C. B. Jones

UNIVERSITY OF LONDON INSTITUTE OF COMPUTER SCIENCE

and

THE MATHEMATICAL LABORATORY, CAMBRIDGE

TECHNICAL REPORT

CPL WORKING PAPERS

JULY 1966

Copy No. 14

CPL WORKING PAPERS

This volume contains various papers concerning CPL. The principal one of these is an unfinished draft of the Reference Manual. This is preceded by a copy of an elementary programming manual and brief notes on the history, present status and future prospects of the language and its compilers, and is followed by various appendices.

These documents are not intended for publication in their present state. They are being given a limited private circulation in this incomplete form for reasons discussed in the preface.

July 1966.

Preface

In October 1962 a joint research project was begun by the University Mathematical Laboratory, Cambridge and the Institute of Computer Science, London (then known as the University of London Computer Unit). The aim of this project was to design and implement a new programming language for use on the Atlas I and II (Titan) computers in the two establishments.

The initial team consisted of D.W. Barron, D.F. Hartley and C. Strachey from Cambridge and J.N. Buxton and E. Nixon from London.

It was intended that the language should possess the advantages of the general structure and precise description of Algol 60 but should be of wider practical utility. Many of its main features were settled at an early stage and were described (in 1963) in a paper in the Computer Journal (ref. 1).

Since the publication of the initial description the language has been subject to extensive redesign and further research work has been carried out on its semantics (see refs. 2, 3).

The design of the language, as opposed to its implementation, has never been more than a part time occupation for any of the authors. As the language gradually approached its present state, the authors' meetings became less frequent and those who were not actively engaged in an implementation found it more difficult to keep in touch with the language. At the same time some of the people who were engaged in an implementation began making substantial contributions to the language itself. During the period 1963 to 1966 D.W. Barron left the CPL language group and D. Park and M. Richards from Cambridge and G.F. Coulouris from London joined it.

The proper description of a programming language is no easy task, and CPL, which is very considerably larger and more sophisticated than Algol, presents a formidable problem. None of the authors have been able to spare the time to document the language adequately, and the fact that it has been evolving continuously has not simplified the problem.

An Elementary Programming manual exists in several versions, the latest of which is included in this volume, but this only describes the simpler parts of the language. Three chapters of an Advanced Programming manual were written (by C.S) but these, too, only cover the easier parts of the language and by now need considerable revision.

In Spring 1965, therefore, the authors decided to prepare and if possible publish a Reference Manual which should contain a complete description of CPL as it then stood. It was not intended that the Reference Manual should be an introductory text and no particular pains were to be taken to make it easy to read by the uninitiated; it was hoped, however, that the result would not be as difficult as the Algol 60 Report.

The first draft was prepared by JNB after a series of long meetings at which the authors discussed the points at issue. The second draft was then prepared by JNB, DFH, MR, GFC, and DP, each writing one or more sections. These were then revised, edited and partly rewritten by CS to produce the incomplete draft which forms the main body of this volume.

At a meeting in June 1966 the present authors (JNB, GFC, DFH, EN, DP, MR and CS) decided that the present volume should be made available to a restricted audience as a research report. The chief reason for this was that many of the authors by now had too many other commitments to undertake any further work on the language. At the same time it was generally appreciated that at least two important additions to the language (Compound Data Structures and Segmentation) were essential if CPL were to be of wide application. With the very limited effort available it was felt that an interim publication of an unfinished document which could be carried out with very little work was all that could be expected if the vital work of developing the language additions was not to be delayed too long to be incorporated in a new implementation which MR hopes to start at MIT in the autumn of 1966. The present volume is the result.

Major developments of CPL are to be expected in the future. In particular a scheme for incorporating Compound Data Structures in a very general way is under development. It is also intended to consider the problems of segmentation and, more generally, of the relationship of the language to its compiler and operating system. Other, less wholesale changes are under consideration and a list of these is given ~~below.~~ as an Appendix.

If these developments prove successful we would hope to extend and revise the current draft of the Reference Manual and, in due course, to publish it.

References

1. Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E., and Strachey, C. (1963) "The Main Features of CPL". Computer Journal 6 134-143.

2. Strachey, C. "Towards a Formal Semantics" in "Formal Language Description Languages" ed. T. Steel. (1966) North-Holland.

3. Burstall, R.M. "Some Aspects of CPL Semantics" (1965) Experimental Programming Report No. 3. Experimental Programming Unit, University of Edinburgh.

4. Barron, D.W. and Strachey, C. "Programming" Ch. 3 pp. 49-82 in "Advances in Programming and Non-Numerical Computation" ed. L. Fox, Pergamon Press.

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

CPL ELEMENTARY PROGRAMMING MANUAL

Edition II (Cambridge)

(including corrections and revisions)

J.M.Buxton
J.C.Gray
D. Park

Corn Exchange Street,
Cambridge.

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 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. 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

$$+10^{113} \text{ to } -10^{113}$$

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; 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:

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 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 / ↑

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 (↑) 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-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 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 _ 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 at the head of 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~~ 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).

NON-LOCAL or FREE

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 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 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
   ------------------------ §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' .

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) where x = 2a + b

let p = f[3a+b]/f[3b+a] where f[x] = axx +bx + c/x

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

A where clause qualifies the largest ~~possible~~ immediately 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 as 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

<expression><relation><expression>

where <relation> denotes one of the following:

= ≠ ≠ > ≥ < ≤ << >>

'=','≠',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.

'>','≥','<','≤', 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 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≤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)
$\wedge$	(and)
$\vee$	(or)
=	(if and only if; equivalent)
$\not=$	(exclusive or; not equivalent)

The operators are given in descending order of precedence: the infixed 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

```
let  Routine R  be

    §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>65 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 → 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 conditonal 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 (section 18), 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 → E1, 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 → E1, 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 → L1, 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 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.

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

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 do C

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'

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:

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

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 some block we write the ROUTINE DEFINITION as follows:

```
routine LINEQ [real a, b, c, a', b', c', x, y, 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

$$V + 2W = 3$$
$$4V + 5W = 6$$

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~~. 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'.

See page B1

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, NOT ....then go to return.

21. EXAMPLES OF ROUTINES.

a)
```
routine Scalarproduct [ref 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 [ real a, b, ref I, val function f ]  be
~~ref f~~
§ 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:

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

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 := 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 → 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]  ℓe
      § ------------
          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]  ℓe
       §2 ---------------
          R [a + x] ; S [a -x]
          ----------------- §2
             and routine S [y]  ℓe
          §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]
```

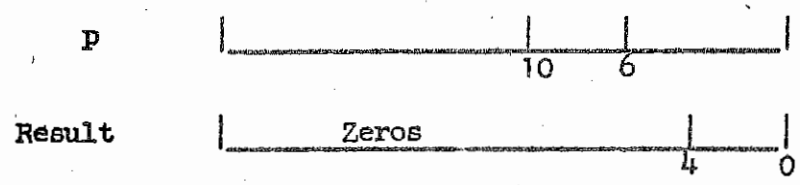
Here, j and k are index variables. Ones [j, k] produces a mask in which bits j to k inclusive are ones and all other bits are zeros; the order in which j,k are written is immaterial, i.e. Ones[j,k] = Ones[k,j].

(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 877 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. ~, ∧, ∨, ≡, ≢.

(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

maskes 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 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 occurences 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 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

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

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.

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 arrey 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'

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:

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

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 some block we write the ROUTINE DEFINITION as follows:

```
routine LINEQ [real a, b, c, a', b', c', x, y, 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

$$V + 2W = 3$$
$$4V + 5W = 6$$

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. 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'.

See page 81

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, NOT ....then go to return.

21. EXAMPLES OF ROUTINES.

a)
```
routine Scalarproduct [ref 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 [ real a, b, ref I, val function f ]  be
                  ~~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:

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

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 := 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]  be
         §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]
```

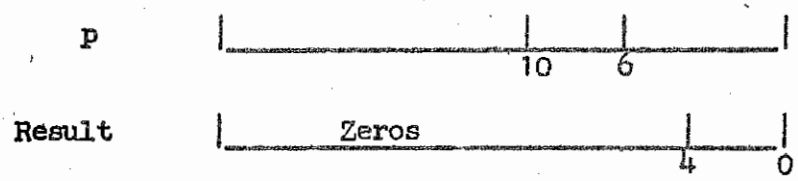
Here, j and k are index variables. Ones [j, k] produces a mask in which bits j to k inclusive are ones and all other bits are zeros; the order in which j,k are written is immaterial, i.e. Ones[j,k] = Ones[k,j].

(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 877 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. ~, ∧, ∨, =, ≠.

(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

maskes 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 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 occurences 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 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

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

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

real	
index	
integer	
double	not initially implemented
complex	"
double complex	"
i	"

Section 4

+
-
x
/
|
*
)
(

Section 5

:=
c
≤
≥

Section 7

let
=
be
all be both be

Section 8

and
where
are all are both are

Section 9

Boolean
true
false
=
≠ ≠
>
<
≥
≤
≫
≪
~
∨
∧
≡
≢ ≢

Section 11

go to	go	jump to	
if			
then	then do	do	
unless			
test			
or	or do	else	otherwise
→			
,			

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

routine rt *
reference ref *
value val *
return

Section 22

value of val of
result is

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

CPL REFERENCE MANUAL

The Editor regrets that due to difficulties in reproduction the page numbers referred to in the following corrections have been removed from some of the pages in the preceding section of the manual. This may lead to some difficulty in correcting these omissions.

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

P11. Line11
'...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

function	fn	fn.	fn.

" 9

routine	rt	rt.	rt.

" 10

reference	ref	ref.	ref.

" 11

value	val	val.	val.

" 19

recursive	rec	rec.	rec.

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,val 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 all be real

  a,b,c := 2,3,4

§ let w,z both 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 REFERENCE MANUAL

(Incomplete Draft)

July 1966

Editor's Apologia

The unfinished draft which follows shows imperfections of several kinds:

1. Missing sections

1.3.4, 1.3.5, 9.2.3, are missing merely because they have not yet been written. They presented some problems in exposition and so were postponed.

1.4, 9.5.1, 9.5.2 were postponed because the content was still under discussion.

Appendices 2 and 3 are still under discussion.

2. Incorrect Sections

The whole of section 2 is an unrevised earlier draft and may need major revision.

9.5 is now largely wrong as we have altered the rules about break, return and result is to allow a more general use. This is too complicated to insert as a manuscript alteration.

3.2, 3.4.6 and 6.5 will need modification when the type character is introduced.

4.6 and 8.3.5 will need modification when the scope of where-clauses is changed.

3. Errors, Misprints and Infelicities

Some of these have been corrected in manuscript. I should be very grateful for a note of any more which come to light.

4. Defects of Style

The draft inevitably still shows signs of its original multiple authorship in the varying styles of the different sections. More serious, perhaps, is the vacillation between a concise definition-like style (such as used in the Algol 60 report) and a more relaxed expository style. I find myself tempermentally averse from writing in English in a very formalised manner, so that the longer the draft has been under my care, the longer it has become. The only cure I can see for this is the development of a precise and symbollically expressed formal theory of semantics which still eludes our grasp.

5. Defects of Exposition

These are probably frequent and are only partly inadvertent. The Reference Manual is not intended to be an exposition or explication of CPL. The definitions of the meaning or effect of some operator or command in terms of other or simpler constructions may be in error in prescribing undesirable actions in infrequent cases. As these are discovered they will be altered.

It is part of our general approach to CPL that logical coherence and convenience in use are not to be sacrificed to brevity or simplicity in explanation.

In the ordinary course of events I should have been most unwilling to publish a report with so many defects. However, in the circumstances, given the desirability of circulating more information about CPL at once and the equally great urgency of working on Compound Data Structures together with the fact that none of the authors has time to spare on the Reference Manual, there seems to be nothing else to do.

I hope it will be possible to publish a revised and corrected version of the Reference Manual incorporating the treatment of Compound Data Structures on which we are now working in the not too distant future. It would be of great assistance to me if readers who discover errors of any sort in this draft would send me a list of them. If a sufficiently large number appear, we shall try to circulate a correction sheet to everyone who has a copy of this report.

C. Strachey

Programming Research Group
45 Banbury Road
Oxford
England.

July 1966

INTRODUCTION

This report is intended to be a complete description of CPL as at present defined. The syntax of the language is given using a system based on that used for ALGOL 60 (see Section 1.2). Semantic descriptions are given in words and by example (see Section 1.3). As a satisfactory formal language for describing semantics has not yet emerged, this document should not be regarded as completely rigorous; it should be read with a modicum of common sense.

One of the principal aims in designing CPL was to make it a practical application of a logically coherent theory of programming languages. It is not the purpose of this report to expound the underlying theory. However, some parts of this report, and in particular, the semantic descriptions, have been much simplified by making use of some of its concepts. As these may be unfamiliar, a brief general discussion of some of the issues involved, particularly as they apply to CPL, has been included in Section 1. Section 2 contains an informal account of the relationship between Publication CPL and the more formal Canonical CPL which underlies the publication language.

Part II (Sections 3-11) contain the description of Canonical CPL which is the main purpose of this report.

× Missing
// May need major revision

1. General Considerations

1.1 Algorithms, Programs and Programming Languages

1.1.1 Commands and Expressions
1.1.2 Equivalence of Algorithms
1.1.3 Equivalence of Programs
1.1.4 Equivalence of Expressions
1.1.5 Rearrangement Rules

1.2 Syntactic Problems

1.2.1 Publication and Canonical CPL
1.2.2 Syntax Rules
1.2.3 Purposes and Limitations of Syntax

1.3 Semantic Problems

1.3.1 General Approach
1.3.2 Data Items, Types
1.3.3 Transfer and Representation Functions
1.3.4 R-values, L-values
1.3.5 Load-Update Pairs

1.4 Relation with the Environment

1.4.1 The Operating System
1.4.2 Input and Output
1.4.3 Compilation
1.4.4 Errors

NOTE Sections 1.3.4, 1.3.5 and the whole of 1.4 are not yet written.

1.1.1. Commands and Expressions

An ALGORITHM is a rule for computation, using the words in a wide and informal sense. A PROGRAM is an algorithm presented in a form in which it can be accepted and, hopefully, executed by a computing machine. A PROGRAMMING LANGUAGE is the formalism in which a program is expressed.

Algorithms are built up in an hierarchical manner (which is not usually described formally) from components which are imperative sentences. These sentences contain verbs and nouns or descriptive phrases which take the place of nouns. For example in the imperative sentence:

"Replace x by the product of x and y."

the phrase "the product of x and y" is a description which is the second object of the verb "replace".

One of the ways in which the hierarchy is built up is by using one algorithm as part of a descriptive phrase used in another. The example just given might have been written "Replace x by the result of multiplying x by y." Here "multiply x by y" is a subsidiary imperative which has become incorporated in the descriptive phrase "the result of....". In this case, however, there is some ambiguity as it is not entirely clear if the command "multiply x by y" means "replace x by the result of multiplying x by y" or merely "discover the result of multiplying x by y". Such ambiguities cannot be tolerated in a program, and it is part of the function of a programming language to make their elimination simple and safe.

The features discussed above make their appearance in all programming languages, though in some the ambiguity has not been wholly eliminated. The terms used in CPL to describe them are the following.

A COMMAND corresponds to an imperative sentence.

An EXPRESSION corresponds to a descriptive phrase.

A NAME corresponds to a noun and is a special case of an expression.

The mechanism for incorporating the result of one algorithm in a descriptive phrase of another is completely explicit (see Section 4.4).

1.1.2 Equivalence of Algorithms

There is no generally accepted rule for determining the equivalence of two algorithms independently of their context. There is, however, one situation when it is quite clear what should be meant by such an equivalence. If an algorithm, A, occurs as part of the descriptive phrase "... the result of ..." as a component of a larger algorithm, then we are only interested in the "result" of A and any other algorithm, B, which produces exactly the same result will be acceptable in its place. In these circumstances we can say that A and B are locally equivalent.

1.1.3 Equivalence of Programs

Two programs are said to be equivalent if they produce indistinguishable effects whenever they are executed. This execution must be complete, not partial, so that, for instance the intermediate effects of two equivalent programs may differ, and in particular all programs which do not terminate may be regarded as equivalent. The question whether two given programs are equivalent or not in this sense is undecidable, but in certain simple cases it may be possible to prove equivalence by one of the following rules.

1. Certain commands are defined in this manual to be equivalent to other, generally simpler, commands or sequences of these.

2. Two commands which differ only in their component parts (which may be expressions or other commands) are equivalent if each of the corresponding component parts of the two commands are equivalent.

These are not the only possible cases in which commands are equivalent. The whole question of equivalence of programs is a complicated and not yet fully understood one and its investigation lies outside the scope of this report.

(Note that equality of routines as defined in section 10.2.4 is a completely different and much stronger concept than that of equivalence between programs.)

1.1.4 Equivalence of Expressions

Two expressions are equivalent when they have the same value. In the case of a block-expression the value of the whole expression is the value of the expression following the basic symbol result is so that any "side effects" such as assignments to non-local variables are ignored when determining equivalence.

When determining the equivalence of expressions involving standard mathematical operators only the properties of the ideal mathematical operators are relevant. Departures from the ideal owing to the fact that only finite representations are used in a computer, are to be ignored. Thus, for example, the expressions x + y - z and y - z + x are to be considered as equivalent irrespective of the precision or range of the internal representations of numbers, and irrespective of the order of evaluation of the components.

1.1.5 Rearrangement Rules

The general rule in CPL is permissive. Any program or expression may be replaced by any other which is equivalent to it in the sense of sections 1.1.3 and 1.1.4.

In order to allow a detailed control over sequencing and the deliberate use of side effects, it has been made possible to prevent any replacement or rearrangement of this sort. This is done by using the note sic at the head of the block. This has the effect of preventing the replacement or rearrangement of the commands inside the block. (See Section 9.8.2)

1.2 Syntactic Problems

1.2.1 Publication and Canonical CPL

Publication CPL contains a number of convenient programming devices. These are described in the Elementary Programming Manuals and a brief summary is given here in Section 2.1. Some of these features are not easily described in a formal system, and so publication CPL is formally defined by describing a transformation which changes a publication CPL document into canonical CPL. A description of this transformation is given in words in Section 2 and as a CPL program in Appendix 1. It should be noted that the transformation from publication to canonical CPL can be carried out by a number of processes and that the one described in this manual is by no means the only possible one.

Canonical CPL, which is introduced in this way as a device to aid in description of the language, is more easily subject to a formal description which occupies Part II of this report. A program in canonical CPL is a context-free character string with no dependence on layout or representation.

The existence of canonical CPL aids the task of specifying other hardware representations, such as those using cards, quite substantially. Logically unimportant changes, for example to the identifier rules, may be advisable in such cases. These alternatives are readily defined by specifying their transforms into canonical CPL.

Any practical implementation of CPL will probably accept an input which differs to some extent from both publication and canonical CPL. One possible way of constructing a compiler for such a system is to transform the locally defined input stream into a representation of canonical CPL before further processing. The program given in Appendix 1 would serve as such a PREPROCESSOR for a system whose local input language was publication CPL. Its inclusion may help to suggest a method for writing similar preprocessors for other local representations of CPL.

Publication CPL with only minor changes in the character set is used as the hardware representation at the London and Cambridge establishments.

1.2.2 Syntax Rules

Syntax descriptions of canonical CPL are given in terms of metalinguistic formulae. Sequences of letters enclosed in brackets < > represent syntactic categories; they are

chosen to be words describing approximately the meaning of the corresponding strings. The sign ::= separates the category being defined on its left from the defining sequence on its right. The vertical bar | is used to separate alternative sequences. Any sign which is not a category stands for itself. Juxtaposition of signs and/or categories signifies juxtaposition of the sequences they denote.

Metalinguistic brackets < > may be nested and thus used to group together more than one constituent sequence (which may contain alternatives). An integer subscript may be attached to a metalinguistic bracket and used to specify repetition; if it is an integer n, then the sequence within the bracket must be repeated at least n times; if the integer is followed by a - sign, then the sequence may be repeated at most n times or it may be absent.

For example, the formula

<ab> ::= <ab><d>|<ab>$_{1-}$(| [

gives a recursive rule for the formation of values of the category <ab>; a legitimate value may be formed by some other value followed by a value of <d>, or by a (possibly absent) value followed by (, or by [. If the values of <d> are the decimal digits, some values of <ab> are

```
[((((1(37(
(12345(
(((
[6
```

Grouping or association rules can be expressed in two ways in the syntax. Unsubscripted recursive formulae or groups of formulae imply a corresponding semantic grouping. Subscripted formulae (which are generally not recursive) imply no such grouping. Thus:

<p> ::= <q>|<p><q>

implies association to the left - i.e., a grouping of the type (((qq)q)q) and

<r> ::= <s>|<s><r>

implies association to the right - i.e. (s(s(ss))) while

<x> ::= <y><<z><y>>$_{o}$

and

<x> ::= <<y><z>>$_{o}$<y>

have the same meaning and imply no association among the components of x so that the only grouping is (yzyzyzy).

Syntax descriptions are given adjacent to those sections of text which describe the corresponding semantics,

and a complete syntax is given in Appendix 5.

For clarity, the ø separators are not included in the syntax formulae.

In order to avoid confusion, the canonical CPL symbols < > << >> and | are replaced in syntactic formulae (where they might be confused with the metasyntactic symbols < > |) by the basic symbols lessthan, greaterthan, muchlessthan, muchgreaterthan, bar.

1.2.3 Purposes and Limitations of Syntax

There is often some confusion about the precise meanings of the words syntax and semantics and some doubt as to where to draw the line between them. In this report we do not attempt either to discuss this issue or to be particularly nice in our distinction between the two; roughly speaking we call anything we express in the notation described in the last section syntax, and everything else in this manual semantics.

The purpose of our syntactical analysis is to allow a CPL program to be divided into smaller segments, known as syntactic categories, in a way which will make the subsequent discussion of its effect (or meaning) simpler and more precise. Any text which cannot be analysed in this way will not be a correct CPL program, and this fact allows a compiler to detect many of the slips and trivial errors in a program at an early stage by a rather superficial 'syntactic' analysis. However, the converse is by no means true, as a text which is syntactically correct in the sense of this paragraph may be semantically meaningless.

The syntactic rules in this report are not even sufficient to group a CPL text completely. In some cases (e.g., infixed-expressions) they leave the final grouping to a later stage of the analysis which we have here included in the semantics. In others, the syntactic categories correspond not to groupings of the text but to semantically related items (e.g., relations).

1.3 Semantic Problems

1.3.1 General Approach

As no satisfactory and generally accepted method of describing the semantics of a programming language has yet emerged, the method adopted in this report is a mixture of informal description in English sometimes illustrated by examples in publication CPL with, in some cases, definitions in terms of other CPL forms. Some of the concepts used in these descriptions are discussed in Sections 1.3.2 to 1.3.5.

When giving examples in publication CPL it is convenient to be able to use variables whose values are individual members of various syntactic categories. Upper case Roman letters with numerical subscripts are used for this purpose. The letter is chosen to give some indication of which syntatic category is being represented, and the suffix is used identify the particular individual member of this category. Thus, for example, when discussing the assignment command which has the syntactic form

<assignment command> ::= <expression> := <expression list>

the example used is

$$E_1 := E_2$$

which makes it possible for the subsequent discussion to refer to the components of the command as E_1 and E_2.

1.3.2 Data Items, Types

This section needs editorial revision

A CPL program is concerned with operations on and relations between certain objects known as DATA ITEMS, (or sometimes merely as ITEMS). These can be thought of in several ways. At one level they may consist of patterns of magnetization in a core store, at another they may be thought of as bit patterns or binary words. At another level they may be divided into 'instructions' and 'data', while at another they may be thought of as being numbers or letters.

The point of view adopted by CPL is that ultimately all data items are represented by bit patterns of various lengths. However, these bit pattern represent other, generally abstract objects, such as numbers or functions and their importance stems from this representation. This attitude allows us to talk about the abstract objects in CPL without being concerned with any particular representation

as a bit-string (which may well be implementation dependent) but at the same time makes it possible to consider explicitly questions which involve this representation.

The TYPE of a data item determines its representation and at the same time constrains the range of entities which may be represented. Thus, for example, the type integer can only be used to represent integers in a certain range. Both the extent of this range and the details of the representation of integers within this range are implementation dependent. The existence of the type integer, however, is a property of the language.

CPL provides for a fairly wide range of types which are discussed in some detail in Section 3.2

It is possible for a single abstract entity to be represented in more than one way as a bit pattern. The integer 3, for example, can be a data item of type real or one of type integer (among other possibilities). These two items would be quite distant and their bit-patterns would, in general, be different. There are a series of TRANSFER FUNCTIONS available in CPL which have the effect of changing from one form of representation to another without altering the abstract entity being represented. These are discussed further in Section 3.3.

The various types in CPL provide representations for a number of well-defined classes of abstract objects. It is these abstract objects which constitute the data items.

In some cases (for example, the type real) the representation of the abstract object is only approximate. The usual situation is that a range of entities all have the same representation. In these circumstances it is important to be clear whether the CPL program is discussing the ideal abstracts or their approximate representations. In other cases, however, where the representation is exact there is no significant difference in outcome whichever view is taken, and it is considerably simpler in these cases to regard CPL as manipulating the abstract entities directly and to leave the questions of representation to the implementation. It is partly for this reason that the rearrangement rules have been included in CPL.

This implies that in normal operation CPL is to be regarded as describing operations on abstract objects and not on their representations. In those cases where the difference between the approximation of the representation and the ideal abstract are of importance, it is possible to specify the exact sequence of elementary operations to be performed on the representations by using the note sic at the head of the block (see Sections ~~1.9,~~ 1.1.5). 9.8.2/

In accordance with this view there are certain specific situations in which transfer functions are inserted automatically by the implementation in order to ensure that a correct representation is used.

1.3.3 Transfer and Representation Functions

Certain functions are available in CPL which deal explicitly with the representation of a data item. One class of these, known as TRANSFER FUNCTIONS, have as their aim the control of the type by which an item is represented. Another, known as REPRESENTATION FUNCTIONS, allow access to the actual bit patterns used in the representations; these, of course, are strongly implementation dependent.

Transfer functions are, in general, polymorphic; they take a single argument, which may be of any meaningful type, and produce a single result which has the same value as its argument (or an approximation to it) and has a specified type. These functions are listed in Section 3.3.1 and are available for explicit use by the programmer. They are also inserted automatically by the compiling system in certain places to ensure that a suitable representation is used.

While the finer details of the transfer functions (and in particular, their alarm conditions) are implementation dependent, their general nature, and the situations in which they are to be inserted automatically by the compiling system are a part of the language. These are described in detail in Section 3.3.

Representation functions are not concerned with the abstract value of their arguments, merely with the bit string which represents them in the current implementation. (see Note at end of this section.) They are not, in general, inserted by the compiling system and their use by a programmer serves as a warning that the program containing them is probably implementation dependent.

There is one exception to the statement that representation functions are not inserted by the compiling system. The ability to use table lookup as a method of processing non-numerical items (such as characters) is so convenient and widely used that the fact that it involves representation explicitly is often overlooked. In order to preserve this facility, the representation functions which take single character strings, logicals and index into each other may be invoked automatically by the compiling system. The details of this, and of the other representation functions are given in Section 3.3.

NOTE: In this section and elsewhere where bit strings and items of type logical and long logical are discussed, it has been tacitly assumed that the implementation will be on a binary machine. For implementations on a decimal machine, considerable revision of the functions and operations dealing with these types would be necessary.

2. The Transformation from Publication to Canonical CPL

NOTE This section is taken from an earlier version and has not been edited. It may well need major revision and is only included to give the reader some idea of the chief features of publication CPL.

2.1 Features of Publication CPL

2.1.1 General Principles

The general principle underlying publication CPL has been to relieve the user, where possible, of the labour of including redundant information in his program. To the user, a program is best regarded as a document written in lines on sheets of paper and, as in studying the content of other documents, its layout may well convey information in the most convenient form. The publication language, therefore, makes use of abbreviations, permits omissions where no ambiguity is possible, and makes use of the layout to convey part of the meaning.

A programming language is more readily described in a formal way if programs are regarded as context-free strings of symbols. Canonical CPL is such a language, and the transformation from publication to canonical is here described. The verbal descriptions in this section describe its main features in an informal way. A more rigorous definition of the transformation is given as a program in publication CPL in Appendix 1.

2.1.2 Terminators and Layout

A command or definition may be terminated in one of three ways:

1) Explicitly, by use of a semicolon.

2) By its layout; an end of line is a terminator unless the end of line occurs at a point in the program where the context indicates that a command could not terminate at that point. Ends of lines which cannot be accepted as terminators or which are adjacent to the symbol c are ignored.

3) Implicitly; a terminator may be omitted if its presence is not essential to avoid ambiguity.

The use of "space" in CPL is derived from its use in written language in general. Spaces may be inserted freely between words to improve the layout. They are not used within words, and they must be used to separate words whose juxtaposition might cause ambiguity.

For example, consider the commands

```
test x = y then z := z + 1 or z := 1 +
z (a + b); Total := Present Factor
```

The semicolon before "or" may be omitted. The end of line following "+" cannot terminate a command and is therefore ignored. A space is necessary only between "Present" and "Factor" to avoid ambiguity (this is a case of implicit multiplication).

2.1.3 Brackets

It is often the case in writing programs that several nested sections of program come to an end at the same point, where a string of closing section brackets has to be inserted. This is an error-prone operation as the number of closing brackets needed is easily miscounted.

Section brackets in CPL may therefore be distinguished from each other by attaching names or tags to them. A closing section bracket corresponds to the nearest opening bracket earlier in the program which bears the same tag. It defines the end of the section thus specified, and it implies the closure at this point of any contained sections of program which have not yet been closed.

Examples of the use of tagged section brackets may be found in the Introductory Manual and in Appendix 1.

2.1.4 Conditional Expressions

In publication CPL, the form of a conditional expression is

$$E_1 \longrightarrow E_2, E_3$$

Any occurrence of a comma used in a conditional expression is replaced in the canonical form by the basic symbol "comma".

2.1.5 Other features

Monadic occurrences of + and - are allowed in publication CPL. In the canonical form, they are replaced by the basic symbols "pos" and "neg".

A wide range of synonyms is available for the basic underlined words in the canonical form. Many examples are found in the Introductory Manual which contains a list of synonyms.

Comments may be inserted freely in publication CPL text. A comment is intoduced by a double bar and continues up to the end of that line; the whole of this text is ignored.

In some cases, symbols may be omitted in the publication form if no ambiguity is caused; for example,

```
if B then goto L
```

may be written

```
if B goto L
```

2.2 Categories recognised during Transformation

All instances of the following six categories are recognised during the transformation process. Their descriptions, and the modifications introduced during transformation, are described here. Note that strings of characters or primes mentioned in this section may ~~include numbers~~ be ~~at~~ null. ~~members.~~

2.2.1 < name >

A single lower case letter followed by any number of primes, or an upper case letter followed by a tag.

A tag is a string of letters of either case, digits and dots, followed by any number of primes.

In the canonical form, the symbol ~~?~~ is inserted before all occurrences of names.

2.2.2 < left section bracket >

The symbol § followed by a tag. The canonical form ~~is unchanged.~~ has no Tag

2.2.3 < right section bracket >

The symbol $ followed by a tag. The canonical form ~~is unchanged~~. has no Tag

Note that two section brackets with identical tags are said to "match".

The closing brackets omitted in publication CPL as in 2.1.3 above are inserted by the transformation.

2.2.4 < string constant >

A string of characters starting and ending with a prime. Within a string constant the symbol $_{10}$ (subscript ten) is used as an escape character:

1010 stands for 10

$10'$ stands for $'$

and the meaning of any other character following a single $_{10}$ is implementation dependent.

In the canonical form, the symbol ß is inserted before all instances of string constants.

2.2.5 <dot string>

A string of dots and spaces which must include at least 2 dots and which is surrounded by commas. The canonical form is as follows:

,...,

2.2.6 <number>

A string of any number of digits and dots which may include spaces and must include at least one digit. The canonical form is preceded by the sumbol #.

2.3 Rules for Transformation

These rules are intended as a brief verbal guide to the operation of the preprocessor program given in Appendix 1. They describe the operations performed on a complete publication text in a series of four passes, the end product being the corresponding canonical text. The operations described under each pass are done together.

Pass 1

1. Remove all comment text.

Pass 2

2. Recognise all instances of the six categories described in 2.2 above, and transform where necessary as described in that section.

Pass 3

3. Insert matching closing section brackets where required.

4. Instances of commas used in conditional expressions are replaced by "comma".

5. Change all new lines to semicolons, replace consecutive semicolons by a single semicolon. Consider the symbols adjacent to each semicolon. If either is c, remove the semicolon. Unless the preceding symbol can legally precede a semicolon and the succeeding symbol can start a command, delete the semicolon.

Pass 4

6. Remove all occurrences of the basic symbol c.

7. Insert do between any pair of symbols such that the first of these cannot precede a command and the second must start a command.

8. Recognise monadic uses of + and - and replace them by pos and neg respectively.

3 PRELIMINARIES

Note The Type character will probably be added. This will involve some alteration to strings and possibly elsewhere.

Compound Data Structures will probably also extend and alter the whole concept of Types so that this section may have to be modified considerably or extended.

3.1 Canonical Form

3.1.1 General

A program in canonical CPL consists of a string of items which are either BASIC SYMBOLS or members of a BASIC CATEGORY. A basic symbol is a single character from the CPL basic alphabet $\mathcal{A}_B$ defined in Section 3.1.3. A member of a basic category consists of a character from $\mathcal{A}_B$ identifying the category followed by a word (string of characters) from the possibly implementation dependent alphabet associated with the category. The categories and their associated alphabets are described in Section 3.1.4.

Every implementation must have a method of representing all the characters in $\mathcal{A}_B$ and the alphabets associated with the basic categories by one or more characters from its IMPLEMENTATION ALPHABET $\mathcal{A}_I$. These are the characters which occupy one print position in the program together with certain extra symbols representing the layout.

The implementation characters may in turn be represented by one or more HARDWARE CHARACTERS from the alphabet $\mathcal{A}_H$. These are the basic hardware units of input and output such as a single row of punched tape, a single column of a punched card or a single keystroke on a directly connected keyboard.

Thus, for example, in a hardware implementation which uses a backspace to provide compound or overstruck characters, the following three strings of length six each produces the same printed result (The symbol ← is used temporarily to denote the hardware character 'backspace').

```
i← _f← _
if← ←__
_←i_←f
```

In each case the result produced should be the two character string from the implementation alphabet i f. This in turn, represents the single character (basic symbol) if from the alphabet $\mathcal{A}_B$.

The transformation from a string of hardware characters to a string of implementation characters is largely implementation dependent. However, it should be arranged so that hardware character strings which would produce the same printed image will produce the same implementation character string as illustrated by the example given above.

The transformation from a string of implementation characters to canonical CPL is done by a preprocessor of the type described in Section 2 and Appendix 1.

3.1.2 CPL Publication Alphabet

There is one 'implementation' alphabet which is not implementation dependent; this is the publication alphabet $\mathcal{A}_p$ which contains the following 162 printing characters:

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
+ - × / ↑
> ≥ = ≠ ≤ < ≪ ≫
~ ≃ → ⇔
∧ ∨ ≡ ≢
( ) [ ] $ $
; : , := 10 | ' . ! _ ‖

and the two non-printing characters 'space' and 'newline'.

These characters are used in the rest of the manual except that the three symbols ≪ ≫ := may be typed so as to occupy two print positions each.

3.1.3 Basic Symbols

The CPL Basic Alphabet $\mathcal{A}_B$ contains the 109 symbols listed below. The symbols are given in the representation used in the syntax tables and also, if necessary, the form or forms by which they can be represented in the publication language. It is a general rule of the publication language that when representing a basic symbol by an UNDERLINED WORD (including words of one letter) no distinction is made between upper and lower case letters and spaces, whether underlined or not, are ignored. Thus the two strings

GO TO go to

in the publication language both represent the same basic symbol which is represented in the table below as goto.

(a) The following 25 symbols from $\mathcal{A}_p$ also represent unique basic symbols from $\mathcal{A}_B$.

× / ↑ ≤ = ≠ ≥ ∧ ∨ ≡ ≢ ~ ≃

; : := ( ) [ ] $ $ → ⇔ 10

(b) the following five basic symbols are represented in the syntax tables by underlined words and in the publication language by single symbols. (see Section 1.2.2)

Syntax Form	Publication Symbol
bar	|
greaterthan	>
lessthan	<
muchgreaterthan	>>
muchlessthan	<<

(c) The following 54 basic symbols are represented in the syntax tables and in the publication by the same underlined word.

and	general	n	test
array	goto	note	~~to~~
			true
Boolean	i	prefer	type
break	if		
	in	r	unless
cpx	index	real	until
	integer	repeat	update
d		repeatuntil	
dcpx	L	repeatwhile	vector
double	label	resultis	
doublecomplex	let	return	where
	load		while
false	logical	sic	
finish	longlogical	step	x
fix			
fixed	matrix		2
for			8
free			

(d) The following 14 basic symbols can be represented in the publication language by any one of a number of synonyms. In the table below, the first form is that used in the syntax tables.

allbe	bothare	bothbe	are
be	is		
constant	const		
do	then	thendo	
forexternal	forext		
function	fn		
or	ordo		
recursive	rec		

reference	ref
referenceof	refof
routine	rt
value	val
valueof	valof
variable	var

To Thru Through

(e) The following six basic symbols are represented by only three symbols in the publication language. The preprocessor decides from the context which is the appropriate basic symbol to use.

Syntax form	Publication Symbol
comma	,
,	,
pos	+
+	+
neg	-
-	-

(f) The remaining five basic symbols are the following:

dotstring which has the publication language representation of at least ~~3~~ 2 dots, possibly interspersed with spaces and surrounded by commas. The following are three examples of dotstrings in the publication language:

,..., ,.. .. , ,.....,

ñ, đ, ß which are used to indicate the basic categories name, number and string-constant respectively.

ø which is sometimes used when writing canonical CPL strings to separate the basic symbols where there might otherwise be confusion. It has no other significance.

3.1.4 Basic Categories

The precise definition of the basic categories is implementation dependent. The definitions given in this section apply to the publication language. The basic categories are recognised by the preprocessor.

(a) Names

A member of the basic category <name> in canonical CPL consists of a name-word from the alphabet $\mathcal{A}_N$ preceded by the basic symbol ñ and followed by the basic symbol ø (both from the alphabet $\mathcal{A}_B$).

For publication CPL the alphabet $\mathcal{A}_N$ contains the following 64 symbols:

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 name-word consists of a single lower case letter followed by any number of primes, or an upper case letter followed by a tag.

A tag consists of a string of letters (of either case), digits or dots of any length followed by any number of primes. A tag may not include spaces - indeed a space is not a symbol in the alphabet $\mathcal{A}_N$.

The alphabet $\mathcal{A}_N$ is a sub-alphabet of $\mathcal{A}_P$ and its symbols are represented in the publication language by the same symbols from $\mathcal{A}_P$.

(b) Numbers

A member of the basic category <number> in canonical CPL consists of a number-word from the alphabet $\mathcal{A}_D$ preceded by the basic symbol đ and followed by the basic symbol ø (both from the alphabet $\mathcal{A}_B$).

For publication CPL the alphabet $\mathcal{A}_D$ contains the following 11 symbols:

0 1 2 3 4 5 6 7 8 9 .

A number-word consists of any string of these symbols containing at least one digit.

The alphabet $\mathcal{A}_D$ is a sub-alphabet of $\mathcal{A}_P$ and its symbols are represented in the publication language by the same symbols from $\mathcal{A}_P$.

(c) String-Constants

A member of the basic category <string constant> in canonical CPL consists of a string-word from the QUOTABLE ALPHABET $\mathcal{A}_Q$ preceded by the basic symbol ś and followed by

the basic symbol ø (both from the alphabet $\mathcal{A}_B$).

For publication CPL the alphabet $\mathcal{A}_Q$ contains the 162 printing characters from the publication alphabet $\mathcal{A}_P$ (see Section 3.1.2) together with the three non-printing characters:

'space' 'backspace' 'newline'

A string-word consists of any number of these symbols (including zero).

In publication CPL a string-constant is distinguished from other parts of the text by being enclosed by primes. This fact, together with the rules for dealing with comments in the preprocessor mean that although the characters of $\mathcal{A}_Q$ and $\mathcal{A}_P$ are very similar, some of the characters from $\mathcal{A}_Q$ cannot be straightforwardly represented by the same character from $\mathcal{A}_P$. The precise rule is the following:

The following five characters from $\mathcal{A}_Q$ have a special representation in characters from $\mathcal{A}_P$ as shown:

Character from $\mathcal{A}_Q$	Representation in $\mathcal{A}_P$
backspace	!b
newline	!n
'	!'
!	!!
||	!||

The remaining 160 characters in $\mathcal{A}_Q$ are represented by the same character from $\mathcal{A}_P$. In addition, the non-printing character 'space' in $\mathcal{A}_Q$ may also be represented by the characters !s from $\mathcal{A}_P$.

Note that string-constants are defined above by the way in which the characters from $\mathcal{A}_Q$ are represented, and not in the way in which an arbitrary string of characters from $\mathcal{A}_P$ can be interpreted. This implies that not all possible strings enclosed in string-quotes (primes) are acceptable. In particular, the ESCAPE CHARACTER ! is only defined if followed by one of the six characters b n ' ! || or s.

Publication CPL has the additional rule (implemented by the preprocessor) that new lines and trailing spaces (at the right end of a line) are ignored in string-constants.

(d) Section-Brackets

In publication CPL the basic symbols § and $ may be followed by a tag taken from the alphabet $\mathcal{A}_N$ (see (a) above). These are used by the preprocessor to insert extra closing section brackets ($) where necessary so that in a canonical CPL program all section brackets nest and match correctly. This means that the tags associated with them in the publication language are no longer necessary so that they do not form a part of the canonical CPL program.

Thus section brackets do not form a basic category in the strict sense. They must, however, be treated as such in

the preprocessor and many implementations will find it desirable to continue to associate a tag with each section bracket even when it is no longer logically necessary in order to give some assistance in locating errors in the program.

3.2 Types

3.2.1 General

The following sections list all the types of data item currently included in CPL and give a brief description of them. The details are implementation dependent.

Further types may be added from time to time, and it is possible that the incorporation of a more comprehensive treatment of compound data structures may force a considerable revision or extension of the concepts of this section.

The rules concerning transfer functions are collected in Section 3.3 for convenience. They are discussed in more general terms in Sections 1.3.2 and 1.3.3.

3.2.2 Numerical Types

real
A real number, whose range and precision are implementation dependent.

integer
An integer, whose range is implementation dependent.

index
An integer, whose range is probably less than that of integer and whose use may improve object program efficiency, for example, in subscripts.

complex
A pair of real numbers taken in order as the real and imaginary parts of a complex number.

double
A real number with approximately double the precision of type real.

doublecomplex
A pair of real numbers taken in order as the real and imaginary parts of a complex number. The components have approximately double the precision of the components of a complex data item.

3.2.3 Logical Types

Boolean
A truth value; its value is either true or false

logical
A bit string of a fixed and implementation dependent length.

longlogical
A bit string of a fixed and implementation dependent length probably greater than that of logical.

The intention of logical and longlogical is that it should be possible to transform reasonably large sections of type index into logical and of type real into longlogical without loss of information.

In an implementation based on a binary word oriented computer where reals occupied one word and index a shorter, possibly address-length, segment, logicals would probably be the length of an index register and longlogicals a whole computer word.

3.2.4 Other Types

string
A string of characters of any length including null (See Sections 3.1.4, 3.4.5 and 3.4.6).

label
A location in the program and a description of an environment of that area of program (See Section 9.3).

function
A representation of a function (See Section ~~10~~. 10.2.2

routine
A representation of a routine (See Section 10).

array
A representation of an array (See Section ~~11~~). 11.1.2

type
A data item whose value is a data item type.

general
A data item whose type may vary dynamically; it may be a data-Type, array-Type or function-Type but not a single-Type-list (See Section 7.1).

3.3 Transfer and Representation Functions

3.3.1 Programmers Transfer Functions

The following transfer functions are available to the programmer:

DoubleComplex	dc
Complex	c
Double	d
Real	r
Integer	n
Index	x
LongLogical	ll
Logical	l
String	s
Boolean	b

Each takes a single argument which may be of any meaningful type and produces as its single result an item of the type indicated by its name. Each function exists in two forms, according to the mode of the context in which it is used.

a) The R-mode Transfer Functions

These take a single R-mode argument and produce a single R-mode result. The details are implementation dependent, but the results can be described with the aid of the following table which uses the abbreviations indicated above for types.

Original Type \ Target Type	x	n	r	d	c	dc	l	ll	s	b
x	-	n	n	n	n	n	l	(l)	s	(b)
n	x	-	r	r	r	r	(x)	(x)	(x)	(x)
r	n	n	-	d	c	c	(n)	(n)	(n)	(n)
d	r	r	r	-	r	dc	(r)	(r)	(r)	(r)
c	r	r	r	r	-	dc	(r)	(r)	(r)	(r)
dc	c	c	c	d	c	-	(c)	(c)	(c)	(c)
l	x	(x)	(x)	(x)	(x)	(x)	-	ll	s	(x)
ll	(l)	(l)	(l)	(l)	(l)	(l)	l	_	(l)	(l)
s	x	(x)	(x)	(x)	(x)	(x)	l	(l)	-	-
b	(x)	(x)	(x)	(x)	(x)	(x)	(x)	(x)	-	-

Table of Initial Transfer Functions

The entry corresponding to a pair of types in this table indicates the first type to which the data item is to be transformed in a transfer from the first type to the second. Thus the transfer from string to doublecomplex would pass through the sequence.

string→index→integer→real→complex→doublecomplex

Further details of the BASIC TRANSFER FUNCTIONS which make up chains of this sort are given in Section 3.3.2.

The transfers indicated by entries enclosed in parenthesis can only be initiated by an explicitly written transfer function. The others may, in suitable circumstances, be invoked automatically by the compiling system (see Section 3.3.3).

There are direct transfer functions from every type (including those not mentioned in the table) to type general which make no change in the representation. Transfers from general to other types can only take place in circumstances when a transfer from the dynamically current type associated with the general would be permitted.

b) The L-mode Transfer Functions

These are ~~load-update pairs~~ and can be defined in terms of the R-mode functions. Thus, for example, the L-mode LH-functions

transfer function which takes real into complex is equivalent to the following.

```
Function RealtoComplex [ref real x] be
        load § result is Complex [x] §
      update § x := Real [rhs] §
```

c) Mode Forcing Functions

The functions

RValue [x]
LValue [x]

which can take an argument of any type, force the evaluation of their argument in the mode indicated. They are not true transfer functions, but may invoke the mode transfers described in Section 4.2. These functions may be of use when it is desirable to specify the mode of evaluation in circumstances where the context would not otherwise do this ~~(eg the argument list for an application of a variable or formal functions).~~

d) Other Types

There are no transfer functions except those mentioned above. Representation functions are discussed in Section 3.3.5.

3.3.2 Basic Transfer Functions

a) index to integer
integer to real
real to double
real to complex
double to double complex
complex to double complex

These are standard numerical transfers with changes of representation but normally no loss of information. With the possible exception of integer to real no alarm conditions should be necessary.

b) integer to index
double to real
double complex to complex

These involve a decrease in precision (and possibly of range) without a change of the nature of the number. They will normally involve a rounding or truncating operation and should give an alarm if the range is exceeded.

c) real to integer
complex to real
double complex to double

These involve a change in the nature of the number probably without alteration of the precision or range. The

two complex to real transformations should show an alarm if the imaginary part of the complex number is not negligible in an implementation dependent sense compared with the real part. The real to integer transfer is intended to allow operations on integers to be carried out using reals even if the representation of reals does not allow exact integer arithmetic to be performed. The transformation should therefore show an alarm whenever the divergence of the real from an integral value is more than a certain implementation dependent amount. It is open to the implementation to make this quantity anything between 0 and 0.5 so that programs which require the nearest integer to a real whose exact value should not be integral should use the basic function Round (see Appendix 3).

An alarm should also be given if the result is out of range.

d) logical to longlogical
longlogical to logical

The logical is taken to be the right hand (least significant) end of a longlogical whose remaining bits are all zeros. As these transfer functions may be inserted automatically, there should be an alarm on an attempt to transform a longlogical whose extra bits were not all zeros. The basic function Mask (see Appendix 3) may be used to ensure this if required.

e) logical to index
index to logical
string to index
index to string
logical to string
string to logical
Boolean to index
index to Boolean

These transfers are made by considering all the types concerned to be integers. For logicals this is done by treating them as unsigned (positive) binary integers. For strings the functions are only defined for strings of length 1 (single characters); for these, the corresponding integer is that given in the collating table (see Section 3.4.6). If several characters have the same number in this table, one of these should be nominated for use by the transfer functions. For Booleans the correspondence is True ↔ 1 and false ↔ 0.

An alarm should be given if the result is out of range or, in the case of transfers to string, if no suitable character exists.

3.3.3 Automatic Insertion of Transfer Functions

The transfer functions corresponding to the unparenthesised entries in the table of Section 3.3.1, those to and from type general, and the mode transfer functions (see Section 4.2) and no others, will be inserted by the compiling system as necessary in the following situations.

(a) When the type and/or mode of an expression is constrained by its context. This arises in the following cases.

(i) The right hand side of an assignment statement. The mode is R-mode and the type must be that of the L-value it is to update (see Section 9.2).

(ii) An argument of a function or routine call in the special case where the written operator is the name of a function or routine data item with the attribute constant, and whose formal parameter types and modes are explicitly (or by default) stated in the definition (see Sections 5.3.2 and 10.1.2).

(iii) An argument of a polymorphic function, routine or operator after the particular version of the function, routine or operator has been determined by the rules given in Section 3.3.4, Appendix 3 or elsewhere.

(Note that programmers' functions and routines come under case (i) and basic and library functions under case (ii) or (iii).)

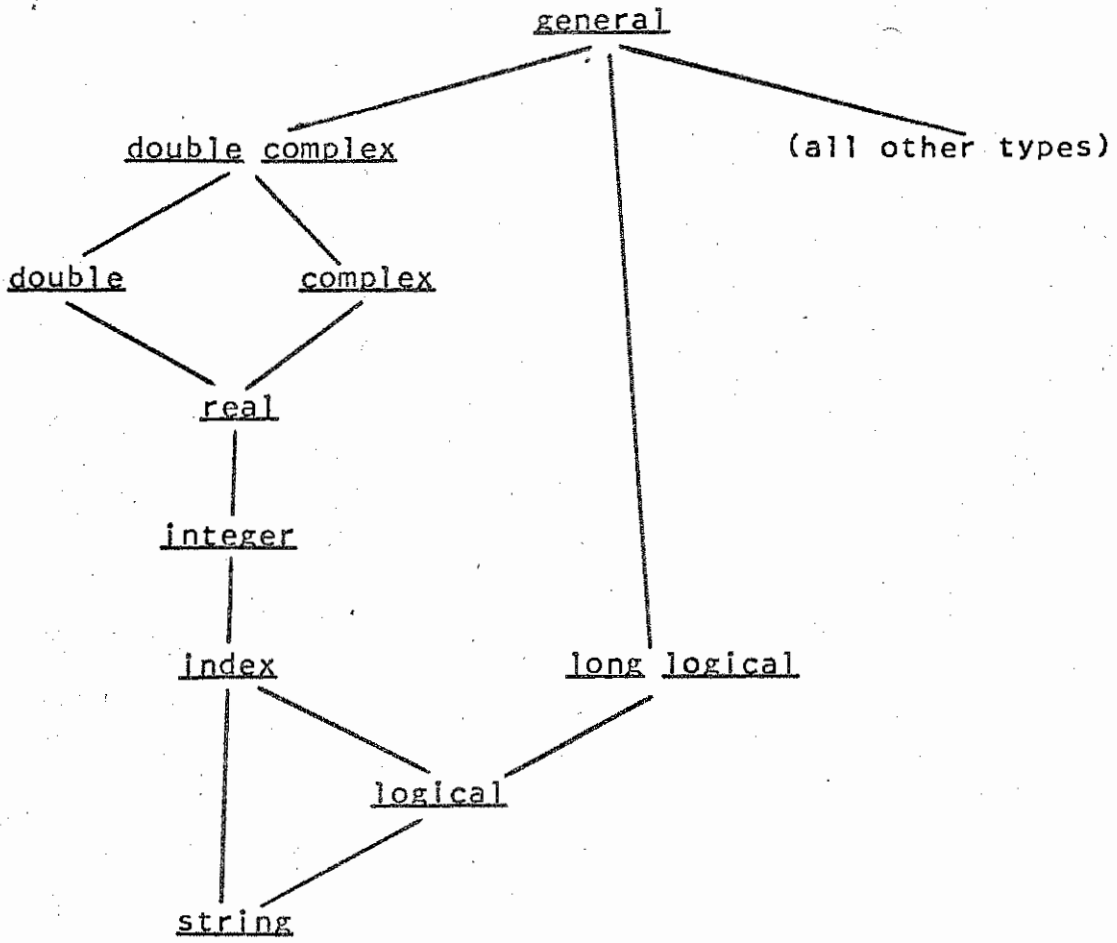
In these situations the compiler will insert the necessary transfer functions to match the mode and type of the expression to its context. If the necessary transfer functions either do not exist or are not allowed to be inserted automatically, the outcome is undefined.

(b) When the value of an expression may be that of one of a number of expressions. The choice being determined dynamically. This arises in the following cases:

(i) The arms of a conditional-expression.

(ii) When a block-expression contains two or more resultis commands.

In these situations each of the alternative expressions is transformed to the type which is the least upper bound of their individual types. This is defined in an obvious way from the diagram on the next page.



Hierarchy of Types

If the required transfer functions do not exist or cannot be inserted automatically, the least upper bound is taken as general.
Mode transfers are not required in these situations.

3.3.4 Polymorphic Operators

Polymorphic operators are operators which exist in several versions, all known by the same name. The choice between these versions is dependent on the types of their arguments. There are two classes of polymorphic operators, those for which a unique version exists for each allowable set of argument types, and those for which only a limited set of versions exist and transfer functions are used to extend the permissible set of argument types. Transfer representation functions are in the first class, most basic

functions and infixed operators are in the second. Polymorphic operators are a part of the language and no mechanism exists at present to allow a programmer to define his own polymorphic functions or routines.

The selection of the correct version of a polymorphic operator of the second class is governed by one of the following rules; the particular rule to be used being determined by the name (or in the case of an infixed-operator, the symbol) of the operator.

If any of the operands are of type general then this rule must be applied dynamically (i.e. at the time at which the operator is to be applied to its operands); the current types of the general arguments must be used to select the correct version of the operator as described below. If the type of the result depends on the version of the operator used in any way, the result must be converted back to type general. If all versions of the operator produce the same type (as occurs, for example, with relations where the result is always Boolean), then this is also the type of the result.

In cases where none of the operands are general, the rule selected can be applied by the compiling system and any transfer functions required may be inserted statically (provided this is allowed under the rules given in Section 3.3.3).

Rule A

The version of the operator selected is the least upper bound (in the sense of Section 3.3.3) of the types of all the operands and of the lowest available operator type. If no version of the operator of this type exists, or if the least upper bound is general , the effect is undefined. If a suitable operator exists, its type will define the types required of the operands all of which must be transformed to this type before the operator is applied. The type of the result is also determined by the version of the operator; in general it is either the same as that of the operands, or restricted to a single type such as Boolean.

Rule B

The operator requires all its parameters except one to be of a unique specified type, so that polymorphism occurs for only one of its arguents. The version of the operator is determined from the actual type of its only polymorphic argument as follows:

i) If a version of the operator exists which accepts an argument of exactly this type, this is the version chosen.

ii) If no such version exists, then the operator is taken to be a single specified version.

In the first case transfer functions may be inserted automatically for those arguments for which the operators is not polymorphic; in the second case they may be inserted for all arguments.

3.3.5 Representation Functions

The following representation functions are available to the programmer.

1. Bitpattern [x]

This is a polymorphic function of the first class (see Section 3.3.4) which has the following versions:

Argument Type	Result Type
real	longlogical
integer	longlogical
index	logical
character	logical

Other implementation dependent versions may also exist.

The effect is to treat the actual (implementation dependent) representation of its argument as a bit pattern of the corresponding type, any spare bits in the result type being filled with zeros. No transfer functions are inserted automatically for this function.

2. Formreal [11]
   Forminteger [11]
   Formindex [1]
   Formcharacter [1]

These functions, which are not polymorphic, together form the inverse of Bitpattern x . They take a longlogical or logical argument as indicated and treat it as if it were the internal representation of the type suggested by the name of the function.

Other representation functions forming the inverse of further versions of Bitpattern may also exist.

3.3.5 Representation Functions

The following representation functions are available to the programmer.

1. Bitpattern [x]

This is a polymorphic function of the first class (see Section 3.3.4) which has the following versions:

Argument Type	Result Type
real	longlogical
integer	longlogical
index	logical
character	logical

Other implementation dependent versions may also exist.

The effect is to treat the actual (implementation dependent) representation of its argument as a bit pattern of the corresponding type, any spare bits in the result type being filled with zeros. No transfer functions are inserted automatically for this function.

2. Formreal [1]
   Forminteger [1]
   Formindex [1]
   Formcharacter [1]

These functions, which are not polymorphic, together form the inverse of Bitpattern x . They take a longlogical or logical argument as indicated and treat it as if it were the internal representation of the type suggested by the name of the function.

Other representation functions forming the inverse of further versions of Bitpattern may also exist.

3.4 Constants

3.4.1 General

Written constants have an R-value which is invariant and independent of any definitions or assignments made in the program. Their natural mode of evaluation (see Section 4.2) is R-mode so that they behave as if they had no L-value.

There are two forms of written constants: ATOMIC CONSTANTS which are usually basic symbols such as True or a single digit, and which have the property that their R-value depends only on their complete written form and cannot be deduced from the R-values of any of its components (if it can be decomposed at all); and COMPOUND CONSTANTS whose R-value can be deduced from the R-values of its components. A number such as 231 is a compound constant and its value can be deduced from the values of its component digits 2, 3 and 1.

Compound constants are a form of constant expression but the rules governing their evaluation are not those governing the other expressions in CPL.

As constants have an R-value which is immediately determinable from their written form, the concepts of scope and extent (see Section 8.2) are not applicable to them. In this respect they differ from named data items which have been defined with the attribute constant (see Section 7.3 and 10.3.2).

3.4.2 Syntax

<basic numerical constant> ::= <number>[₁₀ | ₁₀+ | ₁₀- ><number>]₁₋

<constant prefix> ::= x | n | r | d | cpx | dcpx

<numerical constant> ::= <constant prefix>₁₋
<basic numerical constant> | 1

<logical prefix> ::= 2 | 8 | 2L | 8L | L2 | L8

<logical constant> ::= <logical prefix><number> |
<logical prefix><number> bar |
<logical prefix> bar <number> |
bar <logical prefix><number>

<Boolean constant> ::= true | false

<constant> ::= <numerical constant>|<logical constant>|
<string constant>|<Boolean constant>|
<single type>

3.4.3 Numerical Constants

A basic-numerical-constant of the form

$$N_1 \,_{10}\, N_2$$

where N_1 and N_2 are members of the basic category number, is only defined if N_1 contains at most one dot and N_2 contains no dots. The dot in N_1, if present, is taken as a decimal point; if N_1 contains no dot it is taken as a decimal integer. The $_{10}$ denotes that the N_2 following it is to be interpreted as a decimal exponent. If there is a + or a - between the $_{10}$ and N_2, this is taken as the sign of the exponent; if there is no sign, the exponent is considered positive.

Two unprefixed basic-numerical-constants which have the same numerical values will always be represented in the same way. Thus the constants

$$1, \quad +1, \quad 1.0, \quad +0.1\,_{10}\,+1$$

are interchangeable and only differ in the way in which they are written on the paper.

The type of a numerical-constant prefixed by x, n, r, d, cpx, or dcpx is index, integer, real, double, complex, or doublecomplex, respectively.

The type of an unprefixed numerical-constant is the least upper bound (in the sense of Section 3.3.3) of its PRESUMPTIVE TYPE and its MINIMUM TYPE.

The presumptive type is determined by context: if the numerical constant stands alone on the right hand side of a definition, the presumptive type is the preferred type if this is numerical. In all other cases and contexts the presumptive type is index.

The minimum type of a numerical-constant is determined by its value. If this is integral and within the index range, then the minimum type is index. If the value is integral and outside the index range but within the integer range, then the minimum type is integer. In all other cases the minimum type is real.

The basic symbol i represents a numerical constant of type complex whose value is a complex number with zero real part and unit imaginery part. Note that with this exception there are no unprefixed numerical-constants of type double, complex, or doublecomplex, but that an expression such as

2 + 3i

will be of type complex by virtue of the rules governing the types of infixed-expressions (see Section 6).

3.4.4 Logical Constants

The occurrence of l or L in a logical prefix indicates that the constant is to be represented in the type longlogical. Otherwise it is logical. If the prefix contains 2, the constant is a binary pattern and may only contain digits 0 and 1. If the prefix contains 8, it is an octal pattern and may only contain digits 0 to 7. Logical constants are assumed right justified unless bar is present, in which case justification takes place towards the side on which bar appears.

3.4.5 String-Constants

A string-constant is a member of a basic category and, as such, its syntax is not a part of canonical CPL (see Section 3.1.4).

In Publication CPL in particular a string-constant is identified by being preceded and followed by a prime. Inside these string quotes the character ! is used as a single-character-quote or escape indicator. If it is immediately followed by one of the characters

b n s ' ! ||

the pair is taken as the single string character (i.e., the single character from $\mathcal{S}_Q$)

backspace newline space ' ! ||

respectively.

Thus, for example, a string-constant in Publication CPL which was written as

's!b/!''if!''o!b/'

has a length of 10 string characters

s backspace / ' l f ' o backspace /

and when printed would produce

$'if'$

3.4.6 Character Representation

The internal representation of strings is by characters from the internal alphabet $\mathcal{A}_X$. The number of characters in the alphabet is implementation dependent and the characters have a fixed association with (small) integers which is that used by the transfer functions between strings of length one (characters) and the types index and ~~Boolean~~ logical.

The characters of the alphabets $\mathcal{A}_B$, $\mathcal{A}_I$, $\mathcal{A}_H$, $\mathcal{A}_V$ and $\mathcal{A}_Q$ may all be represented in $\mathcal{A}_X$. The details of the representation are implementation dependent; in particular it is not necessary for the various alphabets to be represented by disjoint sections of $\mathcal{A}_X$ nor is it necessary for single characters to be represented by single characters although this is very desirable at least for the quotable alphabet $\mathcal{A}_Q$.

Tables showing the representation of $\mathcal{A}_P$ and $\mathcal{A}_Q$ which are relevant to publication CPL are given in Appendix 2.

3.4.7 Other Constant Expressions

a) Boolean Constants

The basic symbols true and false are constants of type Boolean.

b) Type Constants

Any expression which falls into the syntactic category <single type> (see Section 7.1) is a constant of type type.

c) Other Types

There are no constants of type label, function, routine, array or general except those introduced explicitly by a definition with the attribute constant.

4 EXPRESSIONS

4.1 Syntax

<prefixed operator> ::= <name>|<prefixed expression>| rhs |
(<expression>)

<prefixed expression> ::= <prefixed operator>[<expression>$_{1-}$]

<individual> ::= <constant>|<name>|<prefixed expression>|
~<individual>|<block expression>| rhs |
pos | neg | (<expression>)

<relation> ::= muchlessthan | lessthan | ≤ | = | ≠ | ≥ |
greaterthan | muchgreaterthan

<infixed operator> ::= + | - | × | / | ↑ | ≡ | ≢ | ∧ | ∨ |
⇔ |<relation>

<infixed expression> ::= <<individual><infixed operator>$_{1->0}$
<individual>

<conditional expression> ::= <infixed expression>→
<basic expression> comma <basic expression>

<block expression> ::= valueof <block>| referenceof <block>

<basic expression> ::= <infixed expression>|
<conditional expression>

<expression list> ::= <basic expression><,<basic expression>>$_0$

<expression> ::= <expression list>|
<expression> where <in definition>

4.2 Evaluation

An expression basically consists of a name, a constant, or an operator with its operands; both the operator and the operands are themselves expressions. More generally an expression may be a sequence of such forms separated by commas.

An expression is a rule for the evaluation of a sequence of one or more quantities or values. Evaluation may in general be performed in either of two modes: L-mode and R-mode. The result of an expression evaluated in L-mode is a sequence of one or more L-values, and the result of an expression evaluated in R-mode is a sequence of one or more R-values (see Section 4.5). The number of members of such a sequence is normally restricted by the context of the expression. In particular, an expression which occurs as the operand of a monadic or infixed-operator (see Sections 5 and 6) must yield a single value.

The mode of evaluation of a particular expression is determined by context, although certain operators may have a natural mode of evaluation. For example, all monadic and infixed-operators require R-values as operands and produce single R-values as results, whereas prefixed-operators may take R-value or L-value operands and produce R-value or L-value results according to context. The natural mode of evaluation of a name is L-mode, and of a constant, R-mode. In both cases, the result is a single value.

In circumstances where the mode required by context conflicts with the natural mode, transfers are automatically effected as follows.

a) L-Value to R-Value

The data specified by the L-value is extracted to give an R-value.

b) R-Value to L-Value

A new (disjoint) L-value is created and the R-value becomes associated with it; the new L-value is given the attribute constant (see Section 7.3).

The following rules apply to the evaluation of operators and operands in expressions.

1. Except where otherwise stated, an operator and its operands (including their component parts) may be evaluated in any order or in parallel. An operator may be applied to its operands at any time after each of them has been evaluated.
2. Expressions may be transformed and re-ordered in any way that would give identical results if all evaluations were performed exactly.

4.3 Conditional-Expressions

4.3.1 Syntax

⟨conditional expression⟩ ::= ⟨infixed expression⟩ → ⟨basic expression⟩ comma ⟨basic expression⟩

In the conditional-expression

$$E_1 \rightarrow E_2, E_3$$

the syntax rules permit E_1 to be an infixed-expression and each of E_2 and E_3 to be a basic-expression. This implies that if either E_2 or E_3 is to represent an expression-list or is to be qualified by a where-clause, then this must be enclosed in parentheses; similarly, E_1 must be bracketed if it is itself a conditional-expression or is qualified by a where-clause.

4.3.2 Semantics

The conditional-expression

$$E_1 \rightarrow E_2, E_3$$

has the same value as the expression

reference of § test E_1 then result is E_2 or result is E_3 §

(see Sections 4.4 and 9.6) Note that on evaluating this conditional-expression E_1 is evaluated in R-mode and that one only of E_2 or E_3 is evaluated in L-mode. Thus the natural mode of evaluation (section 4.2) is L-mode.

4.4 Block-Expressions

4.4.1 Syntax

<block expression> ::= valueof <block>| referenceof <block>

A block-expression enables the computation of one or more results to be written in the form of a command, or block. There are two forms depending on the required mode of evaluation.

value of <block> which produces one or more R-values
reference of <block> which produces one or more L-values

In either case, the block must contain one or more result is commands (for syntax see Section 9.5.4). The type of each result of a block-expression is defined to be the least upper bound (as defined in Section 6.6.2) of the types of the corresponding members of the constituent expressions in all result is commands which appear in the block, but excluding any which appear in further block-expressions within the block.

4.4.2 Semantics

A block-expression is evaluated as follows. The commands (and declarations, if any) are executed in sequence until one of the result is commands is encountered. The constituent expression of this command is then evaluated in the mode determined by the prefix value of or reference of, and the results transformed if necessary to the appropriate types as defined in the previous paragraph. These results form the value of the block-expression and the execution of the block is then terminated.

Commands in a block-expression may include assignments which update L-values which are non-local to the block (i.e. which are not lost when the execution of the block is terminated). In such cases the block-expression is said to have SIDE EFFECTS, and particular care must be taken to ensure that the results of the program are not dependent on the order of evaluations or the rules given in Section 4.2 may imply that the results are unpredictable.

4.5 Expression-Lists

An expression may generally produce a sequence of results; where there are two or more of these, the syntactic class expression-list enables the members to be written explicitly separated by commas. This form is naturally only meaningful in contexts where several values are meaningful: for example, in assignment-commands where several simultaneous assignments are being made (see Section 9.2), and in actual parameter lists where several values are being handed over as the operands of a function or routine call (see Section 10.1).

Note that whenever an expression produces a sequence of two or more values, these values are never considered together as being a single value; and thus such an expression does not have a single type, but a sequence of types.

Several or all of the members of an expression-list may be bracketed: for example, in order to delimit the scope of a where-clause. Otherwise, bracketing in an expression-list has no significance.

4.6 Where-Clauses

The general form of an expression qualified by a where-clause is

$$E_1 \text{ where } D_1$$

where the syntax rules permit E_1 to be any expression (possibly qualified by a where-clause itself) and D_1 to be an in-definition (see Section 8.1). However, since commands and definitions may also be qualified by where-clauses, it is usually necessary for the above form to be enclosed in parentheses. For this reason, the syntactic rules for certain commands and declarations use the forms expression-list or basic-expression to indicate that, if the expression only is to be qualified by a where-clause, they must be enclosed in parentheses.

The semantics of an expression qualified by a where-clause are conveniently defined by the following equivalent form

$$\text{ref of } \S \text{ let } D_1 ; \text{ result is } E_1 \S$$

(See Sections 4.4 and 8.3).

The rules governing the scope of where-clauses may be changed to make them simpler and more logically coherent

5 PREFIXED OPERATORS AND EXPRESSIONS

5.1 Monadic Operators

5.1.1 Syntax

<individual> ::= <constant>|<name>|<prefixed expression>|
~<individual>|<block expression>| rhs |
pos | neg | (<expression>)

5.1.2 Semantics

(a) The operator ~

This is written immediately before its operand, and is less binding than any prefixed-operator, but more binding than any infixed-operator (see Section 6.1). It takes an R-value as an operand (which must be of the type logical, longlogical or Boolean) and produces an R-value of the same type as its result.

If the operand is of type logical or longlogical, the nth bit of the result is determined by the nth bit of the operand according to the following table:

Operand	0	1
Result	1	0

If the operand is of type Boolean the result is defined by the following table.

Operand	false	true
Result	true	false

(b) Operators + -

+ and - may be used as monadic operators in publication CPL. However, the transformation to canonical CPL replaces them by the symbols pos and neg respectively. In the precedence and grouping rules for determining infixed-operators (see Section 6.1), these symbols are formally treated as equivalent to the symbol sequences (+1) and (-1) respectively. Thus the meaning of prefixed + and - is defined to be the same as that of multiplication by the system constants (+1) and (-1) which have the values 0+1 and 0-1 respectively.

5.2 Prefixed-Operators

5.2.1 Syntax

<prefixed operator> ::= <name>| rhs |<prefixed expression>|
(<expression>)

5.2.2 Semantics

A prefixed-operator can be a name, a general expression enclosed in parentheses, or a prefixed-expression. If the prefixed-operator is a name, it must be either a BASIC FUNCTION (see Section 10.1.2) or a PROGRAMMER'S FUNCTION (see Section 10.3) or an ARRAY (see Section 11). If the prefixed-operator is an expression, it must be possible to evaluate it in R-mode to produce a function or array.

5.3 Prefixed-Expressions

5.3.1 Syntax

<prefixed expression> ::= <prefixed operator>[<expression>,...]

A prefixed-expression consists of a prefixed-operator followed by its ARGUMENT LIST which is enclosed in SQUARE brackets. The brackets must be present even if the argument list is empty (i.e. if the operator requires no operands).

5.3.2 Semantics

A prefixed-expression is used to indicate the APPLICATION of a prefixed-operator to its operands. (Note that the EVALUATION of an operator and its APPLICATION are two completely distinct processes.) The evaluation of a prefixed-expression in either L-mode or R-mode is performed in two stages:

(a) The prefixed-operator is evaluated in R-mode to produce a function or array, and the operand (which is the expression enclosed in brackets following it) is evaluated in the appropriate mode to form the argument list. These two evaluations may be performed in any order.

(b) The resulting function or array is applied to its argument in the manner described in Section 10.1 to produce the value of the prefixed-expression. The application of a function to its arguments is known as a FUNCTION CALL.

In general, the operand is evaluated in L-mode and the resulting argument list of L-values, without changes of type, is used in the function call. However, in the special case where the written operator is the name of a function data item with the attribute constant, and whose formal parameter types and modes are explicitly (or by default) stated in the definition, the components of the argument list in a function call are further transformed (if possible) so that they match the corresponding formal parameters in mode and type. (Note that this special case will probably be the most common form of function call.)

A function may only be called with an argument list with the correct number of arguments. (Some basic functions may be called with any number of arguments - see Appendix 3. No programmer's functions can have this property.)

The type of a prefixed-expression is determined by ~~the type of~~ its prefixed-operator (see Section 10.2.2.).

5.3.3 Array References

An array-type data item may be used as a prefix-operator to refer to an element of an array. In this form, the operator takes a single operand of type index in R-mode (called a subscript), which is used to select the required element according to the definition of the array (see Section 11). The natural mode of evaluation of an array reference is L-mode, giving the L-value of an element of the array. The result could itself be an array, which may be applied to a further subscript to obtain a sub-element, and so on.

For example, if A represents a three dimensional array of real elements (i.e. of type real 3 array), and i, j, k represent index values

A[i]	gives an L-value of type real 2 array
A[i][j]	gives an L-value of type real 1 array
A[i][j][k]	gives an L-value of type real

An alternative syntactic form permits an array reference to be written with two or more subscripts to specify directly sub-elements of multidimensional arrays. For example:

A[i,j]	is equivalent to A[i][j]
A[i,j,k]	is equivalent to A[i][j][k]

The result of an array reference is only defined if the value of the subscript lies within the bounds of the array.

6 INFIXED OPERATORS AND EXPRESSIONS

6.1 Syntax and Grouping

6.1.1 Syntax

<relation> ::= muchlessthan | lessthan | ≤ | = | ≠ | > | greaterthan | muchgreaterthan

<infixed operator> ::= + | - | × | / | ↑ | ≡ | ≢ | ∧ | ∨ | ⇔ | <relation>

<infixed expression> ::= <<individual><infixed operator>...><individual>

6.1.2 General

Infixed-operators require two operands; they may require operands of a particular type or be polymorphic (see Section 6.2 and 6.6). With the possible exception of the operators = and ≠, their operands are evaluated in R-mode and the result they produce is always an R-value.

In the written form of expressions, an infixed-operator is placed between its operands and together these may form an operand of a further operator. Thus, an expression may appear as a sequence of alternate operands (i.e. items of the syntactic class individual) and infixed-operators; and therefore rules exist to determine the grouping of operators with operands in any such sequence. These rules may be overridden explicitly by enclosure of an operand in parentheses.

6.1.3 Juxtaposition, Pos and Neg

The symbols pos and neg rank as members of the syntactic class individual. Semantically, they are treated as constant data items with the values +1 and -1 respectively, their type being that numerical type with the lowest precision (i.e. normally index).

The juxtaposition of two individuals in an infixed-expression is taken as implying multiplication and the operator is therefore inserted before any further analysis takes place.

This may need some modification

6.1.4 Grouping

Operator	↑	×	/	-	+	⇔	<relation>	∧	∨	≡	≢
PL	10	9	9	6	6	5	4	3	2	1	
PR	8	9	8	7	6	5	4	3	2	1	

Table of Precedences

Grouping rules are determined by assigning two PRECEDENCES to each infixed-operator (a left precedence PL and a right precedence PR), and by specifying a procedure of analysis on an operand-operator sequence of the general form

$$A_0 x_1 A_1 x_2 \ldots x_n A_n$$

where $A_0, A_1, \ldots A_n$ are individuals and $x_1, x_2, \ldots x_n$ are infixed-operators. The operator-operand sequence is preceded by a dummy operator x_0 (for which PR = 0) and followed by a dummy operator x_{n+1} (for which PL = 0); thus:

$$x_0 A_0 x_1 A_1 x_2 \ldots x_n A_n x_{n+1}$$

The grouping procedure is the following. The sequence is searched for any subsequence $A_i\ x_{i+1} \ldots x_j\ A_j$ such that

$$i < j$$
$$PR[x_i] < PL[x_{i+1}]$$
$$PR[x_j] > PL[x_{j+1}]$$

and, for all $s = i+1, \ldots j-1$

$$PR[x_s] = PL[x_{s+1}] \neq 0$$

A subsequence of this form is termed a PRIME PHRASE. Any prime phrase found is enclosed in parentheses and thus becomes an individual; the operator-operand sequence is then renumbered to take account of this. The process is repeated until no further prime phrases can be found.

This procedure will not group an infixed-expression completely into pairs of operands separated by an operator. In particular, the operators which are associative (viz × + ∧ ∨ ≡ and ≢) may occur in groups with any number of members. These may, in the cases of × and +, be followed by a single occurence of / or - respectively. Thus an infixed-expression such as:

a + b + c - d
p × q / r
x ≡ y ≢ z

are treated as individuals and not grouped any further. This, however, is in accordance with their normal mathematical meaning which implies that the ordering (or grouping) of the operations inside such a group has no effect on its value.

The relations are also combined into a single group. A COMPOUND RELATION, as such a group is called, has the Boolean value True if and only if each of its component relations has the value True. Thus the compound relation

$$E_1 R_2 E_2 R_3 E_3 \ldots E_{n-1} R_n E_n$$

has the value True if and only if all the relations

$$E_1 R_2 E_2$$
$$E_2 R_3 E_3$$
$$\ldots\ldots$$
$$E_{n-1} R_n E_n$$

~~has~~ the value True. have

6.2 Numerical Operators

6.2.1 Types

Infixed-operators are in general POLYMORPHIC; that is they may take operands of various types, and may produce results whose type depends upon that of the operands. However, in all cases, the operator takes two operands of the same type. (Two operands of differing types are permitted in certain circumstances, but these are always converted to a single common type before application of the operator: See Section 6.6.)

In this and following sections, a table is given for each operator (or group of operators) specifying the type of the result as a function of the type of the operands.

Operators + - x

Operand Type	Result Type
index	index
integer	integer
real	real
double	double
complex	complex
double complex	double complex
general	general

Operators / ↑

Operand Type	Result Type
real	real
double	double
complex	complex
double complex	double complex
general	general

6.2.2 Semantics

The meaning of the numerical infixed-operators is intended to be the same as their normal mathematical meaning. In many cases, however, the correspondence is only approximate due to the presence of rounding and similar errors. Such divergences from the mathematical ideal are implementation dependent.

Infixed division (/) produces a minimum type of real. There is thus no infixed-operator producing the integer type of division with a remainder. These results can be obtained by using the basic functions Quot[x,y] and Rem[x,y] (see Appendix 3). The division symbol ÷ is not a part of CPL.

Infixed exponentiation (↑) is somewhat irregular. Like division its minimum type is real, but its result type is required to be the same as its operand type and for operands of types real or double this imposes a restriction on the values of the operands.

More precisely, the value of the expression $E_1 \uparrow E_2$ is one of the values of $\exp(E_2 \log E_1)$ with the following restrictions:

1) If E_1 and E_2 are of type real or double, the expression is undefined unless at least one of the values of $\exp(E_2 \log E_1)$ is not complex.

2) If E_1 and E_2 are of type real or double and $\exp(E_2 \log E_1)$ has two real values, the value of $E_1 \uparrow E_2$ is the positive one of these.

3) If E_1 and E_2 are of type complex or double complex and $\exp(E_2 \log E_1)$ has more than one value, the choice between them is implementation dependent.

6.3 Logical Operators

Operators	$\wedge$ $\vee$ $\equiv$ $\not\equiv$
Operand Type	Result Type
logical	logical
long logical	long logical
Boolean	Boolean
general	general

Where these operators produce logical or long logical results, they are defined to be bit-by-bit manipulations on the individual bits of the operands. Thus, the nth bit of the result is determined by the nth bit of each operand according to the following table.

1st operand	0	0	1	1
2nd operand	0	1	0	1
$\wedge$	0	0	0	1
$\vee$	0	1	1	1
$\equiv$	1	0	0	1
$\not\equiv$	0	1	1	0

When the operators produce Boolean results, their values may be determined from the same table by replacing 0 by false and 1 by true.

6.4 Relations

Operators = ≠

Operand Type	Result Type
any	Boolean

If the operands are one of the numerical types, then

$$E_1 = E_2$$

is true if $(E_1 - E_2)$ is a representation of zero. Thus, as with subtraction, equality of numerical values is implementation dependent.

For all other types, $E_1 = E_2$ is true if the operands yield identical R-values (but see Section 10.2.4).

The expression

$$E_1 \neq E_2$$

is always equivalent to

$$\sim(E_1 = E_2)$$

Note that, if the operands are of different types, transfer functions may be invoked under the rules of Section 6.6. For equality between operands of type function or routine see Section 10.2.4, and between operands of type array see Section 11.1.2.

Operators < ≤ ≥ >

Operand Type	Result Type
index	Boolean
integer	Boolean
real	Boolean
double	Boolean
string	Boolean

If the operands E_1, E_2 are of one of the permitted numerical types (which are all non-complex) the value of the relation is determined from the value of $E_1 - E_2$. Transfer functions may be invoked (see Section 6.6) to make this

possible.

If the operands are of type string, comparison is effected by repeatedly comparing the numerical equivalent (see Section 3.3.5) of successive characters of each string value (beginning with the first or most significant character) until the required attribute is established. If the characters of one of the operands are exhausted before the other in this process, it is assumed to continue with sufficient dummy characters whose numerical equivalents are less than any other character.

Operators << >>

Operand Type	Result Type
index	Boolean
integer	Boolean
real	Boolean
double	Boolean

These operators formally give the result true if the value of one is negligible compared with the value of the other. This normally means that

$E_1 << E_2$ is equivalent to $(E_1+E_2) = E_2$
$E_1 >> E_2$ is equivalent to $(E_1+E_2) = E_1$

The precise definition however is implementation dependent, and in some floating point implementations of the types real and double the decision may be made in the basis of the value of the exponent alone, so that $E_1 << E_2$ may sometimes be stronger than $E_1+E_2 = E_2$ and sometimes be weaker.

If E_1 is of type index or integer and does not have the value zero, $0 << E_1$ is always true. The value of $0 << 0$ is implementation dependent but should normally be false.

possible.

If the operands are of type string, comparison is effected by repeatedly comparing the numerical equivalent (see Section 3.3.5) of successive characters of each string value (beginning with the first or most significant character) until the required attribute is established. If the characters of one of the operands are exhausted before the other in this process, it is assumed to continue with sufficient dummy characters whose numerical equivalents are less than any other character.

Operators << >>

Operand Type	Result Type
index	Boolean
integer	Boolean
real	Boolean
double	Boolean

These operators formally give the result true if the value of one is negligible compared with the value of the other. This normally means that

$E_1 << E_2$ is equivalent to $(E_1+E_2) = E_2$
$E_1 >> E_2$ is equivalent to $(E_1+E_2) = E_1$

The precise definition however is implementation dependent, and in some floating point implementations of the types real and double the decision may be made in the basis of the value of the exponent alone, so that $E_1 << E_2$ may sometimes be stronger than $E_1+E_2 = E_2$ and sometimes be weaker.

If E_1 is of type index or integer and does not have the value zero, $0 << E_1$ is always true. The value of $0 << 0$ is implementation dependent but should normally be false.

6.5 String Operators

Operator ⇔

Operand Type	Result Type
string	string

This operator concatenates two strings. That is, the result is a string consisting of the characters of the first operand, followed by the characters of the second operand.

The introduction of the type character may involve alterations to this section

6.6 Polymorphism and Type Matching

The infixed-operators described in Sections 6.2, 6.3 and 6.4 are all polymorphic. The general rules for choosing the correct version of a polymorphic operator and inserting any necessary transfer functions are given in Sections 3.3.3 and 3.3.4. The choice for all these operators is made using Rule A of Section 3.3.4

7 DEFINITIONS

7.1 Syntax

⟨data type⟩ ::= real | integer | complex | double | logical |
index | longlogical | doublecomplex | type |
Boolean | label | routine | string | general

⟨array type⟩ ::= ⟨single type⟩$_{i-}$ ⟨number⟩ array |
⟨single type⟩$_{i-}$ ⟨ vector | matrix ⟩

⟨function type⟩ ::= ⟨single type⟩$_{i-}$ function

⟨single type⟩ ::= ⟨data type⟩|⟨array type⟩|⟨function type⟩|
(⟨single type list⟩)

⟨single type list⟩ ::= ⟨single type⟩⟨,⟨single type⟩⟩$_{0}$

⟨name list⟩ ::= ⟨name⟩⟨,⟨name⟩⟩$_{0}$

⟨formal parameter⟩ ::= ⟨single type⟩⟨ value | reference ⟩⟨name⟩|
⟨ value | reference ⟩$_{i-}$ ⟨single type⟩$_{i-}$
⟨name⟩.

⟨formal parameter list⟩ ::= ⟨formal parameter⟩
⟨,⟨formal parameter⟩⟩$_{0}$

⟨type definition⟩ ::= ⟨name list⟩⟨ be | allbe ⟩⟨single type list⟩

⟨simple definition⟩ ::= ⟨ variable | constant ⟩$_{i-}$ ⟨name list⟩
⟨ = | ≃ | all = | all ≃ ⟩
⟨expression list⟩

⟨LH function body⟩ ::= ⟨ fix ⟨definition⟩⟩$_{i-}$
load ⟨block⟩ update ⟨block⟩

⟨function definition⟩ ::= ⟨ recursive ⟩$_{i-}$ ⟨ variable | constant ⟩$_{i-}$
⟨name⟩[⟨formal parameter list⟩$_{i-}$]
⟨ = | ≡ ⟩⟨expression list⟩|
⟨ recursive ⟩$_{i-}$⟨ variable | constant ⟩$_{i-}$
⟨ fixed | free ⟩$_{i-}$⟨function⟩⟨name⟩
[⟨formal parameter list⟩$_{i-}$] be
⟨⟨LH function body⟩|
§⟨LH function body⟩§ ⟩

⟨routine definition⟩ ::= ⟨ recursive ⟩$_{i-}$ ⟨ variable | constant ⟩$_{i-}$
⟨ fixed | free ⟩$_{i-}$⟨routine⟩⟨name⟩
⟨[⟨formal parameter list⟩$_{i-}$]⟩$_{i-}$
be ⟨block⟩

⟨basic definition⟩ ::= ⟨type definition⟩|⟨simple definition⟩|
⟨function definition⟩|
⟨routine definition⟩|
⟨ recursive ⟩$_{i-}$ §⟨definition⟩§

7.2 Modes of Definition

One of the functions of a definition is to introduce a name which refers to a data item, and which is then identified with all other occurrences of the same name in the scope of the definition. The scope rules are given in Section 8. Other characteristics depend on whether the definition is by value or reference.

7.2.1 Definition by Type

Type-definitions have the defining operator be or allbe. They create a new L-value (i.e. one which is disjoint from all other existing at the moment of definition) and associate this with the defined name. The R-value associated with this new L-value is undefined.

7.2.2 Definition by Value

A definition by value creates a new L-value and associates this with the defined name; it also associates an initial R-value with the L-value. Definitions by the defining operators = , all = and the special forms of function and routine definitions all define by value.

7.2.3 Definition by Reference

A definition by reference is primarily intended to associate an already existing L-value with the name defined. The L-value is the one obtained by evaluating the defining expression in L-mode. In the exceptional case where the natural mode of evaluation of the defining expression is R-mode, a new L-value of the appropriate type is created, its R-value is the result obtained by evaluating the defining expression in R-mode. This L-value is ~~then~~ associated with the defined name.

given the attribute constant and

7.3 Constant and Variable Definitions

An L-value in CPL has either the property constant or variable, depending solely on how the L-value was created. If an L-value has the property constant, it means that its R-value cannot be changed by assignment. It is meaningful to include the word variable or constant only in definitions by value, if it is omitted from a definition, a constancy attribute is assumed. A function or routine definition is assumed to have the attribute constant unless explicitly specified as variable, whereas a type or simple definition is taken as variable, unless otherwise specified.

7.4 Definitions and Types

Data items of any type may be defined; however the actual type in most cases need not be explicitly stated. The rules for determining the type depend on the kind of definition, and on the current PREFERRED TYPE.

7.4.1 Preferred Type

There are many occasions when it is necessary to determine the type of a data item in CPL. In most cases this can be deduced from known or ascertainable types of its component parts, but there are a few situations where this is not possible. The most important of these are: in the formal parameter lists of function and routine definitions (see Section 10), and in written numerical constants. In these situations it is always possible to specify the type required explicitly.

It is possible for the programmer to select a preferred type which will be used in certain circumstances, where the type of a data item is otherwise unknown.

Initially the preferred type is real; it will be altered only if the programmer writes a directive in the form

prefer <single type>

where the <single type> specifies the new preferred type. This directive may be written in the declaration sequence of a block; its scope consists of any succeeding declarations, together with the command sequence of the block.

If the programmer has specified the type of an array, matrix, vector or function without specifying the single-type which determines the type of its components or result, this is assumed to be the preferred type. This does not apply to functions defined by one of the special forms of definition described in Section 10.3 whose result types are discussed in Section 10.3.6.

7.4.2 Type Definitions

(a) The defining operator be

The syntax of the simplest form of type definition is

<name list> be <single type list>

It is only meaningful if the name-list and single-type-list have the same number of members, in which case the names on the left of the operator be are associated with data items whose types are the corresponding single-types of the single-type-list. Initial values for the data items are not defined.

(b) The defining operator all be

This operator may be used when it is necessary to define by type a number of data items all of the same type. For example, the following two definitions have the same meaning.

```
x, y, z  be  real, real, real
x, y, z  all be  real
```

7.4.3 Simple Initialized Definitions

(a) The defining operator =

The type of each of the data items on the left is the type of the corresponding expression on the right. There must be the same number of defining expressions on the right as names in the name-list on the left. If any of the defining expressions is a single numerical constant and the preferred type is numerical then the type of this constant is determined by the rules given in Section 3.2.2. This form of definition defines by value (see Section 7.2.2).

(b) The defining operator ≃

As in (a) the type of each of the data items on the left is the type of the corresponding expression on the right, and there must be the same number of defining expressions as names defined. For each name defined the corresponding defining expression is evaluated in L-mode, and the L-value obtained is associated with the name. (See Section 7.2.3.)

(c) The defining operators all = and all ≃

The effects of these operators are best described in terms of the operators described above.

For example, the following two definitions are synonymous:

```
a, b, c  all = E,
a, b, c  = (x, x, x  where x = E, )
```

Similarly the following two are also synonymous:

```
a, b, c  all ≃ E,
a, b, c  ≃ (x, x, x  where x ≃ E, )
```

8 DEFINITION STRUCTURE AND SCOPE RULES

8.1 Syntax

<and definition> ::= <basic definition>
< and <basic definition>>$_0$

<in definition> ::= <and definition>|
<and definition> in <in definition>

<definition> ::= <in definition>|
<definition> where <in definition>

<declaration> ::= let <definition>

The scope of where may be changed

8.2 Scope and Extent

8.2.1 Scope

The SCOPE of a definition is a syntactic concept; it is that area of the written program in which the data items defined in definitions may be referred to using the names with which the definitions associate them.

If a definition (the 'inner' definition) occurs within the scope of another definition of the same name (the 'outer' definition), the inner definition supersedes the outer one. The outer definition is said to be 'shielded' from the scope of the inner one, and to have a 'hole' in its scope.

8.2.2 Extent

The EXTENT of a data item is a dynamic concept; it is that part of the dynamic execution of a program through which a named data item maintains a continous existance.

In CPL the extent of a named data item is controlled by the scope of its definition. It continues as long as the command currently being executed lies within the scope (including any holes there may be in it); it terminates as soon as the current command lies outside the scope of the definition.

For this purpose the whole of the execution of a routine call is considered to lie in the extent containing the call, irrespective of the scope which contains the body of the routine.

8.3 Scope Rules for Definitions

The scope of a definition is controlled by its position in a program, and by use of the words recursive, and, in and where.

8.3.1 Recursive

The scope of a definition does not normally include its own definiens (i.e. the right hand side of its own definition). However, by preceeding the definition by the word recursive, it can be made to do so. This may only be used if all the data items defined by the definition are either functions or routines. The effect of recursive on other definitions is undefined.

8.3.2 Composite Definitions

Composite definitions made up of definitions joined by and, in or where together with uses of recursive may be formed into a single basic definition by enclosing them in section brackets. The resulting definition may also be qualified by preceeding it by the word recursive provided it satisfies the conditions of Section 8.3.1.

8.3.3 And

This word is used to combine two or more definitions into one. Unless this combined definition is recursive, none of the component definitions is in the scope of any other component definition. The order of activation of the definitions is undefined; it is intended that they be considered as activated in parallel. Thus, for example, the effect of the definitions.

$$\begin{aligned} & N_1 = E_1 \\ \underline{\text{and}}\ & N_2 = E_2 \\ & \cdots\cdots \\ \underline{\text{and}}\ & N_n = E_n \end{aligned}$$

is the same as the effect of the definition

$$N_1, N_2, \ldots, N_n = E_1, E_2, \ldots, E_n$$

The effect is undefined unless the names $N_1, N_2, \ldots, N_r$ are all different.

8.3.4 in

When this word separates a number of definitions there is an implied association rule to the right. For example, the definition

D_1 in D_2 in D_3

is synonymous with

D_1 in § D_2 in D_3 §

It is therefore only necessary to consider the simple composite form

D_1 in D_2

The scope of D_1 is D_2 alone, and the scope of D_2 is the same as that of the whole definition. If the composite definition is recursive then its scope includes its definiens; that is the scope of D_2 now includes D_1 and D_2, but the scope of D_1 is still only D_2.

8.3.5 Where NB The rules governing the scope of where-clauses may be changed.

As with the word in, the word where separates a number of definitions but in this case there is an implied association rule to the left. For example

D_1 where D_1 where D_3

is synonymous with

§ D_1 where D_2 § where D_3

It is therefore only necessary to consider the simple form

D_1 where D_2

The above definition is exactly equivalent in meaning to

D_2 in D_1

This form is described in Section 8.3.4 above.

The word where may also be used to introduce a where-clause qualifying an expression or command. The general rule is that a where-clause qualifies the longest command, definition or expression (in that order of preference) immediately preceding it; the scope of its component definition is the qualified command, definition or expression. As commands and definitions frequently terminate in expressions, it is generally necessary to enclose expressions qualified by where-clauses in parentheses. (See Sections 4.6 and 9.8.)

8.3.6 Let

A declaration is a definition preceded by the word let. A declaration or a sequence of declarations may only occur at the head of a block. The scope of a declaration is the set of immediately following declarations and the command sequence of the block. If the definition in the declaration is recursive then its scope includes its definiens as well.

8.4 Other Scope Rules

The concept of scope also applies in situations where names are introduced other than by definition. There are two cases, as follows.

(a) Formal Parameters
The scope of a formal parameter in a function or routine definition is the defining expression or command. (see Section 10.)

(b) Labels
The scope of labels declared by colon as in the command

```
L : a := bx+c
```

is the smallest enclosing routine body or block expression (see Section 9.8).

9 COMMANDS

× Section not written
|| May need major revision

9.1 Syntax

<assignment command> ::= <expression> := <expression list>|
<expression> all := <basic expression>

<transfer command> ::= goto <basic expression>

<routine command> ::= <name>|<prefixed expression>

<simple command> ::= <assignment command>|<transfer command>|
<routine command>| resultis <expression>|
break | return |<block>

<if command> ::= < if | unless ><expression> do <basic command>

<test command> ::= test <expression> do <command> or
<basic command>

<conditional command> ::= <if command>|<test command>

<while command> ::= < while | until ><expression> do
<basic command>

<repeat command> ::= <command>< repeatwhile | repeatuntil >
<basic expression>|
<command> repeat

<for element> ::= <basic expression>|
step <basic expression>,<basic expression>,
<basic expression>|
<basic expression> to <basic expression>|
<basic expression>,<basic expression>
dotstring <basic expression>

<for list> ::= <for element><,<for element>>$_o$|
~~<for list> where <in definition>~~

<for command> ::= for <name> = <for list> do <basic command>|
forexternal <individual>< = | := ><for list> do
<basic command>

<cycle command> ::= <while command>|<repeat command>|
<for command>

<basic command> ::= <<name> : >$_o$<<conditional command>|
<simple command>|<cycle command>>

<command> ::= <basic command>|<command> where <in definition>

<note> ::= prefer <single type>| sic | note <string constant>

<block> ::= § <<note>, <declaration>$_o$|<declaration>, |<command>>
<;<command>>$_o$ §

9.2 Assignment-Commands

9.2.1 Syntax

⟨assignment command⟩ ::= ⟨expression⟩ := ⟨expression list⟩|
⟨expression⟩ all := ⟨basic expression⟩

9.2.2 Semantics

(a) The effect of the command

$$E_1 := E_2$$

is as follows. The expression E_1 is evaluated in L-mode to produce a sequence of L-values of any type, say $L_1, L_2 \ldots L_m$; the expression E_2 is also evaluated in R-mode to produce a sequence of R-values of any type, say $R_1, R_2 \ldots R_n$; these evaluations may be performed in any order, but must be complete before the next phase starts. In order for the command to be well-formed, it is necessary that m=n. Each member of the sequence $L_1, L_2, \ldots, L_m$ is then UPDATED by the corresponding member of the sequence $R_1, R_2, \ldots, R_n$.

To update the L-value L_k with the R-value R_k, R_k is first transformed to be of the same type as L_k if the appropriate transfer function exists (if it does not, the effect of the command is undefined). Let the transformed R-value be R'_k. The R-value associated with L_k is now altered to be R'_k so that future evaluations of E_1 in R-mode will produce the R-value R'_k (unless modified by further assignment-commands).

Note that the L-value L_k is not altered by updating it, and that the R-value of any expression which has the L-value L_k is altered by updating L_k. If two L-values are not disjoint - i.e. if they share a part or the whole of their associated R-values - then updating either will effect the R-value associated with both at least as far as their shared part is concerned.

The order in which the sequence $L_1, L_2, \ldots, L_m$ is updated is unspecified (cf. Section 4.2). This means that in general the L-values in the sequence $L_1, L_2, \ldots, L_m$ should be disjoint or the effect of the assignment-command may be undefined.

(b) The effect of the command

$$E_1 \text{ all} := E_2$$

is as follows. The expression E_1 and E_2 are evaluated as described above to yield the sequences $L_1, L_2, \ldots, L_m$ and $R_1, R_2, \ldots, R_n$. In this case the sequence of R-values must have only one member (i.e. n=1).

The command is completed by updating every member of the sequence $L_1, L_2, \ldots, L_m$ with the R-value R_1.

9.2.3 Updating

If L_1 is an L-value and R_1 is an R-value of the same type, there is an operation known as updating L_1 with R_1. This operation depends on the way in which L_1 originally arose.

This section is incomplete

9.3 Transfer-Commands and Labels

9.3.1 Syntax

⟨transfer command⟩ ::= goto ⟨basic expression⟩

⟨basic command⟩ ::= ⟨⟨ name⟩ :⟩ ₒ ⟨⟨ conditional command⟩ |
⟨simple command⟩ | ⟨cycle command⟩⟩

9.3.2 Semantics

In a transfer-command, the basic symbol goto is followed by a basic-expression which is to be evaluated to produce an R - value of type label.

The process of executing a command is known as an "activation" of that command. At any moment in the activation of a command, a number of further activations may be current, since one activation may call for a succession of further activations to be completed before it can be completed itself. Thus activation of a routine-command calls for an activation of the routine body which may call for activation of some sub-block of the body, and so on, each activation being "called for" by the most recent previous activation of the sequence which is not yet complete. If a function or routine is called recursively, the same command may have more than one activation current at the same moment.

An activation may be terminated either normally, as described elsewhere, or by a jump; a jump has an R - value of type label associated with it. This value specifies the destination of the jump, providing the following information:

(a) a point in some configuration of commands at which some activation is to be continued.

(b) a rule for determining which activation is to be continued at the point specified; this is in the form of a rule for determining whether or not an activation is "live" to the destination.

The effect of a transfer-command is then specified by the following rule, which is to be applied iteratively: activation of a transfer-command is immediately terminated by a jump whose destination is specified by the value of the expression associated with the command. If an activation calls for an activation which is terminated by a jump to a destination to which it is not itself live, then it also is terminated by the jump.

(For rules governing the termination of blocks and cycle-commands, see 9.8.5.)

This process is continued until an activation is found which is live to the destination.

The effect of a jump is undefined if there is no current activation to which it is live.

The following trivial example illustrates the way in which ambiguities may arise when recursive routines and functions are involved, and how this rule resolves them. If R is defined by

```
let recursive routine R [integer n, label L, M] be
    § if n>0 do R [n-1, M, N]
      N : goto L §
```

the effect of executing R [10, Even, Odd], say, is to jump to the destination "Even", whereas R [9, Even, Odd] jumps to the destination "Odd". Note that each call of R ultimately has its two label parameters specifying the same point N in two different activations of R.

Once the correct activation has been discovered, the point at which it is to be resumed may still lie within blocks, conditional-commands or cycle-commands which are not yet activated. (The rules of 9.3.3. will imply that these are the only possibilities; in particular, note that the effect of attempting to jump into an expression is undefined.) These activations are achieved by successively "jumping into" the commands, starting with the largest command which is not yet activated and working inwards.

The effect of jumping into the command

$$\underline{\text{if}}\ E_1\ \underline{\text{then}}\ C_1$$

is identical with the effect of jumping into the result of replacing this conditional command, until terminated, by C_1.

The effect of jumping into the command

$$\underline{\text{for}}\ N_1 = F_1\ \underline{\text{do}}\ C_1$$

is to execute the command as written, with the exception that on the first cycle, if any, C_1 is entered by continuing the jump into it, after setting the initial value of N_1.

On jumping into a block, any initialisations are performed as if entering the block normally, before continuing the jump further into the body of the block.

If, while evaluating the expressions occurring in a block or cycle command head for this purpose, a transfer-command is executed which terminates the block or cycle-command, this jump supersedes the former one, which is not pursued further.

The effect of attempting to jump from the head of a block or cycle-command into its body is implementation dependent.

The effects of jumping into other forms of compound-command may be deduced from the rules for expressing such commands in terms of commands of the above form.

The effect of a transfer-command which according to the above rules would terminate a call to the load or update part of a load-update pair, is implementation dependent, and may be undefined. (See Sec.10.3.5.)

9.3.3 Labels

An identifier of type label may be introduced in one of two different ways: by a definition using one of the basic symbols let, and, in and where, as specified in Sections 7 and 8; or by its occurrence as a "command label", in which case it occurs in the text followed by a colon, and preceding some command (or some further command label for a command).

A command label has the status of a definition of type label for the identifier which is used, in that a new data item is associated with that identifier whenever the scope of the command label is entered. The scope is that which a definition by let would have if placed at the head of the smallest routine body or block-expression body in which the command label occurs. The effect when two command label occurrences use the same identifier and have the same scope is undefined (see also Note 1.)

The data item associated with the command label identifier has an R - value which specifies a destination for jumps, providing the information for (a), (b) of 9.3.2 as follows:

(a) the point indicated is the point of the command label occurrence.

(b) the activation of the routine or block-expression body which is being initiated is live to the destination; an activation of a block, conditional-command or cycle-command will be live to the destination if its text contains the command label occurrence, and if called for by a command which is also live to the destination.

The data item associated with a command label is a constant, and may not be updated. But its R - value may be

assigned to variables of type label (declared at the head of a block, say).

Excepting possibly for an implementation dependent set of library labels, every R - value of type label originates as the value associated with a command label in the above way. For the sake of the boolean relations $=, \neq$ applied to labels, two such R - values are equal if they arise from (possibly different) labels for the same command at the same activation. Rules for equality between library labels are implementation dependent.

NOTE

1. For convenience in implementing labels, the interpretation is left implementation dependent in two situations of no great significance.

(a) when, according to the above rules, a command label occurrence does not lie within its own scope.

(e.g.

let routine R be §§let L be real ; L : ..... § § .. go to L .... §)

(b) when the scope is a routine body, and the command label identifier is also used as a formal parameter for the routine.

9.4 Routine-Commands

9.4.1 Syntax

<routine command> ::= <name>|<prefixed expression>

<prefixed expression> ::= <prefixed operator>[<expression>,_]

9.4.2 Semantics

(a) If N_1 is a name the command

N_1

has the same effect as the command

$N_1[]$

(b) If a command is a prefixed-expression, the operator and operand are evaluated in the manner described in Section 5.3.2. The evaluation of the prefixed-operator, in this case, however, must produce a routine. The rules about evaluating the operand and transforming the resulting arguments to the appropriate mode and type are those given in Section 5.3.2.

The application of a routine to its arguments is a form of command; its effect is described in Section 10.1.

Sections 9.5.1 and 9.5.2 not yet written

9.5.3 Break

The basic symbol break is a command which may only appear within the controlled command of a cycle-command. It may not appear in the ~~body of a routine-definition or in the~~ block following the basic symbol update unless the cycle-command immediately enclosing it textually does so also (see Section 10.3).

It has the same effect as a transfer-command to a label immediately following the smallest cycle-command enclosing it textually.

a/

9.5.4 Result-is

The command

result is E

may appear in the block following one of the basic symbols value of, reference of or load. It may not appear in the ~~body of a routine-definition or in the~~ block following the basic symbol update unless the basic symbol with which it is associated does so also (see Section 10.3).

The effect of the command is described in Sections 4.4 and 10.3.5.

These sections may need major revision.

9.6 Conditional-Commands

9.6.1 Syntax

<if command> ::= < if | unless ><expression> do <basic command>

<test command> ::= test <expression> do <command> or
<basic command>

9.6.2 if Commands

The command sequence

if E_1 do C_1
C_2

has the following effect:
The expression E_1 is evaluated in R-mode to produce a result of type Boolean. If the value of E_1 is false, the command C_1 is skipped and the execution continues with the command C_2; if the value of E_1 is true, the execution continues with the command C_1. If the R-value of E_1 is not transformable to type Boolean, the effect is undefined.

The command

unless E_1 do C_1

has the same effect as the command

if $\sim(E_1)$ do C_1

9.6.3 The Test-Command

The command

test E_1 do C_1 or C_2

has the same effect as the command

§ if B_1 do C_1
unless B_1 do C_2 § where $B_1 = E_1$

provided the name B_1 used for the value of the Boolean expression E_1 is chosen to be distinct from all other names occuring in the command.

9.7 Cycle-Commands

9.7.1 Syntax

<while command> ::= < while | until ><expression> do
<basic command>

<repeat command> ::= <command>< repeatwhile | repeatuntil >
<basic expression>|
<command> repeat

<for element> ::= <basic expression>|
step <basic expression>,<basic expression>,
<basic expression>|
<basic expression> to <basic expression>|
<basic expression>,<basic expression>
dotstring<basic expression>

<for list> ::= <for element><,<for element>>$_0$ |
~~<for list> where <in definition>~~

<for command> ::= for <name> = <for list> do <basic command>|
forexternal <individual>< = | := ><forlist> do
<basic command>

<cycle command> ::= <while command>|<repeat command>|
<for command>

Either := or = may be used in either form of for-command

9.7.2 While-Commands

(a) The command

while E_1 do C_2

has the same effect as the command

L_1: if E_1 do § C_1
go to L_1 §

(b) The command

until E_1 do C_1

has the same effect as the command

while ~(E_1) do C_1

9.7.3 Repeat-Commands

(a) The command

C_1 repeat

has the same effect as the command

while true do C_1

(b) The command

C_1 repeat while E_1

has the same effect as the command

§ C_1
unless E_1 do break § repeat

(c) The command

C_1 repeat until E_1

has the same effect as the command

C_1 repeat while ~(E_1)

9.7.4 For-Commands

(a) The commands

for external I_1 := F_1 do C_1 and
for external I_1 = F_1 do C_1

have the same effect as the command

§ let N_1 ≃ E_1
for N_2 = F_1 do § N_1 := N_2
C_1 § §

provided N_1 and N_2 are names which are distinct from any other names occuring in the command.

(b) The command

for N_1 = F_1 do C_1

is interpreted as follows. The for-list F_1 is evaluated in R-mode as described in Section 9.7.5 to specify a sequence of R-values. The name N_1 is taken as the name of a new CONTROL variable whose type is the least upper bound of the types of the elements in this sequence and whose scope is the controlled command C_1.

If the sequence of R-values obtained by evaluating F_1 is void, the for-command has no effect. Otherwise, if the sequence of R-values obtained is V_1, V_2,...V_n ($n \geq 1$) the effect of the for-command is the same as the effect of the command

Does not deal correctly with jump in

$$\$\ \$\underline{\text{let}}\ N_1 = V_1;\ C_1\ \$$$
$$\$\underline{\text{let}}\ N_1 = V_2;\ C_1\ \$$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\$\underline{\text{let}}\ N_1 = V_n;\ C_1\ \$\ \$$$

Note that the values V_1, V_2,...V_n are determined once and for all before the cycle is entered, so that they cannot be altered by assignments within the controlled command, and that the control variable N_1 is redefined with the next value of the sequence at the start of each cycle, so that assignments to N_1 are effective only until the end of the cycle in which they are made.

A jump out of the controlled command either by a transfer-command to a label external to it or by a break command, has the effect of jumping out of a block so that the control variable N is lost.

A jump into the controlled command is treated like a jump into a block (see Section 9.8) and the evaluation of the for-list is treated as part of the declarations at the head of the block. More precisely, the effect of the commands

$$\underline{\text{go to}}\ L_1$$
$$\ldots\ldots\ldots$$
$$\underline{\text{for}}\ N_1 = F_1\ \underline{\text{do}}$$
$$\$\ C_1\ ;\ldots\ldots;\ L_1 :\ C_2;\ldots\ldots\$$$

is the same as that of the commands

$$\underline{\text{go to}}\ L_2$$
$$\ldots\ldots\ldots$$
$$L_2 :\ \$1\ \underline{\text{let}}\ B_1 = \underline{\text{True}}$$
$$\underline{\text{for}}\ N_1 = F_1\ \underline{\text{do}}$$
$$\$2\ \underline{\text{if}}\ B_1\ \underline{\text{do}}\ \$3\ B_1 := \underline{\text{False}}$$
$$\underline{\text{go to}}\ L_1\ \$3$$
$$C_1\ ;\ldots\ldots;\ L_1 :\ C_2;\ldots\ldots\$2\ \$1$$

provided the new names introduced are distinct from all others. Note that this equivalence, unlike many given elsewhere in this section, is only true dynamically. There is no simple textually equivalent sequence of commands as the interpretation of L_1 as a label in a transfer-command

depends on its context, i.e. on whether the transfer-command is inside or outside the for-statement which contains L_1.

9.7.5 Evaluation of For-Lists

The sequence of values specified by a for-list is the sequence of R-values of its component for-elements taken in order from left to right. A for-element which is a basic-expression specifies a single value; the other forms specify an arithmetic progression (which may have no members) determined as follows:

(a) The for-element

$$\underline{\text{step}}\ E_1,\ E_2,\ E_3$$

where E_1, E_2 and E_3 are numerical expressions specifies a sequence with n+1 values where $n=\text{Integer}[\text{Realpt}[(E_3-E_1)/E_2]]$ if $n \geq 0$ and an empty sequence if $n < 0$. Here the function Realpt gives the real part of its argument for complex numbers, and the function Integer gives the nearest integer to its argument (see Appendix 3). If $n \geq 0$ the sequence of values specified is given by E_1+rE_2 for $r = 0,1,\ldots,n$. The type of each element of a sequence which is not void is the type of the expression E_1+E_2.

Round/

(b) The for-element

$$E_1\ \underline{\text{to}}\ E_2$$

specifies the same sequence as the for-element

$$\underline{\text{step}}\ E_1,\ 1,\ E_2$$

(c) The for-element

$$E_1,\ E_2,\ldots,E_3$$

specifies the same sequence as the for-element

$$\underline{\text{step}}\ E_1,\ E_2-E_1,\ E_3$$

~~(d) If the for-list is qualified by a where-clause, the scope of the definition is the for-list only. Thus in the command~~

~~$$\underline{\text{for external}}\ E_1 := F_1\ \underline{\text{where}}\ D_1\ \underline{\text{do}}\ C_1$$~~

~~the scope of D_1 includes all the elements of F_1 but does not include either F_1 or C_1.~~

10 FUNCTIONS AND ROUTINES

10.1 Introduction. Function and Routine Calls

10.1.1 Syntax

<prefixed operator> ::= <name>| (<expression>) | rhs |
<prefixed expression>

<prefixed expression> ::= <prefixed operator>[<expression>,..]

<routine command> ::= <name>|<prefixed expression>

10.1.2 Semantics

A function is a representation of a rule for evaluating function calls, which are forms of expression (see Section 5.3). A routine is a representation of a rule for obeying routine calls, which are forms of command (see Section 9.4).

Functions and routines fall into the following categories:

(a) Programmers' functions and routines: functions and routines introduced by an activation of one of the special forms for definition to be described in Section 10.3.

(b) Basic functions and routines: any of the functions and routines described in Appendix 3 to this manual, introduced into the program by use of the corresponding name, in a context where it is not subject to any definition of that name.

(c) Library functions and routines: an implementation dependent function or routine, introduced by an undefined occurrence of a name which does not correspond to any basic function or routine name. It is assumed that any published program will be accompanied by an adequate, possibly informal account of these functions and routines. In particular, it must be possible to ascertain the data type of any expression involving a reference to such a function or routine.

All functions and routines are regarded directly as data items, of one of the function types (see Section 10.2.2) or of type routine.

A function call is any prefixed-expression in which the prefixed-operator is an expression of one of the function types. A routine call is a routine command consisting either of a single name of type routine, or of a prefixed-expression in which the prefixed-operator is an

expression of type routine. The single name routine call is interpreted as a call without parameters to the routine which is the value of the name. Other function and routine calls indicate an 'actual parameter list', which is the possibly empty list of expressions which follows the prefixed-operator, enclosed in square brackets.

To evaluate a function call, or to obey a routine call, the R-value of the function or routine is obtained, by evaluating the prefixed-operator of the prefixed-expresion (or the single identifier, in the parameterless routine call). The rule represented by this R-value is then invoked using the ordered sequence of data items obtained by evaluating the actual parameter expressions for the call (in L-mode or in R-mode, depending on the mode of call of each parameter, see Section 10.3.3). In the case of a routine, this rule determines a sequence of commands to be obeyed. In the case of a function, the rule determines an evaluation process, from which a list of results is obtained. There are two forms of this rule; one determines how the evaluation is to be carried out in L-mode, and the other in R-mode. The choice between these forms is determined by the mode of evaluation of the function call as an expression.

10.2 Functions and Routines as Data Items

10.2.1 Syntax

<function type> ::= <single type>$_1$ function

<single type> ::= <data type>|<array type>|
<function type>| (<single type list>)

<single type list> ::= <single type><,<single type>>$_0$

10.2.2 Types

There is just one type of routine, the type routine.

There are an infinite number of types of function, two functions having the same data type if and only if they produce the same type of result, or produce results which match in number and in types, if they produce more than one result each.

As data items, those basic and library functions which are polymorphic as to their results (whose result types depend on the types of their parameters), are considered as functions of result type general, or (general, general), or (general, general, general) etc. depending on the number of results. Polymorphic functions with varying numbers of results are not considered as data items, and may only occur in the position of single name prefixed-operators.

Programmers' functions and routines may not be polymorphic. The numbers and types of results and parameters are specified in the special forms for function and routine definition (see Section 10.3).

The data type of a function is symbolised by preceding the basic symbol function by the result types in order, surrounded by parentheses and separated by commas if there is more than one result. Functions specified as type function, without any specification of result types, are assumed to have one result each, of the preferred type. Any of the data types in the result type list may itself be a functional type. For example, the following is an acceptable declaration:

let f be (real, index function, function) function

Values of f within the scope of this definition must be functions producing three results each; in order, a real, and index-valued function, and a function with a single value of the preferred type.

Transfer functions may be defined between those function types for which there are transfer functions between result and parameter types. None of these is implicitly invoked in cases of mismatching, e.g., if a function is assigned to a function variable of a different result type.

Note that the type of a function or routine does NOT depend on the number, types, or modes of calling of parameters, nor on whether the function or routine is fixed or free (see Section 10.3.4).

10.2.3 Expressions and Assignments

The R-value of an expression of a function or routine type is a function or routine which may be applied, assigned, or used as parameter in a function or routine call. At any point in a computation, any function or routine R-value which may arise must have originated in one of the following ways:

(a) as the R-value of a basic or library function or routine.
(b) as an R-value obtained from a call to a basic or library function whose result type involves a function or routine type.
(c) as the initial R-value of a function or routine identified defined previously using one of the special forms of function routine definition to be described in Section 10.3.

Such an R-value may be assigned to any function or routine variable for future assignment or application.

The rules for applying R-values of types (a) and (b) are to be obtained from the specifications of the basic or library functions or routines involved. The rules for applying R-values of type (c) are described in Section 10.3.

In a function or routine assignment, what is assigned is a representation of a computational rule; the rule itself remains unaltered. Thus, if f, g are function identifiers of the same result type, any sequence

```
f := g ; b := g[a]
```

in a program can be replaced without altering the interpretation of the program, by the sequence

```
f := g ; b := f[a]
```

10.2.4 Equality between Functions, Routines (see Note 1)

The only infixed relational operators between function expressions or between routine expressions are the operators

$$= \quad \neq$$

Before defining a notion of equality between function or routine R-values, it will be necessary to define a notion of EQUIVALENCE between two L-values of the same data type. This notion, and the notion of equality between function or routine R-values, will be seen to be interdependent.

Firstly, it should be noted that the only ways in which L-values may arise are the following:

(aa) as constant L-values, i.e., as the L-values of constant unalterable R-values.

(bb) as the L-values allotted to local variables, or to parameters called by value, or to a data item without an L-value (see Section 4.2).

(cc) as results of calls to LH functions.

(dd) as results of calls to basic or library functions.

There are four corresponding rules to define the equivalence of two L-values. Two L-values are equivalent if and only if they are of the same data type and satisfy one of the following conditions:

(aa') they are both constant L-values, whose R-values are equal.

(bb') they are both the same L-value, i.e., the L-value allotted at the same activation to a local variable or parameter called by value, ~~or in an R-value to L-value transfer.~~

(cc') they both result from calls to LH functions, and are compounded from load functions and update routines which are equal in the sense of this section.

(dd') one L-value results from a call to a basic or library function, whose specifications, together with the rules of this section, imply that it is identical with the other L-value.

To define equality between two function or routine R-values, it is necessary, in the same way, to refer to the origins of the two R-values, as analyzed in Section 10.2.3. Two function or routine R-values are equal if and only if one of the following conditions hold:

(a') they are both R-values of the same basic or library function or routine.

(b') both R-values originate from the same special function or routine definition, possibly at different activations, and the L-values of corresponding free variables are equivalent. (see Note 2)

(c') one of the R-values results from a call to a basic or library function, whose specifications, together with the rules of this section, imply equality with the other R-value.

NOTES

1. This specification is given in its present form with an eye to compatibility with any extension of CPL which might deal with more complicated forms of data structure than are at present incorporated. These may possibly be viewed as types of free function, in which case the definition of equality between such functions is of importance. Until that time, the feature is of small significance, and deviant interpretations of this equality notion might not be disasterous. In particular (b') might be amended to:

(b") both R-values originate from the same activation of the same special function or routine definition.

without the roof caving in.

2. In particular, if the two R-values are fixed, they are equal if and only if the R-values of all free variables were equal at defintion time.

10.3 Function and Routine Definitions

10.3.1 Syntax

```
<LH function body> ::= < fix <definition>>,-
                          load <block> update <block>

<function definition> ::= < recursive >,- < variable | constant >,-
                            <name>[<formal parameter list>,-]
                                  < = | = ><expression list>|
                          < recursive >,- < variable | constant >,-
                            < fixed | free >,-<function><name>
                            [<formal parameter list>,-] be
                            <<LH function body>|
                              § <LH function body> $ >

<routine definition> ::= < recursive >,- < variable | constant >,-
                           < fixed | free >,-<routine><name>
                           <[<formal parameter list>,-]>,-
                               be <block>
```

10.3.2 Semantics. General

Functions and routines may be defined by type, or by value or by reference, in any of the standard ways described in Section 7. These forms of definition, however, can only create a new function or routine R-value by a call for one of the basic or library functions; in this case the computational rule specified by the R-value will be determined by the specification of the basic or library function which produces it. Definitions of the special forms of function and routine definitions whose syntax is given in Section 10.3.1 can also produce new function or routine R-values. Each instance of such a definition characterizes a computational rule, and, when activated, sets up a function or routine R-value which represents that rule.

Function and routine definitions have the following features:

(a) an identifier, the name which is the subject of the definition.
(b) a list of identifiers and basic symbols, following (a), enclosed in square brackets, and possibly empty; this is the FORMAL PARAMETER LIST for the function or routine definition.

(c) a BODY; that part of the definition following an occurence of one of the basic symbols

be = ≡

The remaining parts of function and routine definitions have the following significances:

variable signifies that the function or routine is a variable data item which may subsequently be updated. Otherwise the data item is assumed to be constant, with an unalterable R-value. The latter is also signified by using the basic symbol constant in the definition.

function...be are used in LH function definitions (see Section 10.3.5).

routine...be are used in routine definitions.

fixed signifies that the LH function or routine is fixed (see Section 10.3.4).

free signifies that the LH function or routine is free (see Section 10.3.4).

If neither fixed nor free occur in a LH function or routine definition, it is assumed to be free if it has any free variables, and fixed if it has none.

= signifies an ordinary definition of a fixed function.

≡ signifies an ordinary definition of a free function.

For ordinary function definitions, the body of the definition is an expression list; for routine definitions, the body is a command (in the form of a block). In these cases the rule for application of the corresponding function or routine is that the body of its definition is, respectively, evaluated or obeyed, special provision being made for the evaluation of any identifiers which are not defined within the body (which have FREE occurrences in the body, not in the scope of any definition of the identifier concerned within the body). These identifiers are either FORMAL PARAMETERS i.e., those identifiers whose names appear in the formal parameter list of the definition, or FREE VARIABLES i.e., any other identifiers having free occurrences within the body.

10.3.3 Formal Parameters

The identifiers in the formal parameter list are alloted a data type and mode of calling, by use of the basic symbols value and reference, and the data type specifiers described in Section 7.1. The type and mode of calling of any identifier are given by the nearest preceding type specifier and mode specifier (value or reference) in this list. Value signifies that the parameter is CALLED BY VALUE; reference signifies that it is CALLED BY REFERENCE. If there is no preceding type specifier in the list, the type of a parameter is taken to be the preferred type. If there is no preceding mode specifier, the mode is by value.

Before applying a function or routine definition, the formal parameters are provided with L-values, which bear the following relation to the corresponding actual parameters. If the parameter is called by value, its L-value is taken as a fresh L-value, with an initial R-value given by the R-value of the corresponding actual parameter expression. If the parameter is called by reference, it is given the L-value of the corresponding actual parameter expression in the call. (Note the similarity between parameters called by value and local variables defined using =, and between parameters called by reference and local variables defined by ≃ .)

Unless the prefix operator in the function or routine call is a name standing by itself, and occurs as the subject of a special function or routine definition with the attribute constant, the type of each actual parameter exression must be the same as that specified for the corresponding formal parameter in the formal parameter list. In the exceptional case, transfer functions are implicitly invoked where necessary (see Section 5.3.2).

Function and routine calls must result in the same number of actual parameter values as there are formal parameters in the formal parameter list.

10.3.4 Free Variables

The treatment of free variables depends on whether the function or routine is fixed or free. If it is free, the L-values of the free variables are taken to be precisely the L-values of these identifiers at the moment of activation of the definition. If the function or routine is fixed, the free variables are provided, at definition time, with L-values, whose R-values are constant, and are the R-values of those identifiers at definition time. If any free variable in the body of a fixed function or routine is a free array, the constant R-value is taken to be that of a fixed copy of the array, the elements of which are

unalterable (see Section 11.1.2). Fixed functions and routines may not have free variables which are either free functions or free routines or labels. Within the bodies of fixed functions and routines, no assignments may be made to free variables, or to elements of free variable arrays.

10.3.5 LH Functions

In the normal course of events, L-values are disjoint; that is to say, any assignment to one L-value does not affect the R-value associated with any other L-value. This is because L-values are ordinarily created either by activating a definition by type, or definition by =, or at a function or routine call, as the L-value of a parameter called by value, in which case the new L-value is always chosen to be disjoint from all previously created L-values.

In general, however, it is not adequate that any two L-values ae either identical, or disjoint; it is desirable to have ways of creating new L-values, which may SHARE with previously created L-values in arbitrarily complex ways. These are provided by calls to LH functions, and by the special forms for definition provided by LH function definitions.

An L-value may be compounded from two data items of the following sorts:

(a) a parameterless function, of type T_1 function, for some data type specifier T_1; this produces an associated R-value, of data type T_1, when evaluated in R-mode; this is called the LOAD FUNCTION.

(b) a routine with one parameter called by value, of type T_1, this is known as the UPDATE ROUTINE, and, when called, it has the effect of changing the associated R-value to be the R-value of its actual parameter expression.

The result of a call to an LH function is an L-value compounded in this way from a function, defined by the block following the basic symbol load, and a routine defined by the block following the basic symbol update.

To be more precise, where the LH function definition reads:

....load C_1 update C_2 ....

C_1, C_2, standing for blocks, the L-value produced by a call to that LH function is compounded from the function f which would be defined by:

let f[] = value of C_1

and the routine R, which would be defined by:

let routine R[T, x] be C_3

in place of that part of the body of the LH function definition. C_3 here is obtained by substituting x for the basic symbol rhs at all its occurrences in C_2 (outside any nested LH function definition), and T, is the result type of f.

The definition following the basic symbol fix, if it occurs, is activated when the LH function call is evaluated, and has as its scope the remainder of the body. Its purpose is to make it possible to ensure that the L-value concerned (but not, of course, its associated R-value) is not altered by the side effects of other assignment statements between its evaluation at the application of the LH function and its possibly subsequent use to find the R-value or to update the L-value.

LH functions may only be single-valued; i.e., the result type of f as defined above must be a single data type, and not a data type list.

The result of evaluating a call to a LH function in R-mode is otained by evaluating the call in L-mode, and then calling the load function to obtain the current associated R-value.

10.3.6 Determination of Result Types

The data types of functions introduced by special function definitions is determined from the definitions in the following ways:

(a) Non-recursive ordinary functions

The result type list for a non-recursive ordinary function definition is the ordered list of types of constituents of the expression list on the right hand side of the = or = sign. The same rules apply for the types of numerical constants as in simple definitions using = (see Section 7.4.3).

(b) Non-recursive LH functions

Each result type list consists of a single type. When the body of the LH function definition reads:

... load C_1 update C_2 ...

with C_1, C_2 blocks, the type of the LH function is identical with that of the function defined by:

let f[] = value of C_1

The type of rhs in C_2 is taken as the result type of the function as determined in the above manner.

(c) Recursive functions

The result type list for a recursive function, or set of mutually recursive functions cannot necessarily be determined directly from (a) or (b) above, since the type of a defining expression may depend on one of the types to be determined. In this case, the result types are defined as follows:

Given any recursive set of definitions, an assignment of data types to the definitions is CONSISTENT if, assuming those data types on the right-hand side of definitions in the set, the data types for the definitions obtained by following (a), (b) above and the rule of Section 4 are transferable without loss of information to the assumed types. That is to say, corresponding data types that are not functional are directly transferable according to the rules of Section 3.3, and result type lists for functional data types have corresponding components that are also transferable without loss of information in this sense. If, for each definition of the set, there exists a data type which is transferable without loss of information to each data type corresponding to that definition in any consistent assignment of data types to the set, then that is taken as the data type for that definition, provided that in this manner a consistent assignment of data type is obtained. If no unique consistent assignment can be obtained in this way, the data types are implementation dependent.

11 ARRAYS

11.1 Subscripted Expressions. Arrays as Data Items

11.1.1 Syntax

<prefixed operator> ::= <name> | (<expression>) |
<prefixed expression>

<prefixed expression> ::= <prefixed operator>[<expression>,-]

<array type> ::= <single type>,- <number> array |
<single type>,- vector |
<single type>,- matrix

11.1.2 Semantics

An array is a particular sort of function, of one index parameter (by value) defined on just one closed segment of possible parameters. The free variables of an array are in one-to-one correspondence with the set of possible parameter values, and have mutually disjoint L-values; the result of evaluating a call to an array is that free variable which corresponds to the value of the parameter.

Arrays may be either free or fixed.

An R-value of one of the array-types is a representation of an array. The different possible array-types correspond to the different result types of the array, considered as a function, in the following manner:

If T_1 is a single-type, an R-value of type T_1 1 array or T_1 vector represents an array of result type T_1. If T_1 is omitted, the result type is taken as the preferred type. The remaining result types may be deduced from the following synonymies:

matrix, 2 array are synonymous with 1 array 1 array
3 array is synonymous with 1 array 1 array 1 array

and so on. For example, a real 4 array R-value represents an array with result type real 1 array 1 array 1 array, i.e., with result type real 3 array.

A subscripted expression is any prefixed-expression in which the prefixed-operator expression is of an array-type. If A_1 is an expression of some array-type, and $E_1,\ldots,E_n$ are expressions, then

$$A_1[E_1,E_2,\ldots,E_n]$$

is synonymous with

$$A_1[E_1][E_2,\ldots,E_n]$$

The expressions separated by commas are the subscripts of the subscripted expression. A subscripted expression with just one subscript is evaluated in the same way as a function call (see Section 10). If there is more than one subscript, the expression is evaluated in accordance with the reduction to the one subscript case given above.

If the subscript value is outside the range of permitted values, the result is undefined.

The semantics of array expressions, assignments, etc., may be obtained by reference to the corresponding properties for functions (see Section 10.2.3, 10.2.4).

In particular, note the following points:

(a) No automatic copying of arrays is done on assignment. For example, after obeying

$$A_1 := B_1;\ B_1[i] := B_1[i] + 1$$

$A_1[i]$ will have been altered.

(b) Note particularly that equality between arrays is defined in a very strong sense, which, for numerical vectors and matrices, does not coincide with ordinary mathematical usage. Two free arrays are equal only if their free variables have the same L-values; that is to say, only if both have been obtained by sequences of assignments of the same array (only if both are the same array, in every sense). For equality in the mathematical sense between free arrays, the Boolean function Equal must be used (see Appendix 3). Note that two fixed arrays are equal if they are equal in the weak mathematical sense.

There are no special forms available for the definition of arrays, nor are there transfer functions from function types to array types. The only ways in which arrays may be created are as the results of calls to certain basic functions (see Section 11.2.1).

N.B. (a) and (b) above will be seen to be intuitively more acceptable if an array is conceived of, not as the configuration of its elements, but as a POINTER to this configuration. Assignment and equality apply to these pointers rather than to the totalities of elements.

11.2 Basic Functions for Arrays

11.2.1 Array-Creating Functions

(a) Newarray
A call has the form

$$\text{Newarray}[\,T_1, E_1\,]$$

where E_1 is an expression list, and T_1 is a data-type specifier. The function is VARIADIC; that is, it may be called with varying numbers of parameters. On evaluation E_1 produces 2n R-values, $A_1, B_1, \ldots, A_n, B_n$ of type index. The value of the call is then a new array of type

$$T_1 \text{ n array}$$

the L-values in which are disjoint from existing L-values. The values $A_1, B_1, \ldots, A_n, B_n$ determine subscript bounds; after the assignment

$$N_1 := \text{Newarray}\,[\,T_1, E_1\,]$$

the expression

$$N_1[E'_1, E'_2, \ldots, E'_n]$$

is a subscripted expression all of whose subscripts are within bounds if and only if

$$A_i \leq X_i \leq B_i \quad \text{for } 1 \leq i \leq n$$

where each X_i is the R-value of E'_i.

(N.B. Arrays need not be rectangular, although arrays directly produced by Newarray are.)

(b) Formarray:
A call has the form

$$\text{Formarray}\,[\,T_1, E_1\,]$$

with T_1, E_1 as in (a). With the notation of (a), the value of the call is a function of type

$$T_1 \text{ n array function}$$

taking $\prod_{i=1}^{n} (B_i - A_i + 1)$ arguments of type T_1, which, at each call forms a new array; the R-values of actual parameters for the call will be the R-values of elements of the

resulting array arranged in lexicographic order with respect to subscripts. For example, after obeying

N_1 := Formarray [real,(1,2),(1,2),(1,2)][((E_{111},E_{112}),(E_{121},E_{122})),((E_{211},E_{212}),(E_{221},E_{222}))]

each $N_1[i,j,k]$ has the R-value of E_{ijk}, for i,j,k = 1 or 2.

(Note the use of cosmetic parenthesis in the above example; these could be omitted without altering the effect of the command.)

(c) Copy

The value of Copy[E_1], with E_1 an expression of some array-type, is a new free array of the same type, the same spectrum of legitimate subscript sequences, and the same R-values for corresponding elements as the value of E_1. Elements which are arrays are copied.

(d) Fix

The value of Fix[E_1] is a fixed array of the same type as E_1, the same spectrum of subscripts, and the same R-values for corresponding elements. E_1 may be an expression of any array type, except for label n array, (any n). The effect of Fix applied to an array containing a free function or routine is undefined.

11.2.2 Other Functions

(a) Bounds

A function of type (index, index) function; taking one parameter of any array-type. The two values are, in order, the lower and upper bounds for single subscripts of the array (considered as a vector of subarrays).

(b) Equal

A function of type boolean function; taking two parameters of array-types. The value is true if and only if the two arrays are of the same type, have the same spectra of subscript sequences, and equal R-values in corresponding elements.

(N.B. Equal [N_1,Copy [N_1]] always has the value true, but N_1 = Copy [N_1] always has the value false, unless N_1 is a null array.)

This draft has not been fully checked

Appendix 1 The Preprocessor

Part 1 General Description

This is intended to be both a description of the transformation from an implementation alphabet to canonical CPL and a general example of a CPL program. The transformation is written in CPL and appears in Part 2; Part 3 then describes the functions and routines which are used but not declared in Part 2.

The routine Preprocessor uses recursion to model the bracketing structure of the source test; each level of recursion corresponds to a level of brackets. Its actual parameters are the Bracket type (e.g. § ([→), the section bracket tag if the bracket type is § and a label to return to if a closing section bracket is found with a tag which does not match with the current section. It is thus easy to write the preprocessor rule concerned with the insertion of closing section brackets.

The removal of comment is done by the input routine NextChar and since comment is introduced by the pair of symbols || this routine requires a single character buffer which is called InputBuffer. The end of the source stream is recognised by a special end of stream character.

Most of the preprocessor rules are performed by the output routine Output. This routine makes use of the boolean variables TerminatorPending and TerminatorSuppressed to control the insertion or deletion of the command separator ';' . TerminatorSuppressed is true if the most recent symbol processed other than ';' and 'Newline' was the symbol c . TerminatorPending is true if a Newline or ';' has been read since the last canonical CPL symbol was output. The rule for insertion of terminators can most clearly be expressed as follows: a terminator is inserted between two canonical CPL symbols if a terminator is pending and not suppressed by c and if the first symbol can end a command and the second can start a command. It should be noted that all symbols which may end a declaration may also end a command. The last canonical CPL symbol output is always held in the string variable LastOutput. The rules for the insertion of do and the recognition of pos and neg are also incorporated in Output.

The working variable c, is used by Preprocessor and always contains the latest character produced by NextChar. Preprocessor is always called with c set to the next character. The routines ReadTag ReadUnderlinedWord and ReadSpaces all leave c correctly set.

Part 2 The Preprocessor Program

```
let c, s, s' all be string

let LastOutput, InputBuffer both = '' || i.e. the empty string

let TerminatorPending, TerminatorSuppressed = false, true

let rt NextChar [string ref c] be
    §N.ch   c := InputBuffer                                   := / (similarly throughout)

        Readchar [ CPL.Source , InputBuffer]

        If c = '|' = InputBuffer do
          § until c = '*n' do Readchar [CPL.Source,c]
            Readchar [CPL.Source, InputBuffer] §N.ch

let rt Output [string x] be

      §O.p if x = 'c' do § TerminatorSuppressed : = true
                           return §

           if TerminatorPending ∧ ~ TerminatorSuppressed ∧
              CanEndCommand [LastOutput] ∧ CanStartCommand [x] do
                      § Write [ '; ø' ]
                        LastOutput : = ';' §
           TerminatorPending, TerminatorSuppressed both : = false
           x : = MonadicTransformation [LastOutput, x]

           if MustStartCommand [x] ∧ CannotProceedCommand [LastOutput]
                   do § Write [ 'do ø' ]
                        LastOutput : = 'do' §

           Write [ x <=> 'ø' ]
           LastOutput : = x      §O.p

let rt ReadTag [string ref s] be

      §R.T   s : = ''

          § NextChar [c]
            unless Letter.Digit.Dot [c] break
            s : = s <=> c § repeat

            until c ≠ '*'' do § s : = s <=> c
                                NextChar [c]   §R.T.
```

```
let rt ReadUnderlinedWord [string ref s] be
    §R.U.W      s := c
                while UnderlinedLetter [c] do
                    § s := s <=> Small.UnderlinedLetter[c]
                      NextChar [c]      § R.U.W.

let rt ReadSpaces [string ref c] be

    §R.S. § NextChar [c]
            If c ≠ '*s' return § repeat     §R.S

let rec rt Preprocessor [string BracketType, BracketTag,
                         label MismatchReturn] be

    §P.P  Switch : Jump on c into                    See Appendix 1 p. 8 for
                                                     definition of switch-command
      §   case '§' : ReadTag [s]
                     Output [ '§' ]
                     Proprocessor [ '§', s, Mismatch]
                     goto Switch

          case '$' : ReadTag [s]
                     if BracketType ≠ '§' do Report [1]
          Mismatch : Output [ '$' ]
                     if BracketTag ≠ s goto MismatchReturn
                     return

          case ';' : case '*n' :
                     TerminatorPending := true
                     NextChar [c]
                     goto Switch

          case '(' : case '[' : case '→' :
                     Output [c] ; s := c
                     NextChar [c]
                     Preprocessor [s, '', Error]
          Error :    goto Switch

          case ')'   unless BracketType = '(' do Report [2]
                     Output [c] ; NextChar [c]
                     return

          case ']' : unless BracketType = '[' do Report [3]
                     Output [c] ; NextChar [c]
                     return

          case ',' : if BracketType = '→' do
                                § Output ['comma'] ; NextChar [c]
                                  goto Switch §                  Return
                     ReadSpaces [c]
                     unless c = '.' do § Output [','] ; NextChar [c]
                                         goto Switch §
```

```
                    NextChar [c]
                    if Digit [c] do
                        §Dgt  Output [',']
                              s : = '.'
                              while Digit [c] do §  s : = s<=> c
                                                    NextChar [c]  $
                              Output ['#' <=> s]
                              goto Switch    $Dgt
                    if c = '*s' do ReadSpaces [c]
                    if c ≠ '.'  do Report [4]
                    while c = '.' do ReadSpaces [c]
                    unless c = ',' do Report [5]
                    Output ['dot string'] ; NextChar [c]
                    goto Switch

case '*'' :         s : = ''
                    NextChar [c]
                    until c = '*'' do
                        §1  test EscapeCharacter [c]
                                thendo §  NextChar [c]
                                          s : =$<=> SpecialChar [c]  $
                            or do s : = s <=> c
                            NextChar [c]   §1
                    Output ['#' <=> s] ; NextChar [c]
                    goto Switch

case '<' :          NextChar [c]
                    if c = '<' do § Output ['<<'] ; NextChar [c]
                                    goto Switch       $
                    if c = '=' do
                          § NextChar [c]
                            if c = '>' do § Output ['< = >'] ; NextChar[c]
                                            goto Switch    $
                            Report [6]   $
                    Output ['<']
                    goto Switch

case '>' :          NextChar [c]
                    if c = '>' do §  Output ['>>'] ; NextChar [c]
                                     goto ~~Next~~    $
                    Output ['>']
                    goto Switch

case ':' :          NextChar [c]
                    if c = '=' do §  Output [':='] ; NextChar [c]
                                     goto switch    $
                    Output [':']
                    goto Switch
```

```
case '*s' : case '_' :   NextChar [c]
                         goto Switch

the rest :  if BigLetter [c] do
                  §      s : = c
                         ReadTag [s']
                         Output ['*' <=> s <=> s']
                         goto Switch    §
            if SmallLetter [c] do
                  §      s : = c
                         NextChar [c]
                         while c = '*'' do §   s : = s <=> c
                                               NextChar [c]   §
                         Output ['*' <=> s]
                         goto Switch    §
            if Digit.Dot [c] do
                  § s : = ' '
                         while Digit.Dot [c] do §  s : = s <=> c
                                                   NextChar [c]   §
                         Output ['*' <=> s]
                         goto Switch    §
            if UnderlinedLetter [c] do
                  §1     ReadUnderlinedWord [s]
                         s : = Standard [s]
                         while Combinable [s] do
                               §2 if c = '*s' do ReadSpaces [c]
                                  unless UnderlinedLetter [c] do
                                                §  Output [s]
                                                   goto Switch §
                                  ReadUnderlinedWord [s']
                                  s' : = Standard [s']
                                  test  Standardizable [s <=> s']
                                    thendo s : = standard [s <=> s']
                                    ordo §  Output [s]
                                            s : = s'    § 2
                         Output [s]
                         goto Switch    §1
            if c = Endof StreamChar do
                        § unless BracketType = 'StreamStart' do
                                                  Report [7]
                          return    §
            Output [c]
            NextChar [c]
            goto Switch    §P.P
```

```
!! Main program
LastOutput, InputBuffer both : = ''
NextChar [c] ; NextChar [c]
Preprocessor ['StreamStart', '', Error] ; Error : finish
```

Part 3 Additional definitions.

A number of identifiers in Part 2 were left undefined; these are defined informally in this section.

a) The boolean function CanEndCommand [string LastOutput] is true if LastOutput is any of the following:-
Number, Name, StringConstant §]) :
repeat break return rhs finish
index integer real etc.

b) The boolean function CanStartCommand [string x] is true if x is any of the following
Number, Name, StringConstant
pos, neg, + - (§
if unless while until test for forext
break finish return resultis valof refof goto

c) CannotPreceedCommand [string LastOutput] is true if LastOutput is any of the following:-
; § do : or

d) MustStartCommand [string x] is true if x is any of the following:-
if unless while until test
for forext return break finish
resultis goto

e) The string function MonadicTransformation [stringLastOutput, x] is used to recognise when + and - are monadic; if x is not + or - then the result is x, if LastOutput is not any of the following:-
number, name,] , § ,)
and x is + or - then the result is pos or neg respectively.

f) Standard [string x) is a string function whose result is

if x is	are or bothbe
allbe	is
be	constant
const	then thendo dodo
do	forexternal
forext	goto
go	ordo else otherwise
or	recursive
rec	reference
ref	referenceof
refof	resultbe
resultis	

rt	routine
to	thru
val	value
valof	value of

otherwise the result is x

g) Combinable [string x] is true if x is
all d do double for
go long or ref
result repeat val

h) Standardizable [string x] is true if the result of Standard [x] is an underlined word in the basic symbol set of canonical CPL.

i) Letter.Digit.Dot [string c] is true if c is a letter or a digit or a dot.

j) Digit.Dot [string c] is true if c is either a digit or a dot.

k) Digit (string c] is true if c is a digit.

l) EscapeCharacter [string c] is true if c is the escapecharacter for strings.

m) SpecialChar [string c] transforms the special character which occurs after the escape character in strings to the string character it represents.

e.g. the result is the character	'Newline' if c is	n
	'Backspace'	b
	'Erase'	e
	'	'
	etc.	

The switch-command

Syntax

<switch command> ::= switch on <expression> into
<block>
<label> ::= <identifier> : | case <constant> : | default :

The synonyms for switch on are :-
caseswitch and jumpon
and for default are :-
the rest and others

Semantics

The switch-command is a convenient way of controlling a multiway decision in CPL; it is designed as an extension to the test-command which switches to one of two alternatives depending on the value of a boolean expression.

The type of the switch is the type of its controlling expression and this may be string or any of the numerical or logical types. A case-label is said to belong to the smallest enclosing lexicographic switch-command and its constant is transformed to the type of this switch-command.

To execute a switch-command first the controlling expression is evaluated in R mode and then execution is resumed at the case-label which belongs to the switch-command and whose constant equals this value, or if no matching case exists execution continues at the command labelled default: . The constants of two case-labels belonging to the same switch-command may not be equal.

The switch-command has not yet been incorporated in the rest of the reference manual. It is intended to include it in the language.

APPENDIX 2 4

List of Changes Foreseen in CPL

A. Alterations

1. Scope of where-clauses

This is somewhat confusing and unsatisfactory at present. We shall probably distinguish between an "expression-where" and a "command-where" (which probably includes a "definition-where") by some syntactic device similar to that which picks out the conditional commas and replaces them at the preprocessor stage by comma.

2. result is, return and break

"Detached" uses of these will be allowed with a scope which is lexicographically determined. These correspond, inter alia, to Landin's "program points".

3. finish

This exists only in the Elementary Manual. Its use will be extended and generalised.

B. Additions

1. Character

We hope to introduce a type character into the language.

2. Switch-commands

The switch-command described on p.8 of Appendix 1 will be added.

3. Table look-up

Commands for Table look-up seem to be desirable. These may prove to be part of the operations on sets (see next section).

C. Extensions

1. Compound Data Structures

The entire area of Compound Data Structures and related concepts needs to be added to CPL. This will involve a considerable extension of the concept of type. An outline scheme is already under consideration, but much more work is needed before it can be regarded as good enough for incorporation in the language.

2. Segmentation

Methods for segmenting large programs, compiling and testing the segments separately and finally combining them into a larger program are essential to the practical utility of any programming language. We hope to be able to include these facilities in CPL by a natural extension of the language rather than an *ad hoc* device.

3. Sets

It may be possible and desirable to introduce the concept of a set (an unordered collection of identifiable objects) and to provide certain operations on them. (For example a table look-up is very close to a function whose domain is the member of a set).