PREFACE

This manual is a reference manual which describes the Atlas Autocode Compiler currently available (1/3/65) at Manchester University. It is not a teaching manual though we have tried to make it fairly readable. Further compilers may in the future become available both on Atlas and other machines and it is expected that they will be described with reference to this manual.

We would like to thank Mr. G. Riding for his many valuable comments and suggestions and Miss Christina O'Brien who has typed and re-typed the manuscript.

R.A. Brooker
J.S. Rohl.
1st March 1965

CONTENTS

1 INTRODUCTION

An ATLAS AUTOCODE PROGRAM consists of a series of STATEMENTS which describe in algebraic notation the calculation to be executed. The statements are of two kinds, declarative statements giving the nature of the quantities involved, and imperative statements which describe the actual operations to be performed on them, and the sequence in which they are to be carried out. The statements are not immediately recognisable by the computer and must first be converted into an equivalent sequence of basic MACHINE INSTRUCTIONS. This is done by a special translation program called a COMPILER which is held permanently available in the machine. Not until the program has been 'compiled' can it be executed.

The following example gives a general idea of the principles involved in writing a program. We wish to fit a straight line $y = ax + b$ to sets of data of the form X1,Y1; X2,Y2; ----; Xn, Yn which are to be punched and presented on a data tape in this order. Each such set is to be terminated by the number 999999 and the final set by two such numbers. For each set the quantities

$$a = \frac{n\Sigma XiYi - \Sigma Xi\Sigma Yi}{n\Sigma Xi^2 - (\Sigma Xi)^2}$$

$$b = \frac{\Sigma Yi - a\Sigma Xi}{n}$$

$$c = \Sigma Yi^2 - 2(a\Sigma XiYi + b\Sigma Yi) + a^2\,\Sigma Xi^2 + 2ab\Sigma Xi + nb^2$$

are calculated, the last being the sum of the squares of the deviations $\Sigma(Yi - aXi - b)^2$.

The following is the formal program for this calculation. The statements are to be interpreted in the written order unless a statement is encountered which transfers control to another specifically labelled statement. In general each statement is written as a new line, otherwise it must be separated from the previous statement by a semi-colon.

```
        begin
        real   a, b, c, Sx, Sy, Sxx, Sxy, Syy, nextx, nexty
        integer n
        read (nextx)
2:      Sx = 0; Sy = 0; Sxx = 0; Sxy = 0; Syy = 0
        n = 0
1:      read (nexty) ; n = n + 1
        Sx = Sx + nextx ; Sy = Sy + nexty
        Sxx = Sxx + nextx² ; Syy = Syy + nexty²
        Sxy = Sxy + nextx*nexty
3:      read (nextx) ; ->1 unless nextx = 999 999
        a = (n*Sxy - Sx*Sy)/(n*Sxx - Sx²)
        b = (Sy - a*Sx)/n
        c = Syy - 2(a*Sxy + b*Sy) + a²*Sxx - 2a*b*Sx + n*b²
        newline
        print fl(a,3) ; space ; print fl(b,3) ; space ; print fl(c,3)
        read (nextx) ; ->2 unless nextx = 999 999
        stop
        end of program
```

BLOCKS AND ROUTINES

Complete programs are generally split up into a number of self-contained units called ROUTINES, and each routine may be further split into a number of BLOCKS. A detailed description of their construction and use is deferred until later, but in the earlier sections it is sufficient to note that the Autocode statements between begin and end constitute a block. However when a block defines a complete program as in the above example, end is replaced by end of program.

PHRASE STRUCTURE NOTATION

Atlas Autocode is a PHRASE STRUCTURE LANGUAGE and to assist in its description we sometimes have resort to phrase structure notation. In general, whenever a name appears in square brackets in the description of an Autocode statement, we mean that in an actual statement it would be replaced by a particular element of the class defined by the name. For example, in the next section we define [NAME] and [EXPR] to denote a general name and a general expression respectively, and with these definitions we could go on to define a function of a single variable by

[NAME] ([EXPR])

and in an actual program this might be replaced by

g(x + y - 2)

since g is a name, and x + y - 2 is an expression. Further notes on phrase structure notation will be found in Appendix 1.

2. THE BASIC LANGUAGE

SYMBOLS OF THE LANGUAGE

A program is presented to the computer as a length of perforated paper tape, prepared on a Flexowriter keyboard machine, the keys of which are engraved with the following symbols:-

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
α β π
0123456789
= > < | * : , ' & ^{2} / . + - _ ½ ( ) [ ] ?

A back-spacing facility allows underlining and also the formation of compound characters. For example :-

cycle ≠ ≥ ≤ ; ↑

The last of these consists of an asterisk superimposed on a vertical bar . It is usually referred to as a vertical arrow (and would be written as such in a manuscript) and is used to denote exponentiation, thus a↑(n-1) means 'a raised to the power (n-1)'. Such a notation is necessary because we have no means of effecting superscripts and subscripts with a Flexowriter; the format is essentially one dimensional. There is one exception, the superscript ^{2} for which there is a special symbol: it is equivalent to ↑2.

Since the handbook itself is prepared on a Flexowriter the same conventions for exponents will also be used in the text.

NOTE All SPACES and UNDERLINED SPACES in a program are ignored when the program is read into the machine. Thus they may be used freely to assist legibility in the written form of the program.

NAMES

These are used to identify the various operands, functions and routines which appear in the program. A name consists of one or more Roman letters, possibly followed by one or more decimal digits, and possibly terminated by one or more primes('). For example:-

X I Alpha a10 TEMP1 y'' b3'

Underlined names and mixed names such as RK2ST are NOT allowed.

There are certain names, e.g. log, sin, exp, print, read, etc. which have a standard meaning (the PERMANENT routines) but all other names must be declared before any reference is made to them (see below). In future a general name will be denoted by [NAME].

CONSTANTS

Numerical (positive) constants are written in a straight forward notation. For example:-

2.538   1   .25   17.28α-1   1α7

The last two examples mean 1.728 and 10000000.
The numerical part can be written in any number of ways. For example:-

15   015   15.   15.000

are all equivalent. The exponent, where present, consists of α followed by an optional sign and decimal digits.

The symbol ½ is equivalent to the two symbols .5. Thus 2.5 may be punched as 2½.

There is a further specialised type of constant consisting of a symbol (either basic or composite) enclosed in quotes. Its value is that of the internal equivalent of the symbol, a list of which is given in Appendix 5. Thus

'a' ≡ 33
'ø' ≡ 2063

Though this form of constant may be used whenever a constant is relevant it is most often used when reading symbols off a data tape (see Section 5).

DELIMITERS

These are a preassigned set of symbols and underlined words. For example:-

+ - * / (,) > ≥ -> ; π
cycle repeat integer real if then caption comment

Note that -> consists of two symbols, - followed by >

Unlike names whose meaning can be defined by the user, delimiters have fixed absolute meanings in the language.

TYPES

Calculations are performed on two principal types of operand, real and integer (later on we shall introduce complex). Both are represented by floating point numbers (in the form a*8↑b where a is held to a precision of 40 binary digits and b is an 8-bit integer); but those of integer type are kept in an unstandardised form (so that the least significant 24 bits can be used directly for B-modification; the precise method of storage is described in the section on machine instructions).

The locations in the computer store holding numbers are distinguished by assigning names to them (see later), and reference to the number is made by giving the appropriate name. Both real and integer numbers referred to in this way are called variables and denoted by [VARIABLE].

Programs will consist mainly of operations on real operands, the use of integer operands being generally confined to counting and subscript arithmetic.

DECLARATION OF VARIABLES

The names of variables used in a block are declared at the head of the block. For example:-

```
integer  I, max, min
real     t, Temp, VOL 1, VOL 2
```

The effect of these declarations is to allocate storage positions (ADDRESSES) to the named variables, and any subsequent reference to one of the declared names will then be taken as referring to the number stored in the appropriate address. The format of these declarations is formally

```
            [TYPE][NAME LIST]
  where     [TYPE] = integer, real
            [NAME LIST] = [NAME][REST OF NAME LIST]
            [REST OF NAME LIST] = [,][NAME][REST OF NAME LIST],NIL
```

N.B. This means of defining a list consisting of phrases separated by commas is used throughout: See Appendix 1.

One dimensional arrays of elements may be declared by statements such as

```
                array  a,b(0:99),  c(10:19)
```

which reserves space for three arrays of real variables a(i), b(i), c(i).
In the first two the subscript runs from 0 to 99, and in the third from
10 to 19.
To refer to a particular element of an array one might write

```
                a(50)   b(j)   b(2n+2j-1)   c(10+i)
```

It is the computed value of the argument, which may be a general integer
expression (see later), which determines the particular element.
Two dimensional arrays are declared in a similar way. For example:-

```
                array  A(1:20,1:20),  B(0:9,0:49)
```

This defines and allocates storage for a 20 X 20 array A and a 10 X 50
array B. To refer to a particular element, one writes, for example:-

```
                A(1,1)   A(i-1,j+1)   B(9,2K+1)
```

Should an array of integer elements be required, the declaration is
qualified by integer. For example:-

```
                integer array  Ka (1:50).
```

Arrays of more than 2 dimensions may also be declared. For example:-

```
                array CUBE 1, CUBE 2 (1:10,1:10,1:10)
```

reserves 1000 locations for each of the two arrays CUBE 1, CUBE 2.
Storage allocated by all the above declarations has dynamic significance, i.e.
they are implemented at run time and not at compiler time. Consequently,
the arguments in array declarations need not be constants but may be general
integer expressions. The significance of this will be explained in the sections
on block structure and dynamic storage allocation (see later).
The format of an array declaration is

```
                [TYPE'] array [ARRAY LIST]
        where   [TYPE'] = integer , real , NIL
                [ARRAY LIST] = [NAME LIST] ([BOUND PAIR LIST])[REST OF ARRAY LIST]
                [BOUND PAIR] = [EXPR]:[EXPR]
```

Here the [EXPR]'S must be integer [EXPR]'S (see P2.6)

FUNCTIONAL DEPENDENCE

Functional dependence is indicated by writing the name of the function followed by the list of arguments in parentheses (in a similar fashion to array elements). For example:-

sin(2πx/a) arctan(x,y) TEMP(i) a(10,10)

Each argument can be an arithmetical expression (see below).

Within a block all names must be distinct, and it is not possible to have a function with the same name as a scalar. Thus a and a(i) or f and f(x) would NOT be allowed to appear in the same block.

STANDARD FUNCTIONS

Certain standard functions are available and may be used directly in arithmetic expressions (see next section) without formal declaration:

sin(x) cos(x) tan(x) log(x) exp(x) sqrt(x)

arcsin(x)	($-\pi/2 \leq$ result $\leq \pi/2$)
arccos(x)	($0 \leq$ result $\leq \pi$)
arctan(x,y)	(= arctan (y/x), $-\pi \leq$ result $\leq \pi$)
radius(x,y)	(= sqrt (x^2+y^2))
mod(x)	(= $\lvert x \rvert$)
fracpt(x)	(= fractional part of x)
intpt(x)	(= integral part of x)
int(x)	(= nearest integer to x. i.e. intpt(x+.5))
parity(n)	(= (-1)$\uparrow$n)

The last three functions are of type <u>integer</u> (see later), the rest of type <u>real</u>. The arguments of all these functions may be general expressions, except that the argument of the last must be of type <u>integer</u>.
A complete list of standard functions is given in Appendix 2.

ARITHMETIC EXPRESSIONS

A general arithmetical expression is denoted by [EXPR] and consists of an alternating sequence of operands and operators possibly preceded by a sign symbol, thus *

[$\pm$'] [OPERAND][OPERATOR][OPERAND][OPERATOR] .... [OPERAND]

An [OPERAND] is a [VARIABLE], [CONSTANT], ([EXPR]), |[EXPR]|, or [FUNCTION], and an [OPERATOR] is one of + - / $\uparrow$ (the asterisk denoting multiplication).

**Or, more strictly, (See Appendix 1)

[EXPR] = [$\pm$'][EXPR']
[EXPR'] = [OPERAND][OP][EXPR'],[OPERAND]
[OPERAND] = [NAME][APP],[CONST],([EXPR]),|[EXPR]|
[$\pm$'] = +,-,NIL

An explicit multiplication sign is not required when ambiguity could not arise from its omission. For example:-

2.5a1b means 2.5*a1*b

<u>NOTE</u>: When the compiler looks for a name, it finds the longest possible name. Thus ab is taken as a name rather than a*b even if only a and b and not ab were declared. In this case a fault (NAME ab NOT SET) would be indicated. Examples of expressions are:-

$A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1) - 4A(i,j)$
$Z + \log(1 + \cos(2\pi(x/a + y/b + z/c)))$
LENGTH * BREADTH * HEIGHT
$1 + \mathrm{sqrt}(x(i)^2 + y(i)^2 + z(i)^2)$
a * b/c * d/e
(x + y + z)/(a + b + c)
2.5x1b * (c + d)e
e = |x-y| + .00001
(1+x)↑(n-3) * (1-x)↑3

<u>NOTES</u>

1. Multiplication and division take precedence over addition and subtraction and division takes precedence over multiplication. Thus the fifth example means a * (b/c) * (d/e).

2. |[EXPR]| is interpreted as the positive magnitude of the [EXPR]. Thus it is equivalent to mod([EXPR]).

3. An exponent is denoted by ↑ [OPERAND] and exponentiation takes precedence over the other operations. Thus the last example means ((1 + x) to the (n - 3))*((1 - x) to the 3). In the formation of a ↑ n, n must be an <u>integer</u> or <u>integer</u> [EXPR] (see next section); then if

(i) $n > 0$, result = a*a*a.......*a (n times)
(ii) $n = 0$, result = 1
(iii) $n < 0$, result = 1/(a*a*a.......*a)

4. To form a ↑ b, where b is real we must write it in the form exp(b*log(a)),where a must be positive.

<u>INTEGER EXPRESSIONS</u>

An [EXPR] is an <u>integer</u> [EXPR] if all the [OPERAND]'s are scalars, array elements etc. declared to be of type <u>integer</u>, or are integer constants or <u>integer</u> functions (e.g. int, intpt, or parity).

Thus if we assume that x is a real [VARIABLE] and i,n,j,k(1),k(2) are integer [VARIABLE]'s the following are integer [EXPR]'s.

```
n*(n-1)/2
i + j + k(2) + int(x)
j ↑ k
intpt(n*(n-1)/3)
```

The definition given above does not guarantee that an integer [EXPR] will always give an integral result, e.g., 10/3 and j↑(-1) are not integral. There is no guarantee either that expressions like n*(n-1)/2(which is integral) will always yield the exact answer (in this particular case it does). When the result of such an operation is in doubt it is preferable to use 'int' e.g., int(n*(n-1)/2) to give an exact integer result.

Except in certain special cases integer [EXPR]'s are evaluated by floating point arithmetic in exactly the same way as general (real) expressions, but are destandardised on assignment (explicit or implicit) to their integer destination. The definition of an integer [EXPR] is a basis for checking that such assignments are sensible. The special cases mentioned above refer to the subscript expressions in array elements. Such expressions, which should always be integer [EXPR]'s are usually simple linear forms which are dealt with more appropriately by B-modification. It is mainly to facilitate such operations (and the associated operation of counting) that integer's are used. Being destandardised quantities they can be transferred directly to B-registers without using the floating point accumulator.

ARITHMETIC ASSIGNMENTS

The general arithmetic instruction is

```
[VARIABLE] = [EXPR]
```

Examples are:-

```
X(p,q) = 1+2cos(2π(x+y))
a = (b+c)/(d+e)+F
i = i+1
```

The action of the general arithmetic assignment is to place the computed value of the [EXPR] in the location allocated to the l.h.s. [VARIABLE]. If the l.h.s. is a real [VARIABLE], the r.h.s. [EXPR] may be of type real or integer, but if the l.h.s. is integer then the r.h.s. must be an integer [EXPR]. For example, if y had been declared real and i integer then we could write y = i but not i = y even if we knew that y had an integral value.

LABELS, JUMPS AND CONDITIONAL OPERATORS

Normally instructions are obeyed sequentially, but frequently it is required to transfer control to some instruction other than the next in the sequence, or to obey an instruction only if certain conditions are satisfied. The following facilities are provided:

SIMPLE LABELS Any instruction can be labelled by writing an integer [N] before it, separated by a colon. More than one label is permitted. Unconditional jump instructions are written as -> [N]

```
      ->10
      ---
  10: ---
 4:5: ---
      ->4
      ---
      ->5
```

VECTOR LABELS

These are used to provide for a multi-way switch. With reference to the accompanying diagram the instruction -> A(i) will jump to A(1), A(2) or A(3) according as i = 1, 2 or 3. A fault is signalled if the value of i corresponds in any way to a label not set. The general form of the label is [NAME]([N]): The range must be declared at the head of the routine by a statement of the form switch [NAME]([±'][N]:[±'][N]) where the [±'] indicates that the integers may be preceded by a sign if necessary. For example:-

```
      switch A(1 : 3)
      ---
      ---
      ---
      ---
A(1): ---
      ---
A(3): ---
      ---
      ---
      ---
      ->A(i)
      ---
      ---
      ---
A(2): ---
      ---
      ---
```

switch SEGMENT (-4:+4)

A list of switches can be given. For example:-

switch A,B,C(1:3),D(0:2)

The [NAME]'s must not conflict with those of other operands in the same block.

CONDITIONAL LABELS

Another kind of multi-way switch is illustrated by the accompanying diagram. Here the conditions at the places indicated are tested in turn and control passes to the instruction following the first to be successful. If none is satisfied a fault is signalled. The general form of the label is [N] case [COND]: where [COND] denotes the general condition defined in the next section. A simple label [N]: may be used in place of the last alternative(i.e. 6:) in which case control passes directly to the following instructions if it reaches that point.

```
                test 4, 5, 6
                ---
                ---
  4 case x<1:   ---
                ---
  5 case 0≤x≤1: ---
                ---
  6 case x>1:   ---
                ---
                ---
```

NOTE All labels are local to the block containing them and jump instructions may only refer to labels within the block (see later).

CONDITIONAL OPERATORS

A CONDITIONAL OPERATOR of the form

if [COND] then   or   unless [COND] then

may be written before any unconditional instruction. These form the FORMAT CLASS [UI] (see Appendix 1) and include arithmetic, jump and test instructions.

The [COND] phrase takes one of the forms**

```
        [SC] and [SC] and [SC] --- and [SC]
or      [SC] or  [SC] or  [SC] --- or  [SC]
or just [SC]
```

Here [SC] denotes one of the following 'simple' conditions

[EXPR][ø][EXPR] or [EXPR][ø][EXPR][ø][EXPR] or ([COND])

where [ø] denotes one of the comparison symbols $= \neq > \geq < \leq$

If (or unless) the condition is satisfied the instruction is obeyed, otherwise it is skipped and control passes directly to the next instruction.

Examples of conditional instructions and conditional labels are

```
if x < 0  then x = mod(y)
if 0 ≤ x ≤ 1 and  0 ≤ y ≤ 1  then -> 1
case (y > 1 or y < - 1) and x ≥ 0:
```

** or, more strictly, (see Appendix 1)

```
[COND]  =  [SC] and [AND-C], [SC] or [OR-C], [SC]
[AND-C] =  [SC] and [AND-C],[SC]
```

Alternatively, conditional operators may appear AFTER unconditional instructions, in which case they are written

```
                if [COND]   or   unless [COND]
for example     x = 0 if |x| < .0000001
                -> 1 unless z > R or z = 0
```

CYCLING INSTRUCTIONS

These are pairs of statements which allow a group of instructions to be obeyed a fixed number of times. For example:-

```
cycle  i = 0, 1, n-1
___
___
___
___
repeat
```

In the above example the instructions between cycle and repeat are traversed n times, with i successively taking the values 0,1, ...,n-1. After the final cycle, control goes to the statement following repeat. The l.h.s. must be an integer name, but the r.h.s. quantities may be general integer [EXPR]'s which are initially evaluated and stored. Thus within the innermost cycle of the example below, the values of p,q and r may be altered without affecting the number of times the cycle is traversed. The initial value, increment, and final value must be such that

$$\frac{\text{final value} - \text{initial value}}{\text{increment}}$$

must be a positive integer or zero otherwise a fault is indicated.
For example:-

```
cycle i = 1,1,p
cycle k = 1,1,r
c(i,k) = 0
repeat
cycle j = 1,1,q
cycle k = 1,1,r
c(i,k) = c(i,k) + a(i,j)*b(j,k)
repeat
repeat
repeat
```

NOTE   Statements such as cycle x = .2,.1,1 are NOT allowed, and should be replaced by an equivalent permissible form. For example:-

```
cycle i = 2,1,10
      x = .1i
```

where i has been declared integer and x real.

MISCELLANEOUS NOTES

1. end of program is the formal end of the program and appears after the last written instruction; its action is to terminate the reading of the program and to start obeying it from the first instruction.
2. The instruction stop can appear anywhere in the program and signifies the dynamic end of the program; its action is to terminate the calculation.
3. The delimiter comment allows written comments to be inserted in a program to assist other users in understanding it. The information following comment (which may include composite characters) up to the next newline or semi-colon is ignored by the computer. The delimiters page and | are synonyms for comment, though the first has an obvious special use in the pagination of programs.
4. It has been noted earlier that all spaces and underlined spaces in a program are ignored and that Autocode statements are terminated by a semi-colon or a newline. If a line is terminated by the delimiter c then the following newline character is ignored by the computer. Thus a single statement may extend over several lines of the printed page. It is not anticipated that this facility will be frequently used, except when writing comments and possibly long algebraic expressions.
5. If a programmer is prepared to exclude upper case letters from names and captions, then he can effect a saving both in the size of the tape and the speed of compilation, by using the special instruction

upper case delimiters

and then writing all following delimiters in upper case without the underlining. Thus the example of P2.10 could then be written:-

```
CYCLE i = 1,1,p
CYCLE k = 1,1,r
c(i,k) = 0
REPEAT
CYCLE j = 1,1,q
CYCLE k = 1,1,r
c(i,k) = c(i,k) + a(i,j)*b(j,k)
REPEAT
REPEAT
REPEAT
```

The delimiter causes the compiler to replace each upper case letter by the equivalent underlined lower case letter, so that a mixture of normal and upper case delimiters can be used. If this is required only for certain parts of a program then the instruction

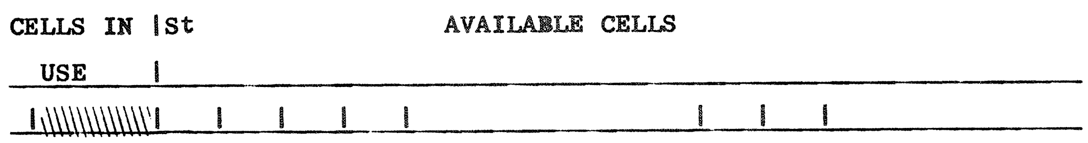
normal delimiters

can be used to return the compiler to its normal operation.

3 STORAGE ALLOCATION AND THE BLOCK STRUCTURE OF PROGRAMS

THE STACK

In order to illustrate the principles of storage allocation, we assume the following simplified picture of the data store (the stack), a fuller description being given in the section on the use of machine instructions.



Each cell or location represents a 48 bit word in the computer store and can be used to hold either a real or an integer variable. At any time during the running of a program, the stack pointer, St, points to the next available location i.e. it contains the address of the next free word.

In the examples that follow, shaded areas represent locations which hold information essential to the program, such as array dimensions and origins, and are not of importance in the context of this section. Each area may in fact consist of several locations. Cells which are allocated to variables are indicated by the presence of the name given to the variable.

STORAGE ALLOCATION DECLARATIONS

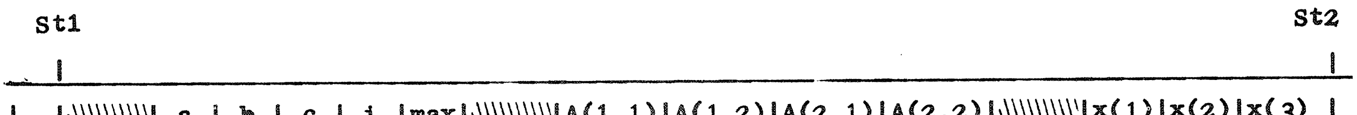
The declarations which allocate storage space are

real integer array integer array

and to illustrate the stack mechanism we consider the following example:

```
begin
real   a, b, c; integer i, max
array  A(1:2,1:2), x(1:3)
```

After the above declarations the stack picture would be as below



St1 is the position of St before begin and St2 its position after the declarations. Any further declaration advances St by an appropriate amount, likewise any activity initiated by the instructions in the body of the block causes St to be advanced(either explicity or implicity) still further. Finally when end or end of program is reached, then St reverts to St1.

Variables declared by real and integer are called FIXED VARIABLES, because the amount of storage space required can be determined at compiler time. Array declarations, however, may have general integer expressions as the parameters and hence have dynamic significance. For example one might have a declaration such as

```
array A,B(1:m, 1:n),x(1:n)
```

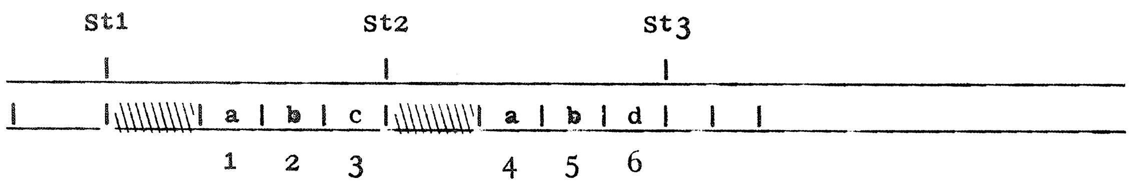
In this case the space allocated will depend on the computed values of m and n and cannot be determined at compiler time. The stack pointer St is thus advanced in several stages following the initial step which reserves space for all the fixed variables.

BLOCK STRUCTURE OF PROGRAMS

This is illustrated by the following example:-

```
begin
real a,b,c
a = 1; b = 2
c = a+b
  begin
  real a,b,d
  a = 2; d = 1
  b = c
  c = 4
  end
a = a+b+c
end
```

The stack picture associated with the above block is given below:



Before the first begin St is at St1, and moves to St2 on entering the outer block. After the second begin St is at St3 and reverts to St2 when end is reached. At the second end, corresponding to the first begin, St assumes its original position, St1.

In the diagram, positions 1, 2, 3 correspond to the declarations of the outer block, and 4, 5, 6 to those of the inner block. After the instruction c = a+b, the value 3 is left in position 3; while the instructions of the inner block leave the values 2, 1, 3, 4 in the positions 4, 6, 5, 3 respectively. The last instruction of the outer block leaves the value 7 in position 1.

Thus the variables a, b of the inner block do not conflict with a, b of the outer block, while a reference to c in the inner block is taken to refer to the variable of that name declared in the outer block. We say that a,b are LOCAL names to the inner block and c is a NON-LOCAL name. We also note that the information stored in the variables of the inner block is lost when the block is left, and that we could not refer in the outer block to a variable declared in the inner block.

Futher details of the structure of programs will be given in the section on routines, and for the present the following notes on blocks will be sufficient.

1. Blocks may contain any number of sub-blocks and blocks may be nested to any depth.
2. Names declared in a block take on their declared meaning in the block and in any sub-blocks unless redeclared in the sub-block.
3. Labels are local to a block and transfers of control are only possible between statements of the same block.
4. The outermost block of a program is terminated by <u>end of program</u>, which causes the process of compiling to be terminated and transfers control to the first instruction of the program.

A simple and common use of block structure arises when reading arrays from tape, each array being preceded by its dimensions.

For example:-

```
      begin
      integer m,n
  1:  read(m,n)
        begin
        array A(1:m,1:n)
        ____
        ____
        _ _
        ____
        end
      ->1
      end
```

If the <u>begin</u> and <u>end</u> defining the sub-block were not included, then the stack pointer would be advanced further each time a new array was read, without ever being reset, and this could be very wasteful of storage space, particularly for very large values of m and n.

4 ROUTINES

BASIC CONCEPTS

A large program is usually made up of several routines each of which represents some characteristic part of the calculation. Such routines may be called in at several different points in the program, and their design and use is a fundamental feature of the language. The introductory example consisted of a main block only (delimited by begin and end of program) although it makes reference to the routines 'read', 'print', 'newline', which are permanently available in the machine. In exactly the same way however, the user may call in routines which he has written himself in Autocode language. Consider for example a routine to evaluate

$$y = a(m) + a(m+1)x + \ldots\ldots\ldots\ldots + a(m+n)x\uparrow n \qquad (n \geq 0)$$

where the coefficients are selected from some vector a.

```
routine poly(real name y, array name a, real x, integer m,n)
integer i
y = a(m+n) ; return if n = 0
cycle i = m+n-1,-1,m
y = x*y+a(i)
repeat
return
end
```

Given the values of x,m,n and the addresses of y and the array elements a(i), it evaluates the polynomial and sets y to this value.

The statement end is the formal or written end of the routine while return is the dynamic end, i.e. it is the instruction which returns control to the main routine. Where the formal end is also a dynamic end as in the present example the return instruction preceding end can be omitted; in this case end serves for both purposes.

NOTES

1: There can be any number of alternative exit points in a routine - i.e. return can occur more than once.

2: return is a member of the FORMAT CLASS[UI] - i.e. it can be made conditional, as above.

This routine can be EMBEDDED and used in a main routine as illustrated below.

```
begin
real U, V, z, x ; integer m
array b(0:15), c(0:50)
routine spec poly (realname y, array name a, real x, integer m,n)
    ,
    ,
    ,
poly(U,b,z,0,m)
    ,
    ,
    ,
poly(V,c,x^2,20,10)
    ,
    ,
stop

  routine poly(realname y, array name a, real x, integer m,n)
  integer i
  y = a(m+n); return if n = 0
  cycle i = m+n-1,-1,m
  y = x*y+a(i)
  repeat
  return
  end

end of program
```

The routine is called in by the main routine whenever the name 'poly' appears. The first reference to 'poly' would cause the poly routine to evaluate

$$U = b(0) + b(1)z + \ldots + b(m)z{\uparrow}m$$

and the second would cause it to evaluate

$$V = c(20) + c(21)x^2 + \ldots + c(30)x{\uparrow}20$$

The parameters in the routine specification and routine heading are the FORMAL PARAMETERS and the parameters in the call statements are the ACTUAL PARAMETERS (see next section).

The body of the routine may be considered as a block delimited by routine and end, and the concepts of storage allocation, local and non-local names etc. apply to routines in exactly the same manner as for blocks. In fact a block may be considered as being an open routine without parameters.

Any number of routines may be embedded in a main routine in the above fashion and they are referred to as SUBROUTINES of the main routine. If the body of a subroutine occurs before any reference to it in the main routine, the routine specification may be omitted, but by convention it is usual to place all the subroutine specifications among the declarations at the head of the main routine and the bodies at the end.

FORMAL PARAMETERS AND ACTUAL PARAMETERS

The parameters of the routine are the items of information which specify the action of the routine whenever it is used. The formal parameters are the names by which this information is referred to inside the routine itself, and the actual parameters are the names or expressions which are substituted for the formal parameters whenever the routine is used in the main program.

For each type of formal parameter there is a permissible form for the actual parameter, as shown in the following table:-

Formal parameter type	Corresponding actual parameter
integer name	name of an integer variable
real name	name of a real variable
integer	an integer [EXPR] (similar to an integer assignment)
real	a general(i.e. real or integer)[EXPR] (similar to a real assignment)
integer array name	name of an integer array
array name	name of a (real) array
integer array	name of an integer array
array	name of a (real) array (the difference between these and the previous pair of parameters is explained below)
routine type i.e. routine real fn integer fn	Sometimes it is required to pass on the name of a routine as a parameter. In this case the actual parameter is the name of a routine which must correspond in type and specification with the formal parameter, the specification of which will b found in the routine body

In the example of a routine to evaluate a polynomial described earlier, the formal parameter y is the name of the variable to which the result is assigned, and the corresponding actual parameter must be a name, in this case the name of a real variable. The formal parameter then is of type real name. A reference to y inside the routine is essentially a reference to the non-local variable named by the actual parameter. The same applies to the array name parameter a, a reference to a inside the routine being a reference to the non-local array whose name is substituted for a in the calling statement.

The formal parameter real x on the other hand can be replaced by a general arithmetic expression, which is evaluated and assigned to the local variable x which is specially created in addition to any local real variables declared in the routine. The same applies to the formal parameters integer m,n. These are essentially local quantities, and expressions are substituted in place of them are evaluated and the resultant values assigned to the local integer variables m and n, which are lost on exit from the routine. Consequently the routine should place the information it produces in variables which are called by NAME (such as x and a).

The formal parameters x, m, n are said to be called by VALUE in so far as it is only the values of the corresponding actual parameters which are of interest. This is the essential difference between the formal parameter types array and array name (or integer array and integer array name). In the former case the array named by the actual parameter is copied into a specially created local array, and a reference to the name in the routine is taken as referring to this local array. As the copying process can be time-consuming and space-consuming, arrays should be called by NAME if at all possible, especially if they are large.

Another example of a routine is the following

```
routine matmult(arrayname A,B,C integer p,q,r)
integer i,j,k ; real c
cycle  i = 1,1,p
cycle j = 1,1,r
c = 0
cycle k = 1,1,q
c = c+A(i,k)*B(k,j)
repeat
C(i,j) = c
repeat
repeat
end
```

This forms the product of a p x q matrix A and a q x r matrix B. The result, a p x r matrix, is accumulated in C. The routine assumes that the first element of each matrix has the suffix (1,1). A typical call sequence might be

```
mat mult(H, x, y, 20, 20, 1)
```

where H, x, y had been declared by

```
array H(1:20,1:20), x,y(1:20,1:1)
```

FUNCTION ROUTINES

When a routine has a single output value it may be written as a function routine and then used in an arithmetic expression in the same way as the permanent functions (cos, sin etc.). For example, the polynomial routine described earlier may be recast as a function routine as follows:-

```
real fn poly(arrayname a, real x, integer m,n)
integer i ; real y
y = a(m+n) ; if n = 0 then result = y
cycle i = m+n-1,-1,m
y = y*x+a(i)
repeat
result = y
end
```

NOTES

1: In general, the exit from a routine is of the form : result = [EXPR] and this causes the EXPRESSION on the right hand side to be evaluated as the value of the function.

2: result = [EXPR] acts as the dynamic end of a function (i.e. it corresponds to return in a routine), and may appear any number of times within the function.

3: result = [EXPR] is a number of the FORMAT CLASS[UI] - i.e. it may be made conditional.

The specification of the above routine would be written

```
real fn spec poly(arrayname a, real x, integer m,n)
```

and the routine can be called in an arithmetic statement, for example

```
y = a*b + 2h*poly(c,1/x,0,16)
```

An example of an integer function is given next. It selects the index of the maximum element x(k) in a set of array elements x(m), x(m+1),....,x(n) $(n \geq m)$

```
    integer fn max(arrayname x, integer m,n)
    integer i,k
    k = m
    ->1 if n = m
    cycle i = m+1,1,n
    k = i if x(i) > x(k)
    repeat
1:  result = k
    end
```

A call sequence for this function routine might be

```
y = 1 + x(max(x,1,100))
```

SCOPE OF NAMES

In general all names are declared at the head of a routine either in the routine heading or by the declarations <u>integer</u>, <u>real</u>, <u>array</u>, etc., and the various routine specifications.

Therefore they are local to that routine and independent of any names occurring in other routines. However, if a name appears in a routine which has not been declared in one of the above ways, then it is looked for outside i.e. in the routine or block in which it is embedded. If it is not declared there it is looked for in the routine or block outside that and so on until the main block is reached.

Now the main block is itself embedded in a permanent block at 'zero level' which contains the PERMANENT material, so that if a name is not found in the main block it is looked for among these. The permanent names may in fact be redeclared locally at any level, but clearly it would be unwise to assign new meanings to such routines as 'log', 'print', etc. This outer block also contains supervisory material for controlling the entry to and exit from the main block. Very often, the only non-local names used in a routine will be the permanent names. The level at which a name is declared is sometimes referred to as its 'textual' level.

PERMANENT ROUTINES

The permanent names include the standard functions, sin, log, int, etc. and the basic input/output routines read, print etc. These routines are used in a program without declaration and without the necessity of inserting the routine bodies, since these are permanently available at level zero. A full list of the permanent routines is given in Appendix 2.

[NOTE : the standard functions (and the same applies to 'read') are not strictly routines : <u>THEIR NAMES CANNOT BE SUBSTITUTED AS ACTUAL PARAMETERS IN PLACE OF FORMAL PARAMETERS OF ROUTINE TYPE.</u> they would first have to be redefined and renamed as formal routines.]

FUNCTIONS AND ROUTINES AS PARAMETERS

This is illustrated by the following example involving an integration routine

```
routine spec integrate(real name y, real a,b,integer n, real fn f)
```

which integrates a function f(x) over the range (a, b) by evaluating

$$y = (f(0) + 4f(1) + 2f(2) + \ldots + 4f(2n-1) + f(2n))(b-a)/(6n)$$

where $f(i) = f(a + \frac{1}{2}i*(b-a)/n)$

An auxiliary routine is required to evaluate f(x) and details of it must be passed on to the integration routine. This is done by means of the formal parameter type [RT] as defined earlier, and the body of the routine might then be:-

```
routine integrate  (real name y, real a, b, integer n, real fn f)
real fn spec f(real x)
real h;  integer i
h = ½(b-a)/n
y = 0
cycle i = 0,2,2n-2
y = y+2f(a+i*h)+4f(a+(i+1)h)
repeat
y = (y-f(a)+f(b))h/3
end
```

To enable instructions such as

```
y = y+2f(a+i*h)+4f(a+(i+1)h)
```

to be translated, a specification of the formal parameter f is required. In this case the delimiter real fn spec can be replaced by spec since the type of the function is given explicitly by the formal parameter itself.
Now consider a programme to evaluate

```
z = exp(-y)cos(b*y)dy
```

for various values of b read from a data tape, the last value being followed by 1000, using for n the integer nearest to 10b.

```
  begin
  routine spec integrate (real name y,real a,b,integer n,real fn f)
  real fn spec aux (real y)
  real z, b
  comment Simpson rule integration
1:read (b)
  if b = 1000 then stop
  integrate (z, 0, 1, int(10b), aux)
  newline
  print (b, 1, 2);spaces(2);print (z, 1, 4)
  -> 1
    real fn aux(real y)
    result = exp(-y) cos(b*y)
    end
   ________________________
  |  routine integrate     |
  |________________________|
  end of program
```

NOTES

1: That the names given to the auxiliary routine and its parameters need not be the same in the integration routine as in the main program, but they must correspond in type.

2: Since the result of the intergration is a single quantity, the routine could be recast as a real fn :-

```
real fn spec integrate(real a,b, integer n, real fn f)
```

and called by, for example:-

```
print(integrate(0,1,int(10b),aux),1,6)
```

RECURSIVE USE OF ROUTINES

The name of a routine is local to the routine or block in which its specification appears, and so the body of the routine is within the scope of its own name. Hence it may call itself. It may also call itself indirectly by invoking other routines which make use of it. On each activation of the routine a fresh copy of the local working space is set up in the stack, so that there will be no confusion between variables on successive calls. (This does not apply however to own variables. See next section.) Some criterion within the body of the routine must eventually inhibit the calling statement and allow the process to unwind. Functions defined recursively, for example:-

$$n! = n(n-1)! \; , \quad n > 1$$
$$\quad = 1 \; , \quad n = 1$$

can be implemented in this way, but it is always more efficient to use recurrence rather than recursive techniques.

OWN VARIABLES

When a routine is left any information stored in variables corresponding to local declarations in that routine is lost, and no furthur reference may be made to it. In some cases it may be desirable to retain some of this information and be able to refer to it on a subsequent entry to the routine. This may be accomplished by prefixing the relevant declaration by own.

For example

```
own real a, b;  own array A (1:10)
```

The effect of own is to allocate storage space for the named variables in a part of the store which is not overwritten when other routines are called in and to set them to zero. This is done during the compiling of the program and hence does not have dynamic significance; as a consequence an own array declaration must have parameters which are integer constants.

5. INPUT AND OUTPUT OF DATA

The input and output of data will generally be accomplished by means of permanent routines. In this section these permanent routines are described and the precise form of data is given.

SELECTION OF DATA CHANNELS

The selection of an input channel is performed by the routine:-

routine spec select input (integer i)

This selects the input channel corresponding to the value of i, and this channel, together with the particular input device assigned to it in the Job Description (see Section 7), remains selected until another 'select input' instruction is encountated.

Output channels are selected in a similar way, by means of the routine:

routine spec select output (integer i)

In both cases channel 0 is initially selected, and in the absence of a channel selection instruction, remains selected during the execution of a program.

BASIC INPUT ROUTINES

Decimal numbers may be read from a data tape by means of the routine

routine spec read([VARIABLE])

This reads a decimal number from the currently selected data channel and places it in the location specified by the [VARIABLE], which may be of type integer or real. The routine reads numbers in either fixed or floating point form, for example:-

-0.3101   18   7.132α-7   3.1872α14

A number is terminated by any character other than a decimal digit, the first decimal point, or an exponent. An exponent consists of α followed by an optional number of spaces, an optional sign, and the decimal digits. Spaces and newlines preceeding a number are ignored, but all other symbols cause the routine to signal a fault (but see NOTE on P5.4). A fault is also indicated if a number assigned to an integer variable is not integral.

It should be noted that a single space is sufficient to terminate a number, and that no spaces are allowed within the mantissa or within the numerical part of the exponent (unlike constants appearing in a program where all spaces are irrelevant and the number is terminated by the following name or delimiter).

Further since 'tabs' are converted to a number of spaces, numbers may be separated by 'tabs'. Several numbers in a sequence may be read by the routine:-

```
routine spec read([VARIABLE LIST])
```

For example, read(a,i,X(i))
This is treated as if it were a series of instructions

```
read (a) ; read(i) ; read(X(i))
```

hence the subscript of X(i) takes the value just assigned to i.

The read routine is an exception to the general form of a routine, as it may have an indefinite number of real name and integer name parameters.

Successive numbers on a data tape may be read so as to fill an array by means of the routine

```
routine spec read array(arrayname A)
```

For example:-
```
array A(1:20, 1:20)
read array (A)
```

would cause the next 400 numbers on the data tape to be read so as to fill the array A, row by row. It is thus equivalent to

```
array A(1:20, 1:20)
integer i,j
cycle i = 1,1,20
cycle j = 1,1,20
read (A(i,j))
repeat
repeat
```

Three permanent routines are provided for manipulating alpha-numeric data. The first:-

```
routine spec read symbol (integername i)
```

reads the next symbol (simple or compound) from the selected channel, converts it into a numerical equivalent and places the result in the specified integer location.

For example, if the next character on the data tape were an asterisk (numerical equivalent 14) the instruction 'read symbol (p)' would set the value of the integer variable p to 14 and move to the next character on tape.

The second allows the next symbol on the data tape to be inspected without moving on to the following one. It is

integer fn spec next symbol

The third:-

routine spec skip symbol

passes over the next symbol without reading it.

A table of numerical equivalents and a description of the formation of compound symbols is given in Appendix 5.

It is in testing symbols that the alternative form of a constant is useful. For example, we could test if the next symbol on a tape were an asterisk by

```
        ->1 if next symbol = 14
or      ->1 if next symbol = '*'
```

Since spaces and underlined spaces are ignored in a program, and newline and semicolon are used as terminaters, special symbols are provided to represent them. Thus a space can be tested for by

```
        ->1 if next symbol = 's̸'
```

The symbols are:-

s̸	$	representing	a space
s̸̲	$̲	''	an underlined space
n̸	n⃒	''	a newline
;̸	;⃒	''	a semi-colon

If the data itself contains these special symbols, then they can be tested only by using the internal equivalent.

Finally there is a permanent input routine which permits the reading of an indefinite number of decimal numbers into successive storage locations, stopping when a particular symbol on the data tape is reached. This routine is

routine spec read sequence (addr s, integer p, integer name n)

The formal parameter type addr is explained in Section 9; for the present purpose it is sufficient to say that the actual parameter will be the name of a variable, representing the first location into which the numbers are to go. p is the numerical equivalent of the terminating character, and on exit from the routine, n contains the number of numbers that have been read.

As an example of the use of the above routine, suppose a data tape contains an unknown number of numbers, but less than 1000, and that the last number is followed by an asterisk. Then the instructions

```
array X (1:1000)
integer n
read sequence (X(1), 14, n) [or :    read sequence(X(1),'*',n)]
```

would cause the successive numbers to be read into X(1), X(2), etc. If there were 800 numbers in the sequence, then n would be set to 800 when the routine was left.

NOTE

On input, each line of data is reconstructed to give an image of the print-out produced by the Flexowriter. Thus 'backspace','tab','upper case' and 'lower case' do not appear as characters in the reconstructed line, since they do not appear on the print-out. 'Tab' produces an equivalent number of spaces, 'backspace' helps form a composite character, and non-significant cases are ignored. Those positions containing an erase are then deleted from this line. The line image is normally 160 characters, but where the tab and backspace facilities are avoided, lines can be of any length, sections of 160 characters being taken serially.

BASIC OUTPUT ROUTINES

The routines for the output of a single decimal number are

```
routine spec print fl (real x, integer m)
routine spec print (real x, integer m,n)
```

The first of these prints the value of x (which may of course be a general [EXPR]) in floating point form, standardised in the range $1 \leq x < 10$, with m decimal digits after the decimal point. The number is preceded by a minus sign if negative, and a space if positive. The exponent is preceded by α and consists of a space or a minus sign and two decimal digits, the first of which is replaced by a space if it is not significant.

The second routine prints the value of x in fixed point form with m digits before the decimal point and n after. Non-significant zeros, other than one immediately before the decimal point, are suppressed, and a minus sign or space precedes the first digit printed. If $|x| \geq 10\uparrow m$ then extra digits are included before the decimal point, the effect being to spoil any vertical alignment of the printed page.

It should be noted that no terminating characters are included by the above routines. They may be included by the user by means of the routines:-

```
routine spec newline
routine spec space
routine spec newlines(integer n)
routine spec spaces (integer n)
routine spec tab
routine spec print symbol (integer i)
```

The first of these resets the carriage of the appropriate printer (or punches the newline character), and the second causes the printer to skip a character position. If a number of consecutive spaces or newlines are required, the third and fourth routines may be used, for example:-

```
spaces (5)
newlines (3)
```

The fifth routine punches the tab character or causes the printer to move to the next tab setting. These settings are at positions 8, 16, 24, 32, 48, 64, 80, 96, 112, 128, 144, and 159. The sixth prints the symbol corresponding to the value i.
The routine:-

```
routine spec newpage
```

causes the lineprinter to commence a new page, if the output device is a line printer, or punches 30 newline characters if it is a punch.
The routine:-

```
routine spec runout (integer n)
```

punches n runout characters (used to seperate sets of results, for example) on the punch. It has no effect if the output is on a line printer.

Arrays of numbers may be output by means of the routines

```
routine spec print array fl (array name A, integer m)
routine spec print array (array name A, integer m,n)
```

For a one-dimensional array, the elements of the array are printed across the page, each number being terminated by two spaces, or a newline if the right hand edge of the page has been reached. The successive rows of a two dimensional array are printed as above, successive planes of a three dimensional array are printed as two dimensional arrays, and so on. Each array is started on a newline and the printing style for the individual numbers is the same as that of the 'print fl' and 'print' routines.

CAPTIONS

There is a special facility for printing captions. For example

caption s̸s̸s̸s̸ TABLE s̸ OF s̸ TEMP s̸ AGAINST s̸ VOL

This prints the information after caption up to, but not including, the terminating symbol 'newline' or 'semi-colon'. Since spaces and underlined spaces are ignored and 'newline' and 'semi-colon' are used as terminators, we also use the special characters:-

s̸	or	$
s̸̲	''	$̲
n̸	''	n⃒
;̸	''	;⃒

Thus

```
newline
caption  A s̸ = s̸s̸ ; print (y,1,3); newline
caption  B s̸ = s̸s̸ ; print (z,1,3); newline
```

would be printed as

```
A =   1.712
B =  -2.380
```

In general c̲ can be used (in its usual sense) in a caption if the information is too long to fit on one line across the page. In view of this if an underlined word ending in c̲ is used at the end of a caption, it must be terminated by 'semi-colon' not 'newline'.

BINARY INPUT AND OUTPUT

Binary tape may be read and punched by means of the routines

```
routine spec read binary (integername i)
routine spec punch binary (integer i)
```

The first reads the next row of holes on the tape as a binary number (in the range 0-127, with the tape so oriented that the sprocket hole comes between the digits of value 4 and 8), and places it in the named variable. Binary data tapes must be preceded by ***B or, if they contain characters of of even parity, by

***P
***B

The second punches the seven least significant binary digits of the integral part of the integer expression as a row of holes on the output tape.

NOTE: Cards or 5-hole tape may be used in which case the operations are on 5 or 12 digits rather than 7.

6 MONITOR PRINTING AND FAULT DIAGNOSIS

FAULT MONITORING

There are two types of fault which can be detected by the compiler, those which can be found during compiling and those which become evident during the running of the compiled program. To aid the programmer in correcting these faults information is automatically printed out where a fault occurs.

COMPILER TIME MONITORING

During compiling an outline of the program is produced as an aid to the finding of faulty instructions. It also associates each block and routine with its serial number, for use in tracing faults found at run time (see later). All faults during compiling are monitored. Those to which a line number can be attached, such as NAME NOT SET, are preceded by it, while those which can only be found at the end of a routine such as TOO FEW REPEATS are monitored after the END. In calculating the line number, blank lines are ignored, and lines joined by the continuation symbol c̲ count as one. Finally at the end of each routine all the non-local variables except the permanent routines and functions are printed out. Although these do not necessarily indicate a fault, they may indicate a name which should have been declared locally. A typical program monitor might be

```
  0      BEGIN BLOCK : SERIAL NO = 89, M/C ADDRESS = 2721
 26*     NAME TEMP NOT SET
 55*     LABEL 7 SET TWICE

 70        BEGIN ROUTINE POLY:SERIAL NO = 90, M/C ADDRESS = 3210
115*       NAME TEMP NOT SET
115*       REAL NAME X IN EXPRESSION
120        END ROUTINE POLY : OCCUPIES 256 M/C INSTRUCTIONS
   *       LABEL 18 NOT SET
           NON-LOCAL VARIABLES A TEMP1 S1

182      END OF PROGRAM' OCCUPIES 800 M/C INSTRUCTIONS
```

The above should be self-explanatory. It indicates that the program started at line 0 and finished on line 182. These are physical lines and exclude all blank lines on the print-out. The outer block is given the serial number 89. The routine POLY started on line 70 and was given the serial number 90. There were mistakes in lines 26 and 55 and two in line 115. Finally label 18 was not set in the routine POLY.

Since there may be more than one statement on a line, it is not possible to tell specifically which statement is involved but the faults are printed in the order in which they are discovered. A full list of faults is given in Appendix 4 together with a brief description of their nature.

RUN TIME MONITORING

During the running of a program certain faults may be detected both by the compiler and by the machine and its supervisor program. For example, the supervisor program detects the case where the square root of a negative argument is being requested and the compiler detects faults connected with switch and test instructions.

The standard procedure is to print out 2 lines of information specifying the fault and the line on which it occurs followed by a list of useful information found in the FIXED part of the stack. For example:-

```
LINE 117 ROUTINE 90
EXP OVERFLOW

ROUTINE 90

ARRAY(1:10,1:5)
ARRAY(1:10,1:5)
10  5
0.3333333341α  0   -1.1249999997α -1   0.0000000000α-99
6  4
CYCLE(CURRENT VALUE = 6, FINAL VALUE = 10, INCREMENT = 1)
CYCLE(CURRENT VALUE = 4, FINAL VALUE = 5, INCREMENT = 1)

BLOCK 89

0.0000000000α-99   3.7152463802α   3
10  5  3  6
ARRAY(1:10,1:5)
ARRAY(1:10,1:5)
```

indicates that an instruction in line 117 routine 90 (the line number refers to the entire program, not just the routine), resulted in exponent overflow. Then follows a list of the scalars, array dimensions and cycles of the routine in the order in which they were originally declared, followed by the list for the routine or block which called this routine, then that of the routine which called it and so on until the main block is reached. Thus the above might correspond to:-

```
begin
real a,b
integer i,j,k,l
array X,Y(1:10,1:5)
       '
       '
       '
       '
matrix fn (X,Y,i,j)
       '
       '
       '
  routine matrix fn(arrayname A,B integer m,n)
  real a,b,c ; integer i,j
       '
       '
       '
  cycle i = 1,1,m
  cycle j = 1,1,n
       '
       '
       '
  repeat
  repeat
  end
 end of program
```

NOTES

1: This fault print out must be interpreted with care. When the fault occurs, the fault print out routine looks in the STACK to find the fixed variables and interpret them (see Section 10). Now every location in the store initially looks as if it contains a real quantity. Thus:-

(i) until an integer is assigned a value, it will appear as (and be printed as) a floating point quantity (probably zero).

(ii) until an array declaration is obeyed, it will appear as 2 floating-point quantities.

(iii) until a cycle has been entered, it will appear as 3 floating-point quantities.

Conversely, since all sub-routines of a program share the same space, then on entry to the second and subsequent routines, the stack will contain the values left by the previous routine and these will be interpreted accordingly, if the current routine does not alter them.

2: The 'CURRENT VALUE' attributed to a cycle is the value of the <u>integer</u> name used on the left hand side of the instruction at the time of the fault. Thus if a program consisted of a number of cycles one after other, controlled by i, and the fault were inside the last cycle, then all cycles would have the same 'CURRENT VALUE' - the current value of i.

3: Only cycles, arrays, integers and reals are distinguished.

(i) for <u>integer name</u>'s and <u>real name</u>'s the address of the actual paramter is printed (as an integer) .

(ii) for <u>array fn</u>'s (see Section 9) its parameters are printed (as integers).

(iii) for <u>routines</u> and <u>function</u>'s used as parameters, six real quantities are printed.

(iv) for <u>complex</u> quantities, the real and imaginery parts are printed in floating point style.

FAULT TRAPPING

The above standard monitoring procedure involving the termination of the program, is not always convenient. For example if a program is dealing with a series of data sets, it may be preferable to restart on the next 'case' in the event of (say) EXP OVERFLOW rather than terminate the entire job.

An instruction is provided which enables the user to trap certain faults and transfer control to some preassigned point in the program. It takes the form:-

```
              fault [FAULT LIST]
    where     [FAULT LIST] = [N-LIST]->[N][REST OF FAULT LIST]
```

For example:-

```
              fault 1,2,5 ->18, 3,4 ->10
```

means 'if a fault of type 1,2 or 5 subsequently occurs then jump to label 18 ; anc
if a fault of type 3 or 4 occurs then jump to label 10.
The effect is to preserve all the necessary control data to enable control to revert to this point in the program (and then jump to label 18 or 10) should one of the specified types of faults occur at some lower (or the same) level. The label must be in the same block as the trapping statement, which will usually be in the 'main' block at (say) level 1 or 2.

The fault instruction has dynamic significance, and a following fault instruction can change this trapping action. All faults not referred to by a fault instruction are dealt with in the usual way (i.e. they cause the program to be terminated).

The first two lines of the standard fault monitoring are printed for faults trapped in this way. Appendix 4 contains the list of faults which can be trapped and the corresponding fault numbers.

FAULT DIAGNOSIS

Provision is made to compile certain checking facilities in selected parts of the program. Having been compiled they can then be switched on or off at run time by means of instructions in the program. The formats are:-

```
              compile[check]
              stop[check]
              [check]on
              [check]off
```

The first pair of statements are DECLARATIVES which delimit the areas of the program in which provision is to be made for the particular checking facility. The second pair are INSTRUCTIONS which turn the facility on or off (initially they are on). They do this by setting a certain switch which is examined whenever the facility is about to be executed. If the relevant switch is on then the facility is executed, if off it is skipped. If the facility has not been compiled in the first place then the instructions have no effect. This switch setting is extremely fast so that there is nothing to be gained from recording the current state of the switch (in some integer, say), and testing this before each setting order. For example, the following sequence causes queries to be printed every tenth time round the cycle.

```
cycle i = 1,1,m
queries off
if fracpt(i/10) = 0 then queries on
    ,
    ,
    ,
    ,
repeat
```

The switch sensing on the other hand is a time consuming operation and it is for this reason that the declaratives are provided to delimit the areas of the program in which this takes place. In most cases, however, the check is compiled over the entire program.

The checking facilities in question are described by the phrase:-

[check] = queries, routine trace, jump trace, array bound check

They will be described in turn.

QUERY PRINTING

Any arithmetic instruction (including complex) can be followed by a ?, for example:-

```
a = b(i) + c?
```

When the facility is operative the new value on the l.h.s. is printed every time the instruction is obeyed. The style of printing will be fixed, floating, or complex floating according as the l.h.s. is of integer, real, or complex type.

[Unlike the other facilities, ?'s are normally compiled so that a compile queries at the head of a program is redundant. Also ignore queries is equivalent to stop queries.]

ROUTINE TRACING

When the routine trace is operative it causes the routine number to be printed each time a routine or block is entered and left. The correspondence between the routine number and the name can be found from the program outline produced during compilation. The printout of a routine trace might appear

R95 R97 RF96 END96 END97 R99.......

Here R, RF denote routine and real fn respectively. The full list of abbreviations is:-

B begin	R routine	
IF integer fn	RF real fn	CF complex fn
IM integer map	RM real map	CM complex map

JUMP TRACING

The jump trace facility allows the flow of the program to be followed in greater detail. For every jump instruction obeyed the label number is printed; for every test the value of the label at which the [COND] is satisfied is printed; for every switch the value of the switching index is printed. Thus a label trace might appear

$\rightarrow$3 T1 $\rightarrow$4 $\rightarrow$6 S3 $\rightarrow$7 $\rightarrow$8 $\rightarrow$9

Here T and S refer to test and switch respectively. If the label and routine trace are both operative the print out might appear:-

R95 $\rightarrow$3 T1 $\rightarrow$4 $\rightarrow$6 R97 S3........

ARRAY BOUND CHECK

If this facility is operative the values of the subscript expressions in all array elements are checked to see if they lie in the range specified by the bound pairs in the array declarations. If not, the program is terminated with the appropriate monitoring.

OTHER CHECKING FACILITIES

Certain checks are built into the object program e.g., whether a cycle instruction calls for an integral number of cycles and whether a switch index is out of range or corresponds to a label not set. All are time and space consuming operations. They can be removed from an object program which is otherwise ready for production by means of the declaration

production run

7 PRESENTATION OF COMPLETE PROGRAMS

JOB DESCRIPTIONS

The running of programs on the computer is controlled by a supervisor program held permanently in the machine. The supervisor accepts complete programs as a series of tapes (program and data) and a JOB DESCRIPTION which may be on a separate tape or included with the program or data. A full description of the system is given elsewhere [1], and in this section we give examples to illustrate the general principles of job descriptions.

PROGRAM AND DATA ON SAME TAPE

The simplest form of job consists of job decription, program and data on the one tape. For example:-

```
JOB
UMA, JONES 5/2
OUTPUT
0 LINEPRINTER 100 LINES
STORE 32 BLOCKS
COMPUTING 10000 INSTRUCTIONS
COMPILER AA

begin
 ________________
|                |
|  PROGRAM       |
|                |
|________________|

end of program

 _________________
|                 |
|      DATA       |
|                 |
|_________________|

***Z
```

NOTES

1. The title (2nd line) identifies the job. The first few characters will be a code to identify the particular organisation and the rest will be information of an arbitrary form to identify the programmer and the program within the organisation.

Reference

[1] 'Documents and Job Description' I.C.T. Ltd., October 1963. This gives a full description of the possible arrangements of program and data tapes and the utilisation of the multi-channel input/output facilities on Atlas.

2. The OUTPUT information says that reference to channel 0 in the program means the lineprinter (if no output channel is selected in the program channel 0 is used). The number of LINES gives an upper limit to the amount of output that is to be permitted.

2. STORE gives an upper limit on the number of 512 word main store blocks used by the program and data.

3. COMPUTING gives a limit on the running time of the program. An 'INSTRUCTION' is equivalent to 2048 machine instructions.

The OUTPUT, STORE, and COMPUTING sections are optional, both individually and collectively. If they are omitted the allowances given in the above example are assumed, i.e., 100 LINES, 32 BLOCKS, 10000 INSTRUCTIONS. These should in fact be adequate for most small problems, except possibly the 100 LINES. The foregoing example could therefore be shortened to:-

```
JOB
UMA, JONES 5/2
COMPILER AA

begin

 ____________
|            |
|  PROGRAM   |
|            |
|____________|

end of program

 ____________
|            |
|    DATA    |
|            |
|____________|

 ***Z
```

A program tape is always assumed to be on input channel 0 so that in the above case, the data for the problem is also on channel 0, which is the channel used in the absence of a contrary 'select input' instruction in the program. ***Z is an end of tape marker and indicates that all the information on that tape has been read. This must be on a line of its own, and must be followed by at least one 'newline'.

PROGRAM AND DATA TAPES SEPARATE

Often when a program is being used for production runs, it is convenient to keep the program on a separate tape which is never changed. For each run the job description and data form a separate tape. For example:-

```
COMPILER AA
(title 1)
begin

 ________________
|                |
|    PROGRAM     |
|                |
|________________|

end of program

***Z
```

The data tape including the job description, would be

```
JOB
(title 2)
INPUT
0 (title 1)
SELF = 1
DATA

 ________________
|                |
|      DATA      |
|                |
|________________|

***Z
```

The input section gives the relevant program as being channel 0 (the program channel) and SELF = 1 indicates that the data tape is to be read as channel 1. Thus an instruction 'select input (1)' is required in the program. This tape could, if necessary, include any OUTPUT, STORE, and COMPUTING information since this is the part of the job description.

Possible titles for the above example might be

(title 1)	UMA, P10
(title 2)	UMA, P10/RUN 26

PROGRAM ON SEVERAL TAPES

It is often convenient to have the program itself on two or more distinct tapes, where, for example, the program may be so long that it would be physically unmanageable to keep it on one tape.

Alternatively the program may contain a large section (declarations and routines perhaps) which is common to many programs and which can conveniently be kept on a separate tape.

The instruction

now compile from input[N]

is used to switch the compiler from one input stream to another. For example:-

```
JOB
(title 1)
INPUT
1 (title 2)
2 (title 3)
COMPILER AA

begin

 ________________
|  FIRST PART    |
|      OF        |
|  PROGRAM       |
|________________|

now compile from input 1
***Z

COMPILER AA

(title 2)

 ________________
|  SECOND PART   |
|      OF        |
|   PROGRAM      |
|________________|
end of program
***Z

DATA
(title 3)

 ________________
|                |
|     DATA       |
|                |
|________________|
***Z
```

8 COMPLEX ARITHMETIC

As indicated previously, facilities exist for the manipulation of complex as well as real and integer quantities. complex quantities are stored as a pair of real numbers in consecutive locations (the real and imaginary parts respectively). The address of the complex quantity is that of the real part.

DECLARATIONS

All quantities must be declared before they are referred to. For example:-

```
real     R1, R2, R3
complex     z
complex array   P(1:10), Q(1:10,1:10)
```

causes 3 locations to be reserved for R1, R2, R3, 2 for z, 20 for P and 200 for Q.

STANDARD FUNCTIONS

The following standard functions are added to those previously given:-

re(z)	(real part of z)
im(z)	(imaginary part of z)
mag(z)	(modulus of z)
arg(z)	(argument of z - in radians)
conj(z)	(complex conjugate of z)

The argument z may be any [EXPR] (in the complex sense as described below) The functions

csin, ccos, ctan, cexp, clog, csqrt

have complex [EXPR]'s as arguments and yield results of complex type. For example if $z = x + \underline{i}y$, $\text{cexp}(z) = \exp(x)(\cos(y) + \underline{i}\sin(y))$
In the case of clog and csqrt it is the principal value which is computed, i.e., the value for which the argument θ lies in the range $-\pi \leq \theta < \pi$

ARITHMETIC EXPRESSIONS

The arithmetic expression [EXPR] is still of the form

[±'][OPERAND][OPERATOR][OPERAND][OPERATOR] .... [OPERAND]

but [OPERAND] is now expanded to be

[VARIABLE],[CONSTANT],([EXPR]),|[EXPR]|,[FUNCTION] or $\underline{i}$

Here $\underline{i}$ is a delimiter denoting the i (or j) of complex algebra notation.

Examples of this more general expression are:-

(V*conj(I) - I*conj(V))/(2$\underline{i}$)

(Z1Z2 + Z2Z3 + Z3Z1)/Z3

Y(1,2) + csin(conj(Y(2,1)))

RO*(1 + 2$\underline{i}$QOd)

$\underline{i}$

When a complex number is written out explicitly (say x + $\underline{i}$y), then it is regarded as 3 operands (x,$\underline{i}$ and y) connected by the two operators + and (implied) *. Thus if the brackets were omitted from the denominator in the first example it would mean

((V*conj(I) - I*conj(V))/$\underline{i}$)2

ARITHMETIC INSTRUCTIONS

The form of an assignment instruction remains

[VARIABLE] = [EXPR]

but [VARIABLE] now includes complex scalars and complex array elements. For example:-

Z = Z1Z2/(Z1 + Z2)

Y = G + $\underline{i}$2πf*c

A(p,q) = 2csin(2πz)

R = R1 + re(Z)

P = ½re(V*conj(I) + I*conj(V))

NOTES

1. Just as real quantities may not appear on the r.h.s. of an integer assignment (except as arguments of integer functions), so complex quantities may not appear in real or integer expressions. However, the functions

re(z), im(z), mag(z), arg(z)

convert from complex to real quantities and may therefore appear on the r.h.s. of a real assignment. In fact any function whose value is real regardless of its arguments may be used in a real expression (just as any integer function, regardless of its argument, may appear in an integer expression). Thus if X and B are real and Y complex then:-

```
X = B + im(Y)
```

is valid.

2. re(z) and im(z) are actual locations in the store and can therefore be used on the l.h.s. of an instruction (whose mode is then real). For example:-

```
re(z) = sqrt(2)
im(y) = 5 + im(z1)
```

However, mag(z) and arg(z), even though they do define z, are not locations in the store and cannot be used on the l.h.s. If a complex quantity is being evaluated by means of the evaluation of its magnitude (m) and argument (a), the assignment is done by

```
          z = m*(cos(a) + i sin(a))
or        z = m*cexp(ia)
```

CONDITIONS

In conditional operators, [EXPR]'s must be real (in the sense of note 1 of the previous section). Hence the following are legitimate:-

```
if arg (z) ≥ π/2 then -> 3
3 case mag(z) ≥ 1 :
```

ROUTINES AND FUNCTIONS

Since routines and functions are allowed to operate on complex quantities, the parameter types have been expanded to include

Formal parameter type	Corresponding actual parameter
complex name	name of a complex variable
complex	any expression (which will be evaluated as if for a complex assignment)
complex array name	name of a complex array
complex array	name of a complex array

The routine types [RT] have also been expanded to include complex fn. As an example we will rewrite the function routine for the polynomial

$$a(m) + a(m+1)x+.......... + a(m+n)x\uparrow n$$

assuming x and the coefficients a(i) to be complex.

```
complex fn poly (complex arrayname a, complex x, integer m,n)
integer i ; complex y
y = a(m+n) ; result = y if n = 0
cycle i = m+n-1, -1, m
y = y*x + a(i)
repeat
result = y
end
```

INPUT-OUTPUT OF COMPLEX NUMBERS

Data is punched in the form

[REAL PART] ± i [IMAGINARY PART]

but the individual parts can be punched in any acceptable 'real' form. Both parts must be punched however. For example:-

3+i4 0+i1 -0.5+i 0 1.17α3 -i2.13α4

They may be read by the instruction

```
read(Z1,Z2,Z3,Z4)
```

The permanent routines

```
print complex(complex z, integer m,n)
print complex fl(complex z, integer n)
```

print the value of z in the form

[REAL PART] ± i [IMAGINARY PART]

the individual parts being printed with the aid of the corresponding real routines, 'print' and 'print fl', using the same digit layout parameters. For 7-hole tape this form of output is compatible with the format for punching complex data.

NOTES

1. Spaces are permitted except in the two number parts themselves. In these they may only appear after an α (see description of basic input routines).
2. The other input and output routines described in Section 5 have not been generalised to deal with complex numbers.
3. One may of course read a pair of real numbers on a data tape as a complex number by the 'real' read instruction

   ```
   read(re(z),im(z))
   ```

9 <u>STORE MAPPING</u>

<u>THE ADDRESS RECOVERY FUNCTION</u>

The absolute address of any variable is not generally known in an Autocode programme, but it may be obtained by means of a standard function. For example:-

```
s = addr(A(0,0))
```

This places the address of A(0,0) into the variable s. The argument may be any variable, <u>real</u>, <u>integer</u>, or <u>complex</u> and the result is an integer giving the absolute address of the storage location allocated to that variable.

Absolute addresses are used in conjunction with array functions (see below) and with the 'storage' functions

```
integer (integer n)
real    (integer n)
complex (integer n)
```

These give the contents of the address in question as an integer, real, or complex number. In the last case the real and imaginary parts of the number are assumed to be in n and n+1. The actual parameter may of course be an <u>integer</u> expression e.g., s+k-1. These functions may be employed on the left hand side of an assignment statement as well as in an expression. Thus the pair of instructions

```
s = addr(a)
real(s) = b
```

are equivalent to

```
a = b
```

<u>ARRAY FUNCTIONS</u>

The declarations of Section 2 define variables and allocate storage space for them. In this section we introduce a declaration which defines variables as the numbers contained in storage locations that have already been allocated. This is of importance in communicating between routines with the <u>addr</u> type of formal parameter and in renaming variables (see below).

An example is

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

which defines X(i) as the <u>real</u> number in the storage location whose address is given by s+i*p. Thus it defines a vector X(i) in terms of an origin s and a dimension parameter p.

Array functions may define rectangular arrays with any number of subscripts. For example:-

<u>array fn</u> Y(s,p,q)
defines Y(i,j) $\underline{=}$ real (s+i*p+j*q)

<u>integer</u> or <u>complex</u> array functions may be defined by prefixing the declaration by <u>integer</u> or <u>complex</u>. (i.e. <u>integer array fn</u> X(s,p))
Array functions may also describe scalars. For example :-

<u>array fn</u> A(s)

defines A to be real (s). In this way, elements of a vector, say, can be given individual names.
The parameters in array functions may be general <u>integer</u> expressions.
As an example, assume that 100 storage locations have been allocated in some way, and that the starting address is given by the <u>integer</u> variable s1. Then to define the contents of these locations as a vector x(i), one could write

<u>array fn</u> x(s1,1)

x(0) would then correspond to the number in address s1, x(1) to that in s1+1 etc. If it is desired that the first location should correspond to x(1), the declaration would be written

<u>array fn</u> x(s1-1,1)

If we had wanted to define a 10 x 10 matrix, stored row by row rather than a vector, we could have written

<u>array fn</u> A(s1,10,1)

and A(0,0) would correspond to address s1.

<u>array fn</u> A(s1-11,10,1)

would define a matrix in the available space whose first element was A(1,1).

<u>NOTES</u>
1. If the suffices of arrays are to start from (1,1,---1) rather than (0,0,---0), an appropriate adjustment must be made to the expression giving the origin in the array function declaration.
2 Space redefined by <u>array fn</u>'s may still be referred to by its original name.

THE RENAMING OF VARIABLES WITHIN A BLOCK

We illustrate this with an example. Suppose we want to define and allocate storage for pairs of real variables x(i), y(i) so that they are in succesive locations. The array declaration will only define a vector or matrix array stored in the conventional manner, so we adopt the following device

```
begin
integer s
array a(1:2000)
s = addr(a(1))
array fn x(s-2,2), y(s-1,2)
-----
-----
-----
```

The first pair of numbers could then be referred to either as x(1), y(1) or a(1), a(2), the second by x(2), y(2) or a(3), a(4) etc. Since the array declaration is for 2000 variables, up to 1000 pairs x(i), y(i) can be accommodated.

As another example, suppose we have defined a matrix A and allocated storage for it by the declaration

```
array A(1:10,1:10)
```

and we wish to define the first column of A as a vector, then we could write

```
array fn y(addr(A(1,1)) - 10,10)
```

which defines y(i) $\equiv$ real (addr(A(1,1)) - 10 + 10*i)
i.e. as the first column of A. Thus y(1) is equivalent to A(1,1), y(2) to A(2,1), - - - -,y(10) to A(10,1).

In the case of complex array functions the user must take into account that a complex number occupies 2 consecutive locations. Thus if s1 is the address of Q(1,1) of a complex array Q(1:10,1:10), then

```
complex array R(s1-20,20)
```

defines a vector R(i) whose elements are the first column of Q, i.e., R(1) $\equiv$ Q(1,1)

STORE MAPPING ROUTINES

Storage functions of arbitrary complexity can be obtained by means of store mapping routines. These are essentially function routines which compute an address. For example:-

```
real map X (integer i,j)
result = s+½i*(i-1)+j-1
end
```

computes the address of the (i,j)th element of a real lower triangular matrix stored by rows starting with X(1,1) at location s. Here s is a non-local quantity, but would probably be local to the routine in which such a statement appeared. Such a function may also be employed on the l.h.s. of an assignment statement, For example:-

```
X(i-1,j+1) = [EXPR]
```

In the same way we can also define integer map and complex map routines.

If the map is placed at the end of a program a specification must be given before the routine can be referred to, for example

```
real map spec X(integer i,j)
```

We can now complete the list of formal parameter types

Formal parameter type	Corresponding actual parameter
addr	the name of any integer,real or complex variable (including an array element). The address of the variable is handed on as the parameter proper. It is equivalent to an integer parameter in the body of the routine. In fact an addr parameter replaced by x is equivalent to an integer parameter replaced by addr (x)
real map integer map complex map	the actual parameter is the name of a mapping routine of the specified type

10:THE USE OF MACHINE INSTRUCTIONS

STACK STRUCTURE

Machine instructions can be used in routines either to make an inner loop more efficient or to effect some operation which cannot easily be done otherwise. It is assumed that the reader is reasonably familiar with the logical structure of the machine, that is with the basic order code. It also essential to know how data is stored in the stack. We illustrate this with reference to the following routine.

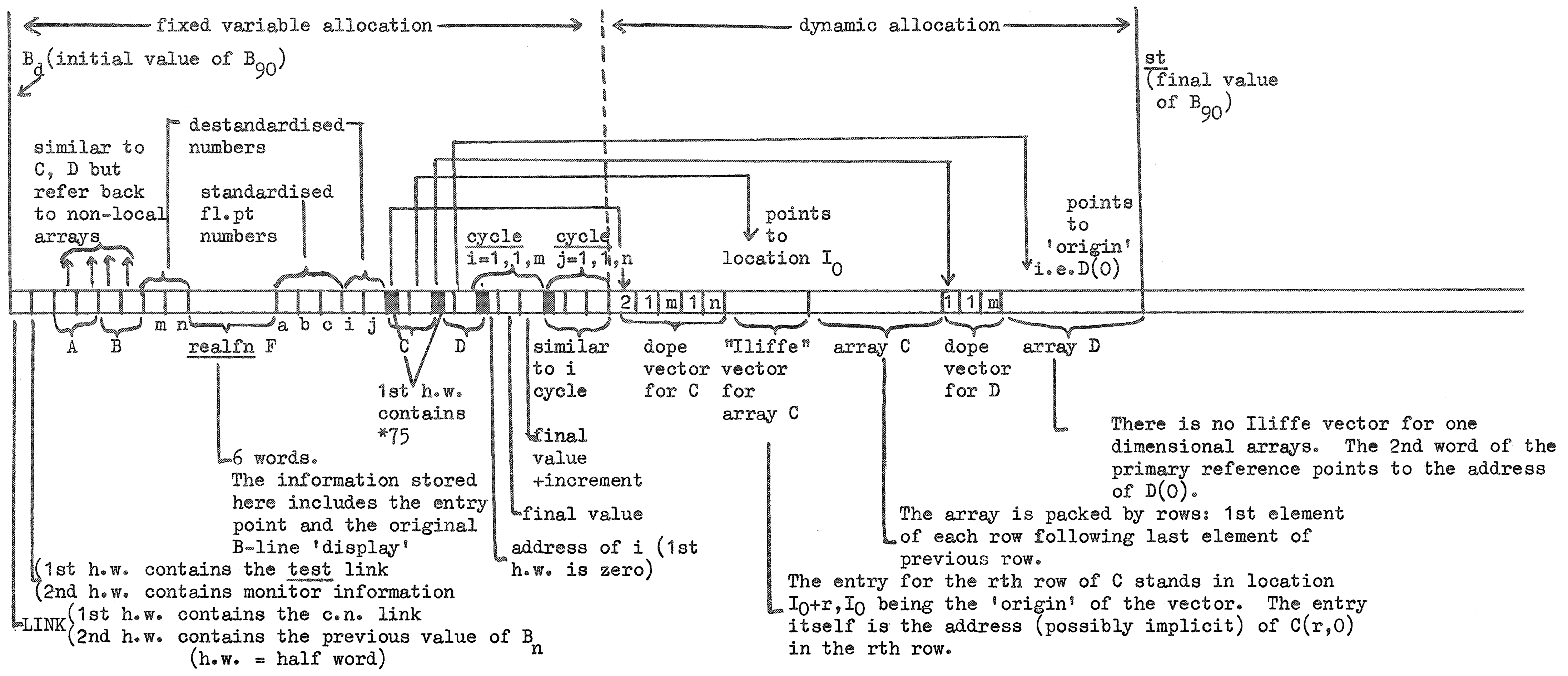
```
routine matrix fn (array name A,B integer m,n real fn F)
real a,b,c ;  integer i,j
array C(1:m,1:n),E(1:m)
real fn spec F (real x)
      .
      .
      .
cycle i = 1,1,m
cycle j = 1,1,n
      .
      .
      .
      .
repeat
repeat
      .
      .
      .
end
```

NOTES

1. The first word of the local stack section contains the control number for returning to the calling routine (the first half word) and the previous contents of Bd, the current level B-line (the second half word). The 1st half word of the second word contains the test link (which records the position within the label list of a test instruction), and the 2nd half word contains information (the number and type of the routine and the number of fixed variables) required by the run-time fault monitor routine. Here Bd refers to the B-line associated with the routine, and corresponds to the textual depth of the routine in the program in which it is embedded. If (say) this is 2 then Bd $\equiv$ B2. The relative locations of the fixed variables A, B, m, n etc., are assigned at compile time. Immediately on entry to the routine the current value of B90, which always points to the next available location in the stack, is recorded in Bd and the previous contents of Bd recorded in the stack (as already noted). B90 is then advanced to the end of the fixed storage allocation. When the declarations for C and D are 'obeyed' it is advanced again to the final value shown.

LAYOUT OF THE STACK FOR THE DECLARATIONS OF 'matrix fn'

Everything except real a, b, c and the (real) arrays themselves are destandardised and held in 'longword' units. In some cases the 1st (or m.s.) half word of the destandardised quantity is 'flagged' to identify its function for the purpose of 'stackprint' (see the shaded h.w's.) The blank sections indicate an indefinite number of words.



fixed variable allocation
dynamic allocation
B_d(initial value of B_{90})
st (final value of B_{90})
similar to C, D but refer back to non-local arrays
destandardised numbers
standardised fl.pt numbers
cycle i=1,1,m
cycle j=1,1,n
points to location I_0
points to 'origin' i.e.D(0)
2 1 m 1 n
1 1 m
A
B
m n
realfn F
a b c i j
C
D
similar to i cycle
dope vector for C
"Iliffe" vector for array C
array C
dope vector for D
array D
1st h.w. contains *75
final value +increment
final value
address of i (1st h.w. is zero)
6 words. The information stored here includes the entry point and the original B-line 'display'
(1st h.w. contains the test link
(2nd h.w. contains monitor information
LINK (1st h.w. contains the c.n. link
(2nd h.w. contains the previous value of B_n
(h.w. = half word)
There is no Iliffe vector for one dimensional arrays. The 2nd word of the primary reference points to the address of D(0).
The array is packed by rows: 1st element of each row following last element of previous row.
The entry for the rth row of C stands in location I_0+r,I_0 being the 'origin' of the vector. The entry itself is the address (possibly implicit) of C(r,0) in the rth row.

2. Destandardised quantities are formed by adding 0*8↑(-12) to the standardised form. This constant will be found in location *1000001. This octal form of the address can be used in machine instruction formats(see later). There are no <u>integer name</u> or <u>real name</u> parameters in this example; if present they would be represented (at the appropriate place among the fixed variables) by single words, namely their addresses, in a destandardised form. They are at present in distinguishable from <u>integer</u>'s. Similarly for <u>complex name</u> parameters. A <u>complex</u> quantity requires two consecutive words, representing the real and imaginary parts.

3. <u>ARRAYS</u>. The primary reference to an array consists of a pair of words. The second half of the 1st word points to the (1st word of the) 'dope vector', that of the second word to the 'Iliffe vector'. The dope vector contains the values of the bound-pair list together with the number of pairs, i.e. the dimensionality of the array. If there is more than one array associated with same bound-pair list they share the same dope vector. The Iliffe vector gives the origin of each row of the matrix (which is stored by rows). The purpose of this is to simplify computation involved in accessing an element of the array. Thus for example to add C(i+5,j-6) into the accumulator the instructions are:-

```
101, 96, d, i + ½        put i in β96
104, 96, 0, C + 1½       β96 = β96 + I0
101, 97, d, j + ½        put j in β97
104, 97, 96, 5½          β97 = β97 + entry for row (i+5)
320, 0, 97, -6           acc = acc + real (β97 - 6)
```

Similarly to add the element D(i+5) of the one-dimensional array D (which has no Iliffe vector) one may write

```
101, 97, d, i + ½        put i in B97
104, 97, 0, D + 1½       B97 = B97 + addr (D(0))
320, 0, 97, 5            acc = acc + real (B97 + 5)
```

In these instructions i, C, j, D refer to the addresses of these quantities (see later)

Arrays of k dimensions (>2) are stored in hierar hical fashion. The primary Iliffe vector points to a set of arrays of k - 1 dimensions stored end to end. Each such array consist of an Iliffe vector referring to a set of k - 2 dimensional arrays, and so on.

4. <u>THE PARAMETRIC FUNCTION F.</u> Six words of information are kept here. In addition to the control number for entering the routine it is necessary to keep a record of the display of the relevant B-lines when the routine is first substituted as an actual parameter. For further details see the Compiler.

5. <u>THE CYCLES.</u> As explained in the text the initial and final values and increment in a cycle are evaluated and checked for compatibility before the cycle is commenced. The increment and final values, together with the address of the controlled integer variable are recorded for use in the execution of the cycle. The diagram illustrates how they are stored.

STACK INSTRUCTIONS

The following autocode formats involving the stack pointer (B90) are available

```
st = st ± [EXPR']
st = [EXPR]
[NAME] = st
```

<u>st</u> represents the contents of B90. In the last instruction the [NAME] must be local to the routine containing the instruction, otherwise a fault is indicated.

MACHINE CODE FORMATS

Some 'machine code' formats are now described.

1. Where there is no symbolic address involved an instruction is written in the form

```
[FD],[N],[N],[ADDRESS PART]
```

(and terminated as usual by ; or newline). Here [FD] refers to the function digits, [N] to the Ba and Bm digits, and [ADDRESS PART] to the address part, which may take a number of forms. It may be written as a constant in the usual way (preceded possibly by a sign) bearing in mind that the binary point is located 3 places from the right hand end. Thus

```
0121, 80, 0, 2.5      is equivalent to
05064000  00000024
```

It may also consist of an octal number which consists of an * followed by up to 8 octal digits, including any significant zeros. Thus

```
        0101, 91, 0, *1001      is equivalent to
        04066600  10010000
```

in octal notation.
Finally it may consist of a label or a (possibly signed) constant plus a label. The label is replaced by the control number corresponding to it. We may refer to labelled constants (see next section) in this way. For example

```
        0334, 0, 0, 14:
        0101, 99, 0, ½ + 14:
              ,
              ,
              ,
        14: *03, *0000012
```

puts an unstandardised 10 in the accumulator, and a halfword 10 in β99
<u>NOTE</u> : The formal definition of [ADDRESS PART] is
[ADDRESS PART] = [±'][CONST]+[N]:,[N]:,[±'][CONST],[OW]

2. The format

```
        [±'][CONST]
```

is used to plant a standardised 48-bit floating point number in the current location of the program.
3. Pairs of 24-bit words may be planted in the object program by means of the forma

```
        [ADDRESS PART][,][ADDRESS PART]
```

Thus we may plant tables of integers or labels, for example:-

```
        3,4
        7:,8:
```

4. We now have an instruction format which uses a symbolic address.

```
        [FD],[N],-,[NAME][±CONST']
  where [±CONST']=[±][CONST], NIL
```

Here the [NAME] can refer to anything which is represented in the fixed storage sections of the stack. The resulting instruction is

```
        [FD],[N], d, p [± CONST']
```

where (Bd, p) is the 'address' of the name, Bd being the B-line pointing to the appropriate section of the stack, and p being the address relative to the origin of that section. Thus an instruction

```
0324, 0, -, a
```

appearing in the routine under discussion would be translated as

```
0324, 0, 2, 14
```

assuming Bd $\equiv$ B2, and that F occupies 6 words.
The effect would be to put a in the accumulator.

If the [NAME] refers to an unstandardised floating point integer then we may wish to select the integral half for use in a B-line. For example

```
0101, 80, -, m+½
```

is equivalent to

```
0101, 80, 2, 6.5
```

If a and m had been <u>real</u> and <u>integer</u> <u>name</u>'s then 2 instructions would be necessary in each case, thus

```
0101, 99, -, a + ½
0324, 0, 99, 0
```

and

```
0101, 99, -, m + ½
0101, 80, 99, ½
```

If a is <u>complex</u> then

```
0324, 0, -, a
```

would put the real part into the accumulator, and

```
0324, 0, -, a+1
```

would load the imaginary part.

In the case of arrays we can select by similar means the two primary reference words, and with their aid obtain access to the dope vector and/or the array itself.

EXAMPLE ON THE USE OF MACHINE ORDERS

The following example forms the sum of three routines A, B, C of similar dimensions. (It is in fact the permanent routine 'matrix add')

```
routine matrix add (array name A, B, C)
comment The routine forms A = B + C
real dump
        0101, 61, -, A + ½              ;comment dope vector of A
        0101, 62, -, B + ½              ;comment dope vector of B
        0101, 63, -, C + ½              ;comment dope vector of C
        0121, 65, 0, 4                  ;comment check dimensions
        0101, 64, 61, ½
        0172, 64, 0, 2
        0225, 127, 0, 1:
2:      0101, 64, 61, ½
        0152, 64, 62, ½
        0225, 127, 0, 1:
        0152, 64, 63, ½
        0225, 127, 0, 1:
        0124, 61, 0, 1
        0124, 62, 0, 1
        0124, 63, 0, 1
        0203, 127, 65, 2:
        0101, 65, -, A + ½              ;comment β65 = dope vector of A
        0324, 0, 65, 1                  ;comment β64 = no of elements in matrix
        0322, 0, 65, 2
        0320, 0, 0, *10000040
        0356, 0, -, dump
        0324, 0, 65, 3
        0322, 0, 65, 4
        0320, 0, 0, *10000040
        0362, 0, -, dump
        0330, 0, 0, *10000010
        0356, 0, -, dump
        0101, 64, -, dump + ½
        0101, 66, 65, 1½                ;comment set β61 = address of 1st element of A
        0104, 66, -, A + 1½
        0101, 61, 65, 3½
        0104, 61, 66, ½
        0101, 66, -, A + 1½             ;comment set β62 = address of 1st element of B
        0120, 66, 61, 0
        0101, 62, -, B + 1½
        0124, 62, 66, 0
        0101, 63, -, C + 1½             ;comment set β63 = address of 1st element of C
        0124, 63, 66, 0
        0122, 64, 0, 1                  ;comment perform addition
5:      0324, 64, 62, 0
        0320, 64, 63, 0
        0356, 64, 61, 0
        0203, 127, 64, 5:
        →4
1:      0121, 91, 0, 34
        fault monitor                   ;comment DIMENSION FAULT
4:      end
```

11. THE PERMANENT ROUTINES

In Section 5, we decribed the input and output routines. The permanent material also includes routines for the solution of linear equations, the solution of systems of ordinary differential equations and operations on matrices. Further routines may be added from time to time.

LINEAR EQUATIONS

routine spec eqn solve(arrayname A,b, realname det)

This routine solves the equations

```
A(1,1)x(1) + A(1,2)x(2) +....+ A(1,n)x(n) = b(1)
A(2,1)x(1) + A(2,2)x(2) +....+ A(2,n)x(n) = b(2)
    '            '                 '
    '            '                 '
    '            '                 '

A(n,1)x(1) + A(n,2)x(2) +....+ A(n,m)x(n) = b(n)
```

(i.e. Ax = b), where the coefficients A(i,j) are stored in the matrix A, and b(i) in the vector b. A is destroyed and the solution is placed in b. If during the elimination process, the equations are found to be linearly dependant, then 'det' is set to zero and the routine is left, with both A and b upset. Otherwise 'det' is set to the determinant of A. Consequently 'det' should be tested after each call of the routine.

MATRIX ROUTINES

The matrix routines operate on two dimensional arrays(i.e. matrices not vectors). The dimensions of the arrays are not required as parameters as the routines automatically find these from the declarations, and check them for compatibility. The programmer may insert similar tests in his own routines by means of the functions

integer fn spec dim (arrayname A)
integer fn spec bound (arrayname A, integer n)

The first gives the dimensionality of the array(1 for a vector, 2 for a matrix etc.), and the second the nth bound (upper or lower) of the array counting from left to right. For example, if A were declared by:-

array A(-5:+5, 1:p) where p = 10 then

```
     dim(A) would have the value 2
bound(A,1)   '    '    '    '  -5
bound(A,2)   '    '    '    '  +5
bound(A,3)   '    '    '    '   1
bound(A,4)   '    '    '    '  10
```

The routines

```
routine spec unit (arrayname A)
routine spec null (arrayname A)
```

set A to be a unit matrix (checking that it is square) and a null matrix respectively. The routines

```
routine spec matrix add (arrayname A,B,C)
routine spec matrix sub (arrayname A,B,C)
routine spec matrix copy (arrayname A,B)
```

set A to B+C, B-C and B respectively. Although the parameters are of type arrayname, the operation of the routines is such that the same array can be substituted for more than one of the parameters. For example:-

```
matrix add(A,A,A)
```

doubles A. The same is not true of the following routines:-

```
routine spec matrix mult(arrayname A,B,C)
routine spec matrix mult'(arrayname A,B,C)
routine spec matrix trans (arrayname A,B)
```

These set A to B*C, B*C' and B' respectively where the ' denotes transposition. If it is required to, say, set a matrix to the product of itself and another then the call

```
matrix mult(A,A,B)
```

will fail. It is necessary to declare another array, 'dummy' say, and then use 'matrix copy' and'matrix mult':-

```
matrix copy (dummy,A)
matrix mult(A,dummy,B)
```

Alternatively, a routine with parameters of type array may be defined which calls the permanent routines :-

```
routine MATRIX MULT(arrayname A, array B,C)
matrix mult(A,B,C)
end
```

In this case a call of the form

```
MATRIX MULT(A,A,B)   or even   MATRIX MULT(A,A,A)
```

is possible.

The routines

    routine spec matrix div(arrayname A,B, realname det)
    routine spec invert(arrayname A,B, realname det)

set A to inv(B) A and inv(B) respectively. In the process B is destroyed and the value of its determinant placed in det. Should the matrix be found to be singular, 'det' is set to zero. Consequently 'det' should be tested after every call for these routines. If B is required at the end of the routine, then the techniques described above should be used. The function

    real fn det (arrayname B)

sets 'det' to the determinant of B and destroys B.

SOLUTION OF DIFFERENTIAL EQUATIONS

There are two routines available for advancing the solution of a system of first order ordinary differential equations

    dy(i)/dx = f(i)(x,y(1),y(2),....,y(n))   i=1,2,...,n

from x to x+h, using the Kutta-Merson fourth-order integration method[2]. The system is defined by means of an auxiliary routine, which must be supplied by the user, of the form :-

    routine spec aux(arrayname f, real x)

which must evaluate the derivatives f(i) in terms of y(1),y(2),...y(n) and x and then place them in f(1),f(2), ..., f(n).

The first routine

    routine spec int step(arrayname y,real x,h, integer n   c
    realname e, routine aux)

advances the solution by a single step of length h of the Kutta-Merson process.

[2] Reference L. FOX (Ed.) Numerical Solution of Ordinary and Partial Differential Equations. Pergamon 1962, P.24.

The parameters are :-

y	name of a (real)array. On input y(1),y(2)...y(n) should contain the solution at x. On output they will contain the solution at x+h.
x	the initial value of the independent variable
h	the increment of the independent variable.
n	the number of equations in the system.
e	the name of a real variable, which on output will contain an estimate of the maximum truncation error over the step.
aux	the name of the routine which evaluates the derivatives at a general point (see above).

The second routine

routine spec kutta merson(arrayname y,real x0,x1, realname e c
integer n,k, routine aux)

advances the solution, by means of a series of calls for 'intstep', from x0 to x1, keeping, if possible the estimate of the maximum truncation error less than e. An initial step length of (x1-x0)/2↑m where 2↑(m+1) >k ≥ 2↑m, is taken. If over a step the local truncation error (given by 'int step') is greater than e, then the step length is halved; if the error is less than .01e then the step length is doubled.
If three successive reductions in step length give no improvement in the estimated truncation error, then e is replaced by twice the smallest error achieved, and the integration process continued. The parameters are :-

y	the name of a (real)array. On input y(1),y(2),....y(n) should contain the solution at x0. On output they will contain the solution at x1.
x0	the inital value of the independent variable.
x1	the final value of the independent variable.
e	the name of a real variable. On input this should contain the accuracy criterion. On output it will be unchanged if this accuracy has been achieved ; if not, it will be replaced by a more realistic value(see above).
n	the number of equations in the system
k	an estimate of the number of steps required to cover the range(see above)
aux	the name of the routine which evaluates the derivatives at a general point(see above).

APPENDIX 1. PHRASE STRUCTURE NOTATION

In describing Atlas Autocode we use square brackets round an entity to denote that it represents a class of entities and may be replaced by any member of the class. We call an entity in square brackets a PHRASE. For example we could define a decimal digit by

```
PHRASE[DIGIT] = 0,1,2,3,4,5,6,7,8,9
```

where the commas are interpreted as meaning 'or'. Thus there are ten different things which can be called [DIGIT], and when we refer to [DIGIT] elsewhere we mean that any of the ten will be legitimate.

We can then build up from this basis and describe, for example, a signed digit as

```
PHRASE[SIGNED DIGIT] = +[DIGIT], -[DIGIT]
```

There are also places where a phrase may or may not appear and to signify this a special phrase 'NIL' may be written as the last alternative in a phrase definition. For example the switch limits in a switch declaration can be preceeded by a + or - sign if desired. (Absence of a sign corresponds to +.) The relevant definition is

```
          [NAME LIST]([±'][N]:[±'][N])
   where  PHRASE[±'] = +,-,NIL

   Thus   -4:+4
          -4: 4
           1: 3
```

are examples of switch limits.

Alternatively we can use the special ? qualifier as follows.

```
PHRASE[±] = +,-
PHRASE[±?] = [±],NIL
```

The last is implicit and we can use [±?] (e.g., in place of [±']) without explicity giving the latter definition.

In the interest of efficiency however, it is preferable to keep the depth of analysis as small as possible and for this reason we use the former scheme.

The phrase structure notation can be used recursively, i.e., phrase definitions may,directly or indirectly, use themselves. For example we may define a 'list of names separated by commas' by

```
PHRASE[NAME LIST] = [NAME][REST OF NAME LIST]
PHRASE[REST OF NAME LIST] = [,][NAME][REST OF NAME LIST],NIL
```

[Since a ',' is used to separate the alternatives of a phrase definition it cannot stand for itself like the other basic symbols. Instead we must write [,]. Similarly [EOL] and [SP] are used to denote 'end of line' and 'space' in the source language.]

The qualifier * also indicates recursiveness and a [NAME LIST] could be defined as

```
PHRASE[NAME LIST] = [NAME][,NAME*?]
PHRASE[,NAME] = [,][NAME]
```

the definitions

```
PHRASE[,NAME*?] = [,NAME*],NIL
PHRASE[,NAME*]  = [,NAME][,NAME*],[,NAME]
```

being implicit. Again, however, for reasons of efficiency we use the former definition.

Given the phrases of the language it is then possible to describe all the formats allowed in a program. For example, if we introduce the phrase[TYPE] as

PHRASE[TYPE] = <u>integer</u>, <u>real</u>, <u>complex</u>

we can define the format for the scalar declarations as

```
FORMAT[SS] = [TYPE][NAME LIST][S]
```

The [SS] indicates that it is a source statement, which means it appears on its own in an Autocode program.

In Atlas Autocode there is a further type or CLASS of format, the unconditional instructions [UI], which have the special property that they may be preceded by the conditional operators <u>if</u> [COND] <u>then</u> and <u>unless</u> [COND] <u>then</u>.

A list of the phrases and formats of Atlas Autocode follows. Note that some phrases ([S],[CONST],[NAME] and [TEXT]) are not formally defined. These are defined by special built-in routines which we will not consider here, but those interested may refer to the references given below.

Finally we should point out that some of the definitions are not completely rigid. For example, the arithmetic assignment statement is defined as

FORMAT[UI] = [NAME][APP] = [EXPR]

In the routine which deals with this format, tests are made to ensure that the [NAME][APP] does in fact describe a variable, and is not, for example, a function.

References

[3]. Brooker,R.A., Morris,D. and Rohl,J.S. ''Trees and Routines'', Computer Journal, Vol. 5. No. 1.

[4]. Brooker,R.A., MacCallum,I.R., Morris,D. and Rohl,J.S. ''The Compiler Compiler'' 3rd Annual Review of Automatic Programming (ed. Goodman), Pergamon Press.

```
PHRASE [EXPR]               = [±'][EXPR']
PHRASE [±]                  = +, -
PHRASE [±']                 = +, -, NIL
PHRASE [EXPR']              = [OPERAND][OP][EXPR'],[OPERAND]
PHRASE [OPERAND]            = [NAME][APP],[CONST],([EXPR]),|[EXPR]|, i, BUT NOT if
PHRASE [APP]                = ([EXPR-LIST]), NIL
PHRASE [EXPR-LIST]          = [EXPR][REST OF EXPR-LIST]
PHRASE [REST OF EXPR-LIST]=[,][EXPR][REST OF EXPR-LIST], NIL
PHRASE [OP]                 = +, -, *, /, ↑, ., NIL
PHRASE (CR)[acc]            = dsa, acc, ca, sac
PHRASE [AO]                 = acc +, acc-, acc*, acc/, acc↑, addr, -, NIL
PHRASE [QUERY']             = ?, NIL
PHRASE (CR)[program]        = programme, program
PHRASE [,']                 = [,], NIL
PHRASE [iu]                 = if, unless
PHRASE [acc']               = acc, ca, sac

PHRASE [TYPE]               = integer, real, complex
PHRASE [TYPE']              = integer, real, complex, NIL
PHRASE [NAME LIST]          = [NAME][REST OF NAME LIST]
PHRASE [REST OF NAME LIST]=[,][NAME][REST OF NAME LIST], NIL
PHRASE [ARRAY LIST]         = [NAME LIST]([BOUND PAIR LIST])[REST OF ARRAY LIST]
PHRASE [REST OF ARRAY LIST]=[,][NAME LIST]([BOUND PAIR LIST])[REST OF ARRAY LIST], NIL
PHRASE [BOUND PAIR LIST]  = [BOUND PAIR][REST OF BOUND PAIR LIST]
PHRASE [REST OF BOUND PAIR LIST]=[,][BOUND PAIR][REST OF BOUND PAIR LIST], NIL
PHRASE [BOUND PAIR]         = [EXPR] : [EXPR]
PHRASE [ARRAY FN LIST]      = [NAME]([EXPR-LIST])[REST OF ARRAY FN LIST]
PHRASE [REST OF ARRAY FN LIST]=[,][NAME]([EXPR-LIST])[REST OF ARRAY FN LIST], NIL
PHRASE [SWITCH LIST]        = [NAME LIST]([±'][N]:[±'][N])[REST OF SWITCH LIST]
PHRASE [REST OF SWITCH LIST]=[,][NAME LIST]([±'][N]:[±'][N])[REST OF SWITCH LIST], NIL
PHRASE [RT]                 = integer map, real map, complex map, integer fn,
                              real fn, complex fn, routine
PHRASE [FPP]                = ([FP-LIST]), NIL
PHRASE [FP-LIST]            = [FP][REST OF FP-LIST]
PHRASE [FP]                 = [FP-DELIMITER][NAME]
PHRASE [REST OF FP-LIST]    = [FP][REST OF FP-LIST], NIL
PHRASE [FP-DELIMITER]       = [,'][RT],[,'] integer array name, [,'] integer array,
                              [,'] integer name, [,'] integer,
                              [,'][real'] array name, [,'][real'] array,
                              [,'] real name, [,'] real,
                              [,'] complex array name, [,'] complex array,
                              [,'] complex name, [,'] complex, [,'] addr, [,]
PHRASE [COND]               = [SC] and [AND-C],[SC] or [OR-C],[SC]
PHRASE [AND-C]              = [SC] and [AND-C],[SC]
PHRASE [OR-C]               = [SC] or [OR-C],[SC]
PHRASE [SC]                 = [EXPR][COMP][EXPR][COMP][EXPR],
                              [EXPR][COMP][EXPR],([COND])
PHRASE [COMP]               = =, ≠, >, ≤, <, ≥
PHRASE [N-LIST]              = [N][REST OF N-LIST]
PHRASE [REST OF N-LIST]      = [,][N][REST OF N-LIST], NIL
PHRASE [ALPHA']              = α, NIL
PHRASE [± CONST']            = [±][CONST], NIL
PHRASE [real']               = real, NIL
PHRASE [ADDRESS PART]        = [±'][CONST] + [N]:,[N]:,[±'][CONST],[LOW]
PHRASE [check]               = routine trace, jump trace, queries, array bound check
PHRASE [FAULT LIST]          = [N-LIST] -> [N][REST OF FAULT LIST]
PHRASE [REST OF FAULT LIST]=[,][N-LIST] -> [N][REST OF FAULT LIST], NIL
PHRASE [SIMPLE LABEL]        = [N]:, BUT NOT [N]:[,]
PHRASE [RT']                 = [RT],NIL

FORMAT CLASS[UI]
FORMAT [UI]                  = [NAME][APP] = [EXPR][QUERY']
FORMAT [UI]                  = [NAME][APP]
FORMAT [UI]                  = ->[N]
FORMAT [UI]                  = ->[NAME]([EXPR])
FORMAT [UI]                  = caption [TEXT]
FORMAT [UI]                  = result = [EXPR]
FORMAT [UI]                  = return
FORMAT [UI]                  = stop
FORMAT [UI]                  = test [N-LIST]
FORMAT [UI]                  = [check]on
FORMAT [UI]                  = [check]off
FORMAT [SS]        = [UI][S]
FORMAT [SS]        = [UI][iu][COND][S]
FORMAT [SS]        = [iu][COND] then [UI][S]
FORMAT [SS]        = cycle [NAME][APP] = [EXPR][,][EXPR][,][EXPR][S]
FORMAT [SS]        = repeat [S]
FORMAT [SS]        = [SIMPLE LABEL]
FORMAT [SS]        = [N] case [COND]:
FORMAT [SS]        = [NAME]([±'][N]):
FORMAT [SS]        = [TYPE][NAME LIST][S]
FORMAT [SS]        = [TYPE'] array [ARRAY LIST][S]
FORMAT [SS]        = [TYPE'] array fn [ARRAY FN LIST][S]
FORMAT [SS]        = [RT'] spec [NAME][FPP][S]
FORMAT [SS]        = [RT][NAME][FPP][S]
FORMAT [SS]        = begin [S]
FORMAT [SS]        = comment [TEXT][S]
FORMAT [SS]        = end [S]
FORMAT [SS]        = end of [program][S]
FORMAT [SS]        = ignore queries [S]
FORMAT [SS]        = production run [S]
FORMAT [SS]        = page [TEXT][S]
FORMAT [SS]        = switch [SWITCH LIST][S]
FORMAT [SS]        = compile [check][S]
FORMAT [SS]        = stop [check][S]
FORMAT [SS]        = own [TYPE][NAME LIST][S]
FORMAT [SS]        = own [TYPE'] array [ARRAY LIST][S]
FORMAT [SS]        = fault [FAULT LIST][S]
FORMAT [SS]        = [FD][,][N][,][N][,][ADDRESS PART][S]
FORMAT [SS]        = [FD][,][N][,]-[,][ALPHA'][NAME][±CONST'][S]
FORMAT [SS]        = [±'][CONST][S]
FORMAT [SS]        = [ADDRESS PART][,][ADDRESS PART][S]
FORMAT [SS]        = [NAME] = st [S]
FORMAT [SS]        = st = [EXPR][S]
FORMAT [SS]        = st = st [±][EXPR'][S]
FORMAT [SS]        = prepare to read perm [S]
FORMAT [SS]        = define compiler
FORMAT [SS]        = define master compiler
FORMAT [SS]        = define special compiler
FORMAT [SS]        = advance β8 [S]
FORMAT [SS]        = pl [NAME]([N][,][N][,][N])[S]
FORMAT [SS]        = real exponentiation [S]
FORMAT [SS]        = |[TEXT][S]
FORMAT [SS]        = now compile from input [N][S]
FORMAT [SS]        = upper case delimiters [S]
FORMAT [SS]        = normal delimiters [S]
FORMAT [SS]        = [TEXT][S]
```

APPENDIX 2 INDEX OF STANDARD FUNCTIONS AND PERMANENT ROUTINES

All the functions and routines listed below are declared at level 0 and hence are permanently available unless the names are redeclared locally by the user. The number in the right hand margin indicates the page on which they are described more fully.

STANDARD MATHEMATICAL FUNCTIONS

integer fn spec intpt(real x)	2.5
integer fn spec int(real x)	2.5
integer fn spec parity(integer n)	2.5
real fn spec sin(real x)	2.5
real fn spec cos(real x)	2.5
real fn spec tan(real x)	2.5
real fn spec log(real x)	2.5
real fn spec exp(real x)	2.5
real fn spec sqrt(real x)	2.5
real fn spec arctan(real x,y)	2.5
real fn spec radius(real x,y)	2.5
real fn spec arcsin(real x)	2.5
real fn spec arccos(real x)	2.5
real fn spec fracpt(real x)	2.5
real fn spec mod(real x)	2.5
real fn spec re(complex z)	8.1
real fn spec im(complex z)	8.1
real fn spec mag(complex z)	8.1
real fn spec arg(complex z)	8.1
complex fn spec csin(complex z)	8.1
complex fn spec ccos(complex z)	8.1
complex fn spec ctan(complex z)	8.1
complex fn spec clog(complex z)	8.1
complex fn spec cexp(complex z)	8.1
complex fn spec csqrt(complex z)	8.1
complex fn spec conj(complex z)	8.1

STORAGE FUNCTIONS

integer fn spec addr(addr s)	9.1
integer fn spec integer(integer s)	9.1
real fn spec real(integer s)	9.1
complex fn spec complex(integer s)	9.1

MISCELLANEOUS FUNCTIONS

integer fn spec control no([ROUTINE NAME])

This gives the address of the first word of the routine in question.

integer fn spec dim(arrayname A)	11.1
integer fn spec bound(arrayname A,integer i)	11.1

NOTE : The above classes of function cannot be substituted as an actual

INPUT ROUTINES

routine spec select input(integer n)	5.1
routine spec read(addr s)	5.1
routine spec read array(arrayname A)	5.2
routine spec read symbol(integername i)	5.2
integer fn spec next symbol	5.3
routine spec skip symbol	5.3
routine spec read sequence(addr s,integer p,integername n)	5.3
routine spec read binary(integername i)	5.6

OUTPUT ROUTINES

routine spec select output(integer n)	5.1
routine spec print(real x, integer m,n)	5.4
routine spec print fl(real x, integer n)	5.4
routine spec space	5.5
routine spec spaces(integer n)	5.5
routine spec newline	5.5
routine spec newlines(integer n)	5.5
routine spec tab	5.5
routine spec newpage	5.5
routine spec runout(integer n)	5.5
routine spec print array(arrayname A, integer m,n)	5.5
routine spec print array fl(arrayname A, integer m)	5.5
routine spec print symbol(integer i)	5.5
routine spec print complex(complex z,integer m,n)	8.5
routine spec print complex fl(complex z,integer m)	8.5
routine spec punch binary(integer n)	5.6

MATRIX ROUTINES

routine spec null(arrayname A)	A=0	11.2
routine spec unit(arrayname A)	A=I	11.2
routine spec matrix add(arrayname A,B,C)	A=B+C	11.2
routine spec matrix sub(arrayname A,B,C)	A=B-C	11.2
routine spec matrix copy(arrayname A,B)	A=B	11.2
routine spec matrix mult(arrayname A,B,C)	A=B*C	11.2
routine spec matrix mult'(arrayname A,B,C)	A=B*C'	11.2
routine spec matrix trans(arrayname A,B)	A=B'	11.2
routine spec matrix div(arraynameA,B,realname det)	A=inv(B)*A	11.3
routine spec invert(arrayname A,B,realname det)	A=inv(B)	11.3
real fn spec det(arrayname B)	result=|B|	11.3

MISCELLANEOUS ROUTINES

routine spec eqn solve(arrayname A,b realname det)	11.1
routine spec kutta merson(arrayname y,real x0,x1,realname e integer n,k routine aux)	11.4
routine spec intstep (arrayname y, real x,h, integer n real name e, routine aux)	11.3

APPENDIX 3 INDEX OF DELIMITERS

APPENDIX 4 LIST OF MONITORED FAULTS

Fault monitoring is very dependent on the form of the compiler used. We describe below the monitoring now given (1/3/65). It will probably change with time but all changes will be designed to give the maximum information

COMPILING TIME FAULTS

1. Faults due to [NAME]'s not having been declared.

NAME[NAME]NOT SET
SWITCH[NAME]NOT SET

2. Faults, found in arithmetical instructions, which give special indications but which are most often caused by [NAME]'s not being declared at the current level. These special indications arise when the [NAME]'s appear in the level above.

NAME[NAME]CANNOT APPEAR ON L.H.S.
SWITCH[NAME] IN EXPR
ROUTINE[NAME] IN EXPR
CALL FOR ADDR OF NON-VARIABLE
CALL FOR CONTROL NO OF NON-ROUTINE[NAME]

3. Arithmetic faults.

COMPLEX[NAME] IN EXPR
i IN EXPR
REAL[NAME] IN EXPR
REAL CONST IN INTEGER EXPR
CALL FOR DIM OF NON-ARRAY NAME
CALL FOR BOUNDS OF NON-ARRAY NAME
NAME[NAME]HAS WRONG NUMBER OF PARAMETERS

(This may be due either to the wrong number of parameters appearing or to the omission of a multiplication sign before a left bracket)

4. Faults found at the end of each block or routine.

LABEL [N] NOT SET	There is a reference to label[N] or
CASE [N] NOT SET	a case [N] which has not been set
NO LABELS SET	
TOO FEW REPEATS	<u>cycle</u>'s do not match <u>repeat</u>'s

5. Other faults.

AP FAULT	An actual parameter fault:the call sequence is not consistent with the routine spec
FP FAULT	A formal parameter fault: the routin heading is not consistent with the routine spec.
LABEL[N]SET TWICE	Two or more instructions
CASE[N] SET TWICE	have been given the same
SWITCH[NAME]SET TWICE	label

NAME[NAME]SET TWICE	The name has been used for more than one purpose at a given textual level
SWITCH[NAME]OUT OF RANGE	A label[NAME]([N]) appears where [N] lies outside the declared range of the switch[NAME]
SWITCH[NAME]OUT OF RANGE	A label[NAME]([N]) appears where [N] lies outside the declared range of the switch[NAME]
TOO MANY REPEATS	Too many repeat's in a block or routine
[NAME] =ST NOT VALID	The [NAME] is non-local
RESULT OUT OF CONTEXT	A result = [EXPR] statement appears in a routine other than a function or map routine
NON-INTEGER CYCLE VARIABLE	The controlled variable is not an integer

RUN TIME FAULTS

1. The following faults are monitored at run time. Normally they cause the program to be terminated but it may be restarted by a fault instruction. The relevant fault numbers appear in the tables below. For those numbers not appearing, reference should be made to the ABL Manual.

DIV OVERFLOW	Division by 0 or a non-standard number	fault 1
EXP OVERFLOW	Exponent overflow	fault 2
SQRT -VE	Sqrt of a negative argument	fault 5
LOG -VE	Log of a negative argument	fault 6
INV TRIG FN OUT OF RANGE	In inverse trig function e.g., arcsin when the argument is not within range(-1,+1)	fault 8
INPUT ENDED	Insufficient data so that a read instruction effectively reads over the end of the data tape	fault 9
SPURIOUS CHARACTER IN DATA	Spurious character (i.e. NOT a decimal digit, point, sign, or α) appears in data.	fault 14
MORE THAN 3 SYMBOLS IN POSITION	A compound character formed from more than 3 superimposed characters has been encounted in textual data.	fault 15
REAL QUANTITY INSTEAD OF INTEGER IN DATA		fault 16
FAULT IN COMPLEX DATA	the complex data is not punched according to the conventions of P8.5	fault 17

2. Faults which indicate programming errors but which always cause the program to terminate

INPUT NOT DEFINED OUTPUT NOT DEFINED	An input or output channel has been selecte which is not mentioned in the Job Descripti
ALL TESTS FAIL	All conditions in a <u>test</u> instruction fail
SWITCH VARIABLE NOT SET	Refers to a multiway switch instruction ->[NAME]([EXPR]) where the value of [EXPR] is out of range or corresponds to a missing label.
ARRAY DIMENSIONS NOT +VE	Refers to a bound pair (L:U) where $U-L+1\leq 0$.
NON-INTEGRAL CYCLE	Refers to the check carried out immediately prior to the execution of a cycle
CALLS FOR NON-EXISTENT ROUTINE	Occurs when the routine and a specification are not at the same level, or the former is missing.
DIMENSION FAULT	Occurs when a matrix routine is called using parameters which are not matrices or are incompatible.
ARRAY SUBSCRIPT OUT OF BOUNDS	Occurs when <u>compile array bound check</u> is used and the subscripts are not within the right bounds.

3. Faults which can arise because of accessing array elements outside the bounds given in the declaration e.g. A(10,3) when A had been declared A(1:3,1:10). If the immediate cause is not obvious the <u>compile array bound check</u> should be used. There are a number of indications such as

SV OPERAND

ILLEGAL BLOCK

APPENDIX 5 NUMERICAL EQUIVALENTS OF BASIC AND COMPOUND SYMBOLS

The numerical equivalents for use in conjunction with the 'read symbol' and 'print symbol' routines are given in the table overleaf. The table gives the numerical equivalents of the basic symbols i.e. symbols comprising of a single (upper or lower case) character.

Up to three basic symbols may be superimposed (by means of the backspace facility) to form a compound symbol. For example:-

$\underline{\neq}$ is formed from = _ /

The numerical equivalent of a compound symbol is

$$a*2\uparrow 14 + b*2\uparrow 7 + c$$

where a,b,c are the numerical equivalents of the individual symbols, ordered so that a>b>c. Thus the numerical equivalent is independent of the order of punching the individual characters.

If only two symbols are used, the formula is

$$b*2\uparrow 7 + c, \; b > c$$

Thus $\underline{\neq}$ is equivalent to $86*2\uparrow 14 + 28*2\uparrow 7 + 15$
and $\underline{>}$ is equivalent to $86*2\uparrow 7 + 27$

TABLE OF NUMERICAL EQUIVALENTS

0		32	'	64		96	
1		33	A	65	space	97	a
2		34	B	66		98	b
3		35	C	67		99	c
4	newline	36	D	68		100	d
5		37	E	69		101	e
6		38	F	70		102	f
7		39	G	71		103	g
8	(	40	H	72		104	h
9	)	41	I	73		105	i
10	,	42	J	74		106	j
11	π	43	K	75		107	k
12	?	44	L	76		108	l
13	&	45	M	77		109	m
14	*	46	N	78		110	n
15	/	47	O	79	:	111	o
16	0	48	P	80		112	p
17	1	49	Q	81	[	113	q
18	2	50	R	82	]	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)	118	v
23	7	55	W	87	|	119	w
24	8	56	X	88		120	x
25	9	57	Y	89		121	y
26	<	58	Z	90	α	122	z
27	>	59		91	β	123	
28	=	60		92	½	124	
29	+	61		93		125	
30	-	62		94		126	
31	.	63		95		127	