PROGRAMMING IN ATLAS AUTOCODE

COMPUTER UNIT REPORT No. 1

by P.D. Schofield and M.R. Osborne

(Revised Edition) 28th June 1965

PREFACE

This is a revised version of Computer Unit Report No. 1, which was originally issued on 3rd. Mar. 1964. The revision was undertaken not only to improve the text, but also to take account of such changing circumstances as

(i) Changes in the compilers available on Atlas.

(ii) The writing of the Edinburgh University Atlas Autocode compiler which has made Atlas Autocode available also on the K.D.F.9.

This book is intended to serve as an introduction to the Atlas Autocode programming language. It is based on courses of lectures given at Edinburgh University, and describes a version of the language acceptable to all current Atlas Autocode compilers.

For a complete beginner, the following should prove suitable for a first reading:-

Chapters 1 - 5	(But see note on page 27 and omit any parts of pages 40, 48 which cause the reader trouble).
Chapter 8	(pages 89-91. These may be read any time after page 44).

We should point out that our examples are chosen to illustrate points of the language: we do not claim that the techniques used are in any sense the best possible.

We should be glad to hear from anyone who discovers or suspects any errors.

In making this revision, we have benefitted considerably from discussions with our colleague Mr. Harry Whitfield, who led the team writing the Edinburgh University compiler.

Our thanks are due to Mrs. Jackie Snashall and Miss Isabel Fraser who bore the burden of re-typing and to Mr. Brian Read who compiled the index and produced some of the diagrams.

P.D. Schofield.

P.D. SCHOFIELD

M.R. Osborne

M.R. OSBORNE

28th June 1965.

TABLE OF CONTENTS

CHAPTER 1 : INTRODUCTION

Stages involved in using a Computer.

A simple view of the Computer.

Information which must be suppled to the Computer.

Analysis of a very simple problem.

PROGRAMMING IN ATLAS AUTOCODE

This book is intended for those who wish to learn to write programs in Atlas Autocode, a language available on both Atlas and KDF9 computers.

A program consists of a detailed set of instructions to the computer, explaining exactly how it is to solve a certain problem. It therefore follows that the programmer must first:-

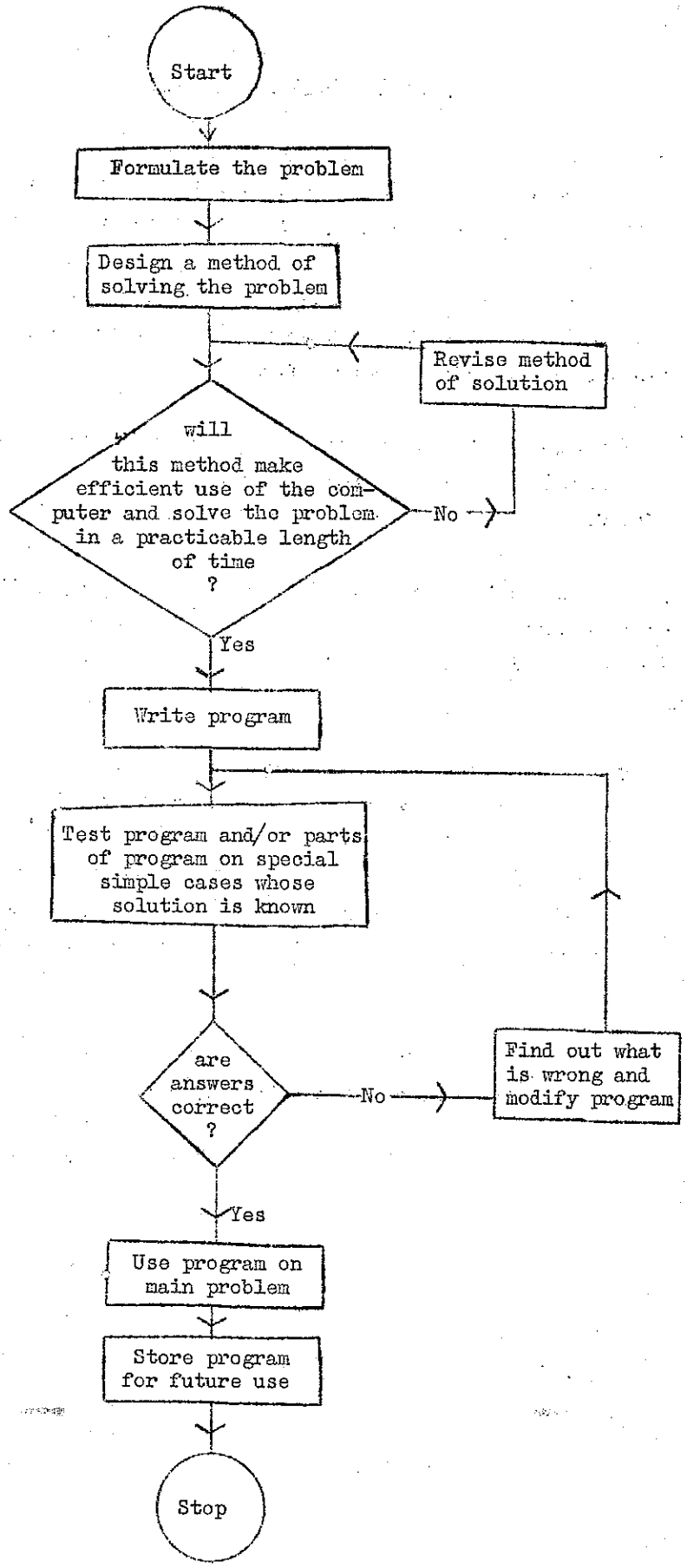
(a) Formulate the problem and decide on the method to be used to obtain a solution. Only then can he
(b) Write a program describing the method already chosen.

Although this book is mostly devoted to describing process (b), it must be emphasised that, in any moderately large problem, it is process (a) which contributes most to the success or failure of a project.

Two general suggestions can be made about this planning stage. Firstly it often pays to draw a 'flow diagram' to help plan the logical connections between different parts of the program. The reader will find many examples of flow diagrams in the subsequent pages. Secondly, considerable effort and money can often be saved by seeking, at the earliest possible stage, the advice of someone who has successfully completed a similar project.

Figure 1 uses the formalism of a flow diagram to indicate the steps involved in using a computer to solve a problem.

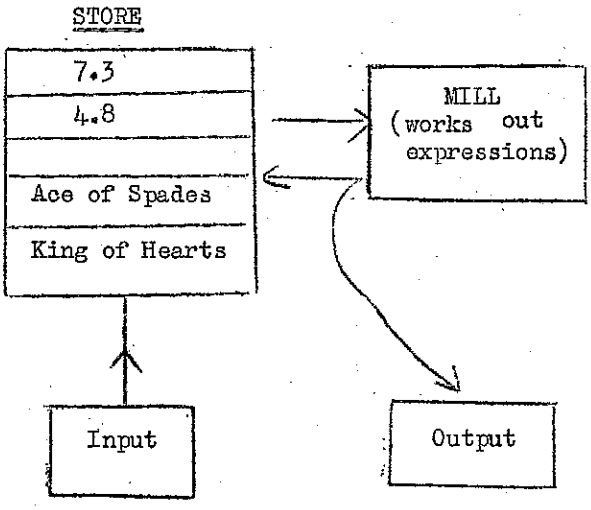
Fig. 1


Start
Formulate the problem
Design a method of solving the problem
Revise method of solution
will this method make efficient use of the computer and solve the problem in a practicable length of time ?
No
Yes
Write program
Test program and/or parts of program on special simple cases whose solution is known
are answers correct ?
No
Find out what is wrong and modify program
Yes
Use program on main problem
Store program for future use
Stop

THE COMPUTER

The basic operation of the computer is most easily understood from the following simplified (and partly fictitious) diagram:-

Fig. 2



The STORE consists of a large number of locations in which information can be deposited. Depending upon the way in which the machine is being used, this information may be thought of as numbers, values of playing cards, letters etc. Some of the store also contains instructions which tell the computer what to do next.

The MILL is a place into which the machine copies pieces of information from the store and works out expressions depending upon this information.

e.g. (1) copy the first two numbers from the store and multiply them.

(2) copy the two cards in locations 4 and 5, and find the higher-ranking.

When an expression has been worked out, it can either be printed out as an answer, or replaced in the store for use later.

Moving information in or out of a location in the store is in some ways similar to the operation of a tape recorder. When withdrawing information ('reading'), we take a copy of the contents of the location. The original information is still there, and can be used again as often as required. When putting information in ('writing') the previous contents of that location are destroyed.

Warning : If we read the contents of a location before putting anything in, we are in danger of obtaining whatever was left behind at the end of the previous program.

INPUT

When we wish to use the computer, we normally need to feed in two 'documents' (1) Program
(2) Data
The difference between the two is shown by the two examples below:-

Program	Data
Method for solving a set of equations	Set of equations
Method for sorting words into dictionary order	List of words

Most of the program consists of a series of instructions telling the computer to carry out various operations. These are kept in the store in a code or 'language' which is not readily comprehensible to the programmer. It is possible, but tedious, to write programs in this language (in the early days of computers, nothing else was available). Nowadays we can write in a more convenient language, Atlas Autocode for example, and the computer is supplied with a COMPILER which translates the program into its own language.

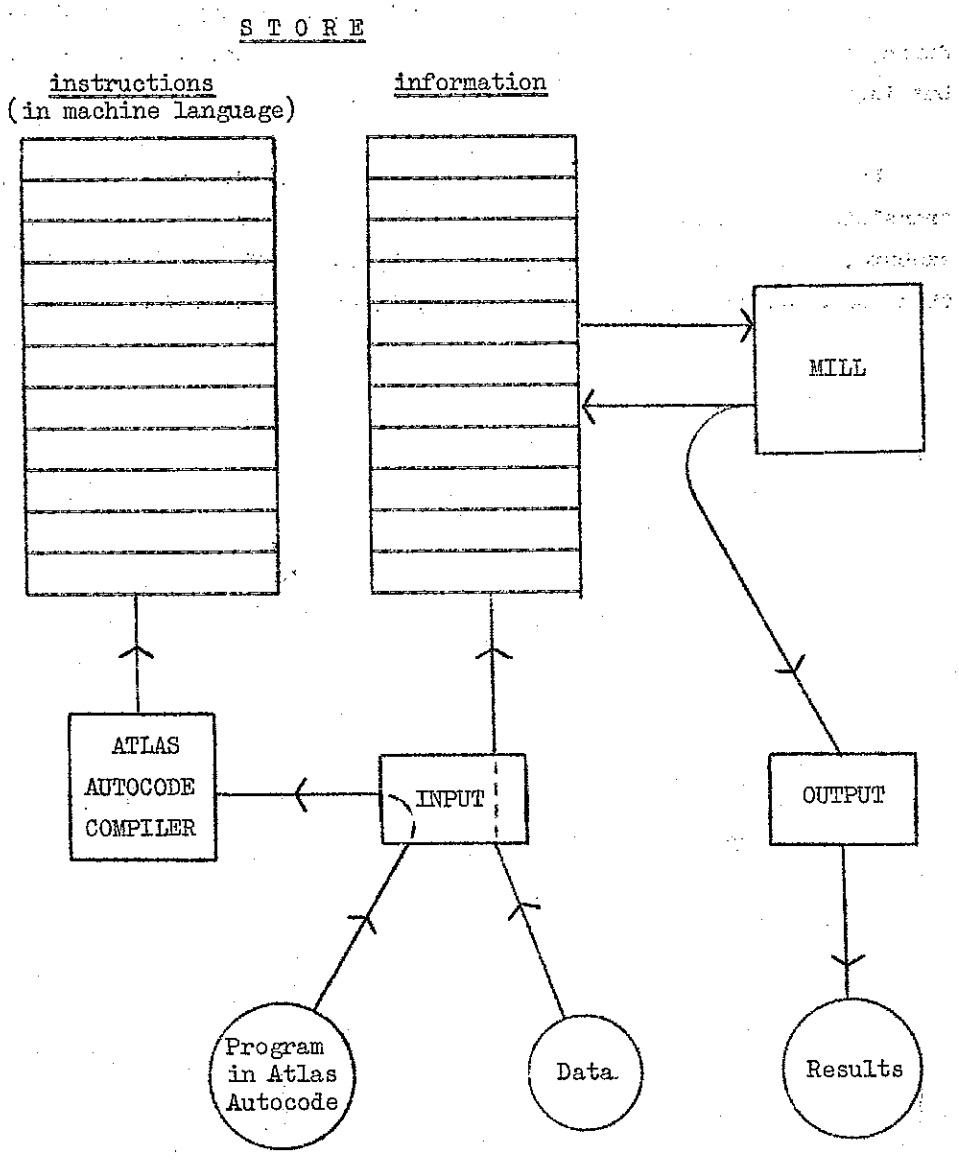
In the first place the program, and often the data as well, will be written down with pencil and paper. After careful checking, this will be converted into suitable form (normally punched paper tape or punched cards) for feeding to an INPUT device.

OUTPUT

The results of the calculation will come out via an OUTPUT device which either prints out answers directly, or produces punched paper tape or cards for subsequent printing. A device for printing out answers directly is known as a LINE PRINTER.

We can now give an improved version of Fig. 2:-

Fig. 3



The sequence of events should be:-

(1) Read in Program.

(2) Compile (i.e. translate) into machine instructions.

(3) Execute the compiled program which will contain instructions to

  (a) Read in Data

  (b) Carry out Calculation/Processing

  (c) Print out Results

However if any violations of the rules of the language are detected during the compiling stage, no attempt is made to execute the program, but instead a list of faults is printed out.

Even after execution of the program has started, some part of the translated program may turn out to be impossible for the machine to execute. For example, it cannot divide by zero. The execution will then cease and an appropriate fault signal will be printed.

ORDER OF PRESENTATION TO COMPUTER

In a small job, the preliminary information (Job Heading), program and data are usually all supplied to the computer on one piece of tape ordered as follows:-

```
*** A                          )  Job Heading giving title of program
JOB                            )  and stating which compiler is to
BLOGGS' FIRST PROGRAM          )  be used to translate the program
COMPILER AA                    )  following.  (AA=Atlas Autocode)

begin                          )
.....                          )
.....                          )
.....                          )  Program
.....                          )
.....                          )
end of program                 )

.....                          )
.....                          )
.....                          )  Data
.....                          )
.....                          )

*** Z                          )  Marks end of tape
```

NOTES

(1) Programs written in Atlas Autocode can at present be run on either an Atlas or a KDF9 computer. If using an Atlas, we have the choice of two compilers, both written at Manchester University

(a) COMPILER AA

(b) COMPILER AB

The latter is a faster but somewhat restricted version of AA. The Atlas Autocode compiler for KDF9 has been written at Edinburgh University.

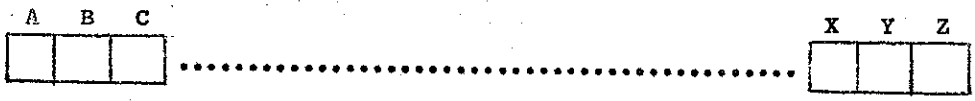
Except where specially indicated, this book describes how to write programs equally acceptable to all three compilers. Precise specifications of each compiler can be obtained from the approprate Computer Unit.

(2) The example of a Job Heading given above is the minimum required. New programmers should consult the Computer Unit of their own University to discover what extra details need be given in any particular case.

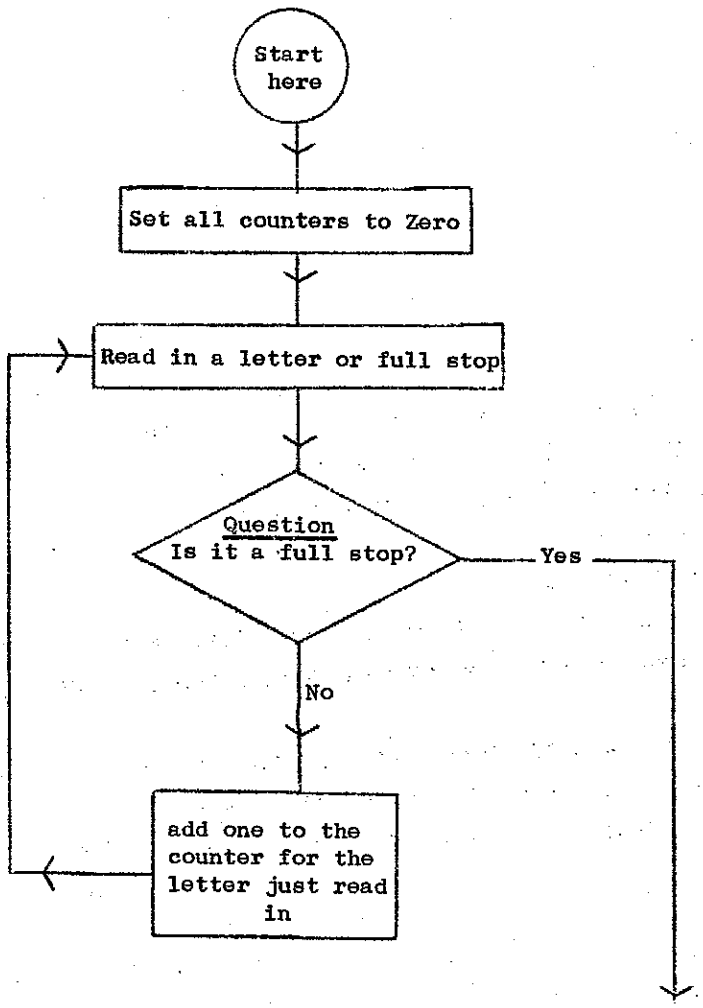
ANALYSIS OF A VERY SIMPLE PROBLEM

Suppose that we wish to read in a string of letters of the alphabet (in any order) and count how many times each letter occurs. To mark the end of the string we shall use a full stop.

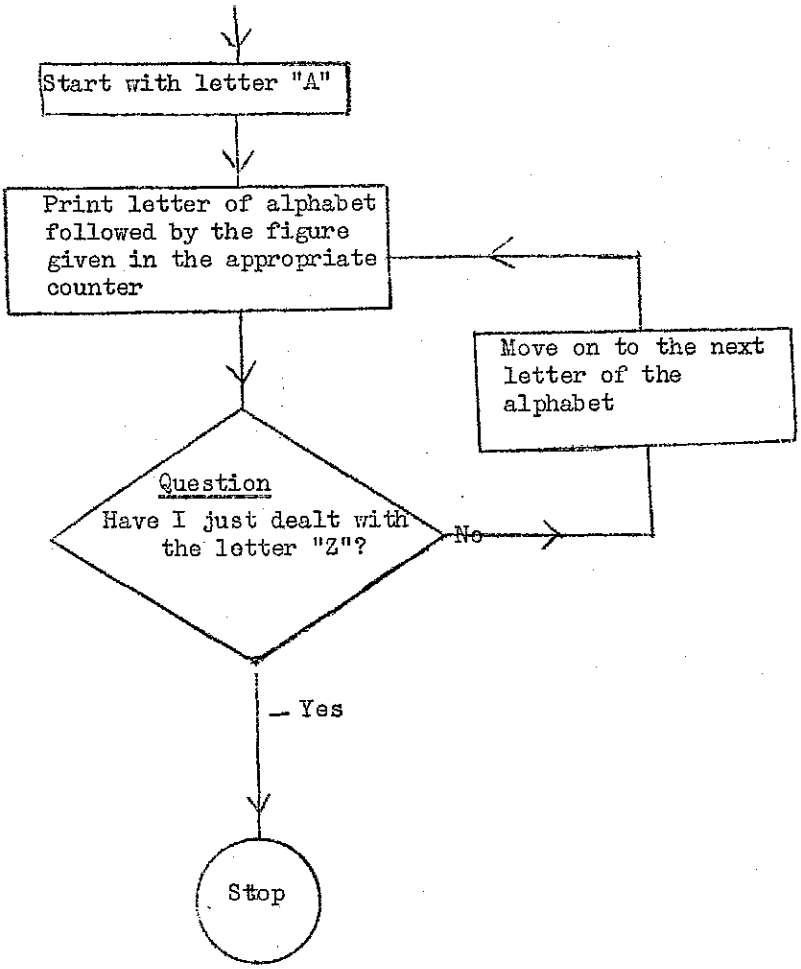
A convenient method of doing this is to set up 26 counters in the machine, one for each letter of the alphabet.



A possible flow diagram is:-



Suppose we now wish to print out the number of times each letter has occurred. The process is:-



NOTE

An Atlas Autocode program corresponding to these flow diagrams will be given in Chapter 4.

CHAPTER 2 : BASIC NUMERICAL OPERATIONS

Names.

Declarations of variables, including simple arrays.

Basic input, output and assignment instructions.

Separators, Comments.

Blocks.

Labels.

Jump Instructions.

Conditional Instructions.

Example Print the average of a list of numbers.

ATLAS AUTOCODE

The Atlas Autocode language is better equipped for dealing with numbers than with other types of information. For this reason, the basic principles of the language will first be explained in terms of very elementary calculations with numbers. Some equivalent orders for manipulating non - numerical symbols will be given in Chapter 3.

NAMES

Before a number or symbol can be placed in a location of the store, this location must be given a name. A name must start with a letter and consist of

(a) one or more letters (a,b,.....z,A, B, ... Z)
(b) possibly followed by one or more digits (0, 1, 2, ... 9)
(c) possibly followed by one or more primes (', '', ''' etc.)

Examples x, a2'', total 3, SUM', Sum

Notes (1) a2c is not permitted as a letter follows a digit.
(2) The compiler completely disregards all spaces (and underlined spaces) in the program. Spaces may thus be used to improve legibility of program.

LINES

The basic units of a program are

(a) Declarations
(b) Instructions
(c) Separators

These will be described in the following pages. We shall refer to them collectively as 'lines', since they are normally written on distinct lines.** However, two or more 'lines' can be written on the same physical line, provided they are separated by semi-colons.

** What is here called a 'line' is often called a 'source statement' in the literature.

DECLARATIONS

Names are allocated to locations in the store by means of declarations such as:-

Declaration	Meaning
real a	set aside the next unused location, call it a and be prepared to put a 'real' number in it later.
integer b, c3	set aside the next two unused locations, call them b and c3 and be prepared to put integers (whole numbers) in them later.

Note (1) Generally speaking, a name allocated to a location will remain fixed throughout the program. However, the contents will vary whenever new number is placed in it.

(2) The word 'variable' is used to describe locations which have been set aside to contain numbers, either real or integer.

(3) There are three distinctions between real variables and integer variables:-

(a) An integer variable can only contain a whole number. A real variable may contain either an integer or a number such as 73.4827, with up to 11 significant figures.

(b) There are certain purposes for which only integer variables are allowed. (e.g. To give the number of times a group of instructions is to be repeated: repeating 1.7 times would be impossible).

(c) When we do multiplications and additions of integer variables, the machine produces the exact answer. When doing arithmetic on real variables, the answers are 'rounded off' to 11 significant figures.

UNDERLINING

Note that the underlining of certain key words (real and integer above, for example) is an integral part of the language. At this stage the reader is advised to accept, as arbitrary rules, that certain words are, and others are not underlined.

DECLARATION OF ARRAYS

We can also declare a whole array of variables, all having the same name, but distinguished from one another by means of a 'suffix' in brackets after it.

real array d(1:4) set aside 4 locations :-

real array e(1:7),f,g (0:4) set aside (7 locations for e(1) to e (7)
(5 locations for f(0) to f (4)
(5 locations for g(0) to g (4)

Notes (1) Arrays of integer locations are declared in a similar manner by writing integer array .....

(2) In the case of real arrays, it is permissible to omit the word real, simply writing array .....

(3) Note the difference between

real array d(1:4) which gives four locations

and real d4 which gives only one location

These four types of declaration are normally written on separate lines, but may instead be separated by a semi-colon.

either real a,b,x3'
integer array y (1:20)

or real a,b,x3' ; integer array y (1:20)

Further types of declaration, allocating names to functions, routines, switches, and multi-suffix arrays will be described later. Declarations are preparatory in nature, and should be contrasted with 'instructions' which, when executed, bring about the transfer of information to locations already prepared.

Note : A name cannot be used simultaneously for two different purposes. For example:-

real A
real array A(1:10)

would cause a fault signal.

INSTRUCTIONS

Some simple types are given below. They are written on separate lines or separated by semi-colons in the same manner as declarations.

Input Instructions (also see p. 85)	Meaning
read (a)	read in the next number in the data and put it in the location whose name is a.
read (b, c3, d(4))	read in the next 3 numbers in the data and put them in b, c3, and d(4)

Assignment Instructions (also see p. 42)

These look like mathematical equations but the meaning is quite different.

Instruction	Meaning
a = b + c	work out the expression on the right (i.e. contents of b plus contents of c) and then put it in the location given on the left (i.e. a)
a = 2a + 1	copy the contents of a, double it, add 1 and place the answer back in a.

Thus

a	7.32	would	a	13.0
b	10.0	become	b	10.0
c	3.0		c	3.0

Notes (1) b + c = a is not permitted since b + c is not the name of a variable.

(2) a = b is quite different from b = a.

(3) the use of more complicated expressions on the right will be explained later.

Output Instructions (Also see p. 86)

print (x, 3, 1) print (2x + y + 7, 3, 1)	work out in the Mill the value of the first expression in the brackets (i.e. x or 2x + y + 7) and print the value of the expression with 3 figures before the decimal point and one after. (The figures 3 and 1 can, of course, be varied).
newline	output printer is to go to the start of a fresh line, moving the paper up accordingly.
newlines (2)	equivalent to:- newline ; newline
space	output printer is to leave one blank space (printing takes place from left to right across the page)
spaces (3)	equivalent to:- space; space; space
caption MORRIS1100	output printer is to print out the set of characters MORRIS1100.

Note The instruction caption ..... is chiefly used to obtain headings and explanatory notes in the output. These notes may be required to include spaces, newlines, etc.

A special method is provided for outputting the symbols space, newline and semi-colon with a caption, since spaces are ignored by the computer and a semi-colon or newline character marks the end of the caption 'line':-

$ or ø	represents a space
¤ or ¶	represents a newline
¦ or ;	represents a semi-colon.

Either

(a) caption ¤ MORRIS $¤ 1100

or (b) newline ; caption MORRIS ; spaces (2) ; caption 1100

will produce an output (at the beginning of a newline):-

MORRIS 1100

SEPARATORS

A few lines, neither declarations nor instructions, have to be written into a program, chiefly to mark the begining and end of blocks and routines. Examples are begin end and end of program, described on the next page.

COMMENTS

Any line starting with comment is disregarded by the compiler. This permits the insertion of explanatory notes, which must not contain a semi-colon, for the benefit of the reader. For example:-

```
read (n)
comment n is the number of cases to be solved.
```

For brevity, a single vertical bar can replace comment. For example:-

```
read (n)
| n is the number of cases to be solved.
```

A third equivalent method of writing the above is:-

```
read (n) ; | n is the number of cases to be solved.
```

BLOCKS

A program is normally split up into a number of blocks. In general a block consists of

```
begin

.....  )
       )
.....  )
       )
.....  )   declarations
       )
.....  )
       )
.....  )

.....  )
       )
.....  )
       )   instructions
.....  )
       )
.....  )

end
```

At the end of the last block of a program, end is replaced by end of program.

Example

```
begin
real array a(1:3)
real b

read (a(1),a(2),a(3))
b = a(1)+a(2)+a(3)
print (b,2,3)
end of program
```

This causes the machine to read in three numbers, add them and print out the total.

Note The machine automatically terminates the calculation on reaching end of program. If it is required to stop the calculation at any other point, the instruction stop is used.

LABELS

Any 'line' in the program can be labelled by writing on the left a positive integer, followed by a colon. The label has no effect other than to give the line a reference number.

EXAMPLE

```
     i=i+j
10:  read (x)
     x=x+i
```

JUMP INSTRUCTIONS

Normally, instructions are obeyed in the order in which they are written. In order to make the machine jump, either forwards or backwards, to a labelled line in the program we can use a jump instruction written, for example:-

instruction	meaning
-> 10	the next line to be obeyed is the one labelled 10:

Notes (1) By making the machine jump back to an earlier part of the program we can make it go round a loop of instructions many times.

(2) Although jumps can be either forwards or backwards, we are not allowed to jump from one block to another.

(3) Jump instructions are frequently made conditional, as described in the next section.

CONDITIONAL INSTRUCTIONS

Assignment and jump instructions may be made subject to a condition.

Examples	Meaning
a = b + c if x = 0	Carry out the instruction if (or unless) the condition is satisfied. Otherwise skip and pass on to the next instruction.
-> 27 unless a > b + 2	
stop if n>100	

If preferred, instructions may be written with the condition first, followed by then:

```
if x = 0 then a = b + c
unless a > b + 2 then -> 27
if  n > 100 then stop
```

Note (1) Note the different uses of '=' in the first example. In x = 0 it has its normal mathematical significance. In a = b + c it means an assignment.

(2) In the condition we may use any of the relations = ≠ > ≥ < ≤

MORE COMPLICATED CONDITIONS

These may be formed

(1) with a two-sided condition **

e.g. if 0 < x < 1 + y then .....

(2) by writing several conditions separated by and with the obvious significance.

e.g. if x>0 and y = 0 and z ≠ 2 then .....

(3) by a similar use of a succession of or's

e.g. if x>0 or y = 0 or z ≠ 2 then .....

(4) by combining (2) and (3) provided and's or or's are separated by brackets.

e.g. if (x>0 or y = 0) and z ≠ 2 then .....

** This form of condition is not accepted by the Manchester Compiler AB.

EXAMPLE OF A SIMPLE PROGRAM

Suppose we want to read in a list of positive numbers and print out their average. Suppose we do not know in advance how many there will be. In order to inform the computer when we have come to the end of the list, we terminate it with the number -1.

We shall need the following variables:-

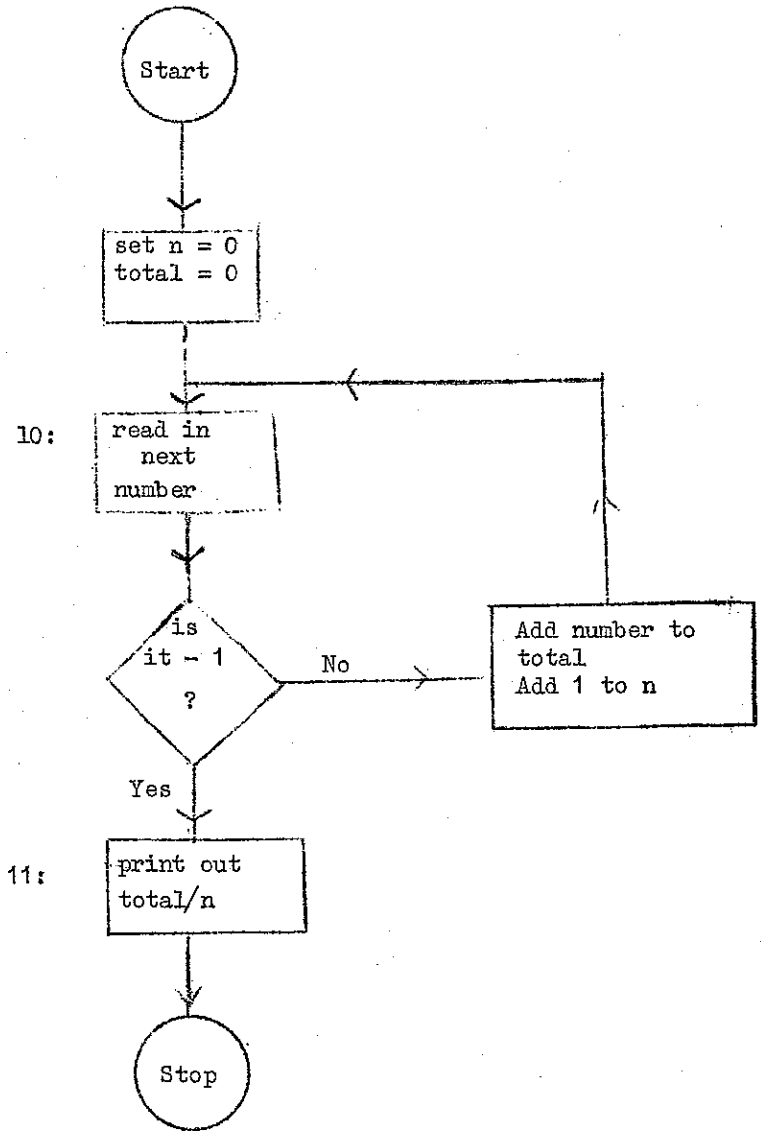
(1) a place in which to put the numbers as they are read in.
(2) a running total.
(3) a count (integer n, say) of how many numbers have been read in.

Note that (2) and (3) must be set to zero before starting.

A possible flow diagram is given on the next page:-

PROGRAM FOR PRINTING AVERAGE

A possible flow diagram is:



and a program to implement this is:-

```
      begin
      integer n
      real total, x
      n = 0;  total = 0
10:   read (x)
      -> 11 if x = -1
      total = total + x
      n = n + 1
      -> 10
11:   newline
      caption Average $ =
      print (total/n,3,5)
      end of program
```

CHAPTER 3 : BASIC SYMBOL OPERATIONS

(NOTE This chapter may be omitted by those solely interested in numerical calculations).

Input of symbols.

Assignment of symbols.

Conditions using symbols.

Output of symbols.

Relationship between symbols and integers.

Example Program to count symbols.

MANIPULATION OF SYMBOLS

INTEGER variables can not only be used to store integer NUMBERS as described in the last chapter, but can also hold SYMBOLS. Possible symbols include

(a) The letters of the alphabet (both upper and lower case).
(b) The numerical digits 0 to 9 (see note 1 below).
(c) ( ) [ ] . , : ; ' ?
(d) + - * / ↑ | = ≠ > ≥ < ≤ α π
(e) space and newline.

Notes

(1) The symbol 9 is NOT the same as the number 9.

(2) Symbols can be stored in integer variables, including elements of integer arrays.

INPUT OF SYMBOLS (See also p. 87)

Input Instruction	Meaning
read symbol (a)	Read the next symbol on the data tape and place it in location a. Move the data tape on by one symbol. Note (1) The instruction read symbol can only read one symbol at a time (unlike the instruction read which may read several numbers). Note (2) a MUST be an integer variable or an element of an integer array.

Important Note When reading numerical data, spaces and newlines simply mark the end of a number. However, both spaces and newlines count as symbols and will be read in by the routine read symbol.

ASSIGNMENT OF SYMBOLS

Instructions to assign symbols to integer variables are written in a form very similar to those which assign numbers, but the symbol concerned is written between a pair of 'quotation marks'. For example:-

```
integer i, j, k
i = '*'
j = 'P'
k = '7'
```

Note that the last two instructions assign the SYMBOLS P and 7 to j and k respectively. On the other hand, the instructions

```
j = P
k = 7
```

assign to j the NUMERICAL value currently stored in the variable named P, and to k the NUMBER 7.

CONDITIONS

Conditions depending upon the equality, inequality, etc. of symbols may be written in a fairly self-evident manner e.g.

```
if i = '*' then stop
-> 9 unless k = '?' or j = 'A'
```

SYMBOLS FOR SPACE, NEWLINE

It was explained on page 19 why it is necessary to write spaces, newlines, etc. in a special manner within a caption. The same problem arises when we wish to write these SYMBOLS in assignment instructions, conditions, etc. and the same special conventions are used. For example,

```
i = '$'
```

assigns the symbol 'space' to the variable i.

A simple method of reading the next 'useful' symbol in data (i.e. disregarding spaces and newlines) would be:-

```
    integer i
i:  read symbol (i)
    -> i if i = '$' or i = '#'
    ..............
    ..............
```

OUTPUT OF SYMBOLS

The instructions

```
(  caption ..........  )
(  space               )
(  spaces  (  )        )
(  newline             )
(  newlines  (  )      )
```

are available for output of symbols, as described on page 19. There is also an instruction print symbol:-

Instruction	Meaning
print symbol ('*')	print out the symbol *
print symbol (i)	print out the symbol currently held in the integer i

Note (1) The first instruction above could equally well be written:-

```
caption *
```

(2) print symbol (i) is useful when we do not know in advance what symbol is going to be stored in the integer i.

RELATIONSHIP BETWEEN SYMBOLS AND INTEGERS

Symbols are stored in integer variables, and in fact each symbol has a numerical value to which it corresponds. However, since this correspondence may vary between compilers, programmers are advised not to make use of this fact. On the other hand, all the compilers are arranged so that 'A' has a value one less than 'B', which is one less than 'C', etc., thus preserving the natural dictionary ordering. The same is true of 'a', 'b',..... etc. and of '0', '1'..... etc.

Example

To test whether the next symbol on the data tape is a lower case letter in the first half of the alphabet we could write:-

```
read symbol (i)
if 'a' ≤ i ≤ 'm' then ..............
```

EXAMPLE OF A PROGRAM TO COUNT SYMBOLS

Suppose that we wish to read in a sequence of symbols as far as the first full stop, and print out the percentage which are capital E's. The program required is almost identical to that used on page 25 to print the average of a list of numbers.

```
        begin ; comment to give percentage occurrence of letter E
        integer n, Number of Es, x
        n=0 ; Number of Es = 0
10:     read symbol(x)
        ->11 if x = '.'
        if x = 'E' then Number of Es = Number of Es + 1
        n = n +1
        -> 10
11:     newline
--      caption percentage $ of $ E's $ =
        print(100*Number of Es/n,2,1)
        end of program
```

For comparison purposes, the program from page 25 is reprinted below:-

```
        begin ;  comment to give average of a list of numbers
        integer n
        real total, x
        n = 0 ;  total = 0
10:     read (x)
        -> 11 if x = -1
        total = total + x
        n = n + 1
        -> 10
11:     newline
--      caption Average $ =
        print (total/n,3,5)
        end of program
```

CHAPTER 4 : EXPRESSIONS, CYCLES, FURTHER ARRAYS.

Arithmetic expressions.

Permanent functions.

Further assignment instructions.

Cycles.

Examples (i) Print the average of list of numbers.

(ii) Counting symbols.

Switch labels.

Multi-suffix arrays.

Example Summary of Examination results.

ARITHMETIC EXPRESSIONS

There are many places in a program where we have to write an arithmetic expression (e.g. on the right of an assignment instruction or in a print instruction). The simplest form of expression is a single variable or numerical constant. More generally, an expression consists of variables, constants and functions, connected together by mathematical symbols. The method of writing constants is given below; variables have already been described - (pages 16 - 18) and functions will be deferred until page 40.

constant	meaning	note
37) 0.25) 2.374)	obvious	0.25 and .25 are equally valid.
1α3	1000(i.e. 1×10^3)	(i) This is called the 'floating point' form for a constant.
1.732α-2	0.01732 (i.e. 1.732×10^{-2})	(ii)The number after α must be an integer constant.
π	3.14159....	
½	0.5	½ is one symbol. Other fractions must be written as quotients(i.e. 1/3)

Mathematical Symbol	Meaning
+	addition
−	subtraction
*	multiplication
/	division
†	raise to a power (a†3 means a^3)
2	squaring
|	used in pairs as modulus signs.

Notes

(1) In normal mathematical notation we often omit the multiplication sign (e.g. ab for a*b). In Atlas Autocode we write the * sign, otherwise the compiler will look for a variable with the name 'ab'. The * can be omitted where a constant is followed by a variable (e.g. 3.5*y and 3.5y are equivalent)

(2) a^2 and a†2 are equivalent. All other powers must be written with †.

(3) In manuscript, the symbol † is usually written ↑.

PRECEDENCE OF OPERATORS (+ - * / ↑)

There may be some uncertainty about the meaning of an expression such as a*b+c. Do we carry out the multiplication first, giving (a*b)+c, or the addition first giving a*(b+c) ?

In the absence of brackets, we have the rule that, of two adjacent operators (like * and + above), the operator of higher precedence in the table below is to be carried out first.

(1) ↑ (highest precedence)

(2) * or /

(3) + or - (equal lowest precedence)

Where two adjacent operators are of equal precedence by the above table, the one appearing to the left (in the expression to be evaluated) is carried out first.

Notes (1) The multiplication operator between a constant and a variable has the same precedence whether written explicitly or 'implied' (see note 1 on previous page)

(2) The symbol ^{2} is treated as equivalent to the pair of symbols ↑ 2 and precedence is given accordingly.

(3) If we wish to over-ride the above rules, we must use brackets as in normal mathematical notation.

(4) When in doubt it is wise to insert brackets for safety and clarity.

(5) The 'left-hand precedence' between + and - agrees with normal usage.

e.g. By a-b+c we mean (a-b)+c and not a-(b+c)

Examples	Meaning
a/b*c	$\frac{a}{b} \times c$
a/(b*c)	$\frac{a}{bc}$
a↑b*c	$a^b \times c$
a↑(b*c)	a^{bc}

Note The first two examples show that it is necessary to bracket denominators containing more than one term. A common mistake is to write a/2b when a/(2b) is intended.

FUNCTIONS

In addition to variables and constants, functions may also be included within expressions. The basic functions available are:-

real function	meaning	note
sin(x))	as in	
cos(x))	elementary	x in radians
tan(x))	trigonometry	
sq rt (x)	$+\sqrt{x}$	
log (x)	logarithm of x	to base e
exp (x)	e^{x}	
mod(x)	modulus of x (i.e. Absolute value of x)	mod(-3.7)=3.7; mod(3.7)=3.7 Can be written |x|, but see note below.
arctan (x,y)	$\tan^{-1}(y/x)$	In radians. Value is in 1st or 4th quadrant if x>0 2nd or 3rd quadrant if x<0
radius (x,y)	$+\sqrt{x^2+y^2}$	
frac pt (x)	fractional part of x	frac pt(3.73)=0.73 frac pt (-3.73)=0.27

integer function	meaning	note
int(x)	nearest integer to x	int (3.73)=4
int pt (x)	integral part of x	int pt (3.73)=3 int pt (-3.73)=-4
parity (n)	+1 if n is even -1 if n is odd	n must be an integer variable.

Notes (1) The first group of functions (down to frac pt) all produce a number of type real, which can only be assigned to a real variable. The last three produce a number of type integer.

(2) In particular, note that the function mod(x) produces a number of type real, irrespective of whether x is of type real or integer. On the other hand, a pair of modulus signs will give the same numerical value as the modulus function, without altering the type. (i.e. integer remains integer). Hence, if n is an integer,

n = |n| is valid

n = mod(n) will be faulted.

(3) The above functions are all understood by the compiler before the program is read in. The method used to define additional functions, if required, will be given later (page. 70)

(4) As the names sin, log etc. are already in use, they should not be used by the programmer in any of his declarations.

INTEGER AND REAL EXPRESSIONS

The differences between integer expressions and real expressions lie not so much in the values of the expressions as in how they are constructed and used.

(1) Any expression consisting entirely of integer variables, integer constants and integer functions is called an INTEGER EXPRESSION. Any other expression is called a REAL EXPRESSION.

(2) We shall meet a number of places where an integer expression is required. In these cases, a real expression is not allowed, not even one whose value may actually work out to be an integer. On the other hand, wherever a real expression is expected, an integer expression will do instead.

Integer Expressions

The main cases where an integer expression is compulsary are:-

(1) When assigning a value to an integer variable.
(2) As the suffix of an array element.
(3) As the power to which a number is to be raised. (Raising to a power is done by repeated multiplication).
(4) In cycle instructions (page 43)

Examples Suppose we have declared

```
integer i
real x,y
real array d(0:10)
```

Example of (1) Although x=i is permitted, i=x will cause a fault signal because x is a real expression.

Example of (2) If we have previously set i=3; x=3 then d(i*i) refers to d(9) but d(i*x) is illegal.

Example of (3) x↑3 and x↑(i+1) are legal expressions but x↑y is not.

Symbols as Integer Expressions

A symbol written between 'quotation marks' is a possible form of integer expression, but should only be used with caution. A simple and safe example of this will be found on Page 47.

FURTHER ASSIGNMENT INSTRUCTIONS

The general form of an assignment is either

(1) assign the value of an INTEGER expression to an INTEGER variable or

(2) assign the value of a REAL or INTEGER expression to a REAL variable.

Examples Suppose we have declared real a,b,c,x
integer i,j,k

then possible instructions are:-

```
x = (-b + sq rt (b*b - 4a*c))/(2a)
i = int(j/k) + j*j
a = log (1 + cos (2π x)) + 3.74b
x = i
```

Notes (1) As already explained i = x will cause a fault signal because x is real. If required, we can write: i = int(x)

(2) On the Atlas versions of the language we can, without a fault signal, assign to an integer variable any integer expression. The responsiblility for ensuring that the expression will work out to be an integer, lies with the programmer. (division or raising to a negative power are possible causes of non-integer results). See the second example above.

(3) When using KDF9 there is the further restriction that, whenever an integer expression is compulsary, each stage in the evaluation of the expression must yield an integer result. Referring to the rules of precedence for operators, we see that, with the previous declarations,

`j = i*(i+1)/2` will always work but that

`j = (i+1)/2*i` , while having the same result as the previous line when (i+1) is even, will be faulted if (i+1) is odd.

(4) It is possible to use expressions inside larger expressions; in particular we can have a function of a function as in the third example.

CYCLES

Suppose that we wish to carry out a certain part of a program 10 times with an integer i taking the values 1,3,5.....,19 on successive occasions.

Clearly it is necessary to indicate

(1) the sequence of values which the integer is to assume, and

(2) the beginning and end of that section of program which is to be repeated.

We achieve this by writing:-

```
cycle i = 1,2,19

(  orders making  )
(  up the section )
(  of program     )
(  to be repeated )

repeat
......
```

NOTES

(1) The sequence of values i is to assume is indicated by writing 'cycle i = ' followed by 3 integer expressions giving the initial value, the increment, and the final value.

(2) The control variable i must have been previously declared to be an integer or an element of an integer array.

(3) The separator repeat is used to indicate the end of the section of program to be repeated.

(4) The expressions for the initial and final values and the increment are evaluated on first reaching the cycle. The number of times the cycle is to be executed is

$$1 + (\text{final value} - \text{initial value})/\text{increment},$$

and a fault is signalled if this is not a positive integer.

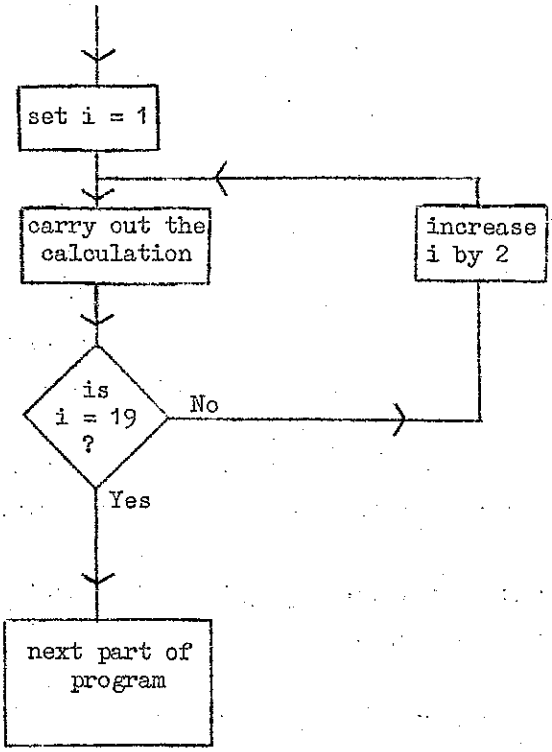
Instead of using cycle and repeat in the above example the same result could have been obtained by using jump instructions. For example:-

```
        i = 1

1: (    orders making    )
   (    up section       )
   (    of program       )
   (    to be repeated   )

        -> 2 if i = 19
        i = i + 2
        -> 1
2:      .......
```

The same flow diagram serves for both methods of writing this program:-



NESTING OF CYCLES

Cycles may be nested to any depth. Each cycle must have exactly one associated repeat.

Example

```
cycle i = 1,1,4
read (A(i))
newline
  cycle j = 1,1,i
  print (A(i)↑j,2,0)
  spaces (2)
  repeat ; comment this refers to cycle j =
repeat   ; comment this refers to cycle i =
```

Supplied with the data

0 1 2 3

the output would be

```
0
1   1
2   4   8
3   9   27   81
```

NOTE

Within cycles, either single or nested, the control variable(s) may be used for two distinct purposes:-
(a) to count the number of times the cycle has been executed, and
(b) to vary systematically quantities occurring in arithemetic expressions. (e.g. the integer i in the above example both counts the number of times the outer cycle is executed, and also enables us to operate on different array elements on each occasion).