ATLAS AUTOCODE

Programming Manual

PREFACE

This document describes an Autocode for the Manchester University Atlas Computer, which has been designed and implemented by the staff of the Computing Machine Laboratory. Thanks are due to Mr. C.V.D. Forrington for drafting and editing this document and to Mrs. B. Duncanson and Miss Carol Clegg for preparing the copy.

R.A. BROOKER
J.S. ROHL
February, 1963.

CONTENTS

8. COMPLEX ARITHMETIC

Declarations; Standard functions; Arithmetic expressions; Arithmetic instructions; Data; Conditions; Routines and functions.

9. HALF-WORD OPERATIONS AND LIST PROCESSING FACILITIES

Operations on half-words; Conversion to integer or real; Lists; List processing facilities; Nests.

10. THE USE OF MACHINE INSTRUCTIONS

Stack structure; Stack instructions; Machine code formats; Example; Use of B-lines.

APPENDICES

1. PHRASE STRUCTURE NOTATION
2. LIST OF STANDARD FUNCTIONS AND PERMANENT ROUTINES
3. NUMERICAL EQUIVALENTS OF BASIC AND COMPOUND SYMBOLS
4. LIST OF MONITORED FAULTS

I 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 XI,YI; 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. Each set is assumed to contain less than I000 pairs. For each set the quantities

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

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

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

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

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, x, y, xx, xy, yy; integer n, i
      array  X, Y (I:1000)
      n = I
   2: read (X(n))
      if X(n) = 999 999 then -> I
   3: read (Y(n))
      n = n + I; -> 2
   I: n = n - I; x = 0; y = 0; xx = 0; yy = 0; xy = 0
      cycle i = I, I, n
      x = x + X(i); y = y + Y(i)
      xx = xx + X(i)^2; yy = yy + Y(i)^2
      xy = xy + X(i)Y(i)
      repeat
      a = (n*xy - x*y)/(n*xx - x*x)
      b = (y - a*x)/n
      c = yy - 2(a*xy + b*y) + xx*a^2 + 2a*b*x + n*b^2
      newline
      print (a, 2, 3); space; print (b, 2, 3); space; print (c, 2, 3)
      n = I
      read (X(I))
      unless X(I) = 999 999 then -> 3
      stop
      end of program
```

Users of Mercury Autocode will note some new features, namely

1. Explicit declaration of all quantities.
2. The nesting of brackets in arithmetic formulae and the obligatory use of brackets as a subscript notation.
3. Explicit use of a multiplication sign where this is necessary to avoid ambiguity.
4. The use of multi-letter identifiers.
5. The underlined delimiter words, e.g. cycle

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 the earlier sections it is sufficient to note that the Autocode statements between begin and end constitute a block. We note however, that when a block

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 I.

2 THE BASIC LANGUAGE

SYMBOLS OF THE LANGUAGE

Programs are presented to the computer as a length of perforated paper tape which is scanned by a photoelectric tape reader, the input unit of the machine. The program tape is prepared on a Flexowriter keyboard machine, the keys of which are engraved with the following symbols:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

α β π

0123456789

= > < ! * : , . ' & $\uparrow$ / + - _ $\frac{1}{2}$ ( ) [ ] ?

A back-spacing facility allows underlining and also the formation of compound characters

e.g. cycle $\neq$ > < -> ; θ

We also make use of a vertical arrow $\uparrow$ which by convention is punched as $\overset{*}{|}$ that is an asterisk superimposed on a vertical bar.

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 letters, possibly followed by one or more decimal digits, and possibly terminated by one or more primes('),e.g.,

x i Alpha aIO TEMPI 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].

DELIMITERS

These are a preassigned set of symbols and underlined words,e.g.,

+ - * / (,) > $\geq$ -> ; 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. An Autocode program consists entirely of names and constants separated by delimiters.

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 [VAR].

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

FUNCTIONAL DEPENDENCE

Functional dependence is indicated by writing the name of the function followed by the list of arguments in parentheses.

e.g. sin(2πx/a) arctan(x,y) TEMP(i) a(IO, IO)

Each argument can be an 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.

DECLARATION OF VARIABLES

The names of variables used in a block are declared at the head of the block

e.g. integer i, j, k, l, m, n, o, p, q, r, s, t
real a, b, c, d, e, f, g, h, u, v, w, x, y, z

which will be familiar to Mercury Autocode users.

Other examples are

integer I, max, min
real t, Temp, VOL I, 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.

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

array a,b(0 : 99), c(I0 : I9)

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 I0 to I9.

To refer to a particular element of an array one might write

a(50) b(j) b(2n + 2j - I) c(I0 + 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

e.g. array A(I:20, I:20), B(0:9, 0:49).

This defines and allocates storage for a 20 X 20 array A and a I0 X 50 array B. To refer to a particular element,one writes, for example

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

should an array of integer elements be required, the declaration is qualified by integer

e.g. integer array TYPE (I:50).

Storage allocated by 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 sectio on block structure and dynamic storage allocation (see later).

STANDARD FUNCTIONS

The following 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) sq rt(x)	
arctan(x,y)	(= $\tan^{-1}(y/x)$)
radius(x,y)	(= $(x^2 + y^2)^{\frac{1}{2}}$)
fracpt(x)	(= fractional part of x)
intpt(x)	(= integral part of x)
int(x)	(= nearest integer to x. i.e. intpt(x+.5))
mod(x)	(= |x|)
parity(n)	(= $(-I)^n$)

The arguments in the above functions may be general expressions, except that the argument of the last must be of type integer (see later).

A complete list of standard functions is given in Appendix 2.

ARITHMETICAL EXPRESSIONS

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

[±?] [OPERAND][OPERATOR][OPERAND][OPERATOR] .... [OPERAND]

An [OPERAND] is a [VAR], [CONST], ([EXPR]), |[EXPR]|, or [FUNCTION]**, and an [OPERATOR] is one of + - * / ↑ (The asterisk denoting multiplication)

Examples of expressions are

A(i-I,j) + A(i+I,j) + A(i,j-I) + A(i,j+I) -4A(i,j)

Z + log(I + cos(2π(x/a + y/b + z/c)))

LENGTH * BREADTH * HEIGHT

sq rt $(x(i)^2 + y(i)^2 + z(i)^2)$

a * b/c * d/e

(x + y + z)/(a + b + c)

2.5 xI b * (c + d)e

e = |x-y| + .0000I

(I + x) ↑ (a-I)(I-x) ↑ b

NOTES

I. 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. An exponent is denoted by ↑ [OPERAND] and exponentiation takes precedence over the other operations. Thus the last example means ((I + x) to the (a - I))((I - x) to the b). A special symbol, superscript^2, is available to represent squares as in the fourth example, and is equivalent to the pair of symbols ↑ 2.

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

4. Constants are written in a straight forward notation, e.g.,

2.538 I .25 I7.28α-I Iα7

The last two examples mean respectively

I.728 and I0 000 000

Fractional exponents are not allowed in floating point constants. The special symbol ½ may be used in constants and is equivalent to the pair of symbols .5

** Or, more strictly, (see Appendix I)

[EXPR] = [±?] [EXPR']

[EXPR'] = [OPERAND] [OPERATOR] [EXPR'], [OPERAND]

5. An explicit multiplication sign is not required when the operands are uniquely separable. In the seventh example it is the quantity 2.5 * xI * b * (c+d) * e that is computed. Note that an explicit multiplication sign is necessary to denote b * (c+d) as b(c+d) would be interpreted as referring to a function b of argument c + d.

6. A full stop . may be used instead of an asterisk to denote multiplication whenever there is no posibility of confusion with a decimal point. Thus in the above examples it is possible to write

```
LENGTH. BREADTH. HEIGHT
a.b/c.d/e
2.5 xIb. (c+d)e
```

The only time when an asterisk must be used is when multiplying two constants together.

Thus 2*5 means (2) x (5)
2.5*3.8 means (2.5) x (3.8)
2.5.3.8 is meaningless

INTEGER EXPRESSIONS

An [EXPR] is an integer [EXPR] if all the [OPERAND]'s are scalars, array elements etc. declared to be of type integer, integer constants or integer functions. Thus if we assume that x is a real variable, and i,n,j,k(I),k(2) are integer variables, the following are integer [EXPR]'s.

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

NOTES

I. All calculations on integer [EXPR]'s are done by floating point operations and the result is destandardised at the end.

2. Exponentiation is performed by repeated multiplication, i.e. j ↑ k = j x j x -------(to k terms). Thus the result is exact.

3. The value of the [EXPR] is assumed to be integral; this will in general be true (if the range is not exceeded) except in division when the result may be non-integral. No check is made for this case so that the function 'int pt' or 'int' should be used.

4. Since the machine can hold negative powers of 2 and since the accumulator gives an exact dividend if the numerator is a multiple of the divisor, the first [EXPR] in the examples gives

ARITHMETIC ASSIGNMENTS

The general arithmetic instruction is

[VAR] = [EXPR]

Examples are

X(p, q) = I + 2 cos(2π(x+y))

a = (b + c)/(d + e) + F

i = i + I

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. [VAR]. If the l.h.s. is a real [VAR], 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].

e.g. 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 integer 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]

```
                    -> I0
                    ---
I0:                 ---
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(I), A(2) or A(3) according as i = I, 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 minus sign if

```
switch A(I : 3)
---
---
---
---
A(I):  ---
       ---
A(3):  ---
       ---
       ---
       ---
       -> A(i)
       ---
```

CONDITIONAL LABELS

Another kind of multi-way switch is illustrated by the accompanying diagram. Here the condition 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 in which case control passes directly to the following instructions if it reaches that point.

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

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

A CONDITIONAL OPERATOR of the form

if [COND] then OR unless [COND] then

may be written before any unconditional instruction (including a jump instruction).

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] by itself **

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 $0 \leq x \leq I$ and $0 \leq y \leq I$ then -> I

case (y > I or y < - I) and $x \geq 0$:

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

if [COND] OR unless [COND]

e.g. x = 0 if |x| < .000 000I

-> I unless z > R or z = 0

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

CYCLING INSTRUCTIONS

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

e.g. cycle i = 0, I, n-I

—

—

—

—

repeat

In the above example the instructions between cycle and repeat are traversed n times with i successively taking the values 0,I, ...,n-I.

After the final cycle, control goes to the statement following repeat.

The r.h.s. quantities may be general integer [EXPR]'s and the l.h.s. must be of integer type.

The initial value, increment, and final value must be such that

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

must be a positive integer or zero otherwise a fault is indicated. It is the initial values of the three expressions which are relevant and the cycle is traversed at least once. Cycles may be nested to any depth.

NOTE Statements such as cycle x = .2,.I,I are NOT allowed,and should be replaced by an equivalent permissable form.

e.g. cycle i = 2,I,IO
x = .Ii

where i has been declared integer and x real.

SOME SIMPLE INPUT AND OUTPUT INSTRUCTIONS

It is convenient to introduce some input and output instructions at this stage in order that the reader may complete his study of the introductory example. Strictly speaking, the input and output instructions are calls for the appropriate routines and as such a formal description of them will come later.

Examples of the instructions to read in decimal numbers from a data tape are

read(aI) read(VOLI, VOL2, TEMP, i) read(X(K))

The first of these reads the next number on the data tape into aI; the second reads the next four numbers into the named variables, and may be used to read in any number of individual numbers. The third is of the same form as the first, but the particular variable depends on the computed value of the argument; thus if K was three, the number

```
array  A(I:I00) ;  integer i
cycle  i = I,I,I00
read  (A(i))
repeat
```

The rules for preparing data tapes are given in the section on Input and Output

To print decimal numbers, instructions of the following form may be used

```
print (x + y. cos(z), 2, 7)
print fl( A, 5)
```

The first of these prints the value of the expression in fixed point style with two digits before the decimal point and seven after. The second prints the value of A, standardised so that $I \leq |A| < I0$ in floating point style i.e. as a fixed point number and a decimal exponent, with five digits after the point. A could be replaced by a general expression. The following instructions may be used to facilitate the programming of page layouts:

```
newline
space
```

The first causes subsequent information to be printed on a fresh line and the second causes a space to be left on the current line.
The above instructions are more fully described in the section on input and output.

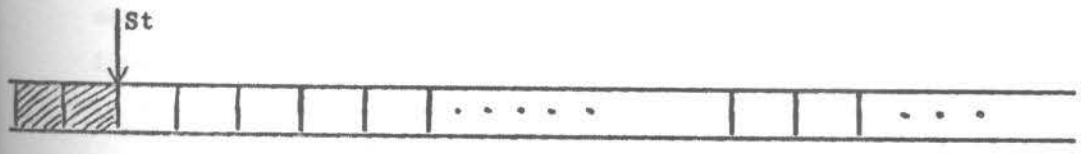
MISCELLANEOUS NOTES

I. With the instructions so far described the reader should be able to construct programs of the level of the introductory example i.e. programs consisting of a single block delimited by begin and end of program.
2. 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.
3. The instruction stop can appear anywhere in the program and signifies the dynamic end of the program; its action is to terminate the calculation.
4. The delimiter comment allows written comments to be inserted in a program to assist other users in understanding it. The information following comment up to the next newline or semi-colon is ignored by the computer.
5. 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

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 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, single shaded cells represent locations which have already been allocated, and double shaded cells represent locations which hold information essential to the compiler, such as array dimensions and origins, and are not of importance in the context of this section. Cells which are allocated to variables are indicated by the presence of the name given to the variable.

STORAGE ALLOCATION DECLARATIONS

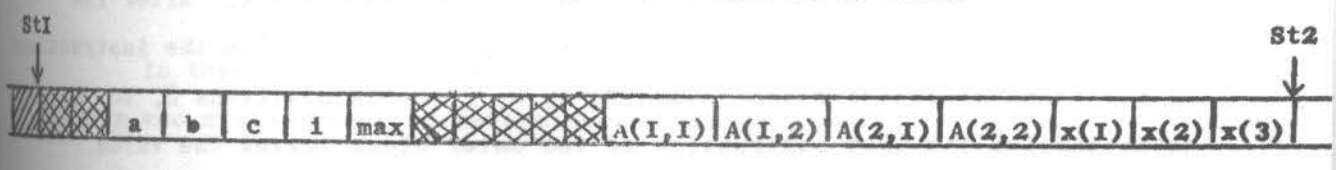
The declarations which allocate storage space are

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(I: 2, I: 2), x(I: 3)
```

After the above declarations the stack picture would be as below



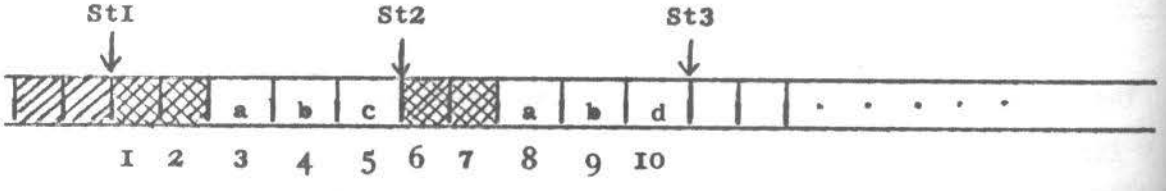
StI is the position of St before begin and St2 its position after the declarations. Any further declaration advances St by an appropriate amount, until end or end of program is reached when St reverts to StI.

BLOCK STRUCTURE OF PROGRAMS

This is illustrated by the following example:

```
begin
real a,b,c
a = I; b = 2
c = a+b
begin
real a,b,d
a = 2; d = I
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 StI, and moves to St2 on entering the first 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 orignal position, StI.

In the diagram, positions 3, 4, 5 correspond to the declarations of the outer block, and 8, 9, I0 to those of the inner block. After the instruction c = a+b, the value 3 is left in position 5; while the instructi of the inner block leave the values 2, I, 3, 4 in the positions 8, I0, 9, respectively. The last instruction of the outer block leaves the value 7 in position 3.

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 end of program, which causes the process of compiling to be terminated and transfers control to the first instruction of the program.

FIXED VARIABLES AND DYNAMIC STORAGE ALLOCATION

Variables declared by real and integer are called FIXED VARIABLES, as are those locations which are double shaded in the stack pictures i.e. those holding the links, array parameters, array function parameters etc. (see later). This is 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(I:m, I:n), x(I:n)

In this case the space allocated will depend on the computed values of m and n and cannot be determined at compiler time.

Owing to the dynamic significance of the storage declarations it is customary to put them at the beginning of a block before the first instruction statement. If they are not put at the beginning it should be remembered that storage will be allocated each time a declaration is reached, without the stack pointer being reset

e.g.
```
begin
integer n
---
---
I: read(n)
array x(I:n)
---
---
---
->I
end
```

In the above example, the stack is advanced and x(i) redefined each time the loop of instructions is traversed, which would almost certainly be undesirable. To define an array in terms of a variable parameter, one could write

```
begin
integer n
I: read(n)
begin
array x(I:n)
---
---
end
->I
end
```

In this case the stack pointer is reset to its original value each time the inner block is left.

ARRAY FUNCTIONS

The declarations of the previous sections 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 and in renaming variables (see later).

An example is

array fn X(s,p)

which defines X(i) as the real 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.

e.g. array fn Y(s,p,q)

defines Y(i,j) = real number in address (s + i.p + j.q)

array fn Z(s,p,q,r)

Z(i,j,k) = real number in address(s + i.p + j.q + k.r)

integer array functions may be defined by prefixing the declaration by integer.

e.g. integer array fn M(s,p)

The parameters in array functions may be general integer expressions.

As an example, assume that IOO storage locations have been allocated in some way, and that the starting address is stored in integer variable sI. Then to define the contents of these locations as a vector x(i), one could write

array fn x(sI,I)

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

array fn x(sI - I,I)

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

array fn A(sI,IO,I)

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

array fn A(sI - IO - I,IO,I)

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

To define the transpose of the above matrix we could write

array fn B(sI - II,I,IO)

when a subsequent reference to B(i,j) would give the same storage location as a reference to A(j,i) of the previous declaration.

It should be noted that, in general, if the suffices of arrays

THE ADDRESS RECOVERY FUNCTION

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.

e.g. s = addr(A(0,0))

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

The address recovery function is used in conjunction with the array function in communicating between routines (see later).

It may also be used in connection with the renaming of variables as in the next section.

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

The first pair of numbers could then be referred to either as x(I), y(I) or a(I), 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 I000 pairs x(i), y(i) can be accomodated.

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

```
array A(I:IO,I:IO)
```

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

```
array fn y(addr(A(I,I)) - IO,IO)
```

which defines y(i) = real number in address (addr(A(I,I)) - IO + IO,i)

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

$$y = a(0) + a(1)x + \ldots + a(n)x^n$$

assuming the coefficients to be stored in consecutive (real) locations. This might take the form

```
routine   poly (real name y, addr s, real x, integer n)
array fn  a(s, I); integer i
y  =  a(n)
cycle i = n - I, -I, 0
y  =  x.y + a(i)
repeat
return
end
```

This routine will be EMBEDDED and used in a main routine as illustrated overleaf.

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

$$U = b(0) + b(1)z + \ldots + b(m)z^m$$

and the second would cause it to evaluate

$$V = c(20) + c(21)x^2 + \ldots + c(30)x^{10}$$

```
begin
real U, V, z, x; integer m
array b(0:5), c(0:50)
routine spec  poly (real name y, addr s, real x, integer n)
---
---
poly (U, b(0), z, m)
---
---
poly (V, c(20), x^2, 10)
---
---
routine  poly (real name y, addr s, real x, integer n)
```

body of poly routine

```
end
---
end of program
```

The parameters in the routine specification and routine heading are the FORMAL PARAMETERS and the parameters in the call sequences are the ACTUAL PARAMETERS, precise definitions of which will be given in the 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 the 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,

We define the TEXTUAL LEVEL of the body of the main block as I and of each subroutine (i.e. within the subroutine body) as 2. Now each subroutine is basically the same in structure as the main block (which is essentially a routine without parameters) and a subroutine may define and use its own subroutines in precisely the same manner. The textual level within these subroutines would be 3, and the progression may continue to any depth.

The most general form of a complete program is thus a nested hierarchy of routines. A typical routine layout is shown below:

```
routine  A(-, -, -)        routine heading
integer    -, -, -
real       -, -, -
array      -, -, -          declarations of local
                            working space and specifications
routine spec  B1(-, -, -)   of subroutines used.
routine spec  B2(-, -, -)
routine spec  BN(-, -, -)
---
---                         body of routine.
---
routine  B1(-, -, -)
---
end
routine  B2(-, -, -)
---
end                         subroutine headings and bodies
routine  BN(-, -, -)
---
end
end                      end of routine
```

The picture is to be treated recursively i.e. it also applies to the bodies of each subroutine, and the whole routine might itself be embedded as a subroutine of a still larger routine.

FORMAL PARAMETERS AND ACTUAL PARAMETERS

The parameters of a routine are the pieces of information (variables, addresses, routine names) which tell it what to do on the different occasions when it is used. The formal parameters are the names by which this information is referred to inside the subroutine itself. The actual parameters are the expressions which are substituted in place of the formal parameters whenever the routine is called in. 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	any expression (which will be evaluated as for an integer assignment)
real	ditto (but for a real assignment)
addr	The name of any integer or real variable (including an array element). The actual address of the variable is handed on as the parameter proper
routine type i.e. routine [RT] real fn integer fn real map integer map	In some cases it is required to pass on a routine as a parameter, and the actual parameter is then the name of the routine, which must be of the same type as the formal parameter and have formal parameters which correspond in type to those of the formal parameter. An example is given in a later section

NOTE An addr parameter is equivalent to an integer parameter in the body of the routine. The difference relates to the corresponding actual parameter. Thus an addr parameter replaced by x is equivalent to an integer parameter replaced by addr(x).

In the polynomial example described earlier, the formal parameter y is an output parameter and the actual parameter must therefore be a name. The formal parameter s is the address of the first coefficient a(0) and the statement array fn a(s, I) establishes the store mapping function as a(i) = real number in address (s+I.i).
The parameter x may be replaced by an expression the value of which is the parameter proper, similarly for n. [The parameters s, x, n are called by VALUE; they correspond to local locations in the poly routine.]

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 the dynamic end as in the present example the return instruction can be omitted; in this case end serves for both purposes.

Another example of a routine is given below:-

```
routine mat mult (addr sI, s2, s3  integer m, p, n)
integer i, j, k;  real c
array fn A(sI, p, I),  B(s2, n, I),  C(s3, n, I)
cycle   i = 0, I, m-I
cycle   j = 0, I, n-I
c = 0
cycle   k = 0, I, p-I
c = c + A(i, k)B(k, j)
repeat
C(i, j) = c
repeat
repeat
end
```

This forms the product of an m x p matrix stored in location sI onwards, and a p x n matrix stored in s2 onwards, and places the resulting m x n matrix in s3 onwards, all three matrices being assumed stored row by row. A typical call sequence might be

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

FUNCTION ROUTINES

When a routine has a single output value only it may be written as a FUNCTION ROUTINE.
e.g. The polynomial routine may be recast as a function routine as follows:-

```
real fn poly (addr s,  real x,  integer n)
array fn a(s, I); integer i; real y
y  =  a(n)
cycle i = n-I, -I, 0
y = y.x + a(i)
repeat
result = y
end
```

Here the (real) result must be assigned immediately before exit. The specification is given by

```
real fn spec poly  (addr s,  real x,  integer n)
```

and the routine might be called in by an assignment statement such as

```
y = a.b + 2h. poly (c(0), I/x, I6)
```

An example of an integer function is given below. It selects the index of the maximum element x(k) in a set x(m), x(m+I) ... x(n) n > m stored in consecutive locations, assuming x(0) is stored in address sI.

```
integer fn  max(addr sI,  integer m, n)
integer  i, k
array fn x(sI, I)
k  =  m
if  n = m  then -> I
cycle i = m+I, I, n
if x(i) > x(k)  then k = i
repeat
I:result = k
end
```

and a call sequence might be

```
y = I + mod(z(max(x(0), 50, I00)))
```

NOTE The delimiter result must be used as the l.h.s. of the assignment statement signifying the dynamic end of the function routine. It does NOT correspond to a store location and cannot be used in any context other than the above.

STORE MAPPING ROUTINES

Finally, store mapping functions may be defined by writing real map or integer map before the function specification and heading.

e.g.

```
real map  X(integer i, j)
    result = s + i.n - i.(i-I)/2 - n + j - I
    end
```

This defines a real triagular matrix of n columns, store

X may appear on the l.h.s. of an assignment statement
e.g. X(i-I, j+I) = [EXPR]
s, n would probably be local to the routine in which such a statement appeared.

SCOPE OF NAMES

In general all names are declared at the head of a routine either in the routine heading or by the declarations integer, real, array, array fn, switch, 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 in the routine outside i.e. in the routine in which it is embedded. If it is not declared in that routine it is looked for in the routine outside that and so on until the main block is reached. The store mapping function of the last section is an example of a routine which uses non-local names.

Now the main block is itself embedded in a permanent block of textual level zero 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 meaning 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. In general, the only non-local names used in a routine will be the permanent names.

PERMANENT ROUTINES AND LIBRARY ROUTINES.

The permanent names include the standard functions, sin, log, addr, etc. and the basic input/output routines read, print etc. These routines are used in a programme 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.

In addition to the permanent routines, there will exist LIBRARY routines which will be permanently stored on magnetic tape. These library routines may be automatically incorporated into a user's program by means of a library routine declaration.

e.g. library routine spec least squares (addr sI,s2,integer m,n)
The appropriate routine is then automatically inserted into the programme at the current level i.e. the name is local to the routine containing the library declaration.

AN EXAMPLE OF A COMPLETE ROUTINE

routine spec line fit (addr s, integer n, real name a, b, c)
The routine takes n pairs of numbers Xi, Yi stored in s onwards and forms the quantities

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

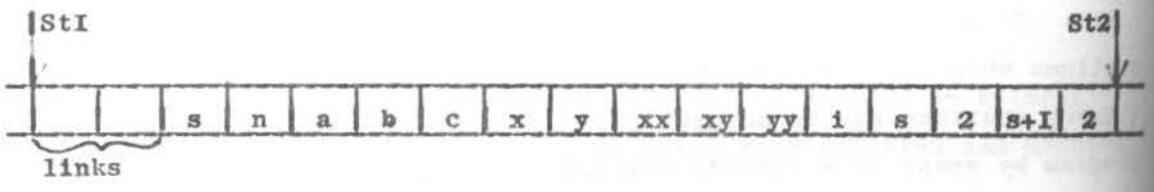
$$b = \frac{[>Yi - a[>Xi}{n}$$

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

[cf introductory example]

```
routine line fit (addr s, integer n, real name a, b, c)
real  x, y, xx, xy, yy; integer i
array fn X(s, 2),  Y(s + I, 2)
x = 0;  y = 0;  xx = 0;  xy = 0;  yy = 0
cycle   i = 0, I, n-I
x = x + X(i);  y = y + Y(i)
xx = xx + X(i)^2;  yy = yy + Y(i)^2
xy = xy + X(i) Y(i)
repeat
a = (n.xy - x.y)/(n.xx - x.x)
b = (y - a.x)/n
c = yy - 2(a.xy + b.y) + xx.a^2 + 2a.b.x + n.b^2
return
end
```

The stack picture associated with the routine above is given below.



When the routine is entered the stack pointer is advanced from StI to St2. The first two locations are used by the compiler and the

others correspond to the formal parameters of the routine, and the declarations in the body of the routine. We note that the locations labelled a,b,c in the diagram will contain the addresses of the named variables, since they correspond to output parameters. The last four locations correspond to the array fn declarations in the body of the routine. The above are the fixed variables of the routine; any array declarations in the routine would cause St to be further advanced by an amount depending on the current values of the parameters in the declarations. When return is reached in the above routine, the stack pointer reverts to StI and the information between StI and St2 is lost. A further description of the mechanics of the stack is given in the section on the use of machine instructions.

OWN VARIABLES

We see from the above description of the stack that when a routine is left the information stored in the local variables of the routine is lost, and no further 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,

e.g. own real a, b ; own array A(I:IO)

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. This is done at compiler time and hence does not have dynamic significance; as a consequence, an own array statement must have parameters which are integer constants.

FUNCTIONS AND ROUTINES AS PARAMETERS

This is illustrated by the following example involving a hypothetical library integration routine

library 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

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

where $f(i) = f(a + .5i.(b-a)/n)$

An auxiliary routine is required to evaluate f(x) and details of it must be passed on to the library 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  h;  integer i
real fn spec  f(real x)
h = .5(b-a)/n
y = 0
cycle i = 0, 2, 2n-2
y = y + 2f(a + i.h) + 4f(a +(i+I)h)
repeat
y = (y - f(a) + f(b))h/3
end
```

Now consider a programme to evaluate

$$z = \int_0^1 \exp(-y)\cos(b.y)dy$$

for various values of b read from a data tape, the last value being followed by I000, using the integer value of n nearest to I0b.

```
    begin
    library routine spec integrate (real name y,real a,b,integer n,real fn f)
    real z, b
    real fn spec aux (real y)
    comment Simpson rule integration
  I:read (b)
    if b = I000 then stop
    integrate (z, 0, I, int(I0b), aux)
    newline
    print (b, I, 2);space;space;print (z, I, 4)
    -> I
    real fn aux(real y)
    result = exp(-y) cos(b.y)
    end
    end of program
```

Note that the names given to the auxiliary routine and its parameters need not be the same in the library programme and the main programme, but they must correspond in type.

The function routine 'aux' in the above example is an example of a routine referring to a non-local name b.

5 INPUT AND OUTPUT ROUTINES

BASIC INPUT ROUTINES

The input of data is handled by permanent routines, some examples of which were given in an earlier section.

The basic input instructions are:

routine spec select input (integer i)

This selects an input channel from which subsequent data is read. This channel remains selected until another 'select input' instruction is encountered. If no channel is specified, channel 0 is automatically selected.

routine spec read ([VARIABLE])

This reads a decimal numbers from the currently selected data channel and places it in the location specified by the [VARIABLE] which may be either a real name or an integer name. The routine reads numbers in either fixed or floating point form.

e.g. −0.3I0I I8 7.I32α-7 3.I872α I4

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. It is terminated by the first symbol which is not a decimal digit. Spaces and newlines preceding numbers are ignored, but all other symbols cause the routine to signal a fault. 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 (c.f. constants appearing in the programme where all spaces are irrelevant and numbers are terminated by the following name or delimiter).

If individual numbers are separated only by spaces or newlines, then a series can be read by the call

read ([VARIABLE LIST])

e.g. read (a,i,X(i))

This is treated as if it were a series of calls

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, since it may have an indefinite number of real names and

Another permanent input routine is

routine spec read symbol (integer name i)

This reads the next symbol (single or compound) from the selected channel, converts it into its numerical equivalent and places the result in the specified integer location. A table of numerical equivalents and a description of the formation of compound symbols is given in Appendix 3.

NOTE. Erases are ignored completely on input to the computer and therefore do not exist as a character inside the machine.

BASIC OUTPUT ROUTINES

The basic permanent output routines are:

routine spec select output (integer i)

which corresponds to the 'select input' routine. Again channel 0 is selected unless otherwise specified.

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 any [EXPR] in floating point form standardised in the range [1, 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 second routine prints the value of x in fixed point form with m digits before the decimal point and n after. Insignificant zeros, other than one immediately before the decimal point are suppressed and a minus sign or space precedes the first digit printed. If $|x| \geq 10$ 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. Terminating characters should be included by the user by means of the permanent routines given below:

routine spec newline

routine spec space

routine spec spaces(integer n)

routine tab

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 successive spaces are required, the third routine may be used e.g. spaces(5) .

The fourth routine causes the printer to move to the next tab setting or punches the tab character.

Another permanent routine is

CAPTIONS

There is a special facility for printing captions,

e.g. caption ⫽⫽⫽TABLE⫽OF⫽TEMP⫽AGAINST⫽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 on input and ; is a terminating symbol, the compound symbols ⫽ ⫽̲ ⫽; are used within the caption statement to denote 'space,' 'underlined space', and 'semi-colon' respectively.

Thus

```
newline
caption  A ⫽ = ⫽⫽ ; print (y,I,3); newline
caption  B ⫽ = ⫽⫽ ; print (z,I,3); newline
```

would be printed as

```
A =   I.7I2
B =  -2.389
```

To enable a caption to appear in the same vertical alignment on the printed programme page and the output page, the following device can be used:

```
newline
caption c
TABLE⫽OF⫽TEMP⫽AGAINST⫽VOL
```

OTHER INPUT / OUTPUT ROUTINES

Input and output of binary information is performed by the routines

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

The first reads the next row of holes on the tape as a binary number and places it in the named variable. 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. In both cases the selected channels must have been designated 'binary' in the job description.

The following two routines are useful in constructing special purpose input and output routines.

```
integer fn spec  next symbol
routine spec     skip symbol
```

The instruction

```
p = next symbol
```

places the numerical equivalent of the next simple or compound symbol on the data tape in the named variable without moving on the data tape. Thus the information is still available for a subsequent read instruction.

The second routine skips a symbol on the data tape and is equivalent to

```
read symbol (p)
```

6 MONITOR PRINTING AND FAULT DIAGNOSIS

FAULT MONITORING

There are two types of fault which can be detected by the Compiler, firstly those which can be found during compiling and secondly 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 in which the beginning and end of each routine are printed against the physical line number. This gives a broad outline of the program as an aid in finding the faulty instructions and also correlates each routine with its routine number, for use in tracing faults found at run time (see later). Also 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. Finally at the end of each routine all the non-local variables except the reserved names 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

```
I     BEGIN
26    NAME NOT SET
55    LABEL 7 SET TWICE
70    BEGIN ROUTINE POLY = 24
115   NAME NOT SET
115   REAL QUANTITY IN EXPR
180   END OF 24
      LABEL 18 NOT SET
      NON-LOCAL VARIABLES
      A
      TEMP
      SI
182   END
```

The above should be self-explanatory. It indicates that the program started at line I and finished on line 182. These are physical lines and include all blank lines on the print-out. The routine POLY started

lines 26 and 55 and two in line II5. 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 instructions and test instructions.

The standard procedure is to print out a line of information specifying the fault followed by a list of the FIXED variables used. For example

```
ALL TESTS FAIL
ROUTINE 38 LINE II7
FIXED VARIABLES
  .
  .
  .
  .
```

indicates that at the test instruction in line II7 routine 38 (the line number gives the line in the whole program not in the routine), none of the [COND]'s on the labels were satisfied. Then follows a list of the fixed variables of the routine.

A full list of run time faults appears in Appendix 4.

FAULT TRAPPING

The above standard monitoring procedure involving the termination of the program, may prove inconvenient. For example, if a program has a number of sets of data, rather than stop if the accumulator overflows (i.e, some number becomes so large as to be out of the machine's range), in the middle of one set, it may be preferable to restart on the next set.

The user may write a set of instructions in his main program (at label I) and label them, for the accumulator overflow case, with

fault (I) :

The action taken by the machine is first to print out the nature of the fault, then to test whether the appropriate fault label has been set; if it has it then obeys the instruction labelled and carries on; if not it gives the standard fault monitoring. The relevant fault label numbers are given in Appendix 4.

FAULT DIAGNOSIS

There are many program faults which manifest themselves only in wrong answers, and the following facilities are incorporated to aid users in tracking down such faults.

QUERY PRINTING

All arithmetic instructions, including complex, may be followed by a query (?).

e.g. a = b(i) + c ?

After obeying such an instruction, the value of the l.h.s. is printed out in floating point style with ten significant figures. The compiling of the query print instructions may be controlled by the statements

compile queries

ignore queries

The first instruction causes the subsequent queries in the program to be compiled, until an ignore queries statement is reached where subsequent queries are ignored.

ROUTINE and LABEL TRACING

There are two tracing facilities available: the routine trace and the label trace. In the areas where the routine trace is operative it causes the routine number to be printed out each time it is entered. The correspondence between routine number and its name can be found from the program outline produced during the compilation. Thus a print-out may appear

RI R5 R3 R2 RI R5 R3 R2 .....

The label trace facility allows the flow of the program to be followed in greater detail. For every simple 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 it is the value of the switch that is printed. Thus a print-out might appear

-> 3 test I -> 4 -> 6 switch 3 -> 7 -> 8 -> 9 .....

If the label and routine trace are both operative the print-out might appear

RI -> 3 test I -> 4 -> 6 R5 switch 3 .....

As with queries the areas in which the traces are operative are delimited by the instructions

compile jump trace

stop jump trace

compile routine trace

stop routine trace

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 in a separate tape or included with the program or data. A full description of the system is given elsewhere **, and in this section we give examples to illustrate the general principles of job descriptions.

We give first an example of a program with its data on the same tape.

```
JOB
(Title)
COMPILER ATLAS AUTOCODE
OUTPUT
0  LINE PRINTER   5 BLOCKS
I  TELETYPE       2 BLOCKS
STORE 30
COMPUTING  I.5 MINUTES

begin
```

```
end of program
```


```
***Z
```

** Howarth, D.J., Payne, R.B., Sumner, F.H., "The Manchester University Atlas Operating System. Part 2: Users' Description" Computer Journal, October 1961.

NOTES

I. The title identifies the program. 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.

2. The OUTPUT information says that reference to channel 0 in the program means the line printer and channel I means a Teletype punch. The block numbers give an upper limit to the number of blocks of output that is to be permitted on each device (4096 characters per block). If the limit is reached the program is terminated. In the absence of an OUTPUT section the following is assumed

OUTPUT

0 ANY I BLOCK

ANY indicates that any of the output devices may be used.

3. The STORE and COMPUTING sections are optional. STORE gives an upper limit on the number of 512 word main store blocks used by the program and data, while COMPUTING gives a limit on the running time of the program. If either limit is exceeded, the program is terminated. If the above information is not present, the following is assumed

STORE 20

COMPUTING 4 SECONDS

4. A program tape is always assumed to be on program channel 0 so that in the above case, the data for the problem is also on channel 0.

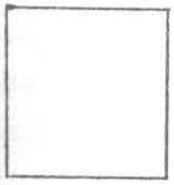
5. ***Z is an end of tape marker and indicates that all the information on the tape has been read.

A second common form of complete program is a program tape and a separate data tape. In this case the program type might be

COMPILER ATLAS AUTOCODE

(Title I)

begin



end of program

***Z

The data tape, which this time includes the job description, might be

```
JOB
(Title 2)
INPUT
0 (Title I)
SELF = I
OUTPUT
0 ANY 10 BLOCKS
STORE 20
COMPUTING 30 SECONDS
DATA
```

```
***Z
```

The INPUT description gives the relevant program as being channel 0 (the program channel) and SELF = I indicates that the data tape is to be read as channel I. Thus an instruction 'select input (I)' would be required in the program.

CORRECTIONS TO PROGRAMS

It is possible to insert corrections automatically in a program by reading in a correction tape, which is designated channel 15 in the job description.

The available instructions are

delete line [N]

replace line [N] by

end of correction

end of corrections

The line numbers refer to the physical line numbers of the original program.

A correction tape might be as below

```
DATA
(Title)
replace line 30 by
a = a*x - b
z = cos (a)
delete line 35
end of corrections
***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 at the beginning of the routine. For example

```
real    RI, R2, R3
complex    z
complex array    P(I:IO), Q(I:IO,I:IO)
```

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

Similarly if sI were the starting address of the matrix Q above, then the declaration

```
complex array fn  for R(sI - 20, IO)
```

would define a complex vector R(i) whose elements were the first column of the matrix Q. Note that the user need only be aware of the method of storing complex numbers for evaluating the origin (here sI - 20) of an array. The other parameters (in this case IO) refer to elements rather than locations and the factor 2 is automatically taken care of.

STANDARD FUNCTIONS

The following standard functions are added to those previously given

```
re(z)        (real part of z)
im(z)        (imaginary part of z)
arg(z)       (argument of z)
conj(z)      (complex conjugate of z)
```

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

sin, cos, tan, log, exp, sq rt, mod

may also have complex [EXPR]'s as arguments and they are interpreted in the normal fashion.
For example if

$$z = x + \underline{i}y$$

then $\exp(z) = \exp(x)(\cos(y) + \underline{i}\sin(y))$

The functions

arctan, radius, frac pt, int pt, int

are meaningless if their arguments are complex.

ARITHMETIC EXPRESSIONS

The arithmetic expression [EXPR] is still of the form

[±?][OPERAND][OPERATOR][OPERAND][OPERATOR] ..... [OPERAND]

but [OPERAND] is now expanded to be

[VAR],[CONST],([EXPR]),|[EXPR]|,[FUNCTION] or i

Here 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))/(i2)
(ZIZ2 + Z2Z3 + Z3ZI)/Z3
Y(I,2) + sin(conj(Y(2,I)))
RO(I + i2Q0d)
i
```

NOTES

I. When a complex number is written out explicitly (say x + iy), then it is regarded as 3 operands (x,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))/i)2

ARITHMETIC INSTRUCTIONS

The form of the instruction remains

[VAR] = [EXPR]

but [VAR] now includes complex scalars and complex array elements. For example

Z = ZIZ2/(ZI + Z2)
Y = G + i2πf*c
A(p,q) = 2sin (2πz)
R = RI + re(Z)
P = ½(V.conj(I) + I.conj(V))

NOTES

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

re(z), im(z), mod(z), arg(z)

convert from complex to real quantities and may therefore appear on the r.h.s. of a real instruction. In fact any function whose value is real regardless of its arguments may be used in a real instruction. Thus if X and B are real 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) = sq rt (2)
re(y) = 5 +im (zI)

However, mod(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 modulus (m) and argument (a), the assignment is done by

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

3. When the functions

sin, cos, exp, mod

appear in a complex instruction, a test is made to see whether the arguments are real or complex, and the appropriate method used.

4. However, when the functions

sq rt, log

appear in a complex instruction, then arguments are regarded as complex. Consequently, in a complex instruction, the evaluation of sq rt (-4) gives i2; and does not cause the machine to stop as it would in a real instruction.

DATA

complex numbers on a data tape are punched using similar conventions as for real numbers. For example

3+i4; i; I.I7α3 - i2.I3α4

Within the number spaces may only appear immediately before or immediately after +i or -i.
They may be read by the usual read instruction

e.g. read (zI,z2,z3)

CONDITIONS

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

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

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	Actual parameter type
complex name	name of a complex variable
complex	any expression (which will be evaluated as if for a complex assignment)

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

$$a(0)+a(I)x+ \ldots\ldots\ldots a(n)x$$

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

```
complex fn spec poly (addr s, complex x, integer n)
complex fn poly(addr s, complex x, integer  n)
complex array fn a(s,I); integer i; complex y
y = a (n)
cycle   i = n-I, -I, 0
y = y.x + a(i)
repeat
result = y
end
```

9 HALF-WORD OPERATIONS AND LIST PROCESSING FACILITIES

OPERATIONS ON HALF-WORDS

In the previous sections we have regarded the 48-bits of a word as a single entity, as a number in fact. There are many applications in non-numerical work where we can conveniently regard the 48-bits split into a number of independent entities and even into single digits. (We will in these cases consider the 48-bit words as two 24-bit words often called half-words). For example, characters are represented by their numerical equivalents (see Appendix 3) in the range 0 - I27 . Since this number can be represented in 7 bits it is possible to pack three such characters to a word. For example a b c could be packed

i.e. *6070 5430 .

Facilities are provided for performing operations on 24-bit words, to pack and unpack information so stored, and for manipulation in list structures etc. These are very similar to those used in the Compiler Compiler, the system used to write the Atlas Autocode Compiler.

The scheme embraces the B-lines of Atlas which are given the special names

βI, β2, β3 .....

The restrictions on the use of B-lines described in the section on machine instructions must be observed. Thus only βII - β59 are generally available to the user. These are 'global' variables and do not have to be declared nor handed down from one routine to another as a parameter. In them the binary point is assumed to be one octal place up from the end.

For local variables we may use the least significant half of an integer type variable. (This is due to the method of storing an integer quantity. As explained in the section on machine instructions, the integer I0 is stored as $8^{12} \times (I0 \times 8^{-12})$ i.e.

03000000 00000I20

and if the least significant half is interpreted in the same way as a B-line, it has the value I0.

If for example K, MIN, I3, I2

have been declared integer quantities then we can refer to their least significant halves as αK, αMIN, αI3, αI2. In the particular case of the names αI3, αI2 ..... we can abbreviate them to α3, α2 ..... This facility is peculiar to the names II, I2, I3 ..... etc. which must still be declared.

Half-word operations are effected by the instruction

[H-VAR] = [H-EXPR]

and [H-EXPR] is defined as

[±?][H][HO][H].....[H]

where [H] denotes a half-word operand and [HO] a half-word operator

[H] is defined as

[H-VAR],[CONST],[OW],([H-EXPR])

and [H-VAR] as

β[N],α[NAME],α([H-EXPR])

β[N] describes the B-lines and α[NAME] the least significant half of integer [NAME]'s as described previously. α([H-EXPR]) denotes the 24-bit word in the address given by the value of [H-EXPR]. Note that this address will not in general be integral but will take the form s/2, [CONST] and [OW] have the meanings described in the section on machine instructions and ([H-EXPR]) has the usual significance of an expression in parenthesis. In addition each half-word operand [H] may be preceded by a negation operator (-) which causes the operand to be negated (i.e. all I's changed to 0's, all 0's to I's) before it is used.

The half-word operator [HO] is defined as

+, -, *, ≠, &, V, /, >, <, (+)

Their operators are described in detail below but note

1. The operators have uniform precedence (c.f. ordinary arithmetic operators) and are applied to successive partial results in order from left to right. Thus

   α1 + α2 - α3 & α4 V α5/6.125

   interpreted

   (((((α1+α2)-α3) & α4) V α5)/6.125)

   If any other interpretation is required it must be indicated by the use of brackets which have the usual over-riding precedence.

2. The operations are carried out on all 24-bits including the three to the right of the binary point.

The operators +, - have their usual meanings. * denotes multiplication and gives a result with the decimal point three places up from the least significant end. Thus

3.5 * 4.5 yields I5.75

Since information can be lost at the least significant end, the exact product is rounded by the addition of 1/16. Information

/ denotes division and gives a quotient with the decimal point three places up.
&, $\underline{\vee}$, $\underline{\not\equiv}$ (Called 'and', 'or' and 'not equivalent' respectively) are logical operations which are performed between each pair of corresponding digits in parallel. For any pair of digits the result is given by the following tables

0 & 0 = 0	0 $\underline{\vee}$ 0 = 0	0 $\underline{\not\equiv}$ 0 = 0
0 & I = 0	0 $\underline{\vee}$ I = I	0 $\underline{\not\equiv}$ I = I
I & 0 = 0	I $\underline{\vee}$ 0 = I	I $\underline{\not\equiv}$ 0 = I
I & I = I	I $\underline{\vee}$ I = I	I $\underline{\not\equiv}$ I = 0

Thus the operation

*0000003I & *00000055

i.e. 000 000 000 000 000 000 0II 00I
& 000 000 000 000 000 000 I0I I0I
would give 000 000 000 000 000 000 00I 00I
i.e. *000000II

The operators ⩾ and ⩽ refer to left and right shifts respectively (note that ⩾ must not be confused with ->, the arrow sign)
These shift the digits up or down circularly; i.e. digits which are shifted over the top reappear at the bottom.
Thus if αI = *0707 0707 then α2 = αI ⩽ 3 sets α2 = *7070 7070
and α3 = αI ⩾ 2 sets α3 = *6I6I 6I6I

These instructions allow us to unpack information packed in the manner escribed at the beginning of the section. Thus if αI contains a, b, and c packed i.e. if αI = *6070 5430
then α2 = αI & *0037 6000 ⩾ 7
sets α2 to *0000 I420, the numerical equivalent of b.

The operation (+) refers to the chaining of information and is described later (see section on lists).

The form of the conditional operators is expanded to allow [H-EXPR][ø][H-EXPR] as a form of [SC]. For example a conditional instruction may be

$\underline{\text{if}}$ βI ≥ αI+I0 $\underline{\text{then}}$ ->I0
or $\underline{\text{if}}$ β3+β4 ≠ I000 $\underline{\text{then}}$ αI = αI+I

CONVERSION TO INTEGER OR REAL

The definition of [EXPR] has been extended to include [H-EXPR] as an alternative form, thus

[EXPR] = [+?][EXPR'].[H-EXPR]

constants. If an
as part of an [EXP
significant half o
to real the result
ised. Note however
bits being assumed
with the treatment
[NAME] = [EXPR'].
that is to say the
permit constants whi

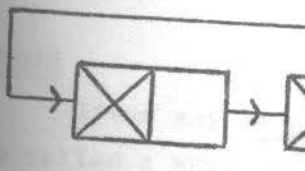
LISTS

Quite often in
sub-sections which m
the total of all the
forward solution is
cover the largest pos
wasteful of storage a

A list consists
as a pair of half-wor
information and the s
is in inverted commas
in the store.
Diagrammatically if

and

a list might appear:-



It is thus possible to
into a chain. In genera
the second half of the

It is to this typ
if αI is set to the addr
word in the chain, αI(+)
Since the lists are circ
lists αI(+)I ≡ α(αI+½).
NOTE lists may only be

constants. If an [H-EXPR] appears in place of an [EXPR] (or within parentheses as part of an [EXPR]) it is converted to 48-bit form by giving it a most significant half 0300 0000 if it is +ve, 0317 7777 if it is -ve. To convert to real the result is then simply standardised, otherwise it is left unstandardised. Note however that all 24 bits are preserved, the three least significant bits being assumed to be zero or in some way relevant. This is consistent with the treatment of the instruction i = 2.5, which is recognised as [NAME] = [EXPR']. Strictly speaking the [EXPR'] should be an 'integer' expression, that is to say the result should be an exact integer. The checks applied, however, permit constants which are a multiple of 1/8.

LISTS

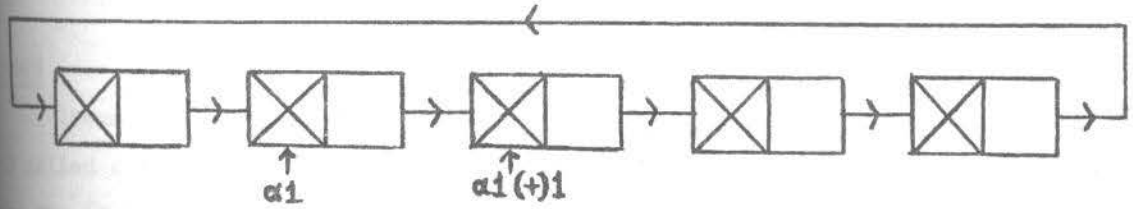
Quite often in non-numeric work, the data may be divided into sub-sections which may have a length which varies over wide limits but where the total of all the sections generally is much more constant. The straightforward solution is to allocate storage for each section big enough to cover the largest possible length which may appear. However, this is very wasteful of storage and the idea of a LIST has been developed to overcome this.

A list consists of a number of full-word registers which may be considered as a pair of half-word registers. The first half-word contains useful information and the second the address of the 'next' word in the chain. 'Next' is in inverted commas since the actual location of the 'next' word may be anywhere in the store.

Diagrammatically if  indicates an 'information' word

and  the linking addresses,

a list might appear:-



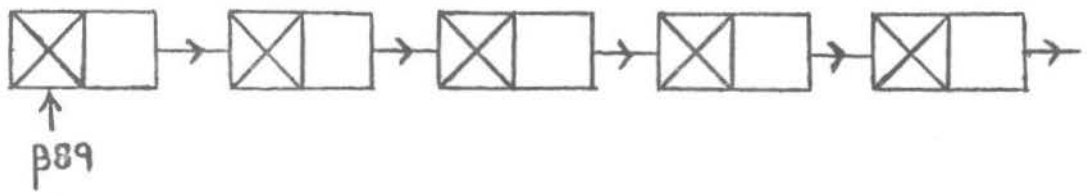
It is thus possible to utilise ALL the working space as all words may be linked into a chain. In general we deal with CIRCULAR chains, i.e. chains in which the second half of the last word contains the address of the first word.

It is to this type of structure that the operator (+) refers. Thus if αI is set to the address of (often shortened to 'if αI points to') a word in the chain, αI(+)1 points to the next, and αI(+)2 to the one after. Since the lists are circular, in the above αI(+)5 ≡ αI. Again from the nature of lists αI(+)I ≡ α(αI+½).

All lists may only be traversed in one direction (the direction of the

LIST PROCESSING FACILITIES

The Autocode contains a number of facilities for forming and manipulating lists. All the operations make use of a central or main chain, whose address is kept in β89



A large area of the store is initially chained up in this way and it is assumed that while registers may be borrowed from and returned to it, it is never exhausted.

Lists are referred to by list [H-VAR]. Thus list αI is the list whose address is given by αI. The main chain then is effectively list β89. Empty lists are characterised by having this address zero.

For setting up lists initially the instruction

set up list [H-VAR]

is provided. It creates an empty list for use later on. To do this, of course, it merely sets [H-VAR] = 0. (In fact for example αI = 0 could be written and it would have the same effect. The above format is essentially to increase readability of programs).

Items can then be added to this list by the instruction

add [H-EXPR] to list [H-VAR]

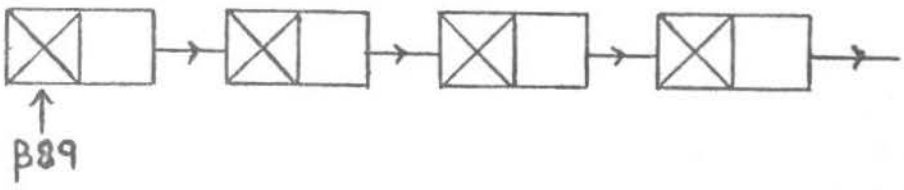
For the first variable to be added the instruction causes the [H-EXPR] to be evaluated and placed in the next word in the main chain; and then the word is detached and made into a circular list of one word.

Thus the instructions

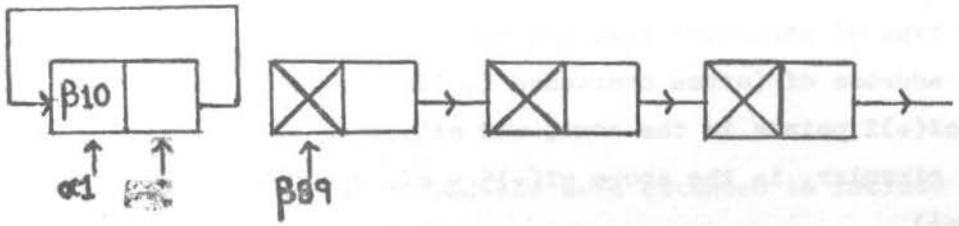
set up list αI

add β10 to list αI

would result in a change from

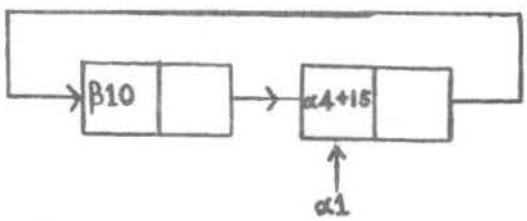


to



A further instruction

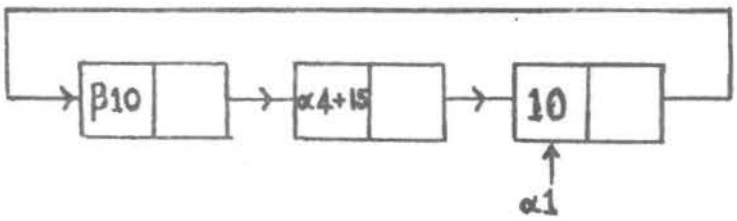
location of the main chain, which is then detached and added to the list, αI being updated in the process. The list then appears:-



A consequent

add I0 to list αI

would cause list αI to become



NOTES

1. In list [H-VAR], the [H-VAR] always points to the last word added.
2. The words are added in such a way that on traversing the chain (which can only be done in one direction), the items are met in the order in which they are entered.

A number of items can be added at the same time by means of the instruction

add [H-EXPR-LIST] to list [H-VAR]

where [H-EXPR-LIST] is defined as a series of [H-EXPR]'s separated by commas. For example

set up list αI

add βI0, α4+I5, I0 to list αI

would give rise to the same list as above.

NESTS

Items may be placed in a chain in another way and the resulting chain is called a NEST. In contrast to a list the nest has the following properties.

1. The [H-VAR] always points to the first word added.
2. On traversing the nest, items are encountered in the reverse order in which they are entered.

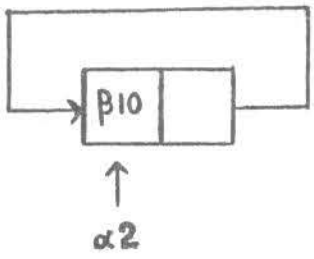
It is this latter property which is important in nests. It is a 'first-in-last-out device'.

The instruction

set up nest α2

like the corresponding listing instruction, sets α2 = 0.

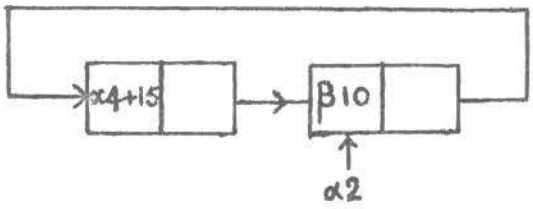
Similarly the instruction



However a following instruction, say,

add α4 + I5 to nest α2

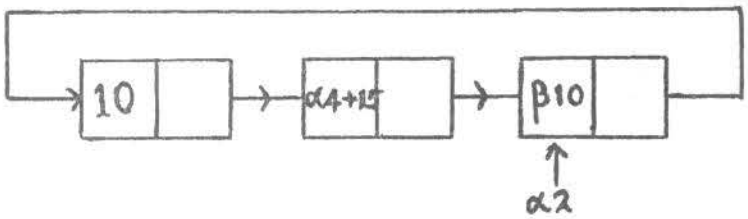
results in



where α2 is unchanged and the consequent

add I0 to nest α2

results in



where the item added is inserted immediately after the item at the head of the chain (i.e. the one whose address is in α2) and where α2 does not change.

The instruction

add [H-EXPR-LIST] to nest [H-VAR]

adds a series of [H-EXPR]'s to the nest.

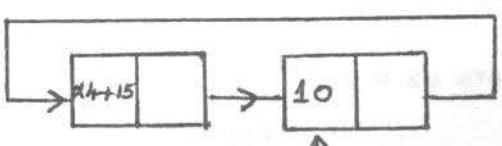
There are two instructions for retrieving information stored in lists and nests.

withdraw [H-VAR] from list [H-VAR]

sets [H-VAR] to the value of the FIRST item entered in the list and returns the word which contained it to the main chain. For example, considering the list αI previously built up, the instruction

withdraw β5 from list αI

sets β5 to the value of βI0 and contracts the list to



same order in which it was entered.

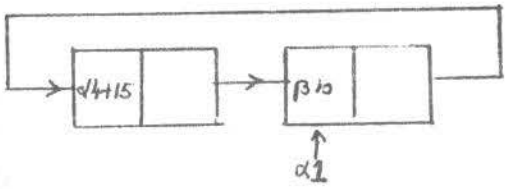
The corresponding nesting instruction

withdraw [H-VAR] from nest [H-VAR]

sets [H-VAR] to the value of the last item entered and returns the word it used to the main chain; for example,

withdraw β5 from nest α2

sets β5 to I0 and contracts the nest to



and further withdraw orders cause the information to be retrieved in the REVERSE order to which it was entered.

NOTES

I. The two operations are exactly the same physically; it is the difference in entering information in a list or nest which results in the different order in which items are withdrawn. The action is to set the [H-VAR] to the information in the next word after the head of the chain and return that word to the main chain.

2. If the list or nest contains only one word, withdrawing a word sets αI or α2 to zero. It is impossible to withdraw information from an empty list or nest and any attempt to do so will cause trouble.

Lists or nests may be deleted by

delete list [H-VAR]

or delete nest [H-VAR]

The effect is to break open the list or nest and return it to the main chain.

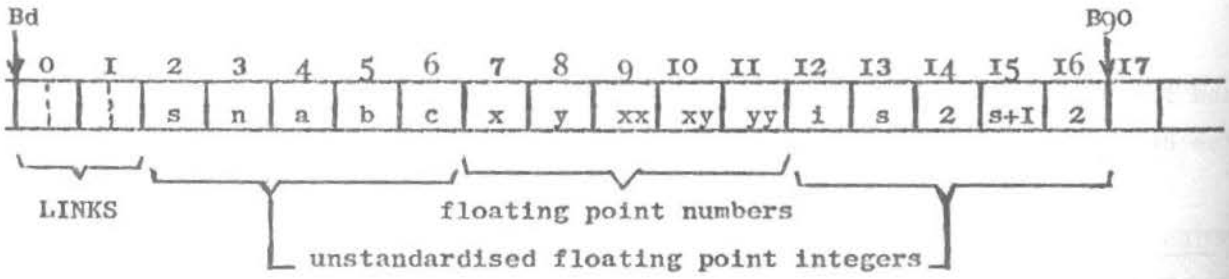
Thus referring to the following diagram

delete list β4 would result in

IO 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 essential however, before using such instructions to know how data is stored in the stack. We shall assume from here that the reader is reasonably familiar with the logical structure of the machine, that is with the basic order code. We shall illustrate the use of machine instructions by recoding the inner loop of the routine 'line fit' described earlier. The local data of the routine is stored as follows:



Here Bd refers to the B-line associated with the routine, and corresponds to the textual depth of the routine in the programme in which it is embedded. If (say) this is 2 then Bd = B2. The addresses 2, 3, 4, ... associated with s, n, a, etc., are assigned of course at compiler time. The coefficients s, 2 and s+I, 2 of the array functions are evaluated at run time however when these statements are encountered. All the above quantities are referred to as the 'fixed' variables. There are no local arrays involved in this routine. B90 always points to the next available location in the stack. Bd points to the old position of B90 before the routine was called in. The previous contents of Bd (if any) are stored in (Bd, $\frac{1}{2}$) and the control number link in (Bd, 0). The location (Bd, I) is used in test instructions while (Bd,3/2) holds the number of fixed variables in the routine. B90 is advanced from its original value to the value shown Bd + I7 immediately on entering the routine. The unstandardised integers are obtained by adding 0.8 to the standardised version with the add instruction 0330. Thus an unstandardised I0 is stored as $8^{12}.(10.8^{-12})$, i.e., as

03000000 00000I20 in octal form

STACK INSTRUCTIONS

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

st 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

We shall now describe some 'machine code' formats.

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

[FD],[N],[N],[±?][CONST]

(and terminated as usual by ; or newline). Here [FD] refers to the function digits, [N] to the Ba and Bm digits, and [±?][CONST] to the address part. This last is 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

0101, 80, 2, 2.5 is equivalent to
04064002 00000024

in octal notation.

The effect of this instruction in the above routine would be to set the address s in B80.

2. The format

[FD],[N],[N],[O?]

is similar to (I) but here the address part is written as an octal number. This consists of an * followed by up to 8 octal digits, including any significant zeros. Thus *0047 is equivalent to the 24 bits

00470000

in octal notation.

3. The format

[±?][CONST]

is used to plant a standardised 48-bit floating point number in the current location of the programme.

plants a pair of 24-bit words, each similar to the address part of (I), in the current location.

5. The format

[OW],[OW]

plants a pair of half words, each similar to the address part of (2).

We now have three instruction formats which use a symbolic address.

6. [FD],[N], - ,[NAME]

Here the [NAME] must refer to an integer (or addr), real, integer name, real name. In the case of an integer or real name, the resulting instruction is as follows

[FD],[N], d, p

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, -, x

appearing in the routine under discussion would be translated as

0324, 0, 2, 7

The effect would be to put x in the accumulator. In case of integer name and real name the symbolic instruction always corresponds to a pair of instructions, thus

0324, 0, -, a is translated as
0I0I, 99, 2, 4.5
0324, 0, 99, 0

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. This can be done by writing α before the [NAME]. Thus

0I0I, 80, -, αs

is equivalent to

0I0I, 80, 2, 2.5

If s were an integer name then the instruction would correspond to the pair of instructions

```
OIOI, 99, 2, 2.5
OIOI, 80, 99, 0.5
```

7. In the instruction format

```
[FD],[N],[N],[N]:
```

[N]: refers to a simple label. It is replaced by the control number corresponding to the label. We may refer to labelled constants in this way. For example

```
0334, 0, 0, I4:
        .
        .
        .
I4: *03, *00000I2
```

puts an unstandardised IO in the accumulator

8. Finally, we have a special purpose format, the use of which requires a more detailed understanding of the operation of the translated programme.

```
[FD],[N], -, [-?][N][NAME][±CONST?]
```

where again the [NAME] can be preceded by α if necessary. Here the actual address is obtained by selecting a particular address associated with the [NAME], adding to it the [±CONST] if present, and finally adding 0.5 if α is present. The particular address referred to is specified by the value of [-?][N]. It is obtained by looking up a property list which is set up by the compiler for each name quantity. The significance of the various addresses which can be obtained in this way is explained elsewhere. For an example however, we may refer to the storage layout of the routine discussed earlier, thus

```
OIOI, 80, -, IαX
```

sets the origin of the X vector in B80. Here X was defined as the array fn X(s, 2). However, in the following version of the 'line fit' routine we dispense with the array functions and work directly from the

```
routine line fit (addr s, integer n, real name a, b, c)
real x, y, xx, xy, yy
x = 0; y = 0; xx = 0; xy = 0; yy = 0

        0101, 80, -, as          load s into B80
        0101, 81, -, an          load n into B81
        0124, 81, 81, -2         form 2n - 2

2:      0324, 80, 81, 0
        0320, 0, -, x            X(i) + x --> x
        0356, 0, -, x

        0324, 80, 81, 0
        0362, 80, 81, 0
        0320, 0, -, xx           X(i)^2 + xx --> xx
        0356, 0, -, xx

        0324, 80, 81, 0
        0362, 80, 81, 1
        0320, 0, -, xy           X(i)Y(i) + xy --> xy
        0356, 0, -, xy

        0324, 80, 81, 1
        0320, 0, -, y            Y(i) + y --> y
        0356, 0, -, y

        0324, 80, 81, 1
        0362, 80, 81, 1
        0320, 0, -, yy           Y(i)^2 + yy --> yy
        0356, 0, -, yy

        0214, 127, 81, 1:        if 'B81 = 0' then -> 1
        0122, 81, 0, 2           B81 = B81 - 2
        0121, 127, 0, 2:         --> 2

1:      a = (n.xy - x.y)/(n.xx - x.x)
        b = (y - a.x)/n
        c = yy - 2(a.xy + b.y) + xx.a^2 - 2a.b.x + n.b^2
        end
```

USE OF B-LINES

The user of machine instructions must take care to avoid using those B-lines which are used by the system. These are

B1, B2, B3, ----

which are associated with routines of textual depth 1, 2, 3, etc. The embedding of routines within each other would not normally extend very far: it would be a very complicated program indeed that ran to 10 deep.

The extracode sequences use B91 - B97. B98, B99 are only used by certain extracodes but on the other hand they are used by almost every autocode instruction and indeed by certain 'machine' instructions of the type 6.

Also B60 - B79 are used by permanent and library routines. Therefore the user should confine himself to B10 - B59.

APPENDIX I 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 many phrases in which a particular component may appear an indefinite number of times (for example an integer may have any number of digits), and a special qualifier * is used in a phrase to indicate that it appears at least once, but may be repeated any number of times. Thus, having defined [DIGIT] we may define an integer and call it [N] by

PHRASE [N] = [DIGIT*]

There are also places where a phrase may or may not appear and the qualifier ? is used. For example a name in Atlas Autocode may be formally defined by

PHRASE [NAME] = [LETTER*][DIGIT*?][PRIME*?]

where PHRASE [LETTER] = a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

and PHRASE [PRIME] = '

The Atlas Autocode language is recursive and phrase definitions may use themselves. For example we may define a 'list of names separated by commas' by

PHRASE [NAME LIST] = [NAME][,][NAME LIST],[NAME]

Thus c is a [NAME LIST] since it is a [NAME],

b,c is a [NAME LIST] since b is a [NAME] and c, a [NAME LIST];

a,b,c is a [NAME LIST] since a is a [NAME] and b,c, a [NAME LIST] etc.

Note that if a comma is a component of a phrase it must appear in square brackets as above, to distinguish it from the comma which separates the alternatives of a phrase.

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

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

where PHRASE [,NAME] = [,][NAME]

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] = integer, real, complex

we can then 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 if [COND] then and unless [COND] then.

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

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

2. 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.

PHRASES AND FORMATS OF ATLAS AUTOCODE

PHRASE [EXPR] = [+?][EXPR'],[H-EXPR]

PHRASE [EXPR'] = [OPERAND][OPERATOR][EXPR'],[OPERAND]

PHRASE [OPERATOR] = +, -, *, /, ↑, .., NIL

PHRASE [OPERAND] = [NAME][APP],[CONST],([EXPR]),|[EXPR]|,i

PHRASE [APP] = ([EXPR-LIST]), NIL

PHRASE [EXPR-LIST] = [EXPR][,][EXPR-LIST],[EXPR]

PHRASE [TYPE] = integer, real, complex

PHRASE [QUERY] = ?

PHRASE [NAME LIST] = [NAME][,][NAME LIST], [NAME]

PHRASE [ARRAY LIST] = [NAME LIST]([BOUND PAIR LIST])[,][ARRAY LIST],
[NAME LIST]([BOUND PAIR LIST])

PHRASE [ARRAY FN LIST] = [NAME]([EXPR][,][EXPR-LIST])[,][ARRAY FN LIST],
[NAME]([EXPR][,][EXPR-LIST])

PHRASE [BOUND PAIR LIST]= [EXPR]:[EXPR][,][BOUND PAIR LIST],
[EXPR]:[EXPR]

PHRASE [OWN ARRAY LIST] = [NAME LIST]([CONSTANT BOUND PAIR LIST])[,][OWN ARRAY L
[NAME LIST]([CONSTANT BOUND PAIR LIST])

PHRASE [CONSTANT BOUND PAIR LIST]=[N]:[N][,][N]:[N],[N]:[N]

PHRASE [COND] = [AND-C],[OR-C]

PHRASE [AND-C] = [SC] and [AND-C],[SC]

PHRASE [OR-C] = [SC] or [OR-C],[SC]

PHRASE [SC] = [H-EXPR][COMP][H-EXPR],[EXPR][COMP][EXPR][COMP][EXPR],
[EXPR][COMP][EXPR],([COND])

PHRASE [COMP] = =, ≠, >, ≤, <, ≥

PHRASE [RT] = routine, integer map, real map, complex map, integer fn,
real fn, complex fn

PHRASE [FPP] = ([FP*]), NIL

PHRASE [FP] = [FP-DELIMITER][NAME]

PHRASE [FP-DELIMITER] = [,?] integer name, [,?] real name, [,?] complex name,
[,?] integer, [,?] real, [,?] complex, [,?] addr,
[,?][RT],[,]

PHRASE [N-LIST] = [N][,][N-LIST],[N]

PHRASE [SWITCH LIST] = [NAME LIST]([-?][N]:[-?][N])[,][SWITCH LIST],
[NAME LIST]([-?][N]:[-?][N])

PHRASE [-] = -

PHRASE [ALPHA] = α

PHRASE [H-EXPR] = [+?][H-EXPR']

ERASE [H-EXPR'] = [NEG?][H][HO][H-EXPR'],[NEG?][H]

ERASE [NEG] = (-)

ERASE [H] = [H-VAR],[CONST],[OW],([H-EXPR])

ERASE [HO] = +, *, &, ∨, ≠, -, /, >, <, (+)

ERASE [H-VAR] = β[N], α[NAME], α([H-EXPR])

ERASE [list or nest] = list, nest

ERASE [H-EXPR-LIST] = [H-EXPR][,][H-EXPR-LIST],[H-EXPR]

ERASE [+ CONST] = [+][CONST]

ERASE [C] = delete line [N], replace line [N] by, end of [CORRECTION]

ERASE [CORRECTION] = corrections, correction

RRAY LIST

MAT [UI] = [NAME][APP] = [EXPR][QUERY?]

MAT [UI] = [NAME][APP]

MAT [UI] = stop

MAT [UI] = return

MAT [UI] = -> [N]

MAT [UI] = test [N-LIST]

MAT [UI] = -> [NAME]([-?][N])

MAT [UI] = -> [NAME]([EXPR])

MAT [UI] = caption [TEXT]

MAT [UI] = [H-VAR] = [H-EXPR]

MAT [UI] = add [H-EXPR] to [list or nest][H-VAR]

MAT [UI] = add [H-EXPR-LIST] to [list or nest][H-VAR]

MAT [UI] = withdraw [H-VAR] from [list or nest][H-VAR]

MAT [UI] = add list [H-VAR] to list [H-VAR]

MAT [UI] = split list [H-VAR] at [H-VAR]

MAT [UI] = set up [list or nest] [H-VAR]

[EXPR],

nteger fn,

ex name,

ddr,

T [SS] = [UI][S]

T [SS] = if [COND] then [UI][S]

T [SS] = unless [COND] then [UI][S]

```
FORMAT [SS]  =  [UI] unless [COND][S]
FORMAT [SS]  =  cycle [NAME] = [EXPR][,][EXPR][,][EXPR][S]
FORMAT [SS]  =  repeat [S]
FORMAT [SS]  =  [TYPE][NAME LIST][S]
FORMAT [SS]  =  [TYPE?] array [ARRAY LIST][S]
FORMAT [SS]  =  [TYPE?] array fn [ARRAY FN LIST][S]
FORMAT [SS]  =  own [TYPE][NAME LIST][S]
FORMAT [SS]  =  own [TYPE?] array [OWN ARRAY LIST][S]
FORMAT [SS]  =  [RT] spec [NAME][FPP][S]
FORMAT [SS]  =  [RT][NAME][FPP][S]
FORMAT [SS]  =  begin [S]
FORMAT [SS]  =  end [S]
FORMAT [SS]  =  end of program [S]
FORMAT [SS]  =  [N]:
FORMAT [SS]  =  [N] case [COND]:
FORMAT [SS]  =  [NAME]([-?][N]):
FORMAT [SS]  =  switch [SWITCH LIST][S]
FORMAT [SS]  =  compile queries [S]
FORMAT [SS]  =  ignore queries [S]
FORMAT [SS]  =  compile jump trace [S]
FORMAT [SS]  =  compile routine trace [S]
FORMAT [SS]  =  stop jump trace  [S]
FORMAT [SS]  =  stop routine trace [S]
FORMAT [SS]  =  comment [TEXT][S]
FORMAT [SS]  =  [FD][,][N][,][N][,][+?][CONST][S]
FORMAT [SS]  =  [FD][,][N][,][N][,][OW][S]
FORMAT [SS]  =  [+?][CONST][S]
FORMAT [SS]  =  [+?][CONST][,][+?][CONST][S]
FORMAT [SS]  =  [OW][,][OW][S]
FORMAT [SS]  =  [FD][,][N][,]-[,][ALPHA?][NAME][S]
FORMAT [SS]  =  [FD][,][N][,]-[,][-?][N][ALPHA?][NAME][+ CONST?][S]
FORMAT [SS]  =  [FD][,][N][,][N][,][N]:[S]
FORMAT [SS]  =  result = [EXPR][S]
FORMAT [SS]  =  [NAME] = st [S]
FORMAT [SS]  =  st = [EXPR][S]
FORMAT [SS]  =  st = st [+][EXPR'][S]
FORMAT [SS]  =  [CORRECTION][S]
```

APPENDIX 2 LIST 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.

1. STANDARD MATHEMATICAL FUNCTIONS

Some of the functions are special cases in that the function type and the type of the formal parameters may depend on context. For example 'sq rt' exists in the following two forms

real fn spec sq rt (real x)

complex fn spec sq rt (complex z) (see Section 8)

In the following lists we shall assume that n represents a formal parameter of type integer, x of type real, and z of type complex.

a) The following functions may appear in integer expressions

int pt(x) int (x) parity (n)

b) The following, together with (a), may appear in real expressions

sin(x) cos(x) tan(x) log(x) exp(x) sq rt(x)

arc tan(x,y) ($-\pi<\theta<\pi$) radius (x,y) frac pt(x,y)

mod(z) re(z) im(z) arg(z)

c) The following, together with (a) and (b), may appear in complex expressions

sin(z) cos(z) tan(z) log(z) exp(z) sq rt(z) conj(z)

2. STORAGE FUNCTIONS

integer fn spec addr ([VAR])

integer fn spec integer (integer n)

real fn spec real (integer n)

complex fn spec complex (integer n)

The address recovery function is a special case since the formal parameter type of the argument may be integer name or real name.

The functions integer (n), real (n), complex (n) give the contents of storage location n as an integer, real, or complex number; in the last case the two parts of the complex number are assumed to be in n and n + 1.

3. INPUT / OUTPUT ROUTINES (see Section 5)

routine spec select input (integer n)
routine spec read ([VARIABLE LIST])
routine spec read symbol (integer name n)
routine spec read binary (integer name n)

routine spec select output (integer n)
routine spec print (real x, integer m,n)
routine spec print fl (real x, integer n)
routine spec print symbol (integer n)
routine spec punch binary (integer n)
routine spec space
routine spec spaces (integer n)
routine spec newline
routine spec tab

integer fn spec next symbol
routine spec skip symbol

The above routines are the basic input/output routines, but it is anticipated that more comprehensive routines will be added shortly.

4. MATRIX ROUTINES

The following basic matrix routines are available. It is assumed that the matrices are stored by rows in the conventional manner.

routine spec mat mult (addr sI,s2,s3, integer m,n,l)

This forms an m x n matrix in sI onwards, as the product of an m x l matrix in s2 onwards and an l x n matrix in s3 onwards.

routine spec mat div (addr sI,s2, integer m,n)

This replaces the m x n matrix A in sI onwards by the m x n matrix $B^{-1}A$, where B is an m x m matrix stored in s2 onwards. B is destroyed by the routine.

routine spec mat trans (addr sI,s2, integer m,n)

This forms in sI onwards the transpose of the n x m matrix stored in s2 onwards.

real fn spec det (addr sI, integer m)

This evaluates the determinant of the m x m matrix stored

5. INTEGRATION OF ORDINARY DIFFERENTIAL EQUATIONS

A permanent routine is available for the integration of systems of first order ordinary differential equations.

routine spec int step (addr s, real name x, integer n,m, real h, routine f)

Given the system of equations

$$yi' = fi(x,yI,y2,.....yn) \qquad i = I,2,.....n$$

where yi at x0 is known, the routine evaluates yi at x0 + mh by m steps of a Runge-Kutta fourth-order integration process. On entry to the routine, yi at x0 is stored in s onwards and x0 in the real name x. On exit, yi at x0 + mh are in s onwards and x0 + mh in x. The routine requires an auxiliary routine to evaluate the derivatives.

routine spec f (addr sI,s2, real x)

This must be written so that when it is entered with a value of x and values of yi in sI onwards, it evaluates the fi and places them in s2 onwards.

APPENDIX 3 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 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.

e.g. $\not\leq$ is formed from = _ /

The numerical equivalent of a compound symbol is

$a.2^{14} + b.2^{7} + c$

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

$$b.2^{7} + c \quad , \quad b > c$$

Thus $\not\leq$ is equivalent to $86.2 + 28.2 + 15$

and $\geq$ is equivalent to $86.2 + 27$

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	¤	43	K	75		107	k
12	?	44	L	76	stop	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	

APPENDIX 4. LIST OF MONITORED FAULTS

COMPILING TIME FAULTS

The following faults are monitored at compiling time.

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

NAME NOT SET

SWITCH VECTOR NOT DECLARED

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

LHS NOT A DESTINATION

ROUTINE NAME APPEARS IN EXPR

CALLS FOR ADDR OF EXPR

CALLS FOR ADDR OF FUNCTION

CALLS FOR CONTROL NO OF NON-ROUTINE NAME

SWITCH VECTOR APPEARS IN EXPR

3. Arithmetic Faults

COMPLEX QUANTITY IN EXPR	i.e. a real or integer [EXPR]
REAL QUANTITY IN EXPR	i.e. in an integer [EXPR]
REAL CONSTANT IN EXPR	"
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.
NON-INTEGRAL EXPONENT	

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

LABEL...NOT SET	There is a reference to label... which has not been set.
NO LABELS SET	No labels to which references are made have been set.
TOO FEW ENDS	begins (or routines) do not match ends
TOO FEW REPEATS	cycles do not match repeats

5. Other faults

AP FAULT	An actual parameter fault: it means

FP FAULT	A formal parameter fault; it means that the routine heading is not consistent with the routine specification.
NOT A ROUTINE NAME	A call sequence is written in which the [NAME] has not appeared in a routine specification or routine heading.
LABEL SET TWICE	
SWITCH OUT OF RANGE	
NAME SET TWICE	
ARRAY OF MORE THAN 2 DIMENSIONS	
WRONG FORMAT IN CORRECTIONS	
NAME NOT VALID	In [NAME] = st , the [NAME] must be local.
INSTR OUT OF CONTEXT	The format result = [EXPR] may only appear in function routines.

RUN TIME FAULTS

I. The following faults are monitored at run time. Normally they cause the program to be terminated but it may be restarted by a sequence labelled

fault (n):

in the top level (label I). The relevant numbers appear in the table below.

DIV OVERFLOW	division by 0 or a non-standard number	fault (I)
EXP OVERFLOW	exponent overflow	fault (I)
SQ RT OF -VE ARG		fault (2)
LOG OF -VE ARG		fault (2)
TRIG FN LOST ACCURACY	a trigonometrical function in which all accuracy has been lost.	fault (3)
INV FN OUT OF RANGE	an inverse trig function e.g. arcsin when the argument is	fault (4)

INPUT ENDED	insufficient data so that a read instruction effectively reads over the end of the data tape.	fault (5)
SPURIOUS CHARACTER IN DATA	spurious character (i.e. not a decimal digit, decimal point, sign or α) appears in data.	fault (5)

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

INPUT NOT DEFINED	
OUTPUT NOT DEFINED	
ALL TESTS FAIL	i.e. all conditions in a <u>test</u> instruction fail.
SWITCH VARIABLE TOO HIGH	
SWITCH VARIABLE TOO LOW	
SWITCH VARIABLE NOT SET	
EXPONENT NEGATIVE	i.e. in an <u>integer</u> expression.
EXPONENT NON-INTEGRAL	

3. Faults which indicate an error in the machine or compiler and should be referred to the compiler staff.
There are a number of these such as

SV INSTRUCTION
SV OPERAND
ILLEGAL BLOCK