UNIVERSITY OF LONDON INSTITUTE OF COMPUTER SCIENCE

****

THE UNIVERSITY MATHEMATICAL LABORATORY, CAMBRIDGE

****

CPL ELEMENTARY PROGRAMMING MANUAL

Edition I (London)

This document, written by the late John Buxton, was preserved by Bill Williams, formerly of London University's Atlas support team. Bill has generously made it available to Dik Leatherdale who has OCRed and otherwise transcribed it for the Web. All errors should be reported to dik@leatherdale.net.

The original appearance is respected as far as possible, but program text and narrative are distinguished by the use of different fonts. Transcriber's additions and "corrections" are in red, hyperlinks in underlined purple.

A contents list and a selection of references have been added inside the back cover.

44 Gordon Square,
London W. C. 1. 2/6d March 1965

CPL (Combined Programming Language, Cambridge Plus London or Christopher's Private Language — according to taste) was a project developed jointly by the Universities of Cambridge and London in the mid-1960s. The main participants in the definition of the language were:

David Barron	Cambridge
John Buxton	London
David Hartley	Cambridge
Eric Nixon	London
Christopher Strachey	Cambridge

Although a compiler for the London University Atlas computer was developed by George Coulouris it saw very little use. The importance of CPL, however, is that it begat a series of programming languages in widespread use over half a century later:

> Firstly, Martin Richard's BCPL (Basic CPL) a "typeless" language with a particularly rich set of control facilities derived from CPL which saw significant use in the 1970s for compiler development and similar applications.
>
> Dennis Richie took BCPL as a starting point, restoring a limited set of types to become B and then C which latter was the implementation language for Unix from the late 1960s onwards. C remains popular, particularly in the Unix community.
>
> Bjrane Stroustrup added object orientation (classes) to the C language in the 1980s to become C++ — widely used on Windows platforms.
>
> Derived from C++ are a number of languages, notably Java and C#

So CPL can be viewed as a bridge between Algol 60 and almost all programming languages used in new software development in the first decades of the 21st century.

Today CPL is little-known — a relic of long ago. But its progeny live on. That is the historical significance of the missing link which is CPL.

[DL]

1. Introduction.

CPL is a programming language which has been produced as a joint undertaking by the Institute and the University Mathematical Laboratory, Cambridge. It owes much to Algol 60 but it is intended to be wider in scope.

This description of the language assumes some knowledge of programming in languages such as one of the Autocodes or Fortran. It does not constitute a formal or complete description, but will be supplemented by a Reference Manual and an Advanced Programming Manual in due course.

The manual contains a description of the central part of CPL. Peripheral matters such as input-output systems, magnetic tape use, program preparation, the operating system, program error checking and the appendices describing the basic functions and character codes will be issued as local users manuals by the 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 3 6 7 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

+ - × / = < >

( ) [ ] ∧ ∨ ~

' ; : , . * → $ | _

≃ ≡ ≠ ≠ ≢ ≢ ≤ ≥ ⋀ $ ∉

CPL programs are made up of those characters alone.

An underlined word is called a BASIC SYMBOL, and should be regarded as a self-contained entity. It may be written in upper or lower case, or a mixture of the two. An underlined space is ignored. Thus **then** may be written **T h eN**. 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. Those 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 sometimes 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

+10^{113} to -10^{113}

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

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

A type **integer** can also be used; 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:

Boolean	a truth value which is **true** or **false**
logical	a string of binary digits, of standard length (On Atlas, 24 bits).
long logical	a string of binary digits, of twice the standard length
string	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, decimal points and primes, e.g.- **A**, **Xyz**, **ALPHA**, **Beta**, **Time**. It may not include spaces.

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 lose its standard meaning.

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 variables or 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 type **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

+ - × / ↑

The use of the multiplication sign is optional, and it is usually omitted, unless by including it the expression can be made easier to read. In particular it. is included between two, large names or between a large name and a constant or a small name. Solidus indicates floating-point division (rounded quotient and no remainder) the remainder is obtained by a standard function. Up-arrow (↑) 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 / b c`	`'''`	`(a/(bc))`
`a ⋀ b + c`	`'''`	`((a ⋀ b)+c)`
`a + b c + d e / 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 also include calls to standard library functions (or 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**
Rem[x,y]	the remainder of **x/y**; i.e. **x-yIntpt[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 expression these have the same form as function calls, with the function name replaced by an **array** name. Arrays will be introduced in Section 14

4.6 Other forms of arithmetic expression are: conditional expressions (Section 11) and **result of** 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 since there is no implicit association of certain names with particular types. Other sorts of definitions will be introduced later.

A definition must be prefixed by the word **let**. The simplest form of definition attaches a type to a list of names, e.g.

```
let a,b,c, be real
```

Definitions of this type can appear on successive lines, or they can appear on the same line, separated by semicolons:

```
let a,b,c, be real; let w,z be complex; let i,f be integer;
```

A command sequence preceded by definitions is called a BLOCK. This is an important concept, and is dealt with in section 6.

5.1 Assignment Commands.

The assignment command causes a value to be calculated and assigned to a particular variable. Examples of assignment commands are:

```
x := x + 1
POWER := VOLTS × AMPS
```

The symbol ':=' is read 'becomes'.

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 is 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. The symbol **c** is used at the end of a line to indicate that the command continues onto the next line.

5.3 Multiple assignments, are allowed, for example

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

This is not quite the same as the previous example as it counts as one assignment and the former counts as three separate assignments.

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**.

5.4 A COMPOUND COMMAND consists of a sequence of commands enclosed in section brackets §.....$

```
§.....x := y + X
      X' := y
      X'' := 0 $
```

Both this form and the multiple assignment are treated as one command by the compiler.

5.5 Section Brackets.

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

```
§2.1
```

Moreover, section bracket pairs may be nested inside other pairs, and the insertion of the closing section bracket is performed automatically for each opening § 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 label. It is recommended that each label be followed by a space. An unlabelled section bracket should also be followed by a space.

6. BLOCK STRUCTURE.

A BLOCK in CPL is an extension of the idea of the 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 fact that the names defined at the head of a block are valid within that block and any block it encloses, but not outside.

The variables defined at its head are said to be LOCAL to the block; variables defined in enclosing blocks are said to be GLOBAL 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 of** clause; this is discussed more fully later). It is important to note that a particular name may not have the same meaning throughout a program.

The use of nesting blocks economises on storage space, since space is obtained for the local variables of a block when a block is entered, and it is relinquished when the block is left.

7. INITIALISED DEFINITIONS.

When a name is included in a definition 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 Pi = 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 at the compiler has a preferred type and if possible it will represent items such as decimal constants in the preferred type. For example,

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

defines two **real** variables in an implementation with the preferred type set to ''**real**''.

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 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
---------------------   $2
```

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**'.

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. For example,

```
    p := (a x ⋀ 2 + b x + c / x) where x = 2 a + b
let p ≡ f[3 a + b] / f[3 b + a] where f[x] ≡ a x ⋀ 2 + b x + c / x
let p ≡ F [2 a + b] where F [x] ≡ G [x , y]
```

A **where** clause qualifies a single definition and it is therefore convenient to be able to group definitions into compound definitions as with commands. This is done by section brackets; one may write, for example,

```
let § f[a] ≡ x + y and g = x − y § where
        § x = 3 z ⋀ 3 and y = 3 z ⋀ -3 §
```

Note that the **let** symbol is placed outside the section brackets: also that in this case a newline does NOT act as a terminator, as the first line alone does not make sense.

9. BOOLEAN VARIABLES AND EXPRESSICNS.

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

<expression><relation><expression>

where **<relation>** denotes one of the following:

= ≠ ≠ > ≥ < ≤ << >>

'=','≠' are equality, inequality signs and are interpreted in the standard manner. They are applicable to all types of expression.

'>','≥','<', '≤' have the obvious meanings, and are applicable to expressions of types **real**, **double** and **index** (not **complex**).

'<<','>>' are applicable only to **real**, **double** expressions. **a << b** is interpreted as **a = 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≤b = c>d

is an acceptable condition, which holds if

a≤b b=c c>d

all hold, and fails otherwise.

(Note that **(a<b)=(c<d)** is also acceptable, but holds under completely different circumstances, when **a<b** , **c<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:

~	(not)
⋀	(and)
⋁	(or)
≡	(if and only if)
≠	(exclusive or)

the operators are given in descending order of precedence: the infixed operators associate from the left. Examples-are:

```
let x, y, z be real;
let p be integer
let b, b', Converging be Boolean
.........
b := x ≥ y ≥ z
b' := (x < y) ⋁ (p > 10)
Converging := b ⋁ b'
```

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 labels are NOT command labels)

Examples of labelled commands are:

```
L1:     Xyz := P + Q
SOLVE:  a := 2
Repeat: 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** and **goto** and **go to**.

The scope of a label is defined as the smallest surrounding routine or **result of** clause, (q.v.) and it is apparent that 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.

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**; indeed the purist will probably prefer. to write '**unless** ..... **do** .....' as being nearer to normal English.

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
```

(An asterisk indicates a decimal exponent).

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 Cl 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** and **or** is accepted in place of **or do**.

10.5 We. can construct multi-level conditional commands,

for example

```
test b1 then do Cl 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
```

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

```
if p > 65 then do $ a := 0
                    b := c + d / c
                    f := g ↑ b $
```

11 CONDITIONAL EXPRESSIONS.

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

The simplest form of a conditional expression is:

```
b → 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 → 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 ≠ 0 → b / c , d)
```

Similarly, we can use it as the argument of a function, thus

```
a := b + FN[a < b + c → 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 → b2 → El, E2, b3 → 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 → (b2 → El, E2), (b3 → E3, E4))
```

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

```
go to (a < 0 → Ll, a = 0 → L2 , L3)
```

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

```
(x > 0 → a , b) := p + q
```

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

12 LABEL EXPRESSIONS.

The full 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 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.

Variants 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
```

where **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. If more than one command is to be repeated, it must be cast in the form of a compound command.

A **for** 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**.

Note that 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
```

Another similar form is written :

```
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 to the command following the smallest repeated command in which it occurs.

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. However, it does NOT follow that variables defined in this way obey the rules of matrix algebra. With a few exceptions (detailed later) 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 define the type of the elements of the array, and 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 = step 1 , 1 , n do
    § 1.1 for j = step 1 , 1 , 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 **1.3** are included for purposes of clarity only: the variable **a** is local to block **1.2**

15. ARRAY DEFINITIONS.

Arrays must have their bounds specified when they are defined. This is done by setting as the initial value of the array the value of a special function, as illustrated 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 that 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. More complex uses of arrays are described in the next section.

16. ARRAY EXPRESSIONS.

Arrays are regarded as being variables in their own right; the dimensionality and the type of its elements is fixed on definition of an array, 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
```

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 now copy is assigned to the array variable '**Place**'.

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.

Another basic function is **Formarray**. This is used to obtain space for an array and to insert values into the array. It is used as follows: for example, the definition

```
let M = Formarray [real, (1 , 2) , (1 , 2)][3 , 10 , -15.4 , 3.1]
```

both defines **M** as a 2 X 2 array and also initialises its values. The second argument list to **Formarray** enumerates the elements row by row. Thus

```
M[1 , 1] = 3        M[1 , 2] = 10
M[2 , 1] = -15.4    M[2 , 2] = 3.1
```

17. FUNCTIONS AND ROUTINES.

Of central importance in CPL are the concepts of FUNCTION and ROUTINE. A FUNCTION, e.g. **f[x] = ax /\ 2 + bx + c** is a named EXPRESSION, (or rule for obtaining a single value) and on evaluation produces a RESULT, which has then to be assigned. A ROUTINE on the other hand is a complicated command sequence, which may require the evaluation of expressions but also performs the assignment of their results: after execution of a routine, the state of the machine will be altered to the extent that the values of certain variables will be changed but no new entities will have been created. (A function does produce something new namely its result.) The importance of these concepts lies in the fact that functions and routines are written in terms of dummy variables (or FORMAL PARAMETERS) and have names; they can thus be called at different places in the same program, usually with different values for their arguments. We consider functions and routines, and the way in which they may be defined, in the next two sections.

18. FUNCTIONS.

A function is a complicated rule for specifying a single value: let us take a specific example. Suppose we wish to evaluate the function

```
f[x] = 3x ⋀ 2+ 4x + 1
```

We give this a name, say **F** and at the head of some appropriate block, when we define it along with the other definitions, we write a FUNCTION DEFINITION

```
let F[x] ≡ 3x ⋀ 2 + 4x + 1
```

Note that the argument is enclosed in SQUARE BRACKETS and that an equivalence sign is used. **x** is a dummy variable, called a FORMAL PARAMETER: when we wish to evaluate the function we write a FUNCTION CALL with the desired argument as an ACTUAL PARAMETER, thus

```
y := F[z]
a := b + cF[d]
```

The actual parameter can be an expression, e.g.

```
g := F[ax + b] / F[cx + d]
```

and a function can have more than one argument, e.g.

```
let Quadform [x, y] ≡ ax ⋀ 2 + 2bxy + cy ⋀ 2
```

The functions so far defined have taken real arguments and produced real results. If the arguments are of any other type than the preferred type of the compiler then this must be indicated in the definition:

```
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 is evaluated.

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

19. FREE VARIABLES IN FUNCTION DEFINITIONS.

A function is defined by an expression, which may include two sorts of variables those which appear in the formal parameter list and those which are 'free', i.e. which are global to the function definition and are either defined in an enclosing block, or are formal parameters of some enclosing function or routine definition.

Formal parameters are always called by value, but there are two possible ways of treating free variables. In the form of function definition so far introduced, free variables are used directly when evaluating the function. However, if the function is defined using = instead of ≡ free variables are evaluated at time of definition and their values are 'frozen'. This is illustrated in the following example,

```
§1 let a , b , c be real
   a , b , c := 2 , 3 , 4
      ---------------
      §2 let w , z be real
         f[x] ≡ ax ⋀ 2 + bx + c
         g[x] = ax ⋀ 2 + bx + c
      ---------------
         a , c , b, := 5 , 6 , 7
      L1 : w , z := f[w], g[z]
      ---------------
                $1
```

When the command labelled **L1** is obeyed, **w** is set equal to **(5w ⋀ 2 + .6w + 7)** and **z** is set equal .to **2z ⋀ 2 + 3z + 4**.

Another way of expressing this difference is to say that the free variables of a '≡' function are called by REFERENCE, whereas free variables of a '=' function are called by VALUE.

20. ROUTINES.

20.1 A function in a self-contained subsection of the program which produces a single result and is essentially a complicated expression. Sometimes we require more complicated processes, possibly with several results, i.e. a complicated command: for this we use a ROUTINE Suppose we wish at various points in a program to solve the equations:

ax by = c
a'x + b'y = c

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 the routine we write the ROUTINE DEFINITION as follows:

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

  y := (ac' - a'c) / DET   $
```

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

```
LINE [P , Q , R , S , T , U , V , W , ERROR]
```

will solve. the equations

PV + QW =R

SV + TW = U

and will send control to the label **ERROR** if there is no solution. The formal parameters are therefore dummies, like the argument(s) in a function definition.

Note that a routine call is a COMMAND, whereas a function call is an EXPRESSION.

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** : 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, a local parameter of the same name is created temporarily and the original value restored when the routine is left.

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 routine 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**, NOT ....**then go to return**.

21. EXAMPLES OF ROUTINES.

a)
```
routine Scalarproduct [real x, vector A., B, index n, label L]
    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 10 elements subscripted from 1 to 10.

If the vectors are orthogonal control goes to the command labelled **ORTH**.

b)
```
routine Gaussquad [real a , b , I , 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 = f(x) dx, using a Gaussian 3-point formula,

22 'RESULT OF', EXPRESSIONS.

Throughout CPL there is a sharp distinction between commands and expressions. '**result of**' is a construction which allows us to obey several commands, which perform some calculation, and to 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) = a[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 := 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 '**result of**' and insert at the end of the command '**result is Sum**'. The function definition this becomes:

```
S[x] ≡ result of § let Sum = 0
          for i = Step [10, -1, 0] do
          Sum := x Sup + a[i]
          result is Sum $
```

Although we have used a function definition as an example **result 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 of**' 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 of** clause.

23. OTHER FORMS OF INITIALISATION.

We have already met the concept of initialised definitions. By analogy with the calling of actual parameters for routines, we can say that a definition such as

```
let x = 3A - 2
```

initialises **x** 'BY VALUE'. Two other forms of initialisation are possible, initialisation 'BY SUBSTITUTION' and 'BY REFERENCE'.

(a) Initialisation by substitution is specified by ≡ e.g.

```
let Abc ≡ a
let A ≡ b + c
```

Loosely speaking, the expression on the right hand side, enclosed, in round brackets if required, is formally substituted for the left-hand variable wherever it occurs.

Strictly, the left hand variable is to be regarded, as replaced by a call to a function without parameters,- defined by '≡'.

(b) Initialisation by reference is indicated by ≃ , e.g.

```
let a ≃ X[i , j}
```

Its main use is when the right hand variable is an array element: every reference to the left hand variable is treated as a reference to the right hand variable with any subscripts evaluated ('frozen') AT THE TIME OF DEFINITION. For example, consider the following:

```
§1 let A be real and i be index
   --------------------
   i := 3; A[3] := 10

   §2 let a = A[i]
      and b ≃ A[i]
      and c ≡ A[i]
       --------
       i := 7
      L: -------
                              $1
```

Let us determine what the values of **a , b , c** are when the command labelled **L** is reached.

a was initialised by value, and therefore has the value **10** (the value of **A[i]** with **i = 3** at the beginning of block **2**).

b was initialised by reference, and is therefore equivalent to **A[3]**, having the current value of **A[3]**.

c was initialised by substitution, and is therefore equivalent to **A[i]**. It thus has the value of **A[7]**.

24. RECURSION.

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

```
f[x] = (x = 0 → 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.

Incidentally, it should be pointed out that a function may be used recursively, without being recursive. As an example of recursive use, consider the following function which evaluates the double integral

$$\int_a^b \int_c^d f(x, y)\, dx\, dy$$

```
Doubleintegral[a , b , c , d , function f] ≡ result of
    §1 let f'[y].≡.f [x , y]
       let g[x] ≡ Gaussquad [c , d , f']
       result is Gaussquad[a , b , g] $1
```

We have assumed a previously defined function **Gaussquad** which evaluates a single definite integral. This function effectively uses itself as an auxiliary. This sort of recursive use of functions is dealt with automatically, and the programmer need not be aware that recursive use is made of such a function or routine.

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 → f[x - a])
let rec routine R x , y]
      § ---------------
         R[x = 1 , a − x]
         -------------- $

let rec routine R[x]
      §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 single in order to specify mutually recursive routines.

If the symbol **rec** is omitted from the definition of a recursive function, the occurrences of the recursive 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 **logicals** are faster than the corresponding operations on **long logicals**,

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. This 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 **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 zeroes.

(b) Masking operations.

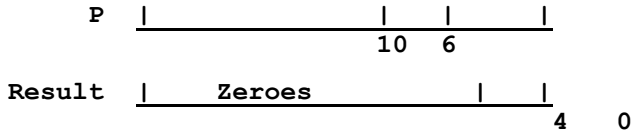
```
Ones[j , k]
```

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 zeroes; the order of j and `k` is immaterial.

(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]` the effect 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 from the left. Thus, `2010111011 = 8273`.) 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 he indicated by a bar, and in this case zeroes at the less significant end can be omitted: thus `8|273` is understood to mean `8 27300000` (assuming, in these examples, that 24 is the standard length.).

Logical variables can be combined to form logical expressions using the same operators as for Boolean expressions, i.e. ~ ∧ ∨ ≡ ≢

(The basic logical operations on bits are :

```
1 ∧ 1 = 1      1 ∨ 1 = 1      ~1 = 0
1 ∧ 0 = 0      1 ∨ 0 = 1      ~0 = 1
0 ∧ 0 = 0      0 ∨ 0 = 0
1 ≡ 1 = 1      1 ≢ 1 = 0
1 ≡ 0 = 0      1 ≢ 0 = 1
0 ≡ 0 = 1      0 ≢ 0 = 0      )
```

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

```
a := c ∧ 8|77
```

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

```
c := (c ∧~ b) ∨ (b ∧ a)
```

replaces the field in `c` specified by the ones in `b`. by the corresponding field of `s`.

26. STRING EXPRESSIONS.

26.1 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 exception that the characters

```
|      '
```

do not signify themselves.

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 this the vertical bar is given special significance: whenever a vertical bar 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 device used. (For Atlas Flexowriters, these conventions are that:

```
        ||    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 infixed 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

```
≤  < = ≠ > ≥
```

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

NOTE that 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 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. LIST PROCESSING OPERATIONS.

These are carried out in CPL by using logical variables and basic functions. NOTE that a storage word on Atlas or Titan can be split into two half-words, each of which corresponds in length to a logical variable and can hold in this length the address of any half-word. The basic functions are as follows:-

In the description **x**, **y** and **z** are **logical** variables or parameters.

```
Head[z] , Tail[z]
```

The value of **z** is assumed to be the address of a word in the store. The functions produce as their results the 'addresses' of the first and second half-words of the word whose address is given by **z**. They may be used on the LH sides of assignment commands.

```
Join[x , y]
```

This function obtains a word from the list free space area, stores in the first and second halves the values given as its parameters, and provides as the values of its result the logical pattern representing the address of this word. Note that if one performs the operation

```
z := Join[x , y]
```

then the function calls

```
Head[z] , Tail[z]
```

give access to the values obtained from x and y.

No automatic garbage disposal can be provided by the system, as it is not possible for the system to detect whether a logical variable holds the address of a list or not. The programmer is advised to return space to the free space list by the following process. The basic logical variable named 'Freespace' is defined within the system similarly to the basic functions and is used to hold the address of the first member on the free space list. To free the word whose address is held in x, obey the commands

```
Tail [x] := Freespace
Freespace := x
```

Using these basic functions, the programmer can construct and manipulate list Structures. He may. for example. write CPL functions and routines to manipulate list structures of some particular nature, such as threaded lists.

28. COMMENTS.

It is sometimes required to include in a program explanatory notes, which are intended for the human reader only, and 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
```

29. COMPLETE PROGRAM LAYOUT.

A complete CPL program consists, basically, of a sequence of commands. It will usually begin with definitions of programmers variables. 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 execution of the program. It would normally occur as the final command in a program, and if not present as the final command, the compiler will insert it.

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

JNB:MD/JML 29 March 1965.

SN: 131

CONTENTS

REFERENCES

A slightly later version of this document produced at Cambridge may be found at www.ancientgeek.org.uk/CPL/CPL_Elementary_Programming_Manual.pdf. It contains some additional material and corrections, but the appearance is poor.

In 1963 the Computer Journal published an account of CPL by the originators. *The main features of CPL* may be found here.

David Hartley's retrospective review of CPL and its progeny can be found (albeit behind a paywall) here.

A longer still (but incomplete) description of the CPL language may be found at www.ancientgeek.org.uk/CPL/CPL_Working_Papers.pdf. It makes a tantalising reference to the notion of a "compound data structure" type (equivalent to C's **`struct`** perhaps?) but gives no detail.