBER";
4Ø INPUT N
5Ø IF N>Ø THEN 8Ø
6Ø PRINT "ERROR NEGATIVE NUMBER !!!"
7Ø STOP    Note: the computer is telling you that it has encountered a STOP
8Ø PRINT "SQUARE ROOT OF";N;"IS";SQR(N)     statement at line 7Ø.
9Ø PRINT
1ØØ GO TO 3Ø
11Ø END

READY
RUN
INPUT NUMBER ? 56.7
SQUARE ROOT OF 56.7 IS 7.52994

INPUT NUMBER ? -65.1
ERROR NEGATIVE NUMBER !!!

STOP AT 7Ø
READY
```

LESSON 32

PROGRAM EXAMPLE: CHECKING YOUR BANK BALANCE

This program enables you to INPUT the amounts on the cheques and be sure that the final balance is correct.

QUESTION

Write an interactive program to check your bank balance. You will need to supply the computer with your old balance and details of the credits and debits in the stubs of your cheque book.

HINT

For people who are not accustomed to reading bank balances, here is what one looks like, and what the printing on it means.

Ø1ØØ59687 - *this is the number of Tim Eiloart's cash account*

Tim Eiloart, Esq. - *here is his name printed by the bank*

This is the year

This refers to the amount of money in the bank account on Feb 11. It has been "carried forward" from the previous balance.

These are amounts paid out of the bank account

These are amounts paid into the cash account

1978

DATE	Detail	Debits	Credits	Balance	
11 Feb	Balance Forward			23.6Ø	*These are the sums remaining in the cash account after each amount is paid to or paid out*
13 Feb	Ø56687	17.9Ø		5.7Ø	
14 Feb	SO	5.ØØ		0.7Ø	
18 Feb	Ø56688	3.ØØ		2.3Ø	O/D
20 Feb			5Ø.ØØ	47.7Ø	
21 Feb	Ø56689	2.ØØ		45.7Ø	

This is a standing order

This means overdrawn

This column gives the date of each transaction

These are cheque numbers, for example: cheque Ø56687 was for 17.9Ø icu on 13 February

Now write your own program, and compare it with the solution given at the end of this Lesson. Don't forget to draw a flowchart, and to document your program properly. For this example, however, you might prefer to work with us through the problem, step-by-step.

Remember the three aspects of the problem definition which were introduced in Lesson 22.

Input - how to get the data into the computer.

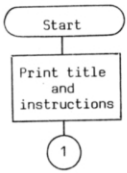
Process - what to do with it.

Output - how to present the results.

Before starting, you should consider an important point; 'Who is going to use the program?' A program for purely personal use can be acceptable, even if it is poorly explained and hard to use, because you, the creator, are the only person who has to master its complexities. A program to be used by others must be better documented, better presented and, most of all, easy to use and understand.

Most programmers write their programs with other users in mind - it's more challenging. Also more fun!

So let us assume that our bank balance program will be used by many people. We must, therefore, print a title and instructions on how to use the program. This is the first box in the flowchart.



QUESTION

What is the input to the program?

ANSWER

The program is to check a bank balance against transactions recorded in your cheque book. The first input is the old balance - which could be positive or negative. Subsequent inputs define each transaction - again either positive or negative to indicate credit or debit.

QUESTION

Since the program is interactive, how do we stop it?

HINT

Think of a number which never needs to be INPUT as a transaction.

ANSWER

A zero-value transaction can be used to stop the program.

QUESTION

What is the 'process' in this program?

ANSWER

Simple, we subtract the debits from the balance, and add credits to the balance.

QUESTION

How are you going to present the results?

ANSWER

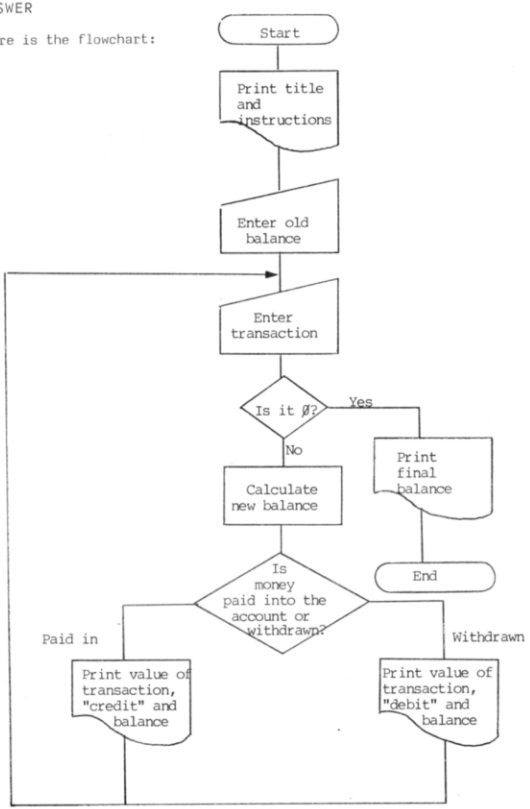
For each transaction we want to print the value of the transaction, to indicate whether it is credit or debit, and to print the current balance. The final balance should be printed separately after the zero transaction has been entered.

PROBLEM

Try drawing the complete flowchart now.

ANSWER

Here is the flowchart:



Now code the program - you might find the Credit and Debit PRINT statements a little difficult - think hard about these before you complete the coding. If necessary, look at the print-out, at the foot of the next page, before examining the program above it.

```
1Ø PRINT "PROGRAM TO CHECK BANK BALANCE"
15 REM       I.WILLIAMSON          31-MAY-99
2Ø PRINT
3Ø PRINT "INDICATE DEBIT BY ENTERING NEGATIVE TRANSACTION"
4Ø PRINT "A ZERO TRANSACTION WILL TERMINATE PROGRAM"
5Ø PRINT
6Ø PRINT
7Ø PRINT "WHAT IS OLD BALANCE";
8Ø INPUT B
9Ø PRINT
1ØØ PRINT "VALUE OF TRANSACTION";
11Ø INPUT T
12Ø IF T=Ø THEN 19Ø
13Ø LET B=B+T
14Ø IF T<Ø THEN 17Ø
15Ø PRINT "CREDIT";T;"I.C.U.S","BALANCE";B;"I.C.U.S"
16Ø GO TO 9Ø
17Ø PRINT "DEBIT";ABS(T);"I.C.U.S","BALANCE";B;"I.C.U.S"
18Ø GO TO 9Ø
19Ø PRINT
2ØØ PRINT "FINAL BALANCE IS";B;"I.C.U.S"
21Ø END

READY
RUN
PROGRAM TO CHECK BALANCE

INDICATE BALANCE BY ENTERING NEGATIVE TRANSACTION
A ZERO TRANSACTION WILL TERMINATE PROGRAM


WHAT IS OLD BALANCE ? 234.56

VALUE OF TRANSACTION ? 25
CREDIT 25 I.C.U.S             BALANCE 259.56 I.C.U.S

VALUE OF TRANSACTION ? -14.21
DEBIT 14.21 I.C.U.S           BALANCE 245.35 I.C.U.S

VALUE OF TRANSACTION ? Ø

FINAL BALANCE IS 245.35 I.C.U.S

END AT 21Ø
READY
```

LESSON 33

A GOOD PROGRAM CAN BE A BAD SOLUTION TO A PROBLEM

Good programmers have to look closely at the problem before they start to code it. You learn a little about statistical programming.

Suppose there are two machines, each putting fish into boxes. Each box is supposed to have 3Ø pounds of fish in it. Jones, the foreman, says to Smith who runs the machines: "Just check the two machines for accuracy, would you?"
Smith decides to weigh ten boxes from each machine and takes the average weight. Later Smith says: "I've checked out the two machines. Number One is OK. The average is exactly 3Ø. But Number Two is faulty, it averages 3Ø.1. I think we should give it an overhaul."

Jones asks to see the figures, and Smith shows the lists of weights:

Number One Machine	Number Two Machine
Weight of fish	Weight of fish
25	3Ø
33	3Ø
26	3Ø
34	31
2Ø	3Ø
4Ø	3Ø
3Ø	3Ø
3Ø	3Ø
29	3Ø
31	3Ø
3ØØ TOTAL	3Ø1 TOTAL
3Ø Average	3Ø.1 Average

"I see" says Jones "and you think Number Two is the faulty machine"
"Yes the average is 3Ø.1 ounces. Not far out, but we got the average just exactly right with Number One."
"But look at the figures!"
Most people can see that Number One Machine is very faulty. It weighs out fish from 2Ø pounds to 4Ø pounds.

Smith was asked to check the two machines for accuracy - and has done so in a thoughtless way. Unfortunately, Smith has chosen a poor solution to the problem - by looking only at the average weight of the packed boxes. It matters little how accurately each box of fish is weighed, since the final results are so misleading. Remember, however good your program, it can be a very poor solution to the problem. Many users of computers are blinded by the power and accuracy of the machine, writing and using programs which produce misleading results. Don't fall into this trap - think about the results you are looking for, and think of ways that you could be misled by your own program.

As a general, though not rigid, rule, *system-programs* (those affecting the workings of the computer) are written in assembly-level or low-level languages, and *user-programs* are written in high-level languages.

In many high-level languages, the range of symbols and mnemonics is extended so much that the program is similar to, and almost as readable as, the English language itself. An important characteristic which distinguishes high-level languages from low-level languages is that the former are generally offered by several computer manufacturers, and are not limited to a single type of computer system as are the latter.

We can emphasise these points by examining a small program in a high-level language. Suppose we want to evaluate the expression AX/4+B and call the value Y, printing out this value.

PROBLEM

Write out a BASIC program to evaluate Y = AX/4+B

ANSWER

```
10 INPUT A,B,C
20 LET Y=A*X/4+B
30 PRINT Y
40 END
```

Now, whilst this small program replaces perhaps 100 machine instructions, its construction is simple, and, what is more important, it is easy to understand. High-level languages such as BASIC are sometimes called *problem-oriented languages* because they are designed to simplify the process of programming for specific types of problems; this example shows an arithmetic problem. Scores of problem-oriented languages have been developed over the last two decades, but only a few have achieved wide acceptance. Problem-oriented languages in common use are BASIC, FORTRAN, APL, and, to a lesser extent ALGOL for scientific problems. COBOL is used for business programs, and PL/1 (called 'P.L.one') for both business and scientific problems. BASIC is one of the most widely used high-level languages. It is also the high-level language which is used by almost all low-cost computers - the kind the man-in-the-street can buy.

PROBLEM

Give three distinctions between assembly languages and high-level languages.

ANSWER

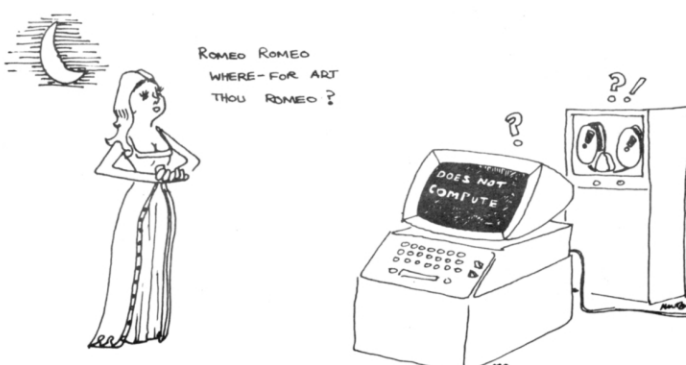
1) A statement in high-level language may generate many machine-code instructions when translated, whereas there is usually a one-to-one relationship between an assembly-language statement and a computer's machine-code.

2) A high-level language is generally available on many different computer systems - whereas an assembly language is specific to one type of computer system.

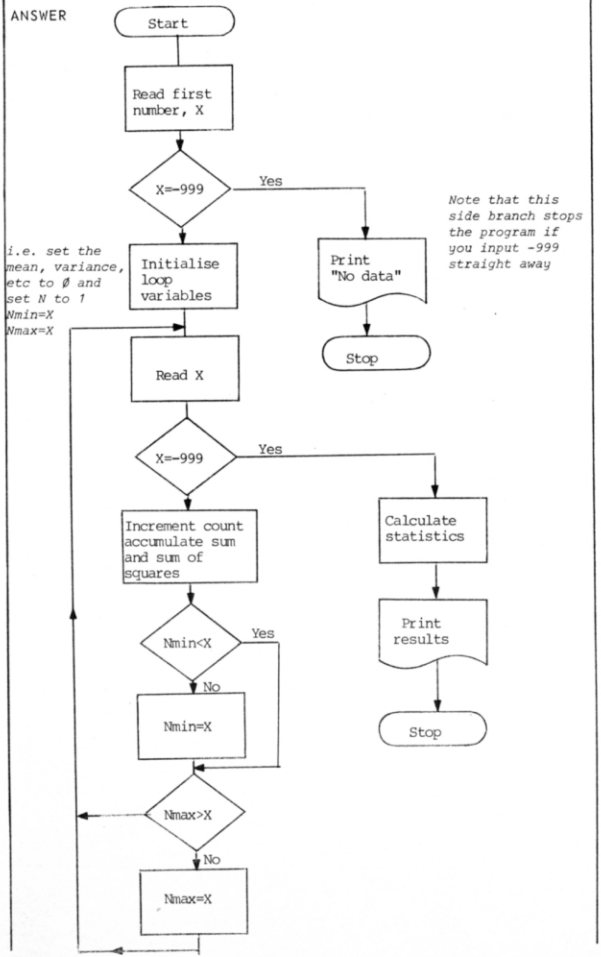
3) A high-level language is easier to learn and understand.

A few more points on languages. When you are programming, various features of the computer make themselves more apparent when assembly language is used than if high-level languages are used. Physical constraints, such as the size of the memory available, make themselves evident.

Although many high-level languages are good, they are not as yet as efficient as a good human assembly-language programmer. For this reason good assembly-language programs are more efficient than high-level programs. However, the cost in time of preparing assembly-language programs is greater than that involved in high-level programming. For programs which are used over and over again (systems-programs), assembly language is often best, as maximum efficiency is obtained whilst running the program, in terms of storage-space and execution-time. This will often offset the extra time for preparation.



ANSWER


Start
Read first number, X
X=-999
Yes
Print "No data"
Stop
Note that this side branch stops the program if you input -999 straight away
i.e. set the mean, variance, etc to Ø and set N to 1
Nmin=X
Nmax=X
Initialise loop variables
Read X
X=-999
Yes
Calculate statistics
Print results
Stop
Increment count accumulate sum and sum of squares
Nmin<X
Yes
No
Nmin=X
Nmax>X
No
Nmax=X

```
1Ø REM PROGRAM TO CALCULATE STATISTICS OF EXPERIMENTAL DATA
2Ø REM -999 IS USED TO TERMINATE DATA BLOCK
3Ø REM          T.EILOART          31-MAY-99
4Ø READ X
5Ø IF X <> -999 THEN 8Ø
6Ø PRINT "NO DATA, PROGRAM HALTED"
7Ø STOP
8Ø LET A=X
9Ø LET B=X
1ØØ REM A AND B ARE MINIMUM AND MAXIMUM VALUES OF DATA
11Ø LET N=1
12Ø LET T=X
13Ø LET S=X*X
14Ø REM N IS NO. OF SAMPLES, T IS SUM, S IS SUM OF SQUARES
15Ø REM ALL VARIABLES INITIALISED
17Ø READ X
18Ø IF X=-999 THEN 27Ø
19Ø LET N=N+1
2ØØ LET T=T+X
21Ø LET S=S+(X*X)
22Ø IF A<X THEN 24Ø
23Ø LET A=X
24Ø IF B>X THEN 26Ø
25Ø LET B=X
26Ø GO TO 17Ø
27Ø LET R=B-A
28Ø LET M=T/N
29Ø LET V=(S-T*M)/(N-1)
3ØØ LET D=SQR(V)
31Ø PRINT "SAMPLES","SUM","SUM OF SQUARES"
32Ø PRINT N,T,S
33Ø PRINT
34Ø PRINT "MEAN","VARIANCE","DEVIATION","RANGE"
35Ø PRINT M,V,D,B-A
4ØØ DATA 1Ø.Ø2,96.48,88.66,63.64,83.9Ø,3Ø.61,28.55,95.82,17.93
41Ø DATA 45.21,9.85,52.21,24.62,77.78,45.Ø5,7Ø.33,16.47,65.81
42Ø DATA 45.67,87.66
5ØØ DATA -999
51Ø END
READY

RUN

SAMPLES         SUM             SUM OF SQUARES

 2Ø              1Ø56.27         72484.45

MEAN            VARIANCE        DEVIATION       RANGE

 52.8135         878.9Ø2         29.646          86.63

END AT 51Ø

READY
```

LESSON 34

THE COMPUTER'S LIMITATIONS

Computers have limited accuracy which may wreck your program.

Computers have many limitations - most of which depend upon the design of the machine and the size of its memory. But, with BASIC programs, the accuracy of the computation is limited.

In arithmetic computations, computers have a limited accuracy, and they vary in performance. Within a certain range (dependent upon the computer system but usually greater than ± 32ØØØ) *integers*, that is, whole numbers with no fractional parts, are treated with absolute accuracy. Integers outside the permitted range, and numbers such as 5.4 and Ø.ØØ2 are stored in *floating point notation* which is equivalent to the exponential introduced in Lesson 11. The limited accuracy is evident in the print-out since non-integer numbers are printed with only six or seven digits.

The computer also computes to a similarly limited accuracy, usually to a greater precision than the print-out but rarely to more than 10 or 11 digits.

```
1Ø LET B=Ø
2Ø LET A=1/3
3Ø LET B=B+A
4Ø PRINT B
5Ø IF B=1 THEN 7Ø
6Ø IF B<2 THEN 3Ø
7Ø END

READY
RUN
 .333333
 .666667
 1                  According to line 5Ø the computer should now
 1.33333            end but it doesn't because the B never
 1.66667            equalled exactly 1.
 2
 2.33333

END AT 7Ø
READY
```

The computer rounds 1/3 down to Ø.333333, and hence the third time the computer tests for line 5Ø IF B=1 THEN 7Ø, it finds that B=Ø.999999 and the equality at line 5Ø is not satisfied. Note that the computer actually prints 1 rather than Ø.999999 although its internal record of B is the latter.

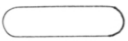
SUPPLEMENTARY LESSON 4

FLOWCHARTING SYMBOLS

A list of many more flowchart symbols and their meanings.

The following flowchart symbols are in common usage.

TERMINAL



This symbol represents a terminal point in the program. A terminal point is a start, stop, or interruption in the logical flow. The appropriate descriptive word should be included within the symbol.

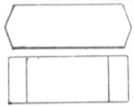
PROCESSING

A rectangular symbol represents a processing function - such as arithmetic operation, transfer of data, edit or sort. A brief description of the task is included within the symbol.
NB: It is wise to draw the symbol of sufficient size to contain the necessary information.

DECISION

A diamond is used to indicate a branch in the program flow. A test condition is included within the symbol and the possible results of the test label the respective flows from the symbol. Results may be 'yes' or 'no', or '<', '=' and '>', etc.

PREDEFINED PROCESS

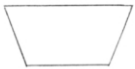


This indicates a number of processing steps - the detail of which is not relevant to the overall program flow. It is usually detailed in another flowchart.

ANNOTATION

A comment to clarify the flowchart is included within the arms of the symbol.

GENERAL INPUT/OUTPUT



This symbol indicates an operation of reading or writing. It may be used for all input/output functions; the precise operation is defined within the symbol.

However, specific symbols are available for some input/output functions as follows:

DOCUMENT INPUT/OUTPUT

This symbol is used for printer output, and also for input using optical character recognition (OCR) equipment.

MAGNETIC TAPE

This symbol indicates that input/output is from or to a magnetic tape.

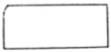
MAGNETIC DISC

This symbol indicates that input/output is from or to a magnetic disc.

MAGNETIC DRUM

This symbol indicates that magnetic drums are in use as a storage medium.

PUNCHED CARDS



This symbol indicates that punched cards are used for input/output.

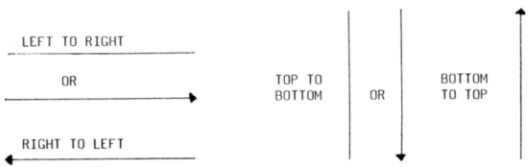
PUNCHED TAPE

This symbol indicates the use of punched tape.

KEYBOARD INPUT

This symbol indicates the use of a keyboard for input of data.

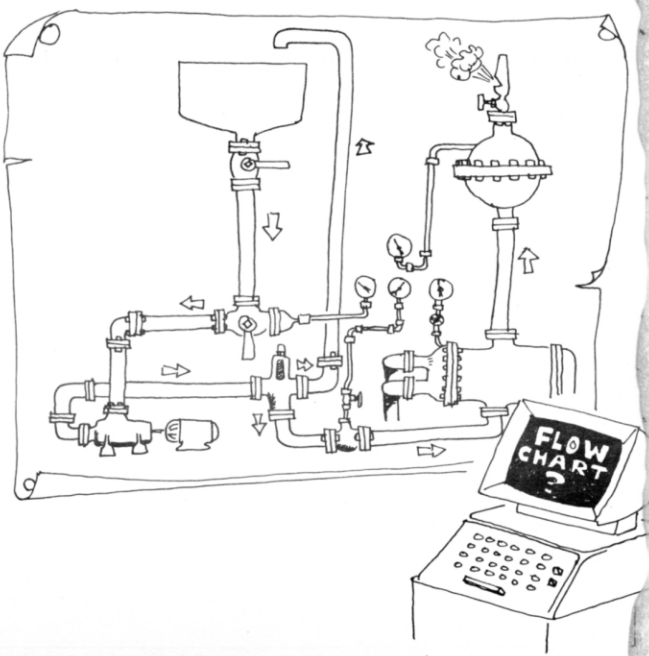
The flow is represented by lines drawn between symbols. Arrowheads are used to indicate the direction of flow and for increased clarity should be used liberally in the flowchart. The normal direction of flow in preparing the flowchart is from left to right and from top to bottom. Lines representing these normal directions do not need to be marked with arrowheads. However, in this course we use arrowheads wherever they might be helpful. It is not written for experts who are accustomed to reading flowcharts. You too may find that it helps to use arrowheads, rather than leaving them out. Your flowcharts may be read by people less expert than yourself.



Symbols must be of sufficient size to include the necessary information. The two symbols which follow are used to assist in the layout of the flowchart.

This is an on-page connector - indicating a reference to another point in the flowchart on the same page. Numbers or letters are used to label the corresponding entry and exit points. This symbol does not represent a program operation. It can point in any direction.

This is an off-page connector which is used in the same way as the on-page connector to connect to another part of the flowchart on a different sheet.



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