

UNIVERSITY OF LONDON INSTITUTE OF COMPUTER SCIENCE
THE UNIVERSITY MATHEMATICAL LABORATORY, CAMBRIDGE

CPL ELEMENTARY PROGRAMMING MANUAL

Edition II (Cambridge)

(including corrections and revisions)

Corn Exchange Street,
Cambridge.

J. Buxton
J.C.Gray
D. Park

January 1966

"I wish to God that these calculations had been executed by steam."

(Charles Babbage, to John Herschel; 1820)

1. INTRODUCTION.

CPL, (Combined Programming Language) has been developed as a joint project between the University of London Institute of Computer Science and the Cambridge University Mathematical Laboratory. In concept it is an extended language intended to cope with all possible classes of program, whether numerical, non-numerical, list-processing, heuristic or clerical.

CPL is not just an extension of any previous language, (although it contains many features of ALGOL 60) but has a distinctive philosophy and logical coherence of its own (see the Advanced Programming Manual and Reference Manual for details). In particular it is intended that, save for those exceptional cases in which an extremely efficient program is required, programmers will not have to escape into machine code.

The following Elementary Programming Manual restricts itself to the central part of the language. Peripheral matters such as input-output, etc., depend on local conditions and will be dealt with in local users' manuals issued by the various establishments using CPL.

2. THE CPL ALPHABET AND BASIC SYMBOLS.

The following is a list of permitted CPL characters, which may occupy single print positions in a printed CPL program:

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

0 1 2 3 4 5 6 7 8 9

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

0 1 2 3 4 5 6 7 8 9

+ - x / = < >

() [] \ ^ ~

' ; : , . * → \$ | _

~ = † ≠ ‡ ≤ ≥ † \$ §

CPL programs are made up of these characters alone.

Certain combinations of these characters have a special significance, and should be regarded as self-contained entities: e.g. certain underlined words, the assignment operator ':=' etc. These are called BASIC SYMBOLS and are listed in Appendix 1. Underlined words may be written in upper or lower case, or a mixture of the two. Spaces in basic symbols are ignored.

Thus then may be written T h e n. The basic symbol if has nothing to do with the letters i and f, and has a completely separate meaning from the combination 'if' appearing without underlining in a program. Basic symbols are introduced as they arise in subsequent sections.

3. ITEMS, TYPES AND NAMES.

A CPL program specifies processes to be carried out on ITEMS of information. These items may be numerical (variables and constants) or non-numerical (e.g. bit-strings and character strings), and if they are numerical they may be real or complex and may be stored to more or less than the standard precision. Every item must, therefore have both a NAME and a TYPE. (Constants are an exception: they have a type but not a name.)

The most common type of number is the real number. (Note the underlining: real is a basic symbol.) A real variable is held in floating-point: (in Atlas this means that it has a range of the order

113 113
+10 to -10

with a precision of about 12 decimal digits). Other numerical types are:

<u>double</u>	A floating point number with a precision about twice that of a real number. (On Atlas, about 24 decimal digits).
<u>complex</u>	An ordered pair of <u>real</u> variables.
<u>double complex</u>	An ordered pair of <u>double</u> variables.
<u>index</u>	An integer used in subscripting operations.

A type integer can also be used; on Atlas, integer arithmetic is carried out in floating-point and the result rounded off when a value is assigned to an integer variable.

Complex and double precision working are not further described as they will not initially be implemented.

Among the non-numerical types of item on which CPL operates are:

<u>Boolean</u>	a truth value which is <u>true</u> or <u>false</u>
<u>logical</u>	a string of binary digits, of standard length. (On Atlas, 24 bits.)
<u>long logical</u>	a string of binary digits, of twice the standard length.
<u>string</u>	a character string.

All items (except constants) appearing in a program are identified by NAMES, which are of two sorts, SMALL and LARGE.

A SMALL name consists of a single lower case letter, optionally followed by one or more primes, e.g. a, b, y, y', y''.

A LARGE name consists of an upper case letter, optionally followed by a string of letters (upper and lower case) and/or digits and decimal points, optionally terminated by primes, e.g. A, Xyz, ALPHA, Beta, Sq.4''. It may not include spaces, as spaces are terminators.

Large names are entirely the programmer's affair and he can invent them to suit his own taste: they may be short alphanumeric sequences, or they may be the actual names of the quantities which they represent, or they may be mnemonics. Some large names, however, are reserved for standard functions, e.g. LShift, Mask, and if used by the programmer with another meaning the standard function will temporarily be inaccessible.

There is no restriction on the number of characters in a name. Sometimes a name can stand for different items in different parts of a program. The important concept of the SCOPE of a name, that is the region over which a name retains its meaning, is discussed under 'Block Structure'.

4. EXPRESSIONS AND ARITHMETIC EXPRESSIONS.

4.1 References to items of information, which may be either variable names or written constants, can be combined together with operators and standard library function names to form EXPRESSIONS. Each expression can be assigned a type, which depends on the types of the component variables and constants. For the moment we will consider simple arithmetic expressions, i.e. expressions involving variables and constants of numerical types real, index, complex, etc.

4.2 Arithmetic expressions are made up by combining numerical variables and constants with the arithmetic operators and ROUND brackets. Brackets may be used to any degree of complexity. The arithmetic operators are

+ - x / \uparrow

The use of the multiplication sign is optional. It is usually omitted, as multiplication is implied by juxtaposition, unless by including it the expression can be made easier to read. In particular it is desirable to include it after a large name; if it is not included the name MUST be terminated by a space. Solidus indicates floating-point division (rounded quotient and no remainder); the remainder is obtained by a standard function. Up-arrow (\uparrow) indicates exponentiation.

Normally, sufficient brackets should be included in an expression to make its meaning unambiguous. However, if brackets are omitted the priority of dealing with arithmetic operators is as follows:

first: multiplication, division and exponentiation
second: addition and subtraction.

Multiplication, division and exponentiation are of equal precedence and associate to the right; that is to say, in the case of any ambiguity they behave as if brackets were inserted so that all the closing brackets are grouped at the right.

Addition and subtraction associate to the left; that is, brackets are inserted so that opening brackets are grouped to the left. Some examples will make these rules clear:

$a + b + c$	is equivalent to	$((a+b)+c)$
a/bc	'''	$(a/(bc))$
$a/b+c$	'''	$((a/b)+c)$
$a + bc + de/f$	'''	$((a+(bc))+(d(e/f)))$

Prefixed + and - are treated as (+1) and (-1) respectively, so that $-a/b$ is equivalent to $(-1)(a/b)$, NOT $(-a)/b$.

4.3 Arithmetic constants are usually written as decimal constants.
* is used to indicate that a decimal exponent follows.
Binary exponents are not allowed.

Some examples are:

153, 10.47, 2.108*8, 27.0, 0.34*-8

4.4 An arithmetic expression may include calls to standard library functions (or to the programmer's own defined functions) in place of variables or constants. Some standard library functions are:

Sqrt[x]	the square root of x
Exp[x]	the exponential function
Log[x]	
Sin[x]	
Cos[x]	
Tan[x]	
Arctan[x]	
Mod[x]	the modulus of x
Intpt[x]	integer part of x; i.e. the integer y such that $0 \leq x-y < 1$
Rem[x,y]	the remainder of x/y; i.e. $x-y\text{Intpt}[x/y]$

Note that each function call has the form of a function name followed, in SQUARE brackets, by a list of arguments separated by commas. The written arguments are themselves arithmetic expressions, and may include further function calls.

4.5 Subscripted variables may also occur in arithmetic expressions; these have the same form as function calls, with the function name replaced by an array name. Arrays will be discussed in section 14.

4.6 Other forms of arithmetic expression are: conditional expressions (Section 11) and result expressions (Section 22). These can also be used in arithmetic expressions in any position where a variable might otherwise occur. If used in this way, they should be bracketed in such a manner as to avoid ambiguities.

5. DEFINITIONS AND COMMANDS.

A CPL program is made up of DEFINITIONS and COMMANDS. The commands specify the arithmetic and logical operations to be performed by the computer: they also control the execution of the program. The definitions provide the information that is necessary for the computer to 'understand' the commands. For example, since the programmer can invent names for variables to suit himself, the program must include definitions associating these names with specific data items. These definitions must also specify the types of variables (there is no implicit association of certain names with particular types). We shall return to the various forms of definition in section 7; for the remainder of this section we shall study the ASSIGNMENT COMMAND.

5.1 Assignment Commands.

In the previous section we have seen how to construct expressions, whose evaluation will produce some numerical value. The assignment operator ':= ' (read 'becomes') can be used to change the value of some variable to the result of such an evaluation. Thus the command

```
x := x+1
```

evaluates the current value of x, adds 1, and assigns this as the new value of x.

Typical examples of assignment commands are:

```
x := ax + by + c
POWER := VOLTS X AMPS
```

N.B. We do not use the '=' sign, as this is reserved for conditions (Boolean expressions) and definitions.

Assignments between variables of differing types are permitted, a transfer function being invoked to transform the value of the right hand side to the type of the left hand variable. This may result in a run-time error, if the right hand side has a value which is out of range, or cannot be transformed in this way.

5.2 A simple assignment command has the form

```
<variable> := <expression>
( <variable> should be read as 'some particular variable' )
```

Usually commands are written on separate lines, and the end of the command is implied by the end of the line.

However, commands may be written on the same line, separated by semi-colons, thus:

```
x := y+x; x' := y; x'' := 0
```

but this form is not recommended, since it is difficult to read.

If it is required to continue a command onto the next line, the symbol c may be used, either immediately before, or immediately after, the newline.

5.3 Multiple assignments are allowed, for example

`x,x',x'' := y+x,y,0`

The items on the left-hand side are names of variables, and those on the right-hand side can be variables, constants or expressions. The items in a multiple assignment need not all be of the same type. For example, if a,b,c are real variables, and d is a Boolean variable, the following command is valid:

`a,b,c,d := 0,0, a+b, true`

Multiple assignments are effectively carried out in parallel, so that

`x,y := y,x`

actually exchanges `x` and `y`, i.e. it is NOT treated as

`x := y; y := x.`

which would assign to both `x` and `y` the previous value of `y`.

A multiple assignment is considered as a single command, however many items are involved.

5.4 A COMPOUND COMMAND consists of a sequence of commands enclosed in SECTION BRACKETS `§.....‡`:

```
§    x := y + x
    x' := y
    x'' := 0    ‡
```

A compound command is considered as a single command.

5.5 Section Brackets.

A feature of pairs of section brackets, used in forming compound commands and blocks, is that they may be tagged:

`§2.1.....‡2.1`

thus increasing the clarity of the written program.

Moreover, section bracket pairs may be nested inside other pairs, and the insertion of the closing section bracket is performed automatically for each nested `§` which lacks a `‡`.

e.g.

```
§2.....
§2.1.....
§2.2.....
..... ‡2
```

Closing section brackets `‡2.1` and `‡2.2` are inserted automatically before `‡2`.

Any sequence of letters, digits and dots, optionally terminated by primes, may be used for a section bracket tag.

A space must NOT appear between the section bracket and its tag.

It is recommended that all section brackets, whether tagged or not, should be followed by a space, after the tag, if any.

6. BLOCK STRUCTURE.

A BLOCK in CPL is an extended form of compound command. It is defined as a command sequence, preceded by definitions, THE WHOLE ENCLOSED IN SECTION BRACKETS.

Wherever we could have a compound command in CPL we can insert a block, and thus a CPL program will usually include blocks nested inside other blocks. The importance of this concept arises from the way in which blocks determine the SCOPES of variable names: the definitions at the head of a block are valid within that block and any block it encloses, but not outside. *Variables declared within a block are said to be the Bound Variables of that block.*

The variables defined at its head are said to be LOCAL to the block; variables which are not local are said to be ~~GLOBAL~~ NON-LOCAL or FREE to the inner block. The whole program is theoretically enclosed in a block containing definitions of the standard functions Log, Exp, etc. (Labels, however, are local to the smallest surrounding routine or result expression; this is discussed more fully later).

It is important to note that, since it may be the subject of more than one definition, a particular name may not have the same meaning throughout a program.

A definition supersedes any previous definitions of the same name within the block in which it occurs.

7. DEFINITIONS.

The definitions in a CPL program must associate with every name introduced by the programmer a type, and possibly a value. The simplest form of definition is simply to define the type of an item, leaving the value to be assigned later. All definitions must start with the basic symbol let, thus:

let a be real

Simultaneous definitions are performed by the construction:

let a, b, c be real, integer, complex

Alternatively, if several items are all of the same type, we may write:

let a, b, c all be real

let x, y both be complex

As with assignment commands, several definitions may appear on the same line, separated by semicolons:

let a, b, c all be real; let x, y both be complex

7.1 Initialised Definitions.

When a name is defined by type at the head of a block this indicates that we intend to use a variable of the specified type and name in the subsequent program, but it does not assign any value to the variable. It is often convenient to assign initial values at the same time as we define variables, and this can be done as part of the definitions, for example:

```
let s, t, n, = 1, 0, 1
```

```
let Fi = 22/7
```

NOTE particularly the use of '=' , NOT ':=' when setting initial values.

The type of the defined variable is deduced by the compiler from the expression used to define it. It should be noted in this context that the compiler has a 'preferred type' and if possible it will represent items such as decimal constants in the preferred type. For example, when the preferred type is set to real,

```
let a, b, = 1, 50
```

defines two real variables.

A variable can be initialised in terms of variables defined in surrounding blocks, for example:

```
$1 let a be real  
  a := .....  
  $2 let b = 2a(a+1)  
    .....
```

When a variable is initialised in a blockhead, then the initial value is assigned to it every time the block is entered.

8. DEFINITIONS BY 'and' AND 'where'.

The full definition system in CPL gives the programmer considerable control over defining his terms. They may be defined 'sequentially' or 'simultaneously', simply or recursively, qualified by other definitions or not to an indefinite degree of complexity.

The simplest forms of definition have already been described:

let a, b be real ; let $c, d = 1, 2$

A sequence of definitions may be activated in parallel and treated as one definition if they are joined by and, as shown below:

A. §1 let $a = 5$

```
§2 let a = 10; let b = a
-----12
```

B. §1 let $a = 5$

§2 let $a = 10$ and $b = a$
§2

The scope of variables defined in definitions (non-recursive) is the body of the block in whose head they occur, and the right-hand sides of any subsequent definitions in the block head. In case A the initial value of 'b' is 10, as the scope of the newly-defined 'a' includes the definition of 'b'.

However, in case B, the initial value of 'b' is 5, since the definition of 'b' is not within the scope of the second definition of 'a'.

The where clause enables us to introduce definitions which apply only to a particular expression, command or definition. It is of particular use in qualifying initialised or function definitions (see Section 18). For example,

$$p := (axx + bx + c/x) \text{ where } x = 2a + b$$

let $p = f[3a+b]/f[3b+a]$ where $f[x] = ax^2 + bx + c/x$

let $p = F[2a + b]$ where $F[x] = G[x, b]$

A where clause qualifies the largest ^{immediately} possible preceding expression, command or definition. This is an important rule when it comes to qualifying a function definition. Thus,

let $f[x] = (1 + yy)/y$ where $y = g[x]$

is probably incorrect, since x in the where clause is not taken as the formal parameter x of f, but is a global variable of the same name. The correct version would be:

let $f[x] = ((1 + yy)/y$ where $y = g[x]$)

in which the where clause is in the body of f, and qualifies an expression. This interpretation is forced by the use of the parentheses.

9. BOOLEAN VARIABLES AND EXPRESSIONS.

9.1 Variables and expressions of type Boolean can take one of just two values when evaluated; the constants true and false.

It is convenient to regard conditions as Boolean expressions. A condition holds if and only if it has the value true when evaluated as a Boolean expression. It fails if it has the value false.

The simplest form of condition is

$\langle \text{expression} \rangle \langle \text{relation} \rangle \langle \text{expression} \rangle$

where $\langle \text{relation} \rangle$ denotes one of the following:

$= \neq > \geq < \leq \ll \gg$

'=' , ' \neq ' , are equality, inequality signs, and are interpreted in the standard manner, *when applied to the simple types mentioned in Section 3.* They are applicable to all types of expression.

'>' , ' \geq ' , '<' , ' \leq ' , have the obvious meanings, and are applicable to expressions of types real, double and index (not complex).

' \ll ' , ' \gg ' , are applicable only to real, double expressions. $a \ll b$ is interpreted as $b = b + a$, in floating point arithmetic. (On Atlas this implies that a is of order 10^{12} smaller than b , if a, b are real.)

In keeping with accepted mathematical notation, conditions may be extended; thus

$a \leq b = c \leq d$

is an acceptable condition, which holds if

$a \leq b$ $b = c$ $c \leq d$

all hold, and fails otherwise.

(Note that $(a \leq b) = (c \leq d)$ is also acceptable, but holds under completely different circumstances, when $a \leq b$, $c \leq d$ have the same truth values).

A condition can be assigned to a Boolean variable, e.g.

let x, y be real. let b be Boolean

.....

$b := x > y$

.....

9.2 The general form of Boolean expression consists of Boolean variables and conditions combined with the operators:

\sim (not)
 \wedge (and)
 \vee (or)
 $=$ (if and only if; equivalent)
 \neq (exclusive or; not equivalent)

The operators are given in descending order of precedence: the infix operators associate to the left.

The main use of Boolean variables is to record the result of a test for later or repeated use. In a conditional expression or conditional command we can write the name of a Boolean variable in place of an expression: thus if b is a Boolean variable,

if b then do C

is read as

'if b has the value true then do C '.

10. LABELS, JUMPS AND CONDITIONAL COMMANDS.

10.1 Any command can be labelled, the label being written before the command and separated from it by a colon. Any name (large or small) can be used as a label, provided that it is not at the same time being used for any other purpose. (Note that NUMERICAL LABELS ARE NOT ALLOWED, and section bracket tags are NOT command labels).

Examples of labelled commands are:

```
L1: Xyz := P + Q
```

```
SOLVE: a := 2
```

```
Loop: a := b + 5
```

The basic form of transfer command (or jump) is go to <label>, e.g.

```
go to SOLVE
```

Alternative forms for go to are goto and go to.

The scope of a label is defined as the smallest surrounding routine or result expression (section 22), so transfers may be written to labels within blocks nested deeper than the position of the transfer command; thus

```
Routine R  
$1 -----  
    go to LB  
  
$2 let a be real  
    -----  
  
LB : -----  ‡1
```

The execution of a transfer which leads into new blocks is understood to cause the activation of all the definitions in the block heads through which it leads.

10.2 It is often required that a jump be conditional on some relation holding, or ceasing to hold; this facility of conditional commands, together with conditional expressions, removes to some extent the need for explicit labels in a program (and should be exploited).

10.3 The first form of conditional command is:

if b then do C

b represents a Boolean condition which may be true or false:
if it has the value true the command C is obeyed, otherwise it is omitted and the next command obeyed.
An alternative form whose meaning is obvious is:

unless b then do C

In both cases then or do are accepted as synonyms for then do.

Here are some examples of conditional commands:

if a<0 then do a:=0

if (a+b) > (c+d) then do X:=Y

unless a >> 1*-7 goto END

Note that for a conditional jump we normally write
'if b goto L', not 'if b then do goto L', (although the latter form would be correctly interpreted by the compiler) as 'then do' may be omitted when followed immediately by 'goto'.

10.4 Another form of conditional command enables us to choose one of two alternatives, depending on some condition. The basic form is:

test b then do C1 or do C2

If b has the value true command C1 is executed, otherwise command C2 is executed.

For example,

test (a+b) > (c+d) then do X:=0 or do Y:=0

Again, then or do are accepted in place of then do; or is accepted in place of or do.

10.5 We can construct multi-level conditional commands, for example

test b1 then do C1 or test b2 then do C2 or C3

If b1 is true the command C1 is obeyed, otherwise b2 is tested and command C2 or C3 is obeyed according as b2 is true or false.
This may also be written:

test b1 then C1

or test b2 then C2

or C3

as the compiler will infer in such cases that the end of the line does NOT imply the end of the command.

10.6 Sometimes it may be desired to skip a whole section of program if a certain condition holds: this is a situation in which a compound command is useful. A compound command is considered a single command, so we can have constructions like:

```

if p>5 then do $ a:=0
                  b:=c+d/e
                  f:=g/h $

```

11 CONDITIONAL EXPRESSIONS.

We have, in conditional commands, a powerful mechanism for performing conditional operations. Conditional expressions offer an alternative way.

The simplest form of a conditional expression is:-

$$b \rightarrow E1, E2$$

Here b is a Boolean condition and $E1$ and $E2$ are expressions (which may, of course, be variables or constants). If condition b has the value true, the value of the expression is $E1$, otherwise it is $E2$.

A conditional expression could be the entire right-hand side of a command, for example:-

$$a := a < 0 \rightarrow 0, a$$

This is equivalent to

```

if a < 0 then do a := 0

```

However, we can include the conditional expression in a more complicated right-hand side; in this case it must be enclosed in brackets, for example,

$$a := a + (c \neq 0 \rightarrow b/c, d)$$

Similarly, we can use it as the argument of a function (section 18), thus:

$$a := b + \text{FN}[a < b+c \rightarrow X[1], X[2]]$$

More elaborate possibilities are introduced by the fact that $E1$ and $E2$ are themselves allowed to be conditional. This permits the writing of extremely complex conditional expressions; for example,

$$b1 \rightarrow b2 \rightarrow E1, E2, b3 \rightarrow E3, E4$$

If such an expression seems unclear, its constituent sub-expressions should be bracketed. The completely bracketed expression whose effect is identical to the example above is written

$$(b1 \rightarrow (b2 \rightarrow E1, E2), (b3 \rightarrow E3, E4))$$

Conditional expressions can be applied to expressions of any type permitted in CPL. For example, with label expressions

```

go to (a < 0 → L1, a = 0 → L2, L3)

```

A conditional expression can also appear on the LEFT of an assignment command, e.g.

$$(x > 0 \rightarrow a, b) := p + q$$

Here, if $x > 0$ a is set equal to $p + q$, otherwise b is set equal to $p + q$.

The form of a transfer command is

go to < label expression >

where a label expression has as its value a command label. In the simplest case it is in fact just such a label, but it can be a label variable. Variables of type label hold as their values command labels; assignments may be made to them in the usual way. As an example of their use, consider the program

```

§ let b be Boolean and L be label
-----
L1: -----
L2: -----
   L := (b → L1, L2)
-----
go to L           †

```

In this instance, L is used as a link which may be set to hold different command labels under different conditions and may later be used in transfer commands.

More complex linking mechanisms can be set up by defining label arrays, and using variable subscripts to transfer to different labels, depending on circumstances.

13. CYCLES AND REPETITIONS

13.1 Various facilities are provided in CPL to cope with cycles and repetitions. If C is a command or compound command, and b is a Boolean condition, then the instruction

C repeat while b

causes C to be obeyed once, and to be repeated as long as b is true. Alternatively we may write

while b do C

In this case, if b is false initially, C is omitted, and control is transferred to the next command in sequence.

Varisnts on these with obvious meanings are:

C repeat until b

until b do C

If several commands are to be repeated, they MUST be enclosed in section brackets.

13.2 Modified repetition of a command, simple or compound, is done by using the for command. One form of this is

for <variable> = Step E1, E2, E3 do C

Here E1, E2, E3 are expressions or constants and C is a command.
~~For example,~~

~~for x = Step 0, 0.1, 1 do C~~

C is executed n+1 times, where n is the value of $(E3 - E1)/E2$, rounded to the nearest integer; the controlled variable takes the values E1, E1 + E2, E1 + 2E2, etc., in turn. Note that the expressions E1, E2, E3, are evaluated once and for all before the cycle is started; it is not possible for the cycle to change the increment or the end condition.

Note also that it is not necessary to write repeat after a for command. A for command has a similar structure to a BLOCK (see section 6). The controlled variable is local to the repeated command, and its TYPE is deduced by the compiler from the types of E1, E2, E3.

This means that when the repetition has finished the controlled variable ceases to exist and it is not possible to use the final value directly. If the programmer wishes an external variable to be used, the for symbol is followed by ext, thus:

for ext x = step 1, 2, 11 do

13.3 A frequent use of the step form is in specifying unit steps, thus:

```
for v = step E1,1, E2
```

This may be replaced by the form E1 to E2, thus:

```
for v = 1 to 20 do C
```

Similarly:

```
for v = 1, 2, ..., 20 do C
```

where the meaning is self-explanatory. The commas are mandatory, and at least two dots must be used.

13.4 Another form for specifying repetition is the explicit list of values, for example:

```
for X = 0, 1.7, 2.51 do C
```

As many values as desired can be included in the list: the command C is obeyed with the controlled variable taking each value in turn.

13.5 With all forms of the for command, the strict definition is that the controlled variable is set before each repetition to the next value in the control sequence, which cannot be altered from within the loop.

13.6 Any of the forms of repeated command in this section may be terminated either by a transfer to a label outside the command, or by obeying the basic command break, which effectively transfers control to the command following the smallest repeated command containing the break order.

14. ARRAYS AND INDICES.

In CPL we have arrays of any number of dimensions, that is to say, subscripted variables with any number of subscripts, though 1- and 2-dimensional arrays are probably the most likely. An array is given a name like any other variable, and must be defined at the head of a block along with the other variables used in the block. The definition must specify the dimensionality of the array, and the type of its elements, for example:

```
let A, XYZ be real 1 array , index 3 array;
```

The symbols vector and matrix are synonymous with 1 array and 2 array respectively. (Note that 1 array is NOT hyphenated). However, it does NOT follow that variables defined in this way obey the rules of matrix algebra. With a few exceptions (detailed later) all array operations must be carried out on the individual elements as in the example at the end of this section.

It is also necessary to set the range of subscripts. The way in which this is done is described in the next section.

An element of an array is referred to by writing the name, followed by the subscripts in SQUARE brackets, separated by commas, thus:

```
A [10] , XYZ [i,j,k]
```

The subscripts can be expressions if required, for example:

```
XYZ[i(i+1),j(j+1),k(k+1)]
```

It is sufficient to use real or integer variables in these expressions, but index variables may always be used. The use of index variables in subscripts sometimes speeds up a program.

As an example in the use of arrays, suppose we have three two-dimensional square arrays A, B, C, whose subscripts go from 1 to n, then the following program sets C equal to the matrix product AB:

```
for i = 1 to n do
  §1.1 for j = 1 to n do
    §1.2 let a = 0
      for k = 1 to n do
        §1.3 a := a + A[i,k] B[k,j] ‡1.3
      C[i,j] := a ‡1.1
```

Note that the section brackets tagged 1.3 are included for purposes of clarity only; the variable a is local to the block with tag 1.2 .

15. ARRAY INITIALISATION.

An array must be initialised before its elements can be used in any way. This can be done by an initialised definition of the array, e.g.

```
let A = B
```

with B already initialised: or by defining the array by type, as in section 14 above, and then assigning to it:

```
let A be real 1 array
```

```
A := B
```

Before initialisation, an array does not possess any elements.

The function Newarray can be used to obtain an array of the required type, dimensionality and subscript range, as shown in the following examples:

```
let A = Newarray [real, (1, 10)]  
let B = Newarray [integer, (-4, 4), (-1, 5)]  
let C = Newarray [real, (1, n), (1, n), (1, n)]
```

A is a one-dimensional array of real elements, with subscripts running from 1 to 10.

B is a 9 by 7 rectangular array of integer elements: the first subscript runs from -4 to +4 and the second from -1 to +5.

C is a dynamic array, that is to say, its dimensions depend on some previously computed quantity, and may be different on the several occasions on which the relevant block is entered. As in any other initialised definition, the array bounds may be expressions involving variables global to the block.

The elements of the array produced by a call to Newarray are not initialised in any way.

If it is required to specify the values of the elements when the array is initialised, the function Formarray can be used: e.g.

```
let M = Formarray[ real, (1,2),(1,2)][ 8,10,12,-16]
```

This definition both defines M as a 2 by 2 array and also initialises the values from the second argument list; i.e.

```
M[1,1] = 8      M[1,2] = 10  
M[2,1] = 12     M[2,2] = -16
```

16. ARRAY EXPRESSIONS.

Arrays are regarded as being variables in their own right; the dimensionality and the type of their elements is fixed on definition, but the bounds may be changed by commands. They may be defined by type only, as in

```
let Work, Place be real 3 array, real 3 array
```

or they may be initialised thus

```
let Work = Newarray [real , (1, 10),(1,10),(1,10)]
```

The right hand side of the initialised definition is an expression of the relevant type; in this case, an array expression. Such an expression consists of either an array name or a function call which produces an array or space for an array.

Array assignment commands may be written thus

```
Place := Work  
Work := Newarray [real, (1,5),(1,5),(1,5)]
```

By the use of such commands the bounds of an array may be changed during operation of the program at any stage. When a command such as the first example above is obeyed, the value of the right hand side is taken, ~~in this case an element-by-element copy of the array 'Work', and this new copy is assigned to the array variable 'Place'.~~ *in this case an indication of the storage area reserved for the array 'Work'.* *Note that after the assignment, this storage area is shared between 'Place' and 'Work'.*

NOTE the distinction between an array expression whose value is an array, and a reference to an array element whose value is a data item, for example, a real number.

If a programmer wishes to copy an array, he can do so using the basic function 'Copy'. The effect of:-

```
B := Copy [A]
```

is to copy the array A and assign the copy to B.

17. FUNCTIONS AND ROUTINES.

The concepts of FUNCTION and ROUTINE are of central importance in CPL. Both are self-contained subsections of the program, written in terms of dummy variables (or FORMAL PARAMETERS); they may therefore be called at different places in the same program, usually with different sets of values for their arguments. A routine is essentially a COMMAND (which can of course be compound, and include assignment commands), which is obeyed.

A function on the other hand is an EXPRESSION, the evaluation of which produces a RESULT.

Both functions and routines are treated as entities in their own right and have names. The type of a function includes the type of its result (e.g. real function).

A FUNCTION or ROUTINE CALL is written in the form of the function or routine name, followed in SQUARE BRACKETS by a list of expressions separated by commas (ACTUAL PARAMETERS).

When a function call is encountered as an expression to be evaluated, the formal parameters take as their values the values of the corresponding actual parameters. The result of evaluating the expression defining the function, with the formal parameters taking these values, is the value of the function call.

Similarly, when a routine call is encountered as a command to be obeyed, the command (usually compound) defining the routine is executed with the formal parameters taking the values of the corresponding actual parameters. (Routines may call their parameters by reference, in which case an 'address' is handed over: see section 20.2)

It should be emphasised that each function call is an expression and is defined by an expression, whereas a routine call is a command and is defined by a command.

It is possible to define PARAMETERLESS functions and routines, which have no formal parameters. Calls to such functions and routines are written using the function or routine name, followed by a pair of square brackets, thus:

```
a := Function1[]  
Routine1[]
```

The square brackets are mandatory for function calls, but may be omitted in routine calls.

18. FUNCTIONS.

A function is a complicated rule for specifying a value: let us take a specific example. Suppose we wish to use the symbol F to stand for the function defined by

$$F(x) = 3x^2 + 4x + 1$$

At the head of some appropriate block, when we wish to define it along with the other definitions, we write a FUNCTION DEFINITION

```
let F[x] = 3x^2 + 4x + 1
```

x is a dummy variable, called a FORMAL PARAMETER: when we wish to evaluate the function, within the block in which it is defined, we write a FUNCTION CALL with the desired argument as an ACTUAL PARAMETER.

If the arguments of a function are of any other type than the preferred type of the compiler then this must be indicated in the definitions: e.g.

```
let P[matrix Alpha, index n] = Alpha[n,n]
```

```
let Q[ index i,j] = i(i+1) + j
```

In the second example, i, j are both taken to be index.

The type of the result is deduced from the definition by the compiler. If at the function call the actual parameters of a function do not correspond in type to the formal parameters, transfer functions are inserted automatically.

For example, if we have

```
$ let a, b be real;  
  let k be index  
  let Q [index i, j] = i(i + 1) + j  
  .....  
  .....  
  a ::= Q[b, k]  
  ..... $
```

then b will be converted to type index before the function Q is evaluated.

The definition of a function is in terms of an expression. By using a result expression (section 22) as the expression, the function may be effectively defined in terms of a command sequence.

Note that function calls, being expressions, can occur anywhere that a simple expression might. Thus:

```
a := Function1[ Function2[ a,b],c]
```

is a legal assignment command, provided that the number of arguments and types of the results of Function1 and Function2 are correct.

'See pages B2, B3 at back of Manual'

'See pages B2, B3 at back of Manual'

20. ROUTINES

20.1 A function call is a notational device for abbreviating an expression. In the same way we need a notational device for abbreviating a compound command. For this we use a ROUTINE. Suppose we wish at various points in a program to solve the equations:

$$\begin{aligned} ax + by &= c \\ a'x + b'y &= c' \end{aligned}$$

with a jump to a specified label if there is no solution. We give the routine a name, say LINEQ, and at the head of some block we write the ROUTINE DEFINITION as follows:

```
routine LINEQ [real a, b, c, a', b', c' ref x, y ref label L] be
ref x, y
$ let DET = ab' - a'b
  if Mod[DET] < 1*-6 go to L
  x := (cb' - c'b)/DET
  y := (ac' - a'c)/DET $
```

This will solve the equations for various values of a,b,c,a',b',c' and assign the solution to x,y.

The first two lines are the ROUTINE HEADING; the remainder is the ROUTINE BODY, and consists of a block with, in this case, one local variable DET. The routine heading gives the name of the routine and the list of FORMAL PARAMETERS; when we wish to use the routine we call it by writing the name followed by the list of ACTUAL PARAMETERS which are to be substituted for the formal parameters.

Thus the command

```
LINEQ [ 1,2,3,4,5,6,V,W,ERROR ]
```

will solve the equations

$$\begin{aligned} V + 2W &= 3 \\ 4V + 5W &= 6 \end{aligned}$$

assign the solution to V,W and send control to the label ERROR if there is no solution. The formal parameters are dummies, like the formal parameters in a function definition.

20.2 In this routine, x and y differ from the other formal parameters in that assignments are made to them. A variable to which a value is assigned corresponds to an address in the computer where that value is stored; x and y are therefore distinguished in the routine heading by the line ref x, y. ^{See page 81.} This means that they are called by REFERENCE. It has the effect that for the duration of the routine they will be regarded as 'address-like'.

The other parameters are called by VALUE; that is, their actual values will be handed over. (Parameters are assumed to be called by value unless it is explicitly stated otherwise.)

If an assignment is made in the routine body to a parameter called by value, the parameter is changed for the remainder of the routine application, but no assignment is made to the corresponding actual parameter.

Free variables of a routine are called by reference, in exactly the same way as the free variables of a '=' function.

20.3 The formal parameters of a routine may themselves be routines or functions.

The end of a routine may be indicated by the end of the command which is its body, and after obeying the command, control returns to the command following the routine call. It may be convenient for the dynamic end of the routine not to be at this point: for this purpose there is a built-in command return, which causes a return to the command following the routine call, e.g.

```
.....
if b then return
.....
```

Note that return is a command, so that we write then return, NOTthen go to return.

21. EXAMPLES OF ROUTINES.

a) routine Scalarproduct [^{ret}real x, ^{val}vector A, B, index n, label L] *be*
~~ref x~~
 $\$$ x := 0
 for i = 1 to n do x := x + A[i] B[i]
 if x << 1 go to L \$

The routine call

Scalarproduct [X, CAT, DOG, 10, ORTH]

will set X equal to the scalar product of two vectors CAT and DOG, each of which has ten elements subscripted from 1 to 10. If the vectors are orthogonal control goes to the command labelled ORTH.

b) routine Gaussquad [^{ret}real a, b, I, ^{val}function f]
~~ref I~~
 $\$$ let s = (b - a)
 I := s (.27778 f[a + .11270s] +
 .44444 f[a + .50000s] +
 .27778 f[a + .88730s])
 \$

This routine sets $I = \int_a^b f(x) dx$, using a Gaussian 3-point formula.

22 RESULT EXPRESSIONS.

Throughout CPL there is a sharp distinction between commands and expressions. Value of is a construction which allows us to obey several commands, which perform some calculation, and treat the result as an expression, to be incorporated in a larger expression. This is particularly useful in function definitions; the form of a function definition requires the body to be an expression and it may well happen that it requires several commands to evaluate the function. For example, suppose we wish to define the function:

$$f(x) = \sum_{n=0}^{10} c_n x^n$$

Let us suppose that the coefficients are available as an array A with subscripts running from 0 to 10. Then given x, the series is evaluated by the following block:

```

$ let Sum = 0
  for i = step 10, -1, 0 do
    Sum := x Sum + A[i] †

```

We have here a block which sets the local variable Sum to the required value. To convert this to an expression we precede it by value of and insert at the end of the command result is Sum. The function definition thus becomes:

```

f[x] = value of $ let Sum = 0
  for i = Step [10, -1, 0] do
    Sum := x Sum + A[i]
  result is Sum †

```

Although we have used a function definition as an example value of can be used anywhere to convert a compound command or block into an expression, which can then be used wherever any other expression could be used.

A result expression may include more than one instance of the command form result is, and the first such command met during execution causes termination of the result expression.

'See pages B2, B3 at back of Manual'

24. RECURSION.

A recursive function or routine is one which explicitly calls itself. Thus

$$f[x] = (x=0 \rightarrow 1, xf[x-1])$$

is a recursive function, if 'f' on the right hand side is interpreted as the function under definition; it computes $x!$, the factorial function. Special facilities must be provided for the definition of recursive functions, since apparently the rule which determines the scope of 'f' would be violated if the necessary interpretation were made. (In non-recursive function definitions, any functions occurring on the right hand side of the definition must have been previously defined).

If we wish to define a recursive function or routine, this must be indicated by preceding the definition with the symbol rec. As an example of this technique, take the Euclidean algorithm for finding the HCF of two integers:

```
let rec HCF [integer n, m] =
    (m > n → HCF m, n],
    m = 0 → n,
    HCF [m, Rem[n, m]])
```

Recursion is only meaningful in the case of functions and routines. For example:

```
let rec f[x] = (x < q → x, f[x - a])

let rec routine R [x,y] be
    § -----
    R [x + 1, a - x]
    ----- §
```

Here, we have defined a recursive function and a recursive routine. As another example:

```
let rec §1 routine R [x] be
    §2 -----
    R [a + x] ; S [a - x]
    ----- |2
    and routine S [y]
    §2 -----
    R [y - b] ; S [y + a]
    ----- |2 |1
```

NOTE the use of section brackets in the last example to force treatment of the two definitions as a single one in order to specify mutually recursive routines.

If the symbol rec is omitted from the definition of a function, the occurrences of that function name in its own definition are taken as referring to a global variable of that name, and not to the function being defined.

25. LOGICAL VARIABLES AND EXPRESSIONS.

25.1 A variable of type logical is a string of bits, of some standard length (24 in Atlas); each bit is processed independently. A variable of type long logical is a string with twice the standard number of bits, also processed independently of each other.

In the remainder of this section we talk about logical variables; everything that is said applies to long logical variables, the only difference being that in general, operations on a logical are faster than the corresponding operations on a long logical.

25.2 Operations on logical variables.

Logical variables provide the means whereby most non-numerical work is carried out in CPL, and it is therefore necessary to have more complicated operations than those so far described. For this purpose there is provided a basic set of built-in functions, in terms of which the more complex operations can be programmed. It is necessary first to define a convention for the numbering of the digits in a logical variable, which is done by numbering the digits upwards from the right hand end starting from 0.

Unless otherwise stated, the functions described below operate on logical or long logical variables, and produce a result of the same type as the logical operand. We use logical without underlining when we do not wish to distinguish between logical and long logical variables.

25.3 Functions for logical operations.

(a) Shifts.

LShift [p, j]

RShift [p, j]

Rotate [p, j]

p is a logical variable, and j is an index variable which defines the number of places shifted. LShift and RShift are logical left and right shifts; Rotate is a circular left shift. In all cases if j is negative the direction of the shift is reversed, so that

LShift [p, j]

is equivalent to

Rshift [p, -j]

With LShift and RShift the bits moved in to fill the gaps are zeros.

(b) Masking operations.

Ones [j, k]

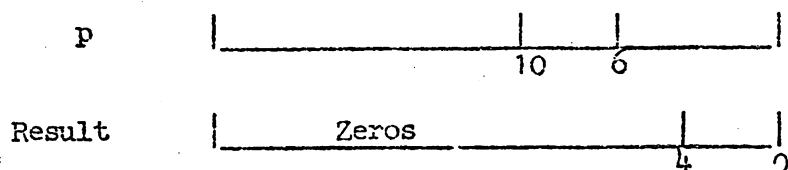
Here, j and k are index variables. Ones [j, k] produces a mask in which bits j to k inclusive are ones and all other bits are zeros; the order in which j, k are written is immaterial, i.e. Ones[j, k] = Ones[k, j].

(c) Other bit-manipulations.

Field [p,j,k]

Bit [p,j]

p is a logical variable, j and k are index variables; as usual, the order of k and j does not matter. The function Field [p,j,k] masks off bits j to k inclusive, and then right justifies the group, so that if j > k, bit k of the argument becomes bit 0 of the result, and bit j becomes bit (j-k). For example, the effect of Field [p,6,10] is shown in the diagram.



Bit [p,j] is equivalent to Field [p,j,j]; it has the effect of specifying and right justifying a single bit. The functions Bit and Field may be used on the left-hand side of assignment commands; their results, therefore, are effectively the 'addresses' of the bit or area specified.

25.4 Logical Constants.

Logical constants can be written in binary or octal, being preceded by the symbols 2 or 8 respectively. (As 7=111 in binary, an octal string is equivalent to a binary string grouped in threes. Thus, 2 010111011 = 8 273.) They are normally assumed to be positioned at the least significant end of a logical variable: thus 8 77 is understood to mean 8 00000077. However, positioning at the more significant end can be indicated by a bar: in this case zeros at the less significant end can be omitted, and 8 |273 is understood to mean 8 27300000 (assuming, in this case, that 24 is the standard length).

Logical variables can be combined to form logical expressions using the same operators as for Boolean expressions, viz. \sim , \wedge , \vee , $=$, \neq .

(The basic logical operations on bits are :

$1 \wedge 1 = 1$	$1 \vee 1 = 1$	$\sim 1 = 0$
$1 \wedge 0 = 0$	$1 \vee 0 = 1$	$\sim 0 = 1$
$0 \wedge 0 = 0$	$0 \vee 0 = 0$	
$1 = 1 = 1$	$1 \neq 1 = 0$	
$1 = 0 = 0$	$1 \neq 0 = 1$	
$0 = 0 = 1$	$0 \neq 0 = 0$	

The specified operation is carried out on all bits independently. Thus if a, b, c, are logical variables,

a := c \wedge 8|77

masks off the top six bits of c and sets a equal to this, while

c := (c \wedge \sim b) \vee (b \wedge a)

replaces the field in c specified by the ones in b, by the corresponding field of a.)

26. STRING EXPRESSIONS.

A string variable is a string of characters with possible local limits on maximum length. The characters must be those of the CPL alphabet.

A string constant consists of the characters of the string enclosed in STRING QUOTES, '.....'; for example,

'this is a string' '123/AB/6'

with the exceptions that the characters

have a special significance; also occurrences of || (double vertical bar) within string constants initiate comments (section 27), which are ignored.

A special mechanism is required for specifying as part of a string a character, other than a space, which does not print, or cannot be represented as a CPL character. For these the ASTERISK is given a special significance. Wherever an asterisk occurs in a string constant it is interpreted together with the following character according to some local convention concerning such characters, the precise nature of which depends on the machine and output devices used.

For Atlas Flexowriters, these conventions are that

*:	stands for	*
*	stands for	
*'	stands for	'
*N and *n	stand for	newline
*S and *s	stand for	space
*T and *t	stand for	tab
*B and *b	stand for	backspace
*E and *e	stand for	erase
*Z and *z	stand for	stopcode
*U and *u	stand for	upper case
*L and *l	stand for	lower case

For example, the string constant

'123*n456**'

is a representation of

123
456*

One infix operator may be used in forming string expressions, the concatenating operator <=>. Other operations on strings are carried out by a set of basic functions, which are described in section 26.2 below.

Relational expressions may be written using strings and the operators

\leq $<$ $=$ \neq $>$ \geq

These have the same form as arithmetic relational expressions and they form a subset of Boolean expressions.

A longer string is 'greater' in relational expressions than a shorter string identical to its starting characters; thus

'ATLAS' > 'AT'

is true. The relations between CPL characters which determine the precise lexicographic ordering on strings is subject to local convention, and may be altered to suit individual programs.

26.2 String manipulation functions.

In the following descriptions, s is a string constant and i is an index parameter. Unless otherwise stated, the result is of type string.

Length [s]

The result is of type index and is the number of characters in s.

First [s]

Last [s]

The result is the first or last character of s, respectively.

Character [i,s]

The result is the i-th character of s.

Initials [i,s]

Finals [i,s]

The result is the first or last i characters of s, respectively.

27. COMMENTS.

It is sometimes required to include in a program explanatory notes which are intended for the human reader only, and which must be ignored by the computer when reading the program. In CPL such comments are introduced by two vertical bars and continue to the end of that line; e.g.

```
|| x is the mean value
```

```
|| if p is zero this is dealt with in §1.2
```

28. COMPLETE PROGRAM LAYOUT.

A complete CPL program consists, basically, of a sequence of commands. It will usually begin with the programmer's definitions; the program is to be thought of as embedded in a global system block which contains all built-in definitions.

The basic command Finish may be used anywhere in the program and terminates the execution of that program. It would normally occur as the final command in a program, and if it is not present the compiler will insert it.

More information on program layout and preparation is given in the local operating manuals.

APPENDIX 1

A SHORT LIST OF CPL BASIC SYMBOLS WITH THEIR SYNONYMS AND ABBREVIATIONS.

Basic symbols may be written in upper or lower case, or a mixture of the two.

Spaces in basic symbols, whether underlined or not, are ignored. (e.g. go to may be written go to or goto.)

Basic symbols are listed under the section number in which they first appear.

Section 1

<u>real</u>	
<u>index</u>	
<u>integer</u>	
<u>double</u>	not initially implemented
<u>complex</u>	"
<u>double complex</u>	"
<u>i</u>	"

Section 4

+
-
x
/
|
*
)
(

Section 5

:=
c
s
t %

Section 7

let
=
be
all be both be

Section 8

and
where
are all are both are

Section 9

Boolean
true
false

=
+
>
<
>
<
>
<
>
<
=
+
>
<
=
+
+

Section 11

<u>go to</u>	<u>go</u>	<u>jump to</u>	
<u>if</u>			
<u>then</u>	<u>then do</u>	<u>do</u>	
<u>unless</u>			
<u>test</u>			
<u>or</u>	<u>or do</u>	<u>else</u>	<u>otherwise</u>
<u>→</u>			
<u>,</u>			

Section 12

label

Section 13

repeat
while
until
step
ext
for
break

Section 14

array
vector
matrix
[
]

In the following sections, abbreviations followed by an asterisk may be followed by a fullstop, possibly underlined.
e.g. routine rt rt. rt.

Section 18

function fn *
= [definition by twobars]
= [definition by threebars]

Section 20

<u>routine</u>	<u>rt</u> *
<u>reference</u>	<u>ref</u> *
<u>value</u>	<u>val</u> *
<u>return</u>	

Section 22

<u>value of</u>	<u>val of</u>
<u>result is</u>	

Section 23

~ [initialisation by reference]
= [initialisation by value]

Section 24

recursive rec *

Section 25

logical
long logical

2

8

| [justify symbol]

Section 26

string

: [string quote]
* [escape character]
<=>

Section 28

|| [comment]

Section 29

finish

P8. Insert after line 9:
'Variables declared at the head of a block are said to be
the Bound Variables of that block.'

P11. Line 11
'...in the standard manner when applied to the simple types mentioned
in section 3. (For equality between functions, routines, labels, etc.,
see the Advanced Programming Manual.)'

P16. Delete lines -15 to -16

P20. Replace lines -6 to -4 by:
'...side is taken, in this case an indication of the storage area
reserved for the array 'Work'. Note that after the assignment, this
array storage area is shared between 'Place' and 'Work'; and assignments
to array elements of the one will assign to array elements of the
other.'

Insert at bottom:

'If a programmer wishes to copy an array, this can be done via the
basic function Copy. Thus the effect of:

B := Copy[A]

is to copy the array A and assign the copy to B.'

P27 Line -10
' for i = step 10, -1, 0 do '

P A3. Delete lines 1 to 3

Line 5	<u>function</u>	<u>fn</u>	<u>fn.</u>	<u>fn.</u>
" 9	<u>routine</u>	<u>rt</u>	<u>rt.</u>	<u>rt.</u>
" 10	<u>reference</u>	<u>ref</u>	<u>ref.</u>	<u>ref.</u>
" 11	<u>value</u>	<u>val</u>	<u>val.</u>	<u>val.</u>
" 19	<u>recursive</u>	<u>rec</u>	<u>rec.</u>	<u>rec.</u>

P25. NOTE: the form for a routine definition is now:

let rt R[<formal parameter list>] be \$......\$

and the formal parameter list now has the form

[ref real a, val index b] with the following rules:

The TYPE of a formal parameter is the nearest type specified to the left.
If no type is specified it is preferred type.

Similarly, the MODE is the nearest mode specified to the left: if no mode is
specified, it is val.

Thus the routine LINEQ now looks like:

let rt LINEQ[a,b,c,a',b',c',ref x,y, label L] be \$......\$

NOTE also that, as formal parameters of functions can now be called by
reference, the parameter lists for functions and routines are now identical.

The following section replaces sections 19 and 23.

INITIALISATION BY VALUE AND BY REFERENCE.

An initialised definition, as in section 7.1, associates a name with an initial value. There are in fact two modes of initialisation, by VALUE and by REFERENCE. These are written with a '=' sign and a '~' sign respectively.

An initialisation by VALUE causes the variable concerned to be associated with a fresh storage location, whose initial contents are given by the right hand side of the definition.

An initialisation by REFERENCE causes the variable concerned to be associated with a storage location which is specified in terms of the storage locations already associated with other variables. This storage location is specified by the expression on the right hand side of the definition. Some simple examples of expressions which may be taken as specifying storage locations are:

a

A[i]

(b → a, A[i])

Some basic functions which may be interpreted as producing storage locations are given in Section 25.3(c). (It is also possible for the programmer to define such functions himself: see the Advanced Programming Manual).

If the expression on the right hand side does not specify a storage location, the variable concerned is associated with a constant, whose value is taken as the current value of the right hand side of the definition. Future assignments to that variable will be construed as errors.

For example, after:

```
$ let a = 1
  let b ~ a + 1
  .....
```

'b' has the constant value 2. An assignment b := 0 is then in error.

A variable initialised by reference shares its value with the variable or expression on the right hand side, and an assignment to either expression has the effect of changing both.

Consider: \$ let a = 1

let b ~ a

let c = a

b := 2 \$

Final values of a, b, c are 2, 2, 1. b and a share the new assignment: c is fixed at the old value of a, namely 1.

The final values had the assignment been a := 3 would be 3,3,1.

Similarly, consider:

\$ let A be real vector and i be index

let i = 3 ; let A[3] = 10

let a = A[i]

and b ~ A[i]

A[3] := 11 \$

With subscripted variables, the subscript is evaluated in both cases and fixed at i = 3: but the value of the element in b ~ A[i] may change. After the assignment, the values of a and b are 10 and 11 respectively.

BOUND VARIABLES, FREE VARIABLES and FORMAL PARAMETERS.

The names that occur on the right hand side of a function definition may be classified as occurrences of the BOUND VARIABLES, FORMAL PARAMETERS, and FREE VARIABLES of the function.

A BOUND VARIABLE is an occurrence of a name in a context in which it is subject to a definition within the function body: this can happen either in an expression qualified by a where clause, or within a result expression (see section 22).

A FORMAL PARAMETER is an occurrence on the right hand side of the function definition of one of the names in the parameter list on the left hand side of the definition, in a context in which it is not a bound variable of the function (i.e. not redefined within the function body.)

A FREE VARIABLE occurrence is any occurrence of a name which does not fall into either of the other two categories. Free variables must be meaningful within the function definition: that is, either the function definition must be within the scope of some definition of the free variables concerned, or those names must be formal parameters of some enclosing function or routine definition or be names of library functions.

Thus, in:

```
$ let a = 2 and b = 3
```

```
let x = aa
```

```
let g[y] = axx+by , $
```

g[y] has three free variables, a, b, and x

We have seen that variables may be initialised by value or by reference. Similarly, parameters in a function or routine may be CALLED BY VALUE or CALLED BY REFERENCE. The mode in which the formal parameters are called is specified in the function or routine heading (see the Note for P25.) The mode in which the free variables of a function are called depends upon whether it is a function defined by twobars (=) or by threebars (≡).

There is a very close analogy between the formal and actual parameters of a function or routine, and the left and right hand sides of an initialised definition. For example, consider:

```
let rt R[x] be $ x := 1 $
```

```
let a = 3
```

```
R[a]
```

.....

The formal parameter, x, may be called either by value or by reference.

If we have R[val x] then after calling R[a], a still has the value 3. The analogy would be:

```
let a = 3  
let x = a
```

```
x := 1
```

which obviously does not affect the value of a.

If we had R[ref x] on the other hand, this would be analogous to:

```
let a = 3  
let x ~ a
```

```
x := 1
```

and as explained above the final value of a will be 1, shared with x.

This may be summarised as:

Assignments to formal parameters called by value do NOT result in assignments to the corresponding actual parameters.
Assignments to formal parameters called by reference DO result in assignments to the corresponding actual parameters.

Assignments to a parameter called by value in any function or routine evaluation has the expected effect of changing the value of that parameter for the rest of the evaluation. In the case of functions this can only be done through the use of a result expression.

The free variables of a function may be called in either mode, depending on whether the definition of the function was by twobars or by threebars. In a twobar (=) function the variables are called by VALUE: that is, the current values of the free variables at DEFINITION time are copied, and during the evaluation of the function the free variables are taken as referring to these private copies and not to the global variables of the same names. Assignments may NOT be made to the free variables of a twobar function from within the body of the function.

In a threebar (=) function the free variables are called by REFERENCE, and take the current values of the variables with the same names at the time of EVALUATION of the function, which may well be different from the values that they had at definition time. Assignments to free variables called by reference can be made from within the function evaluation, via a result expression.

The following example illustrates the difference between the two modes of calling free variables:

```
$ let a,b,c be real
    a,b,c := 2,3,4

let w,z be real
let f[x] = axx + bx + c
let g[x] = axx + bx + c
    a,b,c := 5,6,7

L1:  w,z := f[w], g[z]      $
```

When the command labelled L1 is obeyed, w is set equal to $(5ww + 6w + 7)$

z is set equal to $(2zz + 3z + 4)$.

p.10 line -10 Replace "longest possible" by "longest immediately".

line -5 Replace "is" by "as"

For an account of function definitions, see section 18.

The first example here illustrates a common misuse of where clauses.

Note that y is to be initialised before the function f is defined, so that x in the definition of y must have been defined at that time and cannot depend on future values of the parameter of f.

The first example would be acceptable if it occurred in a context within the scope of some definition of x, e.g.

§ let x = 1

let f[x] = (1 + yy)/y where y = g[x]

. §

f would then be defined as the function with the constant value (1 + yy)/y for all parameter values, where y is obtained by evaluating g[1]. This may not be what the programmer intends.

p.12 line -14 "let routine R be"

p.16 Note: " := ", not " = ", in for ext x := step 1, 2, 11 do

pp.19-20 It is important that an array should be thought of not as the totality of its elements but as an indication of where these elements are to be found, i.e. as a 'pointer' to the relevant element storage area. The effect of an array assignment or initialisation by value is to assign a 'pointer' and not to copy the array elements; any such copying is to be done by a call to the function 'Copy'.

p.26 Example (a) should now start:

"let routine Scalarproduct [ref real x, val vector A,B, index n, label L] be
§ x:=0" etc.

Example (b) should start:

"let routine Gaussquad [real a,b, ref I, val function f] be
§ let s=(b-a)" etc.

p.29 lines -11)

-15) Insert "be" in all routine definitions.
-21)

B4 line -9 § let a,b,c all be real

line -7 § let w,z both be real

line -3 L1: w,z:= f[w], g[z] § §

CPL PRE-PILOT SYSTEM

Supplement to the CPL Elementary Programming Manual (January 1966)

This supplement gives the facilities available in the trial CPL system now in service.

Language Implementation

The following notes give the extent to which features of the language have been implemented, with references to the relevant sections of the Manual.

Section

3 - The data types available are:

real, index, integer, logical, longlogical, string, boolean

4.4 - All these arithmetic functions are available, together with:

Frpt[x] = x - Intpt[x]
Arcsin[x]
Arccos[x]

All these functions, except Rem, take a real argument and give a real result.

Together with Rem, we have:

Quot[x,y] = Sign[x/y]Intpt[Mod[x/y]] where Sign[u] = $u \geq 0 \rightarrow 1, -1$

The definition of Rem is

Rem[x,y] = x-y Quot[x,y], which is not quite as given in the Manual.

Rem and Quot take real or index arguments; if both are index, so is the result; otherwise the result is real.

13.2 - The form of the for ext command is

for ext <expression> :=;

note that there is an assignment operator in it, not an equals.

14 - The dimension number in an array type (e.g. real 4 array) must be a single digit, not 0.

15 - Vector and Matrix are synonyms of Formarray.

17 to 21 - Note the changes on pages B1 to B3, particularly concerning routine definitions and formal parameter lists.

20.3 - This applies to functions as well. Formal parameter functions and routines are not yet correctly dealt with, and care is needed in using them. The following should be noted:

- (a) All functions and routines which are ever likely to be supplied as arguments to a function or routine must call all their parameters by reference;
- (b) in calls of formal functions and routines, the arguments are called by reference, and their types are not checked.

25 - All these bit-handling functions are available (untested), both for logical and long logical variables. Note the spelling of Lshift and Rshift. We also have:

Longones[index i,j] : analogous to Ones, giving a longlogical (or LongOnes) result.
Mask[p,i,j] : result is a bit pattern of the same type as p, with bits i to j the same as bits i to j of p, and the remaining bits zero.

Bit, Mask and Field cannot appear on the left-hand side of an assignment

26.1 - Add to the list of special combinations:

*R and *r stand for Runout (upper case on Flexowriter tape, blank on teleprinter tape).
 *z and *| are not recognized.

The relational operators have not been implemented.

26.2 - These functions are available but not tested.

Input and Output

The following routines are available:

PRINTF[real x, index i] : outputs x to i sig. figs., where $i \geq 0$, in Floating decimal style; width is i+7 character positions.

PRINTD[real x, index i,j] : outputs x in 'fixed point' Decimal style, with i figures before the decimal point, j figures after; i and j must satisfy $i \geq 0$, $j \geq 0$, $i+j < 12$. Non-significant zeros are represented by spaces, and so is a plus sign. If $j = 0$ no point is printed. If $x \geq 10^i$, it is printed in floating decimal style.

WRITE[val a, b, c, ..., z] : outputs an arbitrary sequence of real, index or string arguments. Formats for numbers are:
index : 7 sig. figs., non sig. zeros represented by spaces - width 8.
real : floating decimal style, 7 figures.

READ[ref a, b, c, ..., z] : takes an arbitrary sequence of real or index arguments; reads numbers from an input stream formed according to AUTOCODE conventions, assigning them to successive arguments.

READCHAR[ref logical x] : reads the next CPL string character into x.

WRITECHAR[val logical x] : outputs the CPL string character in x.

The set of CPL string characters contains all the permitted CPL characters of Section 2, possibly overpunched with a stroke or vertical bar or both. To output a single character explicitly, one should write it as a string constant, e.g.

WRITECHAR['a']; WRITECHAR['/'].
 (Note: The original text has a typo 'Z' which has been corrected to '/' based on context.)

The same thing applies when testing for a character after reading; e.g.

READ[x] ; if x = '/' do ...

Input and output streams are referred to by string identifiers. Those

that can be declared in a JD have library string constants associated with them; the following are available:

Input : INPUT4, INPUT5, INPUT6 ;
Output : OUTPUT10, OUTPUT1, OUTPUT2, OUTPUT3 ;

these correspond in the obvious way with the streams declarable in the JD. The selection routines are SELECTINPUT and SELECTOUTPUT, which take a single argument by value. e.g. SELECTOUTPUT[OUTPUT2]

Job Descriptions

Job heading : CPL

Input Streams: Program is on Stream 3

Stream 2 is for editing commands, produced as for E2.

It need not be present.

Stream 1 must not be present.

If the INPUT section is empty, P3 FOLLOWS is assumed.

Output Streams: Streams 0 to 3 will all exist as printer streams at run time; there is no point in declaring any output.

Store : The STORE line must be omitted.

Time : Compiling rate appears to be about 1 line per second.

Execution of Object Program

Output stream 1 is selected at the beginning of execution.

At present a fixed amount of working space, 3500 words, is allocated; about 3000 of these are available for arrays.

Diagnostics

Compile-time error messages are fairly explicit.

Run time diagnostics are primitive. When a program monitors, its output will have to be brought to me for interpretation.

Simple example of a CPL job:

```
CPL
(XYZ123/TEST)
***U
WRITE['*nDEMONSTRATION PROGRAM*n']
$ let x = 1
  WRITE[2x, '*tEnd of test*n'] $
***Z
```

I.J.D. McIntyre
9 May 1965