University of Edinburgh

PROGRAMMING IN ATLAS AUTOCODE

COMPUTER UNIT REPORT No. 1

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

3rd March 1964.

PREFACE

This manual is intended to serve as an introduction to the Atlas Autocode programming language, as implemented on the Atlas Computer at Manchester University. It is partly based on courses of lectures given at Edinburgh University.

No mathematical knowledge or previous experience of computers is assumed, but for a complete beginner we recommend the following programme for a first reading:- pages 1-21, 24-27, 30-39, 41-43, 50-54, 60-65. 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.

Certain sections of the manual are marked (***); these refer to facilities which are not yet available. In this connection, pages 28a. and 40a. have been inserted in the manual to cover facilities introduced since we prepared the original draft.

We understand that some of the (***) facilities may now be deferred until a new and faster version of the compiler is written, and that this new version will exclude half-word operations. Half-word operations will, however, continue to be available to those who are prepared to work with the present compiler.

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

We should like to express our thanks to Mr. Peter Keeping who provided the basis of the material for the section on job descriptions and fault finding, and also provided the vital liason with the staff of the Manchester University Computing Laboratory; also to our colleagues Mr. Sidney Michaelson and Mr. Harry Whitfield for their constructive criticism. Our debt to the authors of the language, Mr. Brooker and Dr. Rohl, is obvious.

Finally, we should like to acknowledge the cheerful help given by Miss Susan Wake, who typed the bulk of the manuscript.

P D Schofield   M.R. Osborne

P.D. SCHOFIELD   M.R. OSBORNE

3rd March, 1964.

TABLE OF CONTENTS

PROGRAMMING IN ATLAS AUTOCODE.

These notes are intended for a complete beginner who wishes to learn to write programs to be run on the Atlas Computer. Such a program can be written in one of several languages: the one described here is ATLAS AUTOCODE.

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

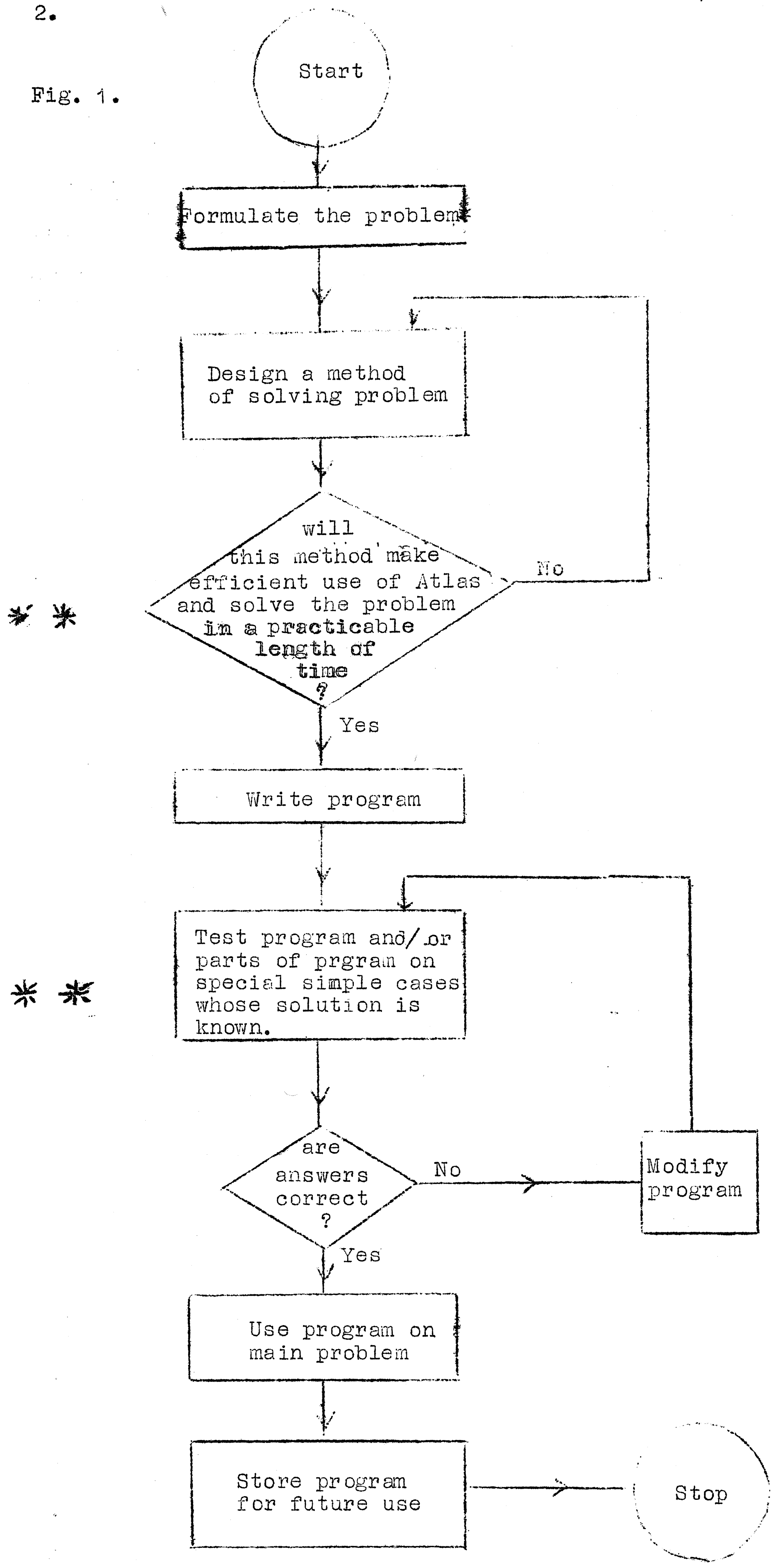
(b) Write a program describing the method already chosen.

Although these notes are primarily concerned with 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. Before passing on to describe the computer and the Atlas Autocode language, two general suggestions can be made about this planning stage.

Firstly, the staff of the Computer Unit will be pleased to give advice. Secondly, it often pays to draw a "flow diagram" to help plan the logical connections between different parts of the program. The customary layout of a flow diagram is given overleaf, where we show the stages through which the human programmer must pass. The same type of diagram is useful to describe the stages of a computation.

Note the two sections marked **.

Fig. 1.


Start
Formulate the problem
Design a method of solving problem
will this method make efficient use of Atlas and solve the problem in a practicable length of time ?
No
Yes
* *
Write program
Test program and/or parts of prgram on special simple cases whose solution is known.
* *
are answers correct ?
No
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:-

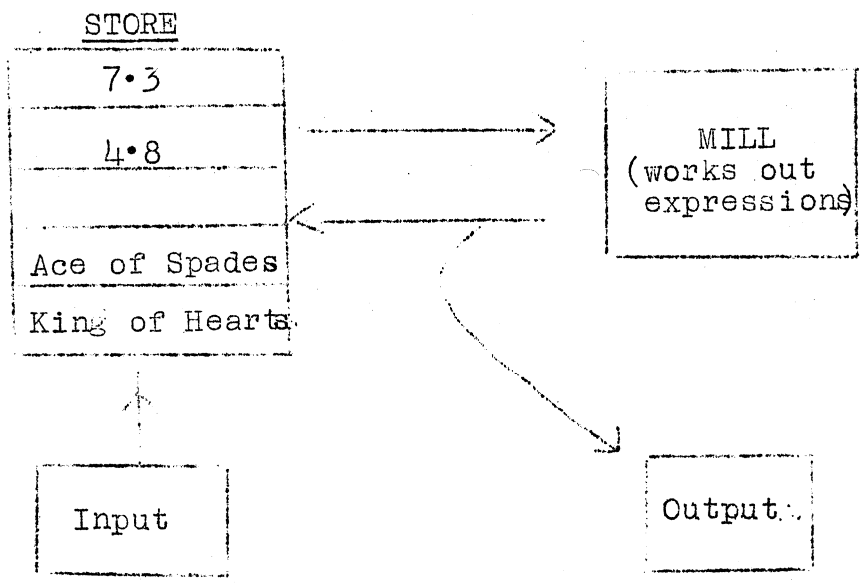


Fig. 2.

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. (i) copy the first two numbers from the store and multiply them.

(ii) 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.

Putting information in and out of the store works in a manner similar to that of a tape recorder. When withdrawing, we make a copy of the contents of a location, so that the original information is still there, and can be used again as often as required. When inserting, the previous contents of that location are destroyed.

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

Documents: When we wish to use the computer, we normally need to feed in two "documents" (1) Program

(2) Data

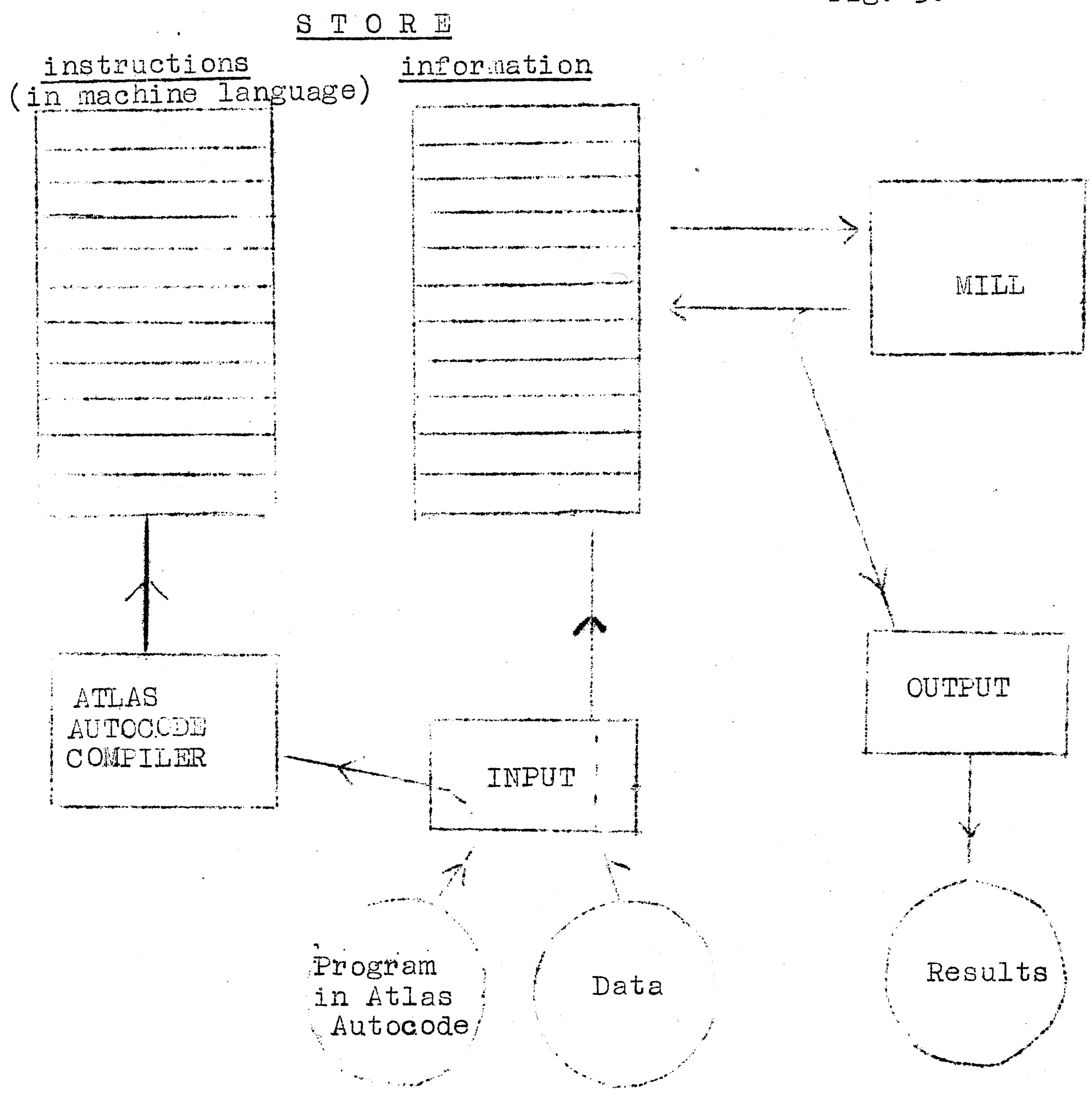
The difference between the two is shown by the two examples below:-

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 verious operations. These are kept in the store in a code or "language" which is not readily comprehensible to the human 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, a set of rules for translating into its own language.

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

Fig. 3.



The sequence of events will 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.

ATLAS AUTOCODE.

The Atlas Autocode language is better equiped for dealing with numbers than other types of information. For this reason, the basic principles of the language will be explained in terms of very elementary calculations with numbers.

NAMES.

Before a number 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 or 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. (i) a2c is not permitted as a letter follows a digit.

(ii) The compiler completely disregards all spaces (and underlined spaces) in the program. Spaces may thus be used to improve legibility of program.

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.

<u>Note</u> (1) Generally speaking, a name allocated to a location will remain fixed throughout the program. However, the contents will vary whenever a 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:-

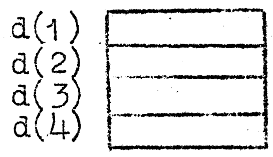
(a) An integer variable must be a whole number. A real variable may also be a number such as 73.4827, with up to 11 significant figures.

(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.

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.

<u>array</u> d(1:4) set aside 4 consecutive locations with names:-



<u>array</u> e(1:7), f, g (0:4) set aside {7 locations for e(1) to e(7)
{5 " " f(0) to f(4)
{5 " " g(0) to g(4)

Notes (1) The declaration array .... automatically prepares for real numbers. If integers are intended, we must write integer array ...

(2) Note the difference between this and the declaration

real d4 which only gives one location:-

d4 $\boxed{}$

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, 2-dimensional arrays and complex numbers 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 (e.g. real and array).

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 Instruction	Meaning
read (a)	read in next number in the data and put it in 'a'.
read (b, c3, d(4))	read in the next 3 numbers in the data and put them in b, c3 and d(4).

Output Instructions.	Meaning
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 answer with 3 figures before the decimal point and one after. (The figures 3 and 1 can, of course, be varied).
spaces (7)	output teleprinter is to leave 7 blank spaces.
newline	output teleprinter is to go to the start of a new line.

ASSIGNMENT INSTRUCTIONS.

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') Thus

a	7·32	becomes	a	13·0
b	10·0		b	10·0
c	3·0		c	3·0

a = 2a + 1	copy contents of 'a', double it, add 1 and replace answer in 'a'

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.

DELIMITERS

These are the punctuation signs of the language and include:-

(a) The normal punctuation signs . , : ; ( )

(b) The mathematical symbols $+ \quad - \quad = \neq < \leq > \geq \quad / * \uparrow | \, |$

(c) The special underlined words:- real if then array comment caption and a few others

Note: (1) All underlined words are delimiters. It would be possible, but highly confusing, to declare a name such as:- real (not underlined)

(2) Delimiters mostly occur as part of either declarations or instructions.

(3) The delimiters stop and return are complete instructions and can therefore appear on lines by themselves. (stop instructs the machine to terminate the calculation: the use of return will be explained later).

(4) A few other delimiters (begin end repeat end of program) also appear on lines by themselves where they are used to separate out groups of instructions or declarations.

(5) As stated on page 8, each declaration and instruction is usually written on a line by itself. However, more than one declaration or instruction can be written on the same line provided they are separated by the delimiter ; . A declaration or instruction can be made to extend over more than one line by terminating all but the final line with the delimiter c.

(6) The delimiter comment, written at the beginning of a line, causes the compiler to ignore the rest of that line. It is used to insert explanatory notes, which must not contain a semi-colon, into the program, for the benefit of the human reader.

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

(7) The delimiter caption written at the beginning of a line, is used in a rather similar manner to make the OUTPUT more readable. It causes the output teleprinter to print out the set of characters following caption.

Example
```
caption   answer =
print (answer, 3, 5)
```

Spaces are ignored by the compiler, and the delimiter ; marks the end of the caption instruction. Spaces and ; can be inserted into a caption by writing $\cancel{s}$ and $\cancel{;}$ respectively.

BLOCKS

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

```
        begin
        .....  )
        .....  )  declarations
        .....  )

        ......  )
        ......  )
        ......  )
        end
```

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

Example

```
begin
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.

EXPRESSIONS

There are many places in a program where we have to write an expression (e.g. on the right of an assignment instruction or in a print instruction). An expression consists of variables, constants and functions, connected together by mathematical symbols.

(a)
The use of variables has already been described (pages 6-8).

(b)

constant	meaning	note
37) 0.25) -2.374)	obvious	0.25 and .25 are equally valid.
$1\alpha3$	1000(i.e. 1×10^3)	(i) This is called the 'floating point' form for a constant.
$1.732\alpha{-2}$	0.01732 (i.e. 1.732×10^{-2})	(ii)The number after α must be an integer.
π	3.14159....	
$\frac{1}{2}$	0.5	$\frac{1}{2}$ is one symbol. Other fractions must be written as quotients(i.e. 1/3)

(c)

function	meaning	note

sin(x))
cos(x))
tan(x))
	as in elementary trigonometry	x in radians

sq rt (x)	$+\sqrt{x}$	
log (x)	logarithm of x	to base e.
exp (x)	e^x	
mod (x)	modulus of x	Can be written $\lvert x \rvert$
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
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) 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 41).

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

(d)

Mathematical Symbol	Meaning
+	addition
–	subtraction
*	multiplication
/	division
↑	raise to a power (a↑3 means a^3)
^{2}	squaring
|......|	modulus of. Alternative to mod (......)

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 only case in which we can omit the * is where a constant is followed by a variable (e.g. 3.5y)

(2) a^2 and a↑2 are alternative forms for a*a. All other powers must be written with ↑.

PRECEDENCE OF OPERATORS

As a result of the rule in Note (1) above, there is uncertainty about the meaning of an expression such as a*b+c. Do we carry out the * first, giving (a*b)+c, or the addition first, giving a*(b+c) ?

In Atlas Autocode we lay down the rule that, of two adjacent operators (like * and + above), the one appearing higher in the table below is to be carried out first.

(i) ↑

(ii) * or /

(iii) + or –

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

(2) When any doubt could arise, it is wise to insert brackets for safety and clarity.

(3) 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}

INTEGER AND REAL EXPRESSIONS

The difference 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 (such as int (x) or parity (n)), 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.

Example We can use an integer expression in place of an integer when referring to an element of an array. Hence if we write

```
integer i
real x
array d(1:10)
i = 3
x = 3
```

then d(i*i) refers to d(9)
but d(i*x) is illegal because i*x is a real expression.

WARNING. Caution is neededwith expressions of the form "a↑b". (a and b may be replaced by expressions). If b is an integer constant or integer variable, all is well as the calculation is done by repeated multiplication. In all other cases, the calculation is performed by taking the logarithm of a: it is therefore important to ensure that a does not assume a negative value.

Example. Suppose that a is negative and that we wish to evaluate a^{j+i} where j is an integer. The instruction y = a ↑(j+i) will be faulted for the reason given above. The simplest solution is to declare another integer k and write:- k = j+i ; y = a↑ k

FURTHER ASSIGNMENT INSTRUCTIONS

The general form of an assignment is either
(i) assign an integer expression to an integer variable or
(ii) assign 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) Although x = i is permitted, i = x will cause a fault signal, because x is real and i is an integer. If required, we can write:- i = int(x)

(2) On the other hand, we can, without a fault signal, assign to an integer variable any integer expression. The responsibility 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) It is possible to use expressions inside larger expressions; in particular we can have a function of a function as in the third example.

LABELS

Any instruction (or a delimiter which appears on a line by itself) can be labelled by writing a positive integer and a colon on the left. (Example in next section).

JUMP INSTRUCTIONS

Normally, instructions are obeyed in the order in which they are written. To make the machine jump to a new position in the program, we write → followed by a label, giving the place to which we wish to jump.

Example

```
    → 10
    .....
    .....
10: a = b + c
    c = sin (x)
    → 12
    .....
    .....
12: end of program
```

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) Labels are local to the block and we cannot jump outside the block (i.e. before begin or after end).

(3) The delimiter stop written on a line by itself instructs the machine to stop the program, and is thus equivalent to the use of "→ 12" in the example above.

CONDITIONAL INSTRUCTIONS.

Assignment and jump instructions may be made subject to a condition, using the delimiters if or unless.

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

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

Example

```
if x = 0 then a = b +c
unless a> b + 2 then → 27
```

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

$= \neq > \geq < \leq$

(3) Stop can be made conditional.
e.g. if n >100 then stop

MORE COMPLICATED CONDITIONS.

These may be formed by

(1) Writing 3 expressions separated by 2 relations

e.g. <u>if</u> $0 < x < 1$ <u>then</u>.......

or <u>if</u> $a + b < 3x < \sin(y)$ <u>then</u>.......

(2) by the use of an unlimited number of <u>and</u> delimiters, with the obvious significance.

e.g. <u>if</u> $x>0$ <u>and</u> $y = 0$ <u>and</u> $z = 2$ <u>then</u>.....

(3) by a similar use of a succession of <u>or</u> delimiters

e.g. <u>if</u> $x>0$ <u>or</u> $y = 0$ <u>or</u> $z = 2$ <u>then</u>....

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

e.g. <u>if</u> ($x>0$ <u>or</u> $y = 0$) <u>and</u> $z = 2$ <u>then</u>...

EXAMPLE OF A SIMPLE PROGRAM

Suppose we want to read in a list of positive numbers and print out their mean. 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:-

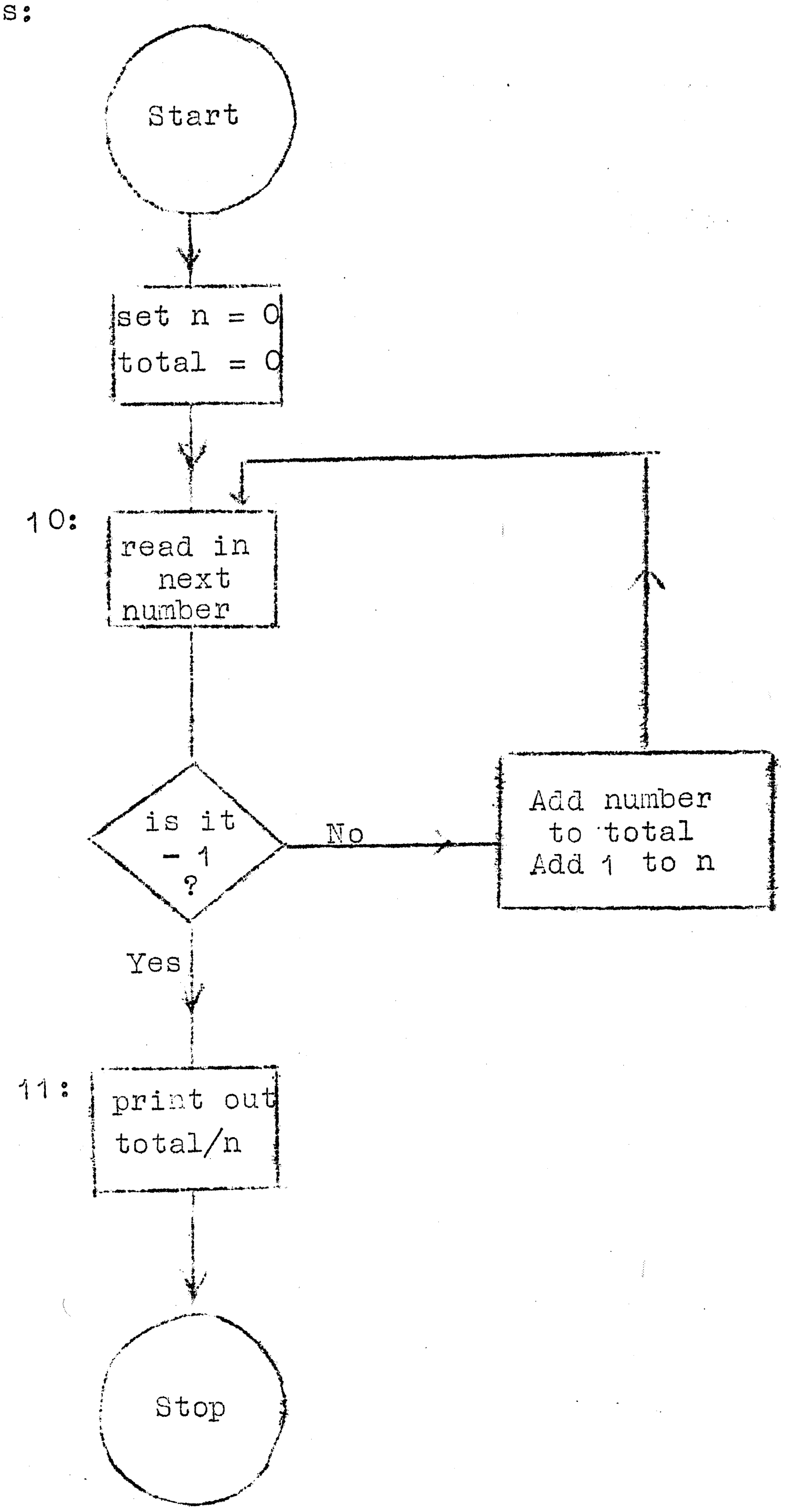
(1) a place in which to put the numbers as they are read in.

(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:



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
    print (total/n, 3,5)
    end of program
```

CYCLE INSTRUCTIONS

```
cycle  i = 1, 3, 3n + 1
a(i) = 0
......
y = y + b(i)
repeat
z = x + y
```

In the example, the group of instructions from "a(i) = 0" to "y = y +b(i)" are traversed n times, i taking in succession the values 1,4,7 .....(3n +1). On reaching "repeat" the machine tests to see if i has reached its final value of (3n + 1).

Then (i) If i $\neq$ 3n+1, it adds 3 to i and jumps back to "a(i) = 0"

or (ii) If i = 3n+1, it continues on with "z = x + y"

Notes (1) The L.H.S. of the cycle instruction must be a variable declared to be of type integer. On the R.H.S. we must have three integer expressions, of which simple integers like 1 and 3 are special cases.

(2) Note that the cycle integer i is used for two purposes
  (a) to count the number of times round the cycle.
  (b) to make the instructions act on different array elements each time round.

There is no obligation to make use of the purpose (b).

(3) The three integer expressions are all evaluated on reaching the cycle instruction, and recorded as integers. Altering the value of n during succesive traverses will not affect them.

(4) The three integer expressions must, of course, have values such that the difference between the first and last is an exact multiple (a positive integer or zero) of the second.

(5) Although "cycle i = 0.2, 0.1, 0.5" is illegal, the required effect is achieved by

```
cycle i = 2, 1, 5
x = 0.1 i
```

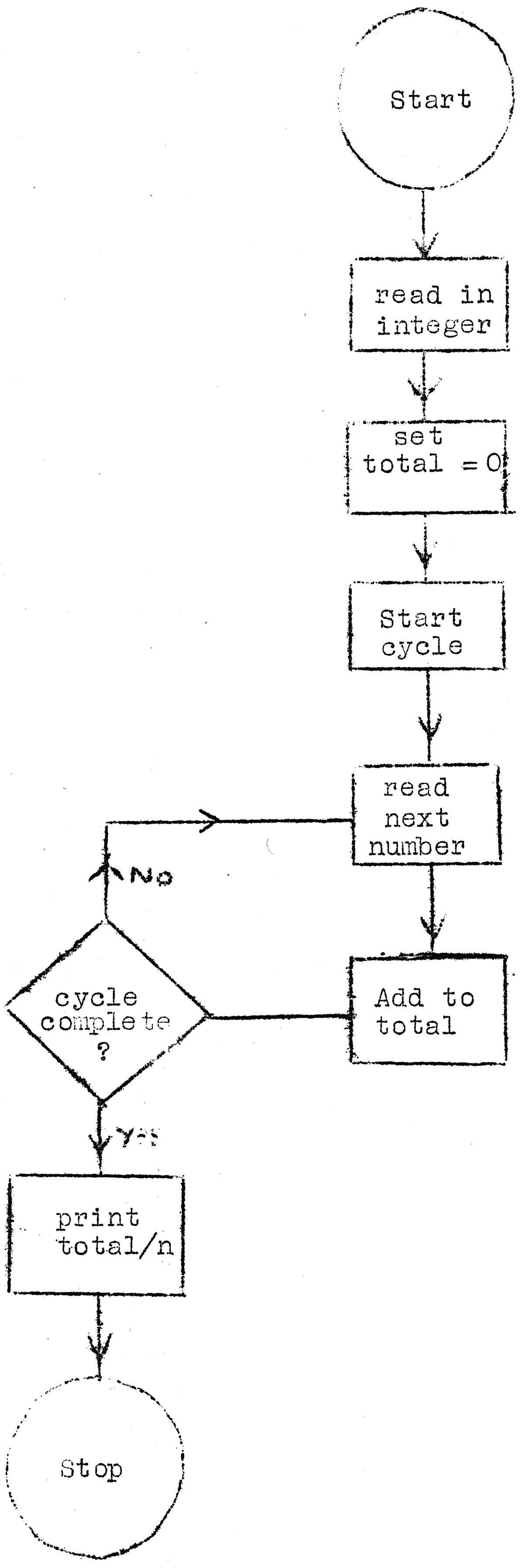
(6) Cycles may be nested to any depth. On the right, the first "repeat" refers back to "cycle j = 1, 1, i" and the second to "cycle i = 1, 1, 10"

```
cycle i = 1,1,10
cycle j = 1,1,i
.....
.....
repeat
repeat
```

(7) Each cycle instruction must have exactly one associated "repeat", which must be in the same block.

Example Using a cycle instruction, write an alternative version of the program for giving the mean of a list of numbers (pages 17-18).

Instead of using a -1 to terminate the list, we head the list with an integer indicating how many numbers are to follow.



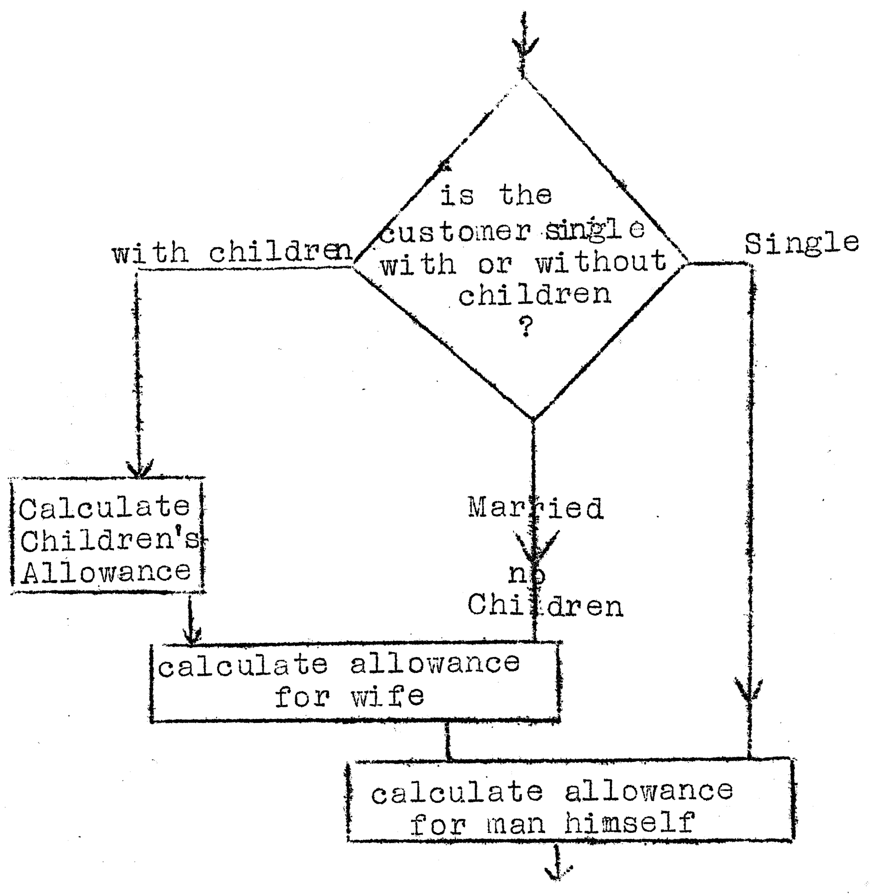
and the program is:-

```
begin
integer n, i
real total, x

read (n)
total = 0

cycle i = 1, 1, n
read (x)
total = total + x
repeat

newline
print (total/n, 3, 5)

end of program
```

MULTI-WAY JUMPS

(a) Conditional Labels

The point of the following examples is best made if we exclude the case of a man with children, but no wife.


is the customer single with or without children ?
with children
Single
Married no Children
Calculate Children's Allowance
calculate allowance for wife
calculate allowance for man himself

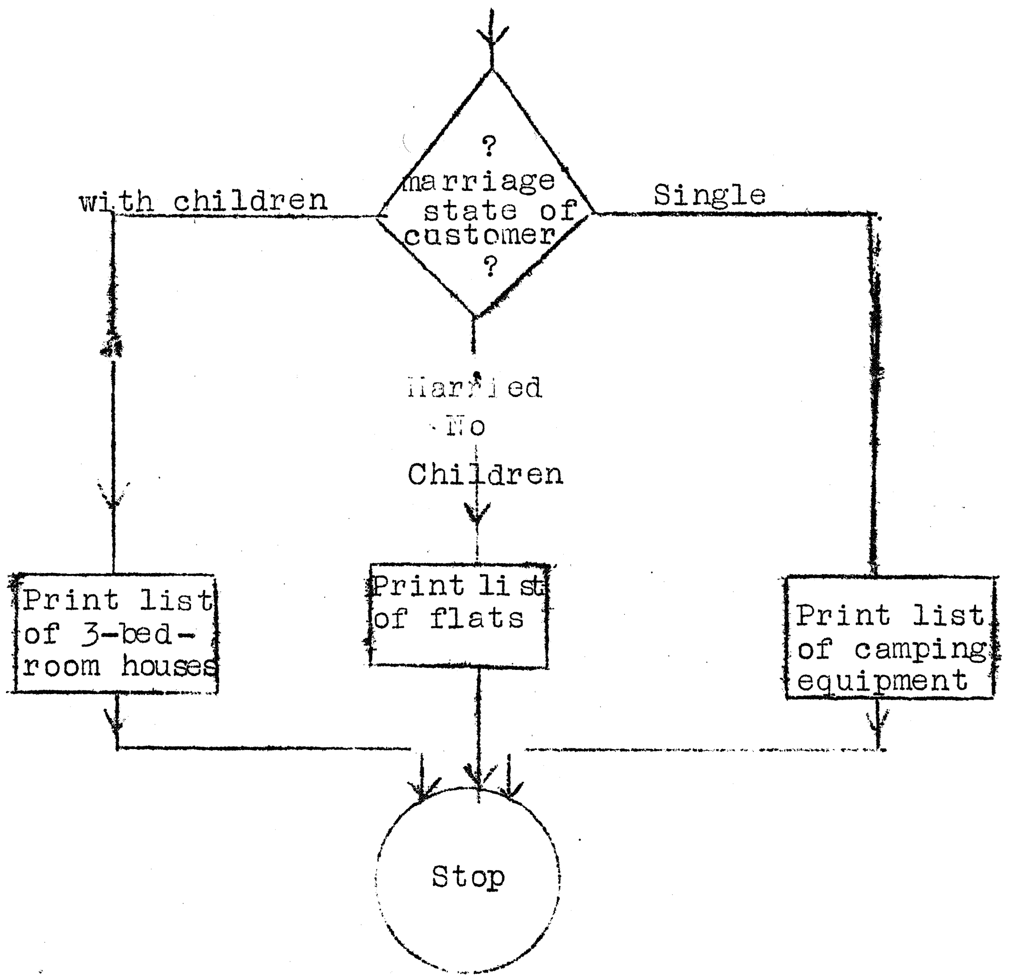
Suppose we have previously placed the number of wives (presumably 0 or 1) and children into variables names "wives" and "children". We can program the 3 way test as follows:-

```
test 4, 5, 6
4 case children >0:.....
                   .....
                   .....


5 case wives  >  0:.....


                 6:.....
```

On reaching the test instruction, the machine tests in turn the conditions at the labels listed (in this case three in number: 4, 5, 6) and jumps to the first one which is successful. As we have omitted a condition at 6, this is taken as a successful condition and the machine jumps here if all other tests fail.

Notes. Conditional labels, like simple labels, are local to the block.

The slight difference in the following case should be noted.



? marriage state of customer ?
with children
Single
Married No Children
Print list of 3-bed-room houses
Print list of flats
Print list of camping equipment
Stop

```
test  7,8,9
7 case children >0: .....
                    .....
                    .....
                     ->101
8 case wives    > 0: .....
                    .....
                    .....
                    -> 101

  9 :               .....
                    .....
                    .....
101 :               end
```

(b) Switch Labels

Another way of obtaining a multi-way jump is to declare a switch at the head of the block. The switch must be given a name ("A" in the case below) which must not clash with any name declared for any other purpose within that block.

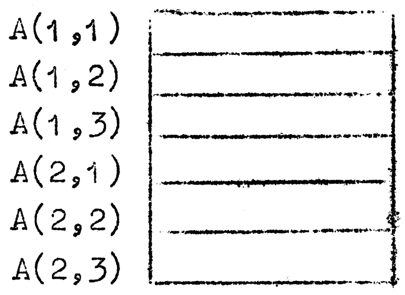
On reaching "$\rightarrow$ A(i)", the machine jumps to one of the four labels, depending on the current value of i. For example if i = 1 it will print the value of y, but if i = 2 it will take the square root first, and then print the answer.

```
switch A (-1:2)
          .....
          .....
A(0)  :   .....
A(-1) :   .....
          .....
          -> A(i)
A(2)  :y = sq rt (y)
A(1)  : print (y, 3, 4)
```

2-SUFFIX ARRAYS

On page 7, we introduced declarations of arrays with one suffix. In a similar manner we can declare arrays with two suffices.

declaration	meaning
array A(1:2, 1:3)	set aside 6 consecutive locations for real variables to be known as follows:
integer array A(1:2, 1:3)	as above, but giving integer variables.



Example

Suppose we wish to form a table giving the number of successes in 'A' level Mathematics, Physics, Latin and French, sub-divided into boys and girls. Let us store the numbers in integer variables A(i,j) as follows:-

	Maths	Physics	Latin	French
Boys	A(1,1)	A(1,2)	A(1,3)	A(1,4)
Girls	A(2,1)	A(2,2)	A(2,3)	A(2,4)

Here the first suffix gives the sex and the second the subject. Although we think of them as a rectangular array and also print them in this shape, the variables are stored in the machine row by row in consecutive locations.

To set all the first row to zero initially, we could write:

```
cycle j = 1, 1, 4
A(1,j) = 0
repeat
```

and then for the second row

```
cycle j = 1,1,4
A(2,j) = 0
repeat
```

It is easier to combine these two processes by means of a cycle within a cycle

```
cycle  i = 1 , 1 , 2
   cycle  j = 1 , 1 , 4
   A(i,j) = 0
   repeat
repeat
```

The same method will be used to print out the results in a rectangular table.

Data Suppose the data is supplied as follows:-

(1) An integer giving the number of results to analyse.

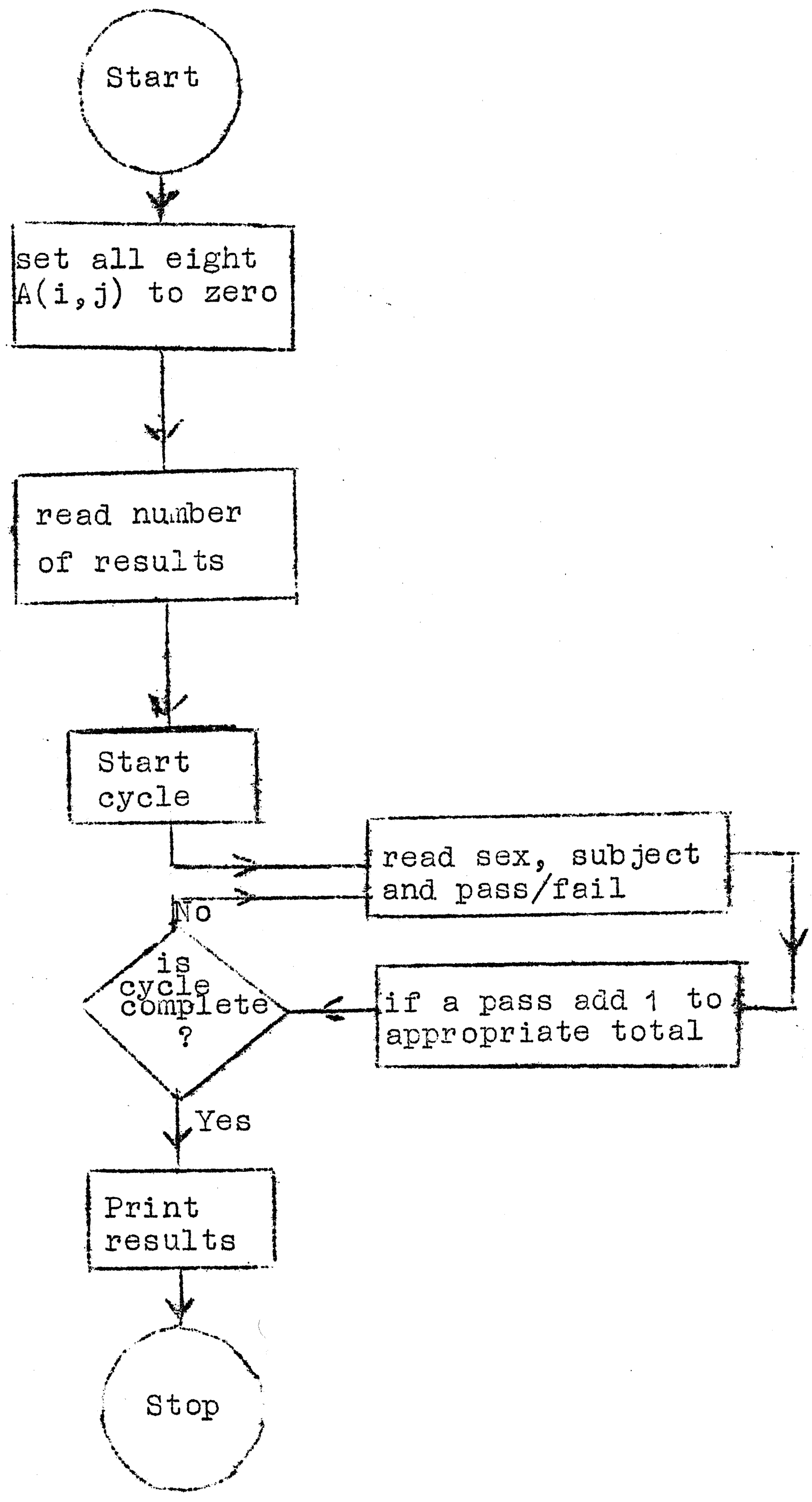
(2) Groups of three integers in which

the first indicates	sex	(1 for boy, 2 for girl)
the second "	subject	(1,2,3,4 as before)
the third indicates	success	(0 for fail, 1 for pass)

For example:-

data	meaning
999	999 results to follow
1 1 0	A boy has taken Maths and failed.
1 4 1	A boy has taken French and passed.
2 3 1	A girl has taken Latin and passed
etc.	

A possible flow diagram and program are given on the following pages.


Start
set all eight A(i,j) to zero
read number of results
Start cycle
read sex, subject and pass/fail
if a pass add 1 to appropriate total
is cycle complete ?
No
Yes
Print results
Stop

```
begin
integer  i,j,k,l,n
integer array A(1:2. 1:4)

cycle i = 1, 1, 2
   cycle j = 1, 1, 4
   A(i,j) = 0
   repeat
repeat

read (n)

cycle l = 1, 1, n
read (i,j,k)
if k = 1 then A(i,j) = A(i,j) + 1
repeat

newline

cycle i = 1, 1, 2
   cycle j = 1, 1, 4
   print(A(i,j), 3, 0)
   spaces (5)
   repeat
newline
repeat

end of program
```

Notes: (1) The inner cycles have been indented on the pages for clarity. This is quite permissible as spaces in the program are disregarded by the compiler.

(2) In order to achieve a rectangular layout of results, spaces (5) are put in the inner cycle, and the newline in the outer cycle.

ADDRESS RECOVERY FUNCTION

In the normal course of events, we do not know where in the store any declared variable is situated. All locations do, however, have a numerical address (like the house number in any one long street) and in the next section we shall need to find out the address of certain variables.

function	meaning
addr (x)	the numerical address of the location allocated to the variable x. (This is an integer function).

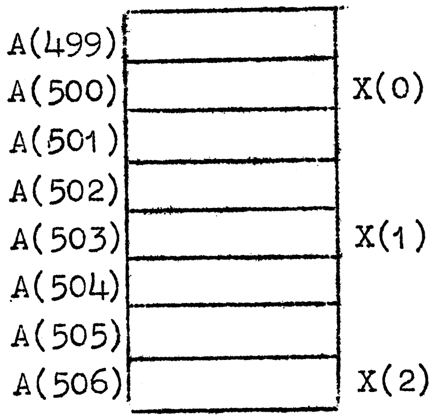
ARRAY FUNCTION DECLARATIONS.

This is a method of renaming storeage locations which have already been named by a declaration of type array or integer array. The old and new names can then be used a synonyms.

Suppose, for example, that we have already made the declaration "array A(1:1000)". Renaming is achieved by

declaration	meaning
array fn X(addr (A(500)), 3)	starting with the location given by "the address of A(500)", rename every third location X(0), X(1), X(2)... so that X(i) is at addr (A(500)) + 3i.

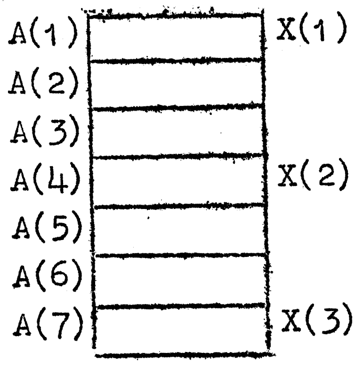
Note (1) The first parameter in the declaration gives the address of the starting loca-tion, the second the increment. Either may be an integer expression.



(2) Since the first parameter is an address, it usually involves the use of the function "addr".

(3) Array function declarations only rename locations already allocated by an initial declaration such as "array A(1:1000)". Hence, in the above example, X(166) corresponds to A(998), but the use of X(167) may produce unexpected results and should be avoided.

(4) array A(1:1000)
array fn X(addr (A(1)) -3, 3)
will give new names X(1).....X(333), where the first X is X(1) and not X(0). We have to write "addr (A(1)) -3" rather than "addr (A(-2))" because A(-2) does not exist.



__2nd MARCH, 1964__

A supplementary page has been inserted overleaf.

Since the date on which page 29 w s written, the compiler has been modified to accept declarations of arrays with an unlimited number of suffices.

Example

```
        array A(0:n, 0:n, 0:n)
        integer array B(1:5, 1:5, 1:20, 1:30)
```

Note It happens to be more economical in storage space if suffices with the smaller range of possible values are written first (as in the second example).

It is permissible to declare array functions having more than one increment parameter. For example

<u>array fn</u> X (s,p,q,r)

Here X (i,j,k) is the variable in location with address s+ip+jq+kr. Again X(0,0,0) is the variable stored in location with address s. If it is required to refer to this variable as X(1,1,1) then the appropriate declaration is

<u>array fn</u> X(s-p-q-r, p,q,r)

If the locations renamed by means of the array functions are to hold quantities of type integer, then the declaration is prefixed by <u>integer</u> in the usual way. For example

<u>integer array fn</u> X(s,p,q,r)

It must be stressed that the array function is a device for renaming locations in the store. It is important that all locations being renamed have already been declared. This ensures that storage allocated for other purposes is not overwritten.

The array function is the only method at present available for generating arrays with an arbitrary number of suffices. The following example indicates the manner of allocating storage for such an array. Consider the array with elements X(i,j,k) as in the last section, and assume that the maximum values assumed by i,j,k are u,v,w respectively. Then the address of the last location in the store used by the array will be s+up+vq+wr. In this case the appropriate declarations are

<u>array</u> A(0:up+vq+wr)

<u>array fn</u> X(addr (A(0)), p,q,r)

<u>ARRAY AND ARRAY FUNCTION DECLARATIONS</u>

As we have seen, there are two methods of declaring two-suffix arrays (commonly known as matrices). For example,

(A) by direct declaration	(B) by renaming
<u>array</u> mat (1:10,1:10)	<u>array</u> A(1:100) <u>array fn</u> mat(addr(A(1))-10-1,10,1)

However, to declare arrays with three or more suffices, the renaming method is the only one permitted at present.

The array function often offers a convenient means of manipulating the elements of two-suffix arrays. For example, if we follow the declaration in either (A) or (B) above by

```
array fn matT(addr(mat(1,1))-10-1,1,10)
```

we produce an array, normally called the transpose of the matrix "mat" with the property that

```
matT(i,j) = mat(j,i)
```

BLOCK STRUCTURE OF PROGRAMS

In many cases the exact storage requirements for the array declarations in the program are not known at the time of writing the program.

Suppose that we wish to write

```
    read (n)
    array A(1:n), B(1:20)
```

In order to bring the array declaration to its rightful place at the head of a block, whilst still ensuring that the value of n is read in beforehand, we use an inner and an outer block:-

```
begin
integer n
read (n)
    begin
    array A(1:n), B(1:20)
    ......
    ......
    ......
    end
end
```

Here the suffix bound for the array A in the inner block is obtained by means of the read instruction in the outer block.

It is sometimes convenient to regard a whole block as one compound instruction. With this view of the above inner block we see that the outer block has, indeed, the basic structure given on page 11, viz:

```
begin
1 declaration
2 instructions
end
```

Note (1) Blocks may be nested within one another to any depth.

(2) Blocks may not be made conditional.

GLOBAL AND LOCAL VARIABLES

It is important to appreciate the sphere of influence of the various declarations.

A declaration appears at the head of a block and normally remains valid throughout that block until cancelled by the end at the bottom of the block. It also remains in force upon descent to an inner block, UNLESS the same name is declared at the head of the inner block. In the latter case, the variable is held in abeyance while the machine is executing the inner block, coming into force again when the end of the inner block is reached.

Within any particular block the term local variable is used when referring to variables declared at the head of this block, and the term global variables when referring to variables declared in any exterior block. These points are illustrated by the following examples.

(i)

```
begin
real A
......
    begin
    real A,B,C
    .....
    .....
    end
print (A,2,2)
end
```

Here the name A refers to quite distinct variables in the inner block and the outer block. The print instruction will print the A of the outer block.

(ii)

```
begin
real A
.....
    begin
    real B,C
    A = B+C
    .....
    end
print (A,2,2)
end
```

Here A is global to the inner block since this time A has not been re-declared.

Notes (1) To communicate between blocks, global variables must be used, since local variables are cancelled upon exit from a block.

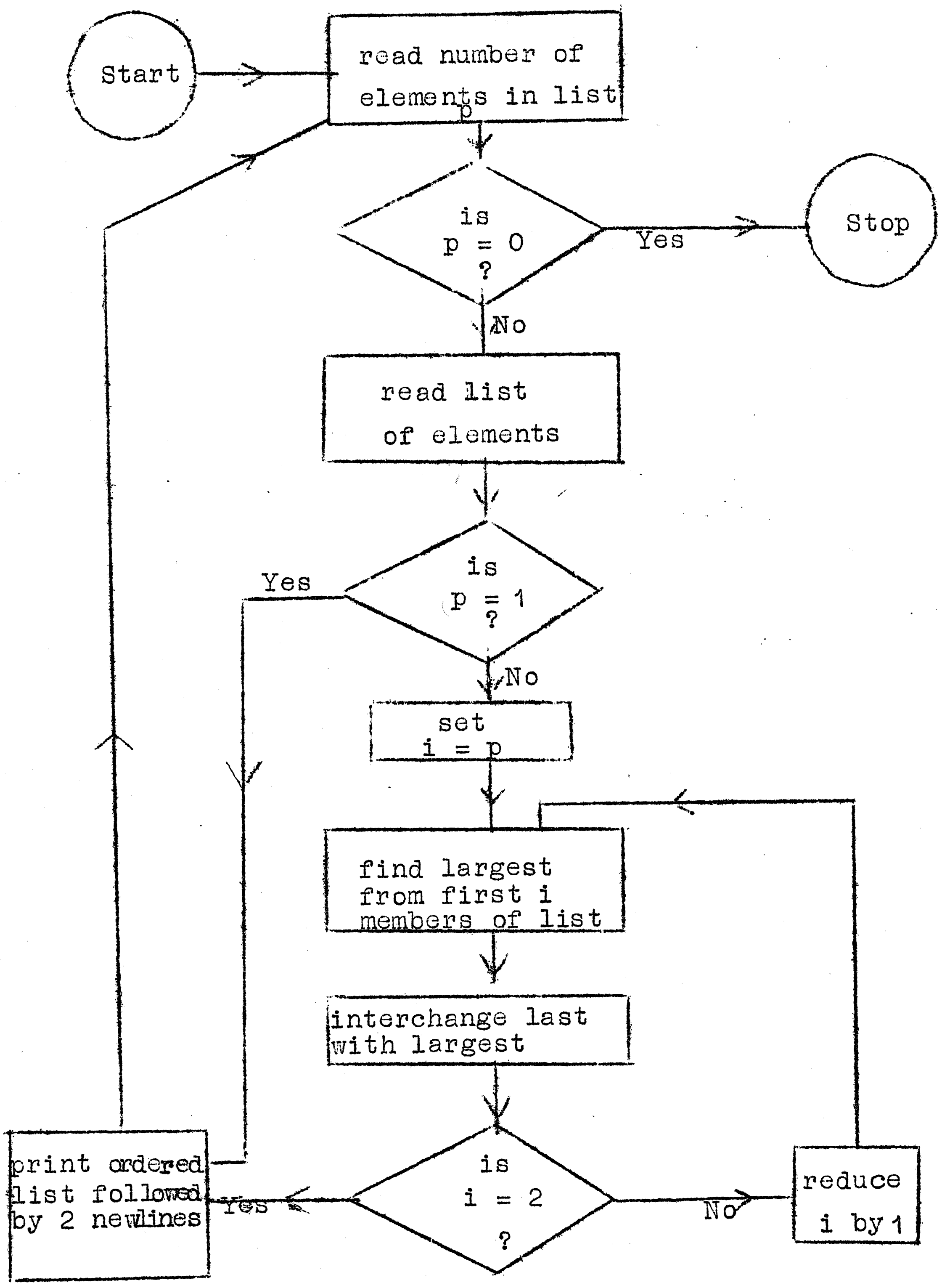
(2) Labels, unlike variables, are always local to a block. It is thus impossible to enter a block except through

the head of the block (which is just as well as the local declarations are written there). It is impossible to jump from one block to another.

EXAMPLE OF A COMPLETE PROGRAM

Read in lists of positive integers and print them out with each list sorted into increasing order of magnitude. Insert 2 blank lines to separate one list from the next.

On input of data, each list will be headed by an integer giving the number of elements in the list. After the last list, a single zero will be fed in, indicating an imaginary list of zero length.



Start
read number of elements in list p
is p = 0 ?
Yes
Stop
No
read list of elements
is p = 1 ?
Yes
No
set i = p
find largest from first i members of list
interchange last with largest
is i = 2 ?
Yes
No
reduce i by 1
print ordered list followed by 2 newlines

EXAMPLE OF A PROGRAM

```
   begin
   comment to arrange lists of positive integers in
   comment increasing order of magnitude
   integer p
 1:read(p)
   if p≤0 then stop
        begin
        integer i,j,AMAX,jMAX
        integer array A(1:p)
        cycle i=1,1,p
        read(A(i))
        repeat
        -> 3 if p=1
        cycle i=p,-1,2
        AMAX=0
             cycle j= 1,1,i
             if  AMAX ≥ A(j) then -> 2
             AMAX= A(j) ; jMAX=j
           2:repeat
        A(jMAX)=A(i)
        A(i)=AMAX
        repeat
      3:cycle i=1,1,p
        newline
        print(A(i),5,0)
        repeat
        end
    newline
    newline
    -> 1
    end of program
```

Notes (i) p is declared in the outer block, but can still be used in the inner block where it is a global variable. (Lines 8, 9, 12, 13, 22).

(ii) The label 1 is in the outer block, so the instruction "→ 1" must also be in the outer block.

(iii) The line numbers given on the left are not printed with a normal program, and should not appear on the program sheet, but the compiler does in fact count lines in this way and will print out the line number of any fault found in a program. In this connection, note that a line may contain more than one declaration or instruction. (e.g. line 17).

(iv) It is often convenient to label delimiters which appear on their own. (e.g. line 18).

OWN VARIABLES

Sometimes a block is entered several times in the same program. All local variables are cancelled upon exit from a block.

Upon re-entry, the same names will be declared again, but their previous contents will have been lost. If it is required to resurrect the old variables, together with their contents, the declarations must be prefixed with the delimiter <u>own</u>

```
e.g.  own real  a,b,c
      own integer array X(1:10)
```

In own array declarations, the suffix bounds must be integer constants (not the more general integer expressions which are permitted in ordinary array declarations).

(***) <u>Note</u> Own array declarations are not yet available

ROUTINES AND FUNCTIONS

There are many occasions on which it is necessary to perform an operation several times in different contexts within a program, or even in different programs (perhaps written by different people). A possible method of programming this operation as a unit is to write it as a block.

ROUTINES WITHOUT PARAMETERS

On page 30 we explained that a block can be regarded as one compound instruction. Instead of writing out the block in full every time it is required, we can give the block a name. We then simply write down this name (as a single instruction) every time we wish the block to be carried out. A whole block described by a name is called a ROUTINE.

There are three operations involved in incorporating a routine into a program
(1) Declaration
(2) Calling
(3) Description

As an example, we use a routine to interchange the values of two variables x and y.

(1) Declaration

```
routine spec interchange
```

meaning

the name "interchange" is to be the short title for a routine (a block of declarations and instructions) which will be described later

(2) Call

```
interchange
```

meaning

carry out the routine which has the short title "interchange".

(3) Description

```
routine interchange
integer z
z=x
x=y
y=z
 .
end
```

meaning

the routine "interchange" consists of the one declaration and three instructions given opposite.

Notes (1) A routine description has the same structure as a block except that begin is replaced by routine followed by its name.

(2) In the routine description, x and y are global variables.

(3) The first line of the description is always the same as the declaration, but with the delimiter spec omitted.

(4) A routine may be called in any block interior to the one in which it is declared (and described). In this way we can think of local and global routines, in just the same way as local and global variables.

(5) A routine call is an instruction and may be made conditional:

e.g. if p $\neq$ 10 then interchange

(6) Normally, instructions in the routine are obeyed in sequence until reaching end. If it is desired to stop the routine at any other point, the delimiter return may be used. This is equivalent to a jump to end. (Compare with stop which is equivalent to a jump to end of program). return, like stop, may be made conditional.

Example Interchange x and y and square them if they are both positive.

```
routine interchange and square
integer z
z = x; x = y; y = z
if x≤0 or y≤0 then return
x = x*x; y = y*y
end
```

Note A second return could be written immediately before end, but would be redundant.

STRUCTURE OF BLOCKS CONTAINING ROUTINES

Routine descriptions are placed at the end of the block in which they are declared. The general structure of a block can now be extended to :-

```
begin
.....   )                      declarations, including
.....   )                      declarations of routines.
.....   )

.....   )                      instructions, including
.....   )                      routine calls
.....   )

routine  )
.......  )                     routine descriptions, each
end      )                     having a block-like structure
routine  )                     of its own.
.......  )
end      )

end
```

ROUTINES WITH PARAMETERS

The previously described routine "interchange" will exchange the values of x and y, but will be of no use if we wish to interchange any other pair of variables.

In Atlas Autocode, to facilitate the use of the same routine in different contexts within a program, the user is permitted to write the routine using formal (or dummy) names for some or all of the variables global to it. In each call of the routine, these formal names are replaced by the appropriate actual names.

If formal names are used in the writing of a routine, then the following modifications must be made to the procedures for declaring, describing, and calling the routine.

(a) In declaring and in describing the routine its name must be followed by a bracketed list of the formal parameters used, together with a statement of their type.

(b) In calling the routine the name must be followed by a bracketed list of the actual parameters which are to replace the formal parameters on this occasion.
The designation "parameter" has been used above in anticipation of facilities which permit quantities other than names (for example elements of arrays and arithmetic expressions) to be passed on to routines.

Example 1

```
    integer a,b
    integer array A(1:10)
    routine spec interchange (integer name x,y)     Declaration
    ............
    ............
    interchange (a,b)                               Call 1
    ...........
    interchange (A(1) ,  A(2))                      Call 2
    ...........
    ...........
    routine interchange (integer name x,y)          Description
    integer z
    z = x; x = y; y = z
    end
```

Note (1) Here x and y are the formal parameters.

(2) The actual parameters must be placed in the same order as the formal parameters to which they correspond. In call 1, x is replaced by a and y by b. In call 2, x is replaced by A(1) and y by A(2).

(3) The statement of parameter type is omitted in calling the routine, but the compiler checks to see that the actual parameters listed are of the type indicated in the declaration.

The following example illustrates the use of a different kind of formal parameter.

Example 2

```
........
routine spec  FACT (integer name y, integer n)
............
............
FACT  (shriek, 10)
............
............
routine  FACT (integer name y, integer n)
integer i
y = 1
if n = 1 then return
cycle i = n, -1, 2
y = i* y
repeat
end
```

The difference in usage between the formal parameter types integer and integer name is profound and must be carefully noted.

(a) integer name. When a routine call is made the first action is to replace the formal integer name parameter at every place where it occurs within the routine body by the corresponding actual parameter given at the time of the call. This must have been declared in the usual way either as an integer variable or as an element of an integer array (see the two routine calls in example 1 above).

(b) integer. In this case the first action is the declaration of a new variable of type integer local to the routine. This variable is now filled with the value of the actual parameter which may be a general integer expression. This new local variable is now inserted at each occurence of the formal parameter in the routine body. As it is a local variable, its contents are lost on leaving the routine.

The parameter types real name and real are used in a similar manner. The actual parameter corresponding to the parameter type real name must have been declared as a real variable or as an element of an array. The actual parameter corresponding to the parameter type real may be a general (i.e. integer or real) arithmetic expression.

Parameters of type integer name and real name are said to be CALLED BY NAME.

Parameters of type integer or real are said to be CALLED BY VALUE.

PASSING ON ARRAYS TO ROUTINES

A parameter of type array name or integer array name is used in the same manner as those of type real name and integer name. We can describe a routine in terms of elements of an array with a formal(or dummy) name. In each call, we give the actual name of the array which is to be used in place of the dummy array on that particular occasion. The actual array must be one which has been declared directly by writing:-

```
          array ........      )
     or   integer array.....  )  and not one formed
```

by means of the array function or store map function.

Example The following routine will double the first 10 elements of any one-suffix array (assumed to start with suffix 1).

```
routine double (array name X)
integer i
cycle i = 1,1,10
X(i) = 2 X(i)
repeat
end
```

The routine is called by instructions such as:-

```
double (A)
double (B)
```

which will double A(1), A(2)....A(10) and B(1),....B(10).

Note The routine and the two arrays would, of course, have been previously declared in the usual manner.

Example In the next example it is assumed that a number of square arrays have been declared and a routine is required to print out certain sums of consecutive diagonal elements such as A(5,5) + A(6,6) + ...+A(10,10).

```
          routine trace (array name X, integer m,n)
          integer i
          real z
          z = 0
          cycle i = m,1 ,n
          z = z + X(i,i)
          repeat
          newline
          print (z, 5, 5)
          end
```

and instructions to call this routine might be

```
     trace (A , 5, 10)
     trace (B, 1, 50)
```

PASSING ON ARRAYS GENERATED BY ARRAY FUNCTIONS
(or Store Mapping Functions)

An array formed by renaming the elements of an array (as on pages 28 and 48) cannot be passed on as an array name parameter. Instead, we give, as actual parameters, the value of the address of the first element of the array and the values of the increments. These can be passed on as type integer. This information permits the array to be re-constituted within the routine by means of the array function.

Example

```
          routine Form (integer p,q,r,s)
          .......
          array fn V(p,q,r,s)
          .......
```

which might be called by:-

```
          Form(addr(A(1,1)), 10, 4, 1)
```

An alternative method of passing on the address is by use of the type addr in the routine heading. In this case, the actual parameter is the name of the first element of the array.

Example

```
          routine Form (addr p, integer q,r,s)
          .......
          array fn V(p,q,r,s)
          .......
```

which might be called by:-

```
          Form (A(1,1), 10, 4, 1)
```

Note (1) The array function declaration is unaffected by the method used for passing on the address.

(2) These two methods are also available as alternatives to the array name method, whenever that method is allowed.

FUNCTIONS

The function facilities are closely related to the routine facilities. However, the result of a function call is a number (real or integer) and function calls occur in arithmetic expressions. The declaration, call and description of routines and functions are compared in the following table:-

	Routine	Function
Declaration	routine spec...	a. real fn spec.... b. integer fn spec...
Result of call	execution of an instruction	a. real number b. integer
Description	routine....	a. real fn... b. integer fn ...

In place of return the delimiter result is used to terminate the evaluation of a function. However, the use of result is obligatory.

Example The routine FACT can be rewritten as an integer function which we rename FACT'

```
integer shriek
integer fn spec FACT' (integer n)
...............
shriek = FACT'(10)
...............
integer fn FACT'(integer n)
integer prod,i
prod = 1
if n = 1 then result = prod
cycle i = n,-1, 2
prod = i*prod
repeat
result = prod
end
```

Note (1) the assignment of the value of the function to result.

(2) Both the routine call FACT(shriek, 10), and the assignment shriek = FACT'(10) produce identical results.

Like routines, functions have the property of being global to any block interior to the one in which they have been declared and described. In particular, the functions listed on page 12 have the property of being global to the user's program so that neither declaration nor description is required.

Example

The specimen program for ordering lists of positive integers can be rewritten to illustrate the use of the routine and function facilities. A function MAX will be defined which finds the suffix of the largest element in the list. In terms of these, the cycle which achieves the ordering is written

```
cycle i=p,-1,2
j = MAX(A,i)
interchange(A(j), A(i))
repeat
```

with a considerable gain in legibility.

The full program is given on the next page.

Note (1) The integer array A is passed on to the function MAX in exactly the same manner as was indicated previously for routines.

(2) The function MAX could have been written in terms of elements of the global array A. To give the function a more general application, we write it in terms of a (formal) local array V. When calling the function, we pass on the name A as the actual parameter to replace V.

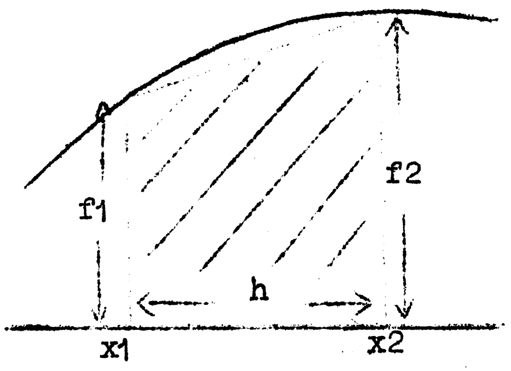
(3) The instruction-> 1 in the outer block refers to label 1 of the outer block. The same instruction in the integer function MAX refers to label 1 of that function.

```
begin

comment to order lists of positive integers

integer p

1: read(p)

if p≤0 then stop


        begin

        integer i,j

        integer array A(1:p)

        integer fn spec MAX(integer array name V, integer k)

        routine spec interchange (integer name a,b)

        cycle i = 1,1,p

        read (A(i))

        repeat

        -> 2 if p = 1

        cycle i = p,-1,2

        j = MAX(A,i)

        interchange (A(j), A(i))

        repeat

     2: cycle i = 1,1,p

        newline ; print (A(i),5,0)

        repeat


        integer fn  MAX(integer array name V, integer k)
        integer p,q,r
        r = 0
        cycle p = 1,1,k
        if r ≥ V(p) then -> 1
        r = V(p) ; q = p
     1: repeat
        result = q
        end


        routine interchange (integer name a,b)
        integer z
        z=a ; a=b ; b=z
        end

        end ; comment end of inner block

newline ; newline ; -> 1


end of program
```

ROUTINES AND FUNCTIONS AS PARAMETERS

It is possible to include routines and functions among the formal parameters of a routine or function by means of the type statements <u>routine</u>, <u>real fn</u>, and <u>integer fn</u>. When calling the routine or function the actual parameters must be the names of routines or functions declared either at the head of the block in which the call is made or in any exterior block. Note, however, that all quantities, other than the formal parameters, used in a routine or function description must be global to this description. It is not sufficient for them to be global at the time of call.

<u>Example</u>



Calculate approximately the area under the graph of y=f(x) between x = x1 and x = x2 using the trapezoidal rule. This is illustrated in the accompanying diagram. The area of the shaded part of the panel is $h(\frac{f2 + f1}{2})$.
The calculation is performed by dividing the area under the graph into a number of such panels, applying the formula to each panel and summing the results.

The program opposite carries out the approximate calculation five times, with the area divided into, 10, 20 30, 40 and 50 panels. The curve used is a quarter circle given by $y = \sqrt{1-x^2}$, from x = 0 to x = 1. The exact area is $\pi/4 = 0.785398163$.

```
begin

integer i

real fn spec TRAP SUM(real x1,x2, integer n, real fn f)

real fn spec circle(real x)


cycle i=1,1,5

newline

print(10i,2,0); spaces(2)

print(TRAP SUM(0,1,10i,circle),1,9)

repeat

    real fn TRAP SUM(real x1,x2, integer n, real fn f)

    real fn spec f(real y)

    real h,SUM

    integer i

    h=(x2-x1)/n; SUM=f(x1)

    cycle i=1,1,n-1

    SUM=SUM+2f(x1+i*h)

    repeat

    SUM=SUM+f(x2)

    result =h*SUM/2

    end


    real fn circle(real x)

    result =sq rt(1-x*x)

    end

end of program
```

Notes (1) The print-out from the program was:-

```
10   0.776129582
20   0.782116220
30   0.783610789
40   0.784236934
50   0.784567128
```

(slowly approaching the true value of 0.785398163).

(2) Because of anomalies in the compiler:-

(a) The standard functions (page 12) cannot be used as actual parameters to replace formal parameters of type real fn or integer fn. (They can, of course, appear in expressions used to replace formal parameters of type real or integer).

(b) It is necessary to declare the formal function parameter f, as on line 2 of the description of routine TRAPSUM.

RECURSIVE USE OF ROUTINES AND FUNCTIONS

Routines and functions have the property of being global to any block interior to the one in which they are declared. As any call of a routine or a function involves the execution of a block, a routine or function is global to itself. In particular a routine or function can call itself recursively.

Example The function FACT' can be defined recursively as $n(n-1)(n-2)\ldots2\ 1 = n\left[(n-1)(n-2)\ldots2.1\right]$

```
integer fn RECFACT (integer n)
if n = 1 then result = 1
result = n*RECFACT(n-1)
end
```

Note (1) Although the description of RECFACT is considerably shorter than that of FACT' (given on page 41) the latter is more efficient as it takes less time to execute the steps of a cycle than to make an equivalent number of function calls.

Example The program to order lists of positive integers can be made recursive, for, if a program can order a list of (n-1) integers, then it can order a list of n integers by taking the largest into the nth place, and then calling itself to order the remaining (n-1).

```
begin
comment to order lists of positive integers in
comment increasing order of magnitude
integer p
1:read(p)
if p≤0 then stop

   begin
   integer i
   integer array A(1:p)
   routine spec ORDER(addr s, integer k)

   cycle i=1,1,p
   read(A(i))
   repeat
   ORDER(A(1),p)
   cycle i=1,1,p
   newline
   print(A(i),5,0)
   repeat
```

```
        routine ORDER(addr s, integer k)
        integer j
        integer fn spec MAX(addr s, integer k)
        routine spec interchange(integer name a,b)
        integer array fn V(s-1,1)

        if k=1 then return
        j=MAX(V(1),k)
        interchange(V(j),V(k))
        ORDER(V(1),k-1)

           integer fn MAX(addr s, integer k)
           integer p,q,r
           integer array fn V(s-1,1)
           r=0
           cycle p=1,1,k
           if r>V(p) then->1
           r=V(p); q=p
           1:repeat
           result =q
           end

           routine interchange(integer name a,b)
           integer Z
           Z=a; a=b; b=Z
           end

        end ; comment end of routine ORDER

        end ; comment end of inner block

     ->1
     end of program
```

Note (1) The comments made concerning the efficiency of RECFACT apply also to ORDER.

(2) The routine ORDER has the general structure described on page 35.

Example The game of HANOI. In the two previous examples it is easy to avoid writing the program recursively. This is not true of this example. In this game one is given three pegs, and on one of the pegs are a number of circular discs of different sizes, graded so that the largest is at the bottom and the smallest at the top. The aim of the game is to transfer the discs to one of the other pegs (making use of the third as required) in such a way that there is never a larger disc on a smaller one. Only one disc at a time may be moved.

If the solution to the game for (n-1) discs is known, then the solution for n can readily be obtained. Let the pegs be labelled 1,2 and 3 and let it be required that the n discs on 1 be transferred to 3. This can be done by transferring the first (n-1) to 2, the last to 3, and then the first (n-1) from 2 to 3. In the following program n is the number of discs which have to be moved from peg i to peg j. The moves are printed out as pairs of integers.

PROGRAM FOR GAME OF HANOI

```
    begin
    integer n,i,k
    routine spec hanoi(integer n,i,k)
    read(n,i,k)
    hanoi(n,i,k)
    newline
    stop
    routine hanoi(integer n,i,k)
    if n=0 then ->1
    hanoi(n-1,i,6-i-k)
    newline
    print(i,1,0); space; print(k,1,0)
    hanoi(n-1,6-i-k,k)
  1:end
    end of program
```

STORE MAPPING FUNCTIONS

The store mapping function provides the user with the possibility of renaming storage locations in a more general manner than is possible with array functions.

The mapping function must be declared by either <u>real map spec</u> or <u>integer map spec</u> depending on the nature of the variables to be renamed. The mapping function can appear in an arithmetic expression, but, as the result of the store map is the name of a variable, it can also be written on the left hand side of an assignment.

Example

```
    real x
    real map spec  W(integer i,j)
    .............
    x = W(2,3)
    W(1,2) = 1 + x↑3
    .............
```

Note This illustrates the use of the store map both on the left and the right hand sides of an assignment.

The description of the store mapping function has the form

```
real map  W(integer i,j)
result = addr(X) + ("integer expression involving i and j")
end
```

Here X is an element of the array to be renamed. X must be global to the description of the mapping function.

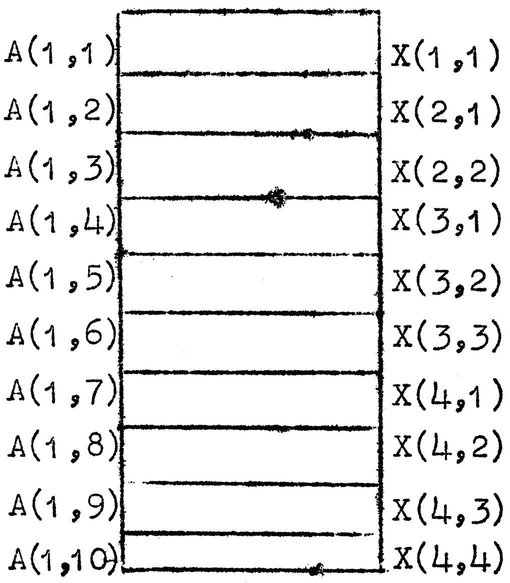
Example

```
array A(1:1000)
real map spec W(integer i)
.........
real map  W(integer i)
result = addr(A(1)) -1 + i*i
end
```

Note (1) In this example W(i) and A(i*i) are equivalent. For example W(10) and A(100) are both valid titles for the same storage location.

Example   To store a triangular array of numbers

```
array A(1:10; 1:10)
real map spec X (integer i,j)
.....
.....
.....
cycle i = 1, 1, 13
cycle j = 1, 1, i
read (X (i,j))
repeat
repeat
.....
.....
real map  X (integer i,j)
result = addr(A(1,1)) + i*(i-1)/2 + j-1
end
```



Note The diagram illustrates the storing of the first four rows of the triangular array.

INPUT AND OUTPUT

Input and output of numbers and symbols is achieved by permanent routines whose descriptions are held in the machine. These routines are global to the whole program and may therefore be called without further declaration and description.

A distinction must be drawn between numbers and symbols.

(a) A NUMBER usually occupies more than one space on the printed page, and consists of certain permitted sequences of the symbols 0 1 2 3 4 5 6 7 8 9 . + − α as given on page 12.

(b) A SYMBOL occupies one space on the printed page.
5 Examples of basic symbols are:- A a & = 7

Compound symbols consist of two or three basic symbols, superimposed upon one another by means of the backspace on the teleprinter. Examples of compound symbols are:-

$\underline{a}$ (the 2 symbols a and _ superimposed)

$\neq$ (the 2 symbols = and / superimposed)

$\underline{\neq}$ (the 3 symbols = / and _ superimposed)

Some of the following routines have already been explained but are included here for the sake of completeness.

INPUT OF NUMBERS

read (a)	Read the next number on the data tape into location a, and move the tape on, ready for the next number. (Parameter called by name).
read (a,b,c,d)	Read the next four numbers into a,b,c and d. There is no limit to the number of parameters allowed, and each one may be either real or integer, or an element of either an array or an integer array.

OUTPUT OF NUMBERS

print ((a+b+c)/n, i+1, j)	Print out the value of the first expression, with (i+1) figures before and j figures after the decimal point. Parameters are called by value, and so may be an expression. The last two must be integer expressions.
print fl ((a+b+c)/n, j)	Print out the value of the first expression in floating point form. One figure is printed before and j after the decimal point, additional powers of 10 being indicated by the symbol α, as on page 12.

INPUT OF SYMBOLS

read symbol (a)	Read the next symbol (simple or compound) on the data tape and place its numerical equivalent (as given on the next page) in location a. Note (1) unlike read, only one parameter is allowed. Note (2) a must be an integer or an element of an integer array.
skip symbol	Move the data tape on one symbol, without reading anything into the machine.
a = next symbol	Read the next symbol into integer location without moving the data tape on. (So the same symbol can be read in a second time).

OUTPUT OF SYMBOLS

print symbol (a+b+7)	Print the symbol whose value is given by the integer expression in brackets.
<u>caption</u> answer	Print the string of symbols following the delimiter <u>caption</u>, as far as the next semi-colon or "newline" character. (Spaces and ; are indicated by $\not{s}$ and $\not{;}$ respectively).
newline	This causes the teleprinter to go to the start of a fresh line.
spaces (5)	Print out 5 blank spaces.
space	Equivalent to spaces (1)

Note At the moment, compound symbols printed out via paper tape will not appear correctly superimposed.
For example, <u>caption</u> x ≠ 0 <u>and</u> y ≠ 0
will cause a print-out x = 0 and y = 0 / ___ /

Compound symbols are reproduced correctly when the LINE PRINTER is used for output.

TABLE OF NUMERICAL EQUIVALENTS

0			32	'		64			96		
1			33	A	|	65	space	a	97	a	o
2			34	B	|	66			98	b	
3			35	C		67			99	c	
4	newline	a	36	D		68			100	d	
5			37	E		69			101	e	
6			38	F		70			102	f	
7			39	G		71			103	g	
8	(	o	40	H		72			104	h	
9	)	o	41	I		73	colourshift		105	i	
10	,	o	42	J		74			106	j	
11	π	o	43	K		75			107	k	
12	?		44	L		76	stop		108	l	
13	&		45	M		77			109	m	
14	*	o	46	N		78			110	n	
15	/	|	47	O		79	:	o	111	o	
16	0	|	48	P		80			112	p	
17	1	|	49	Q		81	[	o	113	q	
18	2	|	50	R		82	]	o	114	r	
19	3	|	51	S		83			115	s	
20	4	|	52	T		84			116	t	
21	5	|	53	U		85			117	u	
22	6	|	54	V		86	_(underline)	a	118	v	
23	7	|	55	W		87	|		119	w	
24	8	|	56	X		88			120	x	
25	9	|	57	Y		89			121	y	
26	<	o	58	Z	|	90	α	|	122	z	o
27	>	o	59			91	β		123		
28	=	o	60			92	½		124		
29	+	|	61			93			125		
30	-	|	62			94			126		
31	.	|	63			95			127		

Note To find the numerical equivalent of a compound symbol let the numerical equivalents of its 2(or 3) constituents be a,b (or a,b,c) arranged in increasing order of magnitude. Then the value of the compound symbol is $a + b.2^7$ (or $a + b.2^7 + c.2^{14}$)

Example $\underline{a}$ has the value $86 + 97.2^7 = 12416$

while $\underline{\neq}$ has the value $15 + 28.2^7 + 86.2^{14} = 1412623$

<u>Notes</u> (1) Although spaces in a program are disregarded, this is not true on a data tape, where spaces can be used to separate numbers from one another. Numbers written in floating point form may have spaces between α and the exponent, but in all other cases a space or a newline character indicates the end of a number.

(2) When using the print and print fl routines, a negative number is preceded by - and a positive number by a space (to give correct vertical alignment of a column).

(3) To print a space followed by the symbol 7, we could use one of three methods:-

(a) print (7, 1, 0)
(b) <u>caption</u> $\not{s}$ 7
(c) print symbol (65); print symbol (23)

(4) <u>caption</u> is usually easier than print symbol if the symbol is known in advance. But if, for example, we wish to print B or b depending upon whether an integer n is even or odd, the simplest method is

print symbol (66 - 32 * parity (n))

(5) A maximum of 120 characters can be printed on any one line of output.

(6) Programmers should note that they are charged for each line of output, irrespective of the number of characters in it.

<u>(***) BINARY INPUT/OUTPUT</u>

Routines for the input and output of binary information will be available later.

BINARY REPRESENTATION OF NUMBERS

In normal usage, numbers are constructed from the ten digits 0,1,.....9. 4703 means $(4 \times 10^3) + (7 \times 10^2) + (0 \times 10^1) + 3 \times 10^0)$

In the binary notation used in the machine, numbers are constructed from the two binary digits (or "bits") 0 and 1. Here 10111 means $(1 \times 2^4)+(0 \times 2^3)+(1 \times 2^2)+(1 \times 2^1)+(1 \times 2^0)$ (i.e. 23 in normal notation).

A location in the store contains 48 bits, but for the present purposes we only consider locations declared of type <u>integer</u>. For the moment we may disregard the left hand 8 bits and the right hand 3 bits. Of the remaining 37, the left hand bit indicates the sign of the integer (0 for positive, 1 for negative).

The value of the integer is calculated by giving the sign bit a weight of -2^{36} and the remaining 36 weights of $+2^{35}$, $+2^{34}....+2^0$

Hence 111....111 means $-2^{36} + 2^{35} + 2^{34}....+2^1 + 2^0 = -1$

If we now add -1 and 1 we get

```
    111 ........ 111
    000 ........ 001
    ----------------
(1) 000 ........ 000
    ----------------
```

and as the left hand 1 is spilt over the end, we get the result 0 as required. (Note that in binary arithmetic 1 + 1 gives 0 "and carry one").

As described above, integers are stored in 37 bits, and the last 3 bits are normally 0. However, these last three bits may be regarded as of weights 2^{-1}, 2^{-2}, 2^{-3}, that is $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, and so numbers with whole multiples of $\frac{1}{8}$ may be stored in so-called "integer" variables.

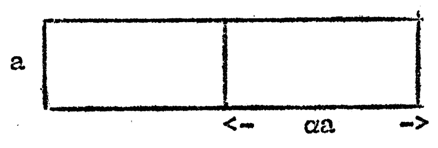
In an integer location, the left hand 8 bits are normally fixed as 00001100. (These represent the "exponent", not required for the present purpose, but called into play when an integer is copied into a real location).

HALF WORD VARIABLES

It is sometimes convenient to store information in, and carry out certain operations upon, 24-bit locations, known as HALF-WORDS.

The declaration <u>integer</u> a

allocates the name 'a' to a 48-bit location. At the same time, without further declaration, the right-hand half of a becomes known as αa. (Note: This has no connection with the use of α in floating point numbers). Provided the current value of a lies in the range $-2^{20} \leq a < 2^{20}$, all the significant information lies in αa. The left-hand half-word merely contains the fixed exponent 00001100 followed by 16 bits which are all the same as the left-hand bit of αa. (0 if $a \geq 0$, 1 if $a < 0$).



When dealing with half-words, the left-hand bit becomes the sign bit with a weight of -2^{20}. In this way, the numerical value of αa is the same as that of a whenever a lies in the range quoted above.

<u>Notes</u> (1) The half-word variable αa is local to the block in which a is declared, and may be used, as usual, in any interior block.

(2) Elements of integer arrays cannot be used in the same way. (e.g. αa is permitted, $\alpha A(1)$ is not). A device to avoid this difficulty will be given on page 59.

CONSTANTS FOR USE IN HALF WORDS

These can be written into a program in either decimal or octal form. To distinguish one from the other, octal numbers are preceded by *. Whichever form is used, the resulting bit pattern in the half-word is the same.

<u>Examples</u>

Decimal Form	Octal Form	Resulting bit pattern
17	*00000210	000000000000000010001 000
17.5	*00000214	000000000000000010001 100
17.625	*00000215	000000000000000010001 101

<u>Notes</u> (1) 17.5 is expressed as $(1 \times 2^4) + (1 \times 2^0) + (1 \times 2^{-1})$.

(2) The octal form is obtained by grouping the 24 bits into eight groups of three, and working out each group as a number in the range 0-7. (e.g. 100 becomes 4, 101 becomes 5).

HALF WORD EXPRESSIONS

These are formed from half-word variables and constants, connected by half-word operators, which are:-

operator	example	meaning
+ - * /	αa + αb	normal meaning but acting on only 24 bits.
&	αa & αb	AND. Compare αa and αb bit by bit, and put a 1 in the answer whenever αa AND αb have a 1. Otherwise 0.
$\underline{V}$	αa $\underline{V}$ 7	OR. Compare αa and 7, and put a 1 in the answer whenever αa OR 7 has a 1. Otherwise 0.
≢	13 ≢ αc	NOT EQUIVALENT. Compare 13 and αc, and put a 1 in the answer if, and only if, they have NOT EQUIVALENT bits. (i.e. one 0 and one 1).
≻	αd ≻ 5	SHIFT RIGHT. Copy αd and shift all bits 5 places right. Bits lost at the right hand end re-appear on the left. (Note: The symbol consists of > and - superimposed).
≺	αa ≺ αn	SHIFT LEFT. Copy αa and shift all bits cyclicly n places to the left.
(-)	(-) αb	NEGATE. Copy αb and turn all 0's into 1, and vice versa. Note:(-) and - are not the same.

Example (of an & operation).

```
          000000111111000000111 111
     &    000000000000111111111 111
   give   000000000000000000111 111
```

Precedence of Operators.

The precedence rules given on page 13 do not apply in half word expressions. All half word operators except (-) have equal precedence, but are carried out in sequence from the left, unless brackets are inserted to over-ride this rule.

i.e. αa $\underline{V}$ αb + αc means (αa $\underline{V}$ αb) + αc

The operator (-) has higher precedence than the rest, and is slightly different as it has only one operand.

i.e. αa & (-) αc means αa & ((-) αc)

HALF WORD INSTRUCTIONS

(1) Half-word expressions may be assigned to half-word variables. Such instructions may be made conditional.

e.g. $\alpha x = \alpha y$ & αz

$\alpha x = \alpha y \not\equiv \alpha z$ if a>b

(2) Half-word expressions may be used anywhere where an integer (or real) expression is expected.

e.g. print(αx V αp, 5, 0)

a = b - c if αx & $\alpha y \neq 0$

(3) If a half-word expression is assigned to an integer variable, 24 bits are added onto the left. The first 8 are 00001100 (the exponent) and the remaining 16 are the same as the sign bit of the half-word being expanded into a full word(0 if positive, 1 if negative). This ensures that the numerical value of the half word is preserved.

e.g. a = αx & αy

Example

Take the integer n and reduce it modulo 16 (i.e. add or subtract multiples of 16 until the result is in the range 0-15).

All except the last 7 bits of the integer represent multiples of 16. The last 3 bits represent the fractional part (as n is a true integer, they are all 0). The required bits are therefore 4 in number and we can extract them by means of the 'mask':-

000000000000000001111 000 (i.e. 15).

The required instruction is

n = αn & 15 or in octal n = αn & *00000170

This method is suitable for reducing integers modulo 2,4,8 or any power of 2. In particular, αn & 1 will give 1 or 0 for odd or even values of n, respectively.

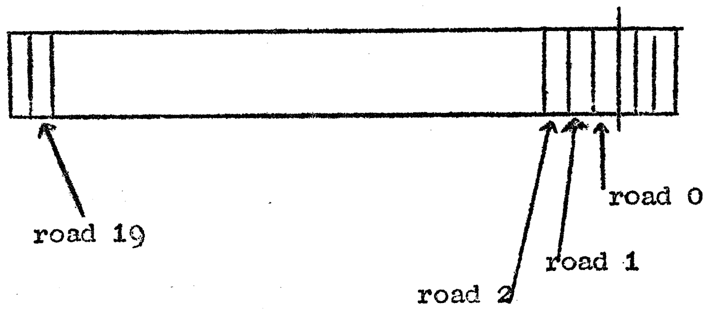
The next example has been designed to make use of all the half-word operators.

Example

There are 20 roads, numbered 0,1,.......19 all leading to a traffic bottleneck area J. The state of each road is to be kept in an appropriate bit of the integer J. (0 if road is clear, 1 if blocked). Reports are being received from many sources, stating that a given road has become blocked, or clear, or even that the previous situation has been reversed. Allowance must be made for the possibility of receiving the same information twice or more.

Solution

The half-word expression (1 ◂ αn) has a 1 in the bit representing road n, and a 0 in all 23 other bits. Similarly, the expression (-)1 ◂ αn has 0 in the bit representing road n, and 1 elsewhere. Hence suitable instructions are:-



situation	declaration/instruction
Initial situation. All roads clear.	<u>integer</u> J,n,i,p J = 0
Road n is blocked	J = 1 ◂ αn <u>V</u> αJ
Road n is clear	J = (-)1 ◂ αn & αJ
State of road n changed	J = 1 ◂ αn <u>≢</u> αJ
All roads blocked	J = -1
Roads 0-5 all blocked	J = αJ <u>V</u> 63
Request to print out state of all roads.	p = J <u>cycle</u> i = 0,1,19 print symbol(31+3*(αp & 1)) space p = αp ▸ 1 <u>repeat</u>

Notes (1) print symbol(34) produces a B (for blocked road).
print symbol(31) produces a full stop (for clear road).
Hence the output will take the form:-

```
. B B . B B B . . B . . . B . B B B B .
```

(2) The remaining details of the example are left as an exercise for the reader.

FURTHER HALF-WORD VARIABLES

Immediate access to right-hand half-words by means of the symbol α is possible for locations declared to be of type integer, but not for elements of integer arrays (e.g. αa is permitted, but αA(i,j) is not). To obtain the left-hand half of an integer location, or either half of an integer array element, it is necessary to make use of the address function. The address of the left-hand half of a is the same as addr(a): that of the right-hand half is addr(a)+$\frac{1}{2}$.

In the following, it is important to notice the two distinct uses of α:-

(i) α('half-word expression')
(ii) α followed by a name.

a	means	The 48-bit word stored at addr(a).
αa	means	The 24-bit word stored at addr(a)+$\frac{1}{2}$.
α(...)	means	The 24-bit word stored at the address given by the bracketed HALF-WORD expression.

If we write:-

```
integer a,y
y = addr(a)
```

then as all addresses lie in the range $-2^{20} \leq y < 2^{20}$, αy is a half-word variable with value equal to addr(a). Hence

$\alpha(\alpha y)$	means	The 24-bit word stored at addr(a). (i.e. the left-hand half of a).

This can be used anywhere where a half-word variable is allowed.

Example

Declare 20 half-word variables, and place the numbers 0-19 into them.

```
integer array a(1:10)
integer y,i
y = addr(a(1))
cycle i = 0,1,19
α(αi/2 + αy) = i
repeat
```

DOCUMENTS AND JOB DESCRIPTIONS

To make efficient use of the high speed of Atlas, it is controlled by a master program called the supervisor. Each complete task the computer handles is called a "job" and may be presented to the computer in several parts, each of which is called a "document". Each document must start with certain information so that the supervisor may associate it with the appropriate job.

SINGLE DOCUMENT JOBS

These are the simplest and the most common form of job.

Example The program given on pages 46-7 was transmitted to Atlas as follows:-

	meaning
JOB	Start of a new job.
U EDIN, COU OSBORNE, T.4/101 ORDER INTS	Department: Computer Unit Author: Osborne Transmission No: 101 of 1964 Brief Description: ORDER INTS
OUTPUT	Details of output to follow
0 SEVEN-HOLE PUNCH 100 LINES	Channel no. 0. 7-Hole paper tape. Printed output not to exceed 100 lines.
COMPUTING 500 INSTRUCTIONS	Computing is not to exceed 500 instruction interrupts (of 2048 machine code instructions each).
STORE 15 BLOCKS	Total store used is not to exceed 15 blocks (each of 512 locations)
COMPILER AA	Program below is written in Atlas Autocode.
<u>begin</u> . . . . . . <u>end of program</u>	Program as explained on pages 46-7.

15
12 3 4721 59 123 67 253
97 82478 97 0 23 1 2 3 -1
	Data

*** Z	Marker for end of document

Notes (1) The second line of the document (U EDIN.....INTS) is known as the TITLE. The title is limited to a total of 80 characters none of which must be a backspace.

(2) The most convenient medium for large quantities of output is the fast line printer. To make use of this, LINE PRINTER is written in the job heading in place of SEVEN-HOLE PUNCH. The results are printed out directly (top and one carbon) and posted to Edinburgh. However, output required as data for a subsequent job must be on paper tape (normally 7-hole punch).

(3) It may be more convenient to specify maximum output in blocks (each of 4096 characters).

```
e.g.  OUTPUT
      0 LINE PRINTER  11 BLOCKS
```

(4) Execution of the job ceases immediately if the maximum specified output, instruction interrupts or store is exceeded. This is largely a precaution against faulty programs running on and incurring vast expense. If these maxima are not specified in the job heading, the supervisor automatically allows 1 block of output, 5000 interrupts and 20 blocks of store.

RESULTS FROM SINGLE DOCUMENT JOBS

The complete print-out of results from the ORDER INTS program is given below, with an explanation on the following page.

```
00.00.03 /          17.02.64   21.52.05
OUTPUT   0
U EDIN, COU OSBORNE, T.4/101 ORDER INTS


ATLAS AUTOCODE WITH FINAL SUPERVISOR 9TH JAN 1964


   0    BEGIN M/C ADDRESS = 2240

   6      BEGIN

  18        BEGIN ROUTINE  ORDER   = 47  M/C ADDRESS = 2365

  27          BEGIN ROUTINE  MAX   = 48  M/C ADDRESS = 2445
  36          END OF 48  OCCUPIES 69 M/C INSTRUCTIONS

  37          BEGIN ROUTINE  interchange   = 49  M/C ADDRESS = 2517
  40          END OF 49  OCCUPIES 20 M/C INSTRUCTIONS

  41        END OF 47  OCCUPIES 175 M/C INSTRUCTIONS
               NON-LOCAL VARIABLES ORDER

  42      END
             NON-LOCAL VARIABLES p

  44    END OF PROGRAM  OCCUPIES 305 M/C INSTRUCTIONS

 PROGRAM ENTERED


      0
      1
      2
      3
      3
     12
     23
     59
     67
     97
     97
    123
    253
   4721
  82478


 INSTRUCTION 209     196
 STORE 15        / 14
 INPUT 0         1        BLOCKS
 OUTPUT 0        TT PUNCH 1        BLOCKS

 END OUTPUT      1 BLOCKS
```

INTERPRETATION OF RESULT PRINT-OUT

(a) Heading	Meaning
00.00.03 / 17.02.64 21.52.05	Supervisor No. Date. Time.
OUTPUT 0	Output channel No. 0.
U EDIN, COU OSBORNE, T.4/101 ORDER INTS	Title of job.
ATLAS AUTOCODE WITH FINAL SUPERVISOR 9TH JAN 1964	Details of compiler and supervisor used, with date of latest amendment to compiler.

(b) Program Map

The program map gives the line numbers of the begining and the end of each block and routine (functions and store maps are treated as routines, and are called 'ROUTINE.......' in the print-out). The first begin is line 0.

Each routine is given a serial number. After each block and routine, the number of machine instructions involved is printed out, followed by a list of non-local (i.e. global) names used in that block or routine. This list includes the names of global routines as well as variables. An unusual feature of the example given is that the routine ORDER is called recursively, and so the name ORDER appears as a non-local name used in routine ORDER.

(c) Results of the computation.

(d) Terminal information	Meaning
INSTRUCTION 209 196	Instruction interrupts used. (a) Total (b) For compiling.
STORE 15 / 14	Store requested. Store used.
INPUT 0 1 BLOCKS	Amount of input on channel 0.
OUTPUT 0 TT PUNCH 1 BLOCKS	Amount of output on channel 0.
END OUTPUT 1 BLOCKS	Amount of output on all channels.

ESTIMATES OF OUTPUT, COMPUTING AND STORE

(a) OUTPUT It is normally easy to estimate the number of lines of output required for results. To this must be added an allowance to cover the program map, whose size depends largely upon the number of blocks and routines. Unfortunately, each line of program map counts as approximately five lines of output.

(b) COMPUTING

For the first run of a program, it is necessary to make a cautious over-estimate based upon experience. Subsequently, the total instructions used in the first run will give a better guide.

(c) STORE

The total storage required consists of blocks for

(1) The compiled program. (Total M/C INSTRUCTIONS/512)
(2) Variables. (Total locations/512)
(3) Certain permanent routines (including print, read) (9 blocks).

Note One 35 x 35 array occupies 1225 locations, which is more than 2 blocks.

(***) MULTIPLE DOCUMENT JOBS

At a later date it will be possible to run jobs using several channels of input and/or several channels of output.

FAULTY PROGRAMS (a) Compiler Time Faults

The program map has an asterisk against the line number of any illegal instruction or declaration, and indicates the nature of the fault. Note that one faulty line can easily cause subsequent correct lines to be signalled as faulty.

Example

```
begin
integer N
N =0
```

Here the declaration will not be understood, as the delimiter integer must be completely underlined. The instruction N = 0 will then be faulted as N has not been declared correctly. The print-out will appear:-

```
0         BEGIN
1 *       INSTRUCTION NOT RECOGNISED
          integerN
2*        NAME N NOT SET
```

Notes (1) The indication of type of fault is usually self-evident. AP-FAULT (or FP-FAULT) indicates that the actual parameter (or formal parameter) used is of a type inconsistant with the declaration.

(2) As spaces are disregarded by the compiler, the print out from any instruction not recognised will be devoid of spaces. (See integerN above). Also, for the moment, compound symbols will not be correctly superimposed if paper tape output is used. (See note on page 51).

(3) If a fault is recorded in the program map, then PROGRAM ENTERED is replaced by PROGRAM FAULTY, immediately followed by the terminal information about the number of instructions and blocks used.

FAULTY PROGRAMS (b) Execution Time Faults

A fault occuring during the running of the program (i.e. after PROGRAM ENTERED) also produces a fault print. The execution of the program is terminated. Common faults, generally caused by an attempt to divide by zero, are:-

```
DIV OVERFLOW
EXP OVERFLOW
```

The serial number of the routine and the line number of failure are printed. Again most of the faults are self-explanatory.

FAULT FINDING FACILITES

Some programming errors only show their presence by producing incorrect results. The following facilities are of use in tracing errors.

(1) QUERY PRINTING

A question mark written on the right of an assignment instruction will cause the machine to print out the value of the number assigned. The number is printed in floating point form, with one figure before and ten after the decimal point.

e.g. a = b + c ?

Query printing is possible with assignments to integer, real and complex variables but not half-word variables.

Query printing can be suppressed in two ways:-

(a) <u>During Execution</u> The following machine instructions are available.

<u>instruction</u>	<u>meaning</u>
127,84,0,-4.125	No more query printing until further orders.
167,84,0,4	Switch on query prints again.

With the above instructions, we can, for example, execute a loop of instructions many times, but only query print the first time round.

(b) <u>During Compilation</u> By inserting the following delimiters, we can effectively erase query prints from a part or the whole program without having to re-punch the tape.

<u>DELIMITER</u>	<u>Meaning</u>
<u>ignore queries</u>	Subsequent query print symbols are to be ignored by the compiler.
<u>compile queries</u>	This cancels the effect of the previous delimiter.

(2) <u>ROUTINE AND LABEL TRACING</u>

<u>DELIMITER</u>	<u>Meaning</u>
<u>compile routine trace</u>	In the following section of program, print out the routine number every time a routine is entered or left.
<u>compile jump trace</u>	In the following section of program, print out details of every jump instruction obeyed. (simple jumps, <u>test</u> and <u>switch</u>).
<u>stop routine trace</u> <u>stop jump trace</u> }	These cancel the effects of the previous two.

<u>Example</u> In a part of the program where both routine and jump traces were in force, the print-out might be:-

```
→ 2  R47  → 3  → 5  END 47  → 1
CASE 3      switch 1
```

(***) COMPLEX ARITHMETIC FACILITIES

The following facilities for carrying out calculations which involve complex quantities will be available shortly.

Declarations Variables may be declared as type complex. In this case two consecutive locations are set aside for each name. The first location holds the real and the second the imaginary part. Complex arrays may be used. They must be declared by complex array....

Examples

```
complex Z
complex array z (1:10)
```

Renaming locations is possible by means of the complex array function:-

Example

```
complex array fn X(addr(A(1,1))- 2n-2,n,1)
```

renames the complex array A(1:n,1:n) so that

X(i,j) = A(i,j)

Notes (1) The address recovery function can have a complex argument. The result is the address of the real part.

(2) In calculating the origin for the array function, it must be remembered that each element occupies two locations.

(3) Apart from the exception mentioned in (2) above, the complex array function is used in exactly the same manner as the array function.

Complex Arithmetic Expressions

These consist of constants, variables and functions connected by delimiters.

(a) Constants. Real or integer constants can be used in complex expressions. In forming complex constants the role of $\sqrt{-1}$ is played by the delimiter i.

Examples

```
5
i
1.75 + i0.573
```

Note The delimiter i is written to the left of the number.

(b) Variables. Complex, real and integer variables may all be used in complex expressions.

(c) Functions. The standard functions which produce complex results when they have complex arguments are available. However, in this case they are preceded by c.

Examples ccos, cexp, clog

Complex Assignment Instructions

Complex, real or integer expressions can be assigned to complex variables. If a real or integer expression is assigned, the imaginary part is, of course, set to zero.

Examples

```
integer j,k
real x,y
complex z,Z
complex array X(1:10)

Z = ccos(z) + i csin(z)
X (1) = cos(x) + i sin(x)
X (2) = x + 2y
X (3) = j + k
```

Complex functions in real expressions

Complex functions may not appear in real expressions. However, the functions

re(Z), im(Z), mod(Z), arg(Z)

convert from complex to real. Note that re(Z) and im(Z) are a pair of actual locations in the store and so may appear on the left hand side of real assignments.

Conditions. Only real or integer expressions can appear in conditional expressions.

Examples

$\rightarrow$ 3 unless im(Z) > 0

if arg(Z) $\geqslant \pi/2$ then $\rightarrow$ 5

Routines and Functions

The routines types are to be expanded to include types complex function and complex map.

The parameter types complex name and complex are to be defined in a manner analgous to that of the previously considered parameter types. It is hoped that the type complex array name will also be available.

Data

Complex numbers are punched on the data tape observing conventions similar to those for real numbers.

Examples $3 + \underline{i}4$

$\underline{i}$

$1.17\alpha 3 + \underline{i}2.13\alpha 4$

Within the number, spaces may only appear immediately before or immediately after $+ \underline{i}$ or $-\underline{i}$. They may be read by the usual read instruction.

(***) LIST PROCESSING

A number of list processing facilities will be available later. No details are given here, since the currently planned instructions may be withdrawn and replaced by new instructions written in the form of routines.

LIBRARY ROUTINES

A number of library routines and functions have been collected at Manchester. For the time being, copies of these on paper tape can be obtained from the Computer Unit, and included in a program exactly as if written by the user.

A list of library routines available on 18th February, 1964, is given on the next page.

(***) LIBRARY ROUTINES ON MAGNETIC TAPE

Eventually, the library routines will be stored on magnetic tape on Atlas, and then the programmer will only need to declare and call the routine, the description being unnecessary. The declaration will then take the form:-

<u>library routine spec</u>..............

LIST OF LIBRARY ROUTINES AVAILABLE ON 18th FEBRUARY, 1964

(Further details of the use of these routines can be obtained from the Computer Unit).

real fn spec erf (real x)

Computes the error function.

real fn spec random (real name x, integer n)

Generates successive numbers rectangularly distributed in (0,1) or normally distributed in(-1,1)

routine spec ber and bei (real x, real name ber, bei, integer n)

Computes the Bessel functions $\mathrm{ber}_n x$ and $\mathrm{bei}_n x$

routine spec householder (array name a,w, integer n,k)

Householder's method for eigenvalues and eigenvectors of real symmetric matrix. 4 Components of this routine are also available:-

(1) routine spec householder tridiagonalisation (.......)

(2) routine spec back transformation (.......)

(3) routine spec tridibisection (.........)

(4) routine spec tridiinverse iteration (..........)

routine spec sym pos def mat inv (...........)

Solves AX = B by Choleski method, where A is symmetric positive definate matrix stored as upper triangle.

routine spec int step (array name y, real x,h, integer n, routine aux)

routine spec int step 2 (array name y,e, real x,h, integer n, routine aux)

Single step of kutta-merson method for system of first-order equations.

routine spec kutta-merson (array name y, real x0, x1, real name e, integer n,k, routine aux)

Integration system of first-order equations from x0 to x1.

real fn spec GAUSS 5 (real fn f, real a,b)

real fn spec GAUSS 6 (real fn f, real a,b)

real fn spec GAUSS N (real fn f, real a,b, integer n)

(5, 6 and N-point GAUSS quadrature for $2 \leq n \leq 9$)

APPENDIX

Summary of instructions which can be made conditional

Any unconditional instruction can be made conditional by writing an if or an unless clause either before or after it. It is convenient to list all possible types of unconditional instructions.

(1) ASSIGNMENTS	to	(a) real variables
		(b) integer variables
		(c) half word variables
		(d) result (in a function)
		(e) complex variables (***)
(2) JUMP INSTRUCTIONS		(a) to a simple label (e.g. → 9)
		(b) to switch label (e.g. → A(i*j)
		(c) to conditional label (e.g test 1, 2, 3)
		(d) return (i.e. jump to end of block)
		(e) stop (i.e. jump to end of program)
(3) ROUTINE CALLS		(a) permanent routines (e.g. print, read)
		(b) routines declared in program
		(c) library routines

(4) caption INSTRUCTIONS

(5) LIST PROCESSING INSTRUCTIONS (***)

Notes (1) As assignment to result is also a jump instruction (to end of function) and so might also have been placed in the 2nd group.

(2) A conditional clause placed after a caption will be indistinguishable from the test of the caption, and so will be disregarded by the machine except that it will print out "if...... "

(3) cycle ......
repeat } do not appear on the list of unconditional instructions, and so cannot be made conditional.