Technical Memorandum No 69/1

The BCPL Reference Manual

by

Martin Richards

January 1969

The University Mathematical Laboratory
Corn Exchange Street,
Cambridge.
Abstract

BCPL is a language which is readable, easy to learn and efficient. It is made self consistent and easy to define accurately by an underlying semantic structure based on a simple idealized object machine. The treatment of data types is unusual and it allows the power and convenience of a language with dynamically varying types and yet the efficiency of FORTRAN. BCPL has been used successfully to implement a number of languages and has proved to be a very useful tool for compiler writing. The BCPL compiler itself is written in BCPL and has been designed to be easy to transfer to other machines; it has already been transferred to more than seven different systems.
Contents

1.0 Introduction

2.0 Hardware Representations and Syntax
  2.1 Table of Canonical Symbols
    2.1.1 Syntactic Notation
  2.2 The Canonical Syntax of BCPL
  2.3 Hardware Representations
    2.3.1 Names and System Words
    2.3.2 Section Brackets
    2.3.3 Automatic Insertion of SEHICOLON
    2.3.4 Automatic Insertion of DO
    2.3.5 Comments
    2.3.6 The 'get' Directive
    2.3.7 An Example Program

3.0 Fundamental Concepts of BCPL
  3.1 The Object Machine
  3.2 Variables and Manifest Constants
  3.3 Ivalues and Modes of Evaluation
  3.4 Simple Assignment
  3.5 The Iv Operator
  3.6 The Rv Operator
  3.7 Data Structures
  3.8 Data Types
4.0 Expressions

4.1 Primary Expressions
  4.1.1 Names
  4.1.2 Numbers
  4.1.3 String Constants
  4.1.4 Character Constants
  4.1.5 Truth Values
  4.1.6 Bracketed Expressions
  4.1.7 Result Blocks
  4.1.8 Lv Expressions
  4.1.9 Rx Expressions
  4.1.10 Vector and Structure Applications
  4.1.11 Function Applications

4.2 Arithmetic Expressions
4.3 Relational Expressions
4.4 Shift Expressions
4.5 Logical Expressions
4.6 Conditional Expressions
4.7 Tables
4.8 Constant Expressions
5.0 Commands

5.1 Simple Assignment Commands
5.2 Assignment Commands
5.3 Routine Commands
5.4 Labelled Commands
5.5 Goto Commands
5.6 If Commands
5.7 While Commands
5.8 Test Commands
5.9 Repeat Commands
5.10 For Commands
5.11 Break Commands
5.12 Finish Commands
5.13 Return Commands
5.14 Resultis Commands
5.15 Switchon Commands
5.16 Blocks

6.0 Definitions and Declarations

6.1 Scope and Scope Rules
6.2 Extent and Space Allocation
6.3 Let Declarations
6.3.1 Simple Variable Definitions
6.3.2 Vector Definitions
6.3.3 Function and Routine Definitions
6.4 Manifest Declarations
6.5 Static Declarations
6.6 Global Declarations

7.0 Example Programs
1.0 Introduction

ECPL (Basic CPL) is a general purpose programming language which is particularly suitable for large non-numerical problems in which machine independence is important. It was originally designed as a tool for compiler writing and has, so far, been used in three compilers. ECPL is currently implemented and running on CTSS at Project MAC, the GE 635 under GEOS and on MULTICS. There are also ECPL compilers on the RDF 9 at Oxford and on Atlas 2 at Cambridge. Other implementations are under construction.

ECPL is related to CPL (or Combined Programming Language [1, 2]) and was developed using experience gained from work on a CFL compiler.

The ECPL compiler is written in ECPL and is designed for fairly easy transfer to any other machine. Where possible the implementation dependent parts of the compiler have been separated out, and so only a small proportion (about 1/5th) of the compiler needs to be rewritten for a new implementation. In addition to modifying the compiler, it is necessary to design and write the interface with the new operating system; this is usually written in assembly language and its length is likely to be between 200 and 1000 instructions.

The cost of transferring ECPL to new machine is usually between 2 and 5 man months.

1/1

15/1/69
2.0 Hardware Representations and Syntax

Since BCPL is implemented on many machines having different hardware character sets, it is useful to separate the machine dependent hardware representation of a BCPL program from the canonical syntax of the language. The details of the hardware representation provided for any implementation can be found in the corresponding implementation notes. In this chapter we give the machine independent canonical syntax of BCPL and provide guidelines on which any hardware representation should be based.

A BCPL program can be thought of as a stream of canonical symbols laid out on a page. The canonical symbols are the basic words, operators and symbols of the language and they are the terminal symbols of the canonical syntax. Some canonical symbols are given below:

\[ \text{let} \quad \text{and} \quad "P3\text{-'n}" \quad 36 \quad \& \quad + \quad ; \quad \text{while} \]

Different hardware representations of the same canonical symbol does not affect its meaning. Thus the symbol let may equally well be represented in different implementations by any of the following:

\[ \text{let} \quad \text{LET} \quad \text{let} \quad \text{.LET} \]

### 2.1 Table of Canonical Symbols

The table given below gives the names of all the ACHL canonical symbols together with some examples of their possible hardware representation.

<table>
<thead>
<tr>
<th>Name of Canonical Symbol</th>
<th>Hardware Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>103 2 6000</td>
</tr>
<tr>
<td>NAME</td>
<td>Abc 112 i Tax_rate</td>
</tr>
<tr>
<td>STRINGCONST</td>
<td>&quot;A&quot; &quot;36+t&quot;</td>
</tr>
<tr>
<td>CHARCONST</td>
<td>'p' 'n'</td>
</tr>
<tr>
<td>TRUE</td>
<td>true false</td>
</tr>
<tr>
<td>FALSE</td>
<td>Oct</td>
</tr>
<tr>
<td>OCT</td>
<td>$8 ex $8</td>
</tr>
<tr>
<td>VALUF</td>
<td>Valof</td>
</tr>
<tr>
<td>VECAP</td>
<td>V!</td>
</tr>
<tr>
<td>LV</td>
<td>Lv</td>
</tr>
<tr>
<td>RV</td>
<td>Rv</td>
</tr>
<tr>
<td>DIV</td>
<td>/</td>
</tr>
<tr>
<td>R24</td>
<td>ran</td>
</tr>
<tr>
<td>MULT</td>
<td>*</td>
</tr>
<tr>
<td>PLUS</td>
<td>+</td>
</tr>
<tr>
<td>MINUS</td>
<td>-</td>
</tr>
<tr>
<td>EQ</td>
<td>= eq</td>
</tr>
<tr>
<td>NE</td>
<td>&gt; ne</td>
</tr>
<tr>
<td>LS</td>
<td>&lt; la</td>
</tr>
<tr>
<td>GR</td>
<td>&gt; fr</td>
</tr>
<tr>
<td>LE</td>
<td>&lt; le</td>
</tr>
<tr>
<td>GE</td>
<td>&gt; ge</td>
</tr>
<tr>
<td>NOT</td>
<td>= not</td>
</tr>
<tr>
<td>LSHIFT</td>
<td>lshift</td>
</tr>
<tr>
<td>RSHIFT</td>
<td>rshift</td>
</tr>
<tr>
<td>LOGAND</td>
<td>&amp; logand</td>
</tr>
<tr>
<td>LOGOR</td>
<td>v logor</td>
</tr>
<tr>
<td>EQV</td>
<td>= eqv</td>
</tr>
<tr>
<td>NEQV</td>
<td># neqv</td>
</tr>
<tr>
<td>COND</td>
<td>+ cond</td>
</tr>
<tr>
<td>COMMA</td>
<td>, table</td>
</tr>
<tr>
<td>TABLE</td>
<td>end</td>
</tr>
<tr>
<td>ASS</td>
<td>:=</td>
</tr>
<tr>
<td>Name of Canonical Symbol</td>
<td>Hardware Examples</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>GOTO</td>
<td>goto</td>
</tr>
<tr>
<td>RESULT</td>
<td>result</td>
</tr>
<tr>
<td>COLON</td>
<td>:</td>
</tr>
<tr>
<td>TEST</td>
<td>test</td>
</tr>
<tr>
<td>FOR</td>
<td>for</td>
</tr>
<tr>
<td>IF</td>
<td>if</td>
</tr>
<tr>
<td>UNLESS</td>
<td>unless</td>
</tr>
<tr>
<td>UNTIL</td>
<td>until</td>
</tr>
<tr>
<td>REPEAT</td>
<td>repeat</td>
</tr>
<tr>
<td>REPEATAWHILE</td>
<td>repeatwhile</td>
</tr>
<tr>
<td>REPEATAUNTIL</td>
<td>repeatuntil</td>
</tr>
<tr>
<td>BREAK</td>
<td>break</td>
</tr>
<tr>
<td>RETURN</td>
<td>return</td>
</tr>
<tr>
<td>FINISH</td>
<td>finish</td>
</tr>
<tr>
<td>SWITCHCHON</td>
<td>switchon</td>
</tr>
<tr>
<td>CASE</td>
<td>case</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>default</td>
</tr>
<tr>
<td>LET</td>
<td>let</td>
</tr>
<tr>
<td>MANIFEST</td>
<td>manifest</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>global</td>
</tr>
<tr>
<td>STATIC</td>
<td>static</td>
</tr>
<tr>
<td>BE</td>
<td>be</td>
</tr>
<tr>
<td>SECTORIA</td>
<td>$(6)</td>
</tr>
<tr>
<td>SECTORST</td>
<td>$(Trans)</td>
</tr>
<tr>
<td>HINA</td>
<td>()</td>
</tr>
<tr>
<td>NEXT</td>
<td></td>
</tr>
<tr>
<td>SEMICOLON</td>
<td>;</td>
</tr>
<tr>
<td>INTO</td>
<td>into</td>
</tr>
<tr>
<td>TO</td>
<td>to</td>
</tr>
<tr>
<td>DO</td>
<td>do then</td>
</tr>
<tr>
<td>OR</td>
<td>or</td>
</tr>
<tr>
<td>VEC</td>
<td>vec</td>
</tr>
</tbody>
</table>

The symbols NUMBER, NAME, STRINGCONST, CHARCONST, SECTORIA and SECTORST denote composite symbols which have associated variable length parts.

Throughout this manual syntax and programming examples will be given in some suitable hardware representation.
2.2.1 Syntactic Notation

The syntax given in this manual is Backus Naur Form with the following extensions:

(1) Some common syntactic categories are not surrounded by meta linguistic brackets.

(2) The symbols $\{\}$ and $\{\}$ are used to indicate repetition, for example:

$$E \{ , E \}^*$$ means

$$E \mid E, E \mid E, E' \mid E \ldots \ldots \ldots$$

The syntax given in the next section is ambiguous and is simply intended to list all the syntactic constructions available. The ambiguities are resolved later in the manual.
2.2.2 The Canonical Syntax of ECPL

\[ E ::= \text{<name>} | \text{<stringconst>} | \text{<charconst>} | \text{<number>} | \text{true} | \text{false} | (E) | \text{valof} \text{<block>} | \text{lv} E | \text{rv} E | E(E(<E\text{list}>)) | E() | E \text{<diadic op>} E | \text{<monadic op>} E | E + E, E | \text{table} <\text{constant}> \frac{1}{2}, <\text{constant}> \frac{3}{2}\]

<diadic op> ::= \times | / | \text{rem} | + | - | = | \neq | \leq | \geq | \text{ro} | \text{ishift} | \text{rshift} | \text{\&} | \text{\textasciicircum} | \text{\_} | \text{\_}

<monadic op> ::= + | - | \text{not}.

<E list> ::= \text{E}\{, E\}_{m}^{n}

<constant> ::= E

C ::= <E list> ::= <E list> | \text{E}(<E\text{list}>) | E() | \text{goto} E | <\text{name}> : C | \text{if} E \text{do} C | \text{unless} E \text{do} C | \text{while} E \text{do} C | \text{until} E \text{do} C | C \text{repeat} | C \text{repeatuntil} E | C \text{repeatwhile} E | \text{test} E \text{then} C \text{or} C | \text{break} | \text{return} | \text{finish} | \text{resultis} E | \text{for} <\text{name}> = E \text{to} E \text{do} C | \text{switch} E \text{into} <\text{block}> | \text{case} <\text{constant}> : C | \text{default} : C | <\text{block}> | <\text{empty}>
\[ \text{declarations} \]

\[ D : = <\text{name}>(<\text{FPL}>) = \Sigma \mid <\text{name}>(<\text{FPL}>) \text{ be } C \]

\[ <\text{name list}> = <\text{C list}> \mid <\text{name} > = \text{vee} <\text{constant}> \]

\[ <\text{FPL}> : = <\text{name list}> \mid <\text{empty}> \]

\[ <\text{name list}> : = <\text{name}> ; , <\text{name}> \]

\[ <\text{block}> : = \{ <\text{block body}> \} \]

\[ <\text{block body}> : = \text{C} \{ ; \text{C} \} \mid <\text{declaration}> \{ ; \text{C} \} \]

\[ <\text{declaration}> : = \text{let } D \{ \text{ and } D \} \mid \text{static } <\text{decl body}> \]

\[ \text{manifest } <\text{decl body}> \mid \text{global } <\text{decl body}> \]

\[ <\text{decl body}> : = \{ <\text{C def} > \{ ; <\text{C def} \} \} \]

\[ <\text{C def}> : = <\text{name} > : <\text{constant}> \mid <\text{name} > = <\text{constant}> \]

\[ / <\text{program}> : = <\text{block body}> \]
2.3 Hardware Representations

Since the hardware character sets for different implementations differ, it is only practical to give an outline of the hardware conventions which are common to most versions of BSPL.

2.3.1 Names and System Words

System words are sequences of letters used to denote canonical symbols for which there are no suitable graphics. The set of reserved system words is implementation dependent. Names are also mainly composed of letters and may be coined and used by the programmer to denote variables and constants within his program. If the available character set includes small letters then system words and names are syntactically distinct.

For character sets with capital and small letters:

(1) A system word is any sequence of 2 or more small letters,

(2) A name is either
   (a) a single small letter
   (b) a capital letter followed by any sequence of letters, digits and possibly other suitable characters (e.g. # * #)

For character sets with only capital letters:

(1) An identifier is a capital letter followed by any sequence of letters, digits and possibly other suitable characters (e.g. # * #)

(2) Some identifier which is not a system word.

(3) A name is an identifier which is not a system word.
Thus on some implementations let and logor are system words while let, LIT, Logor and LLOGR may be used as names; but with a more restricted character set LIT and LLOGR would be reserved system words and the programmer would have to represent the names in some other way, perhaps by:

```
F_INT, S_INT, F_LOGOR, S_LOGOR
```

2.3.2 Section Brackets

Section brackets are used to bracket blocks and comments. To aid the readability of programs, section brackets may be tagged with any sequence of characters which may occur in identifiers. A closing section bracket matches an earlier open section bracket with the same tag and any outstanding sections will be closed automatically.

For example:

```
$(1 \text{ until } i=0 \text{ do}
  $(2 \text{ R}(i)
    i := i + 1 \text{ } $(1
```

is equivalent to:

```
$(1 \text{ until } i=0 \text{ do}
  $(2 \text{ R}(i)
    i := i + 1 \text{ } $(2 \text{ } $(1
```

2/8.

15/1/69
2.3.3 Automatic Insertion of SEMICOLON

The symbol SEMICOLON is used as a separator for delimiting commands, it is however usually optional and may be omitted in most circumstances. When two commands are juxtaposed both the programmer and compiler can almost always deduce where the first ends and the second starts; however, cases of ambiguity can arise as in the following:

\[ R(x, y)(B \rightarrow f, g)(1, 2) \] // possible omission of ';' before (3

A simple rule which is guaranteed to be safe is:

only omit semicolons between commands if they are written on different lines.

In order that this rule should always work the following minor restriction was imposed:

a diadic operator may not be the first symbol on a line.

Example: the following two programs are equivalent

\[
\begin{align*}
x &:= x + 1 \\
\text{if } x > y \text{ do } y &:= 0 \\
R(x)
\end{align*}
\]

\[
\begin{align*}
x &:= x + 1; \\
\text{if } x > y \text{ do } y &:= 0; \\
R(x)
\end{align*}
\]
2.3.4 Automatic Insertion of DO

As with SEMICOLON the canonical symbol DO is optional except in situations where ambiguity arise from its omission.

Example: the following programs

```
unless 0 ≤ T ≤ Tmax resultis true
if x = 0 goto L
```

is equivalent to

```
unless 0 ≤ T ≤ Tmax do resultis true
if x = 0 do goto L
```

But beware of the very rare situations where DO may not be omitted as in the following

```
if F(x) (B + f_i) (g)(1,2,3) // erroneous omission of do
```

2.3.5 Comments

User's comments may be included in a program between a double slash '//' and the end of the line. Example:

```
let R() be // this is a routine which refills Symb
$( for i = 1 to 200 do
    Readch (INPUT, 1v Symb \ i) $
)$
```
2.3.6 The "Get" Directive

A directive of the form

```
get <specifier>
```

may occur anywhere in a HCL program on a line by itself; it directs the compiler to replace the characters of the directive by the text of the file referred to by the specifier. The syntactic form of the specifier is implementation dependent but is usually a string constant.
2.3.7 An Example Program

// This is an example of a hardware representation of EDPL using
// both capital and small letters.
get 'HEID2'  //This 'gets' the file called HEID2 which presumably
            //declares
            //Checkdistinct, Report and Dvec
let Checkdistinct(E,S) be
 $(1 until E=S do
  $( let p = E + 4

  and N = Dvec\!p // ! represents the VECAP operator
  while p ls S do // Note that ls is a
    // system word, p is a name.
  $( if Dvec\!p = N do Report(142, N)
    p := p + 4 $)

E := E + 4 $)1 // Note that this closes
            // two sections.

2/11 15/1/69
3.0 Fundamental Concepts of BCPL

3.1 The Object Machine

BCPL has a simple underlying semantic structure which is built around an idealised object machine. This method of design was chosen in order to make BCPL easy to define accurately and to facilitate machine independence which is one of the fundamental aims of the language.

The most important feature of the object machine is its store and this is represented diagrammatically in fig. 1.

```
  n     n+1    n+2    n+3
```

**Fig. 1 - The Machines Store**

It consists of a set of numbered boxes (or storage cells) arranged so that the numbers labelling adjacent cells differ by one. As will be seen later, this property is important.

Each storage cell holds a binary bit pattern called an Rvalue (or Right hand value). All storage cells are of the same size and the length of Rvalues is a constant of the implementation which is usually between 24 and 36 bits. An Rvalue is the only kind of object which can be manipulated directly in BCPL and the value of every variable and expression in the language will always be an Rvalue.
Rvalues are used by the programmer to model abstract objects of many different kinds such as truth values, strings and functions, and there are a large number of basic operations on Rvalues which have been provided in order to help the programmer model the transformation of his abstract objects. In particular, there are the usual arithmetic operations which operate on Rvalues in such a way that they closely model integers. One can either think of these operations as ones which interpret their operands as integers, perform the integer arithmetic and convert the result back into the Rvalue form, alternatively one may think of them as operations which work directly on bit patterns and just happen to be useful for representing integers. This latter approach is closer to the BCPL philosophy. Although the BCPL programmer has direct access to the bits of an Rvalue, the details of the binary representation used to represent integers is not defined and he would be losing machine independence if he performed nonnumerical operations on Rvalues he knew to represent integers.

An operation of fundamental importance in the object machine is that of Indirection. This operation has one operand which is interpreted as an integer and it locates the storage cell which is labelled by this integer. This operation is assumed to be efficient and, as will be seen later, the programmer may invoke it from within BCPL using the \texttt{RV} operator.
3.2 Variables and Manifest Constants

A variable in BCPL is defined to be a name which has been associated with a storage cell. It has a value which is the Rvalue contained in the cell and it is called a variable since this Rvalue may be changed by an assignment command during execution. Almost every form of definition in BCPL (including function and routine definition and label declarations) introduce variables. The only exception is the manifest declaration which is used to introduce manifest constants.

A manifest constant is the direct association of a name with an Rvalue; this association takes place at compile time and remains the same throughout execution. There are many situations where manifest constants can be used to improve readability with no cost in runtime efficiency.

3.3 Lvalues and Modes of Evaluation

As previously stated each storage cell is labelled by an integer; this integer is called the Lvalue (or Left hand value) of the cell. Since a variable is associated with a storage cell, it must also be associated with an Lvalue and one can usefully represent a variable diagrammatically as in fig. 2.

```
\[\text{Name} \rightarrow \text{Storage Cell} \]
\[\text{LVALUE} \rightarrow \text{RVALUE} \]
```

Fig. 2. - The form of a variable
Within the machine an Ivalue is represented by a binary bit pattern of the same size as an Rvalue, and so an Ivalue can represent an Ivalue directly. The processes of finding the Ivalue and Rvalue of a variable are called Inode and Rnode evaluation respectively. The ideas of code of evaluation is useful since it applies to expressions in general and can be used to clarify the semantics of the assignment command and other features in the language.

3.4 Simple Assignment

The syntactic form of a simple assignment command is:

\[ E_1 := E_2 \]

where \( E_1 \) and \( E_2 \) are expressions. Loosely, the meaning of the assignment is to evaluate \( E_2 \) and store its value in the storage cell referred to by \( E_1 \). It is clear that the expressions \( E_1 \) and \( E_2 \) are evaluated in different ways and hence there is the classification into the two modes of evaluation. The left hand expression \( E_1 \) is evaluated in Inode to yield the Ivalue of some storage cell and the right hand side \( E_2 \) is evaluated in Rnode to yield an Rvalue; the contents of the storage cell is then replaced by the Rvalue. This process is shown diagrammatically in fig. 3.
Fig. 3. - The process of assignment

The only expressions which may meaningfully appear on the left hand side of an assignment are those which are associated with storage cells, and they are called Ltype expressions.

The terms Lvalue and Rvalue derive from consideration of the assignment command and were first used by Starchey in the CPL reference manual [2].

3.5 The L Operator

As previously stated an Lvalue is represented by a binary bit pattern which is the same size as an Rvalue. The L operator provides the facility of accessing the Lvalue of a storage cell.

The syntactic form of an L expression is:

\[ L \text{E} \]

where \( E \) is an Ltype expression. The evaluation process is shown in fig. 4.
Fig. 4. - The evaluation of an lv expression

The operand is evaluated in Inode to yield an Ivalue and the result is a bit pattern identical to this Ivalue. The lv operator is exceptional in that it is the only expression operator to invoke Inode evaluation, and indeed in all other contexts, except the left-hand side of the assignment, expressions are evaluated in Rnode.

3.6 The Ry Operator

The ry operator is important in BCPL since it provides the underlying mechanism for manipulating vectors and data structures; its operation is one of taking the contents (or Rvalue) of a storage cell whose address (or Ivalue) is given.

The syntactic form of an ry expression is as follows:

ry E

and its process of evaluation is shown diagrammatically in fig. 5.
Fig. 5. — The evaluation of an rv expression

The operand is evaluated in Rmode and then the storage cell whose Lvalue is the identical bit pattern is found. If the rv expression is being evaluated in Rmode, then the contents of the cell is the result; however, it is also meaningful to evaluate it in Imode, in which case the Lvalue of the cell is the result. An rv expression is thus an Ltype expression and so may appear on the left hand side of an assignment command, as in:

```
rv p := t
```

and one can deduce that this command will update the storage cell pointed to by p with the Rvalue of t.
3.7 Data Structures

The considerable power and usefulness of the rv operator can be seen by considering fig. 6.

![Diagram of RV operator]

**Fig. 6. - An interpretation of V + 3**

This diagram shows a possible interpretation of the expression $V + 3$. Some adjacent storage cells are shown and the left hand one has an Lvalue which is the same bit pattern as the Rvalue of $V$. One will recall that an Lvalue is really an integer and that Lvalues of adjacent cells differ by one, and thus the Rvalue of $V + 3$ is the same bit pattern as the Lvalue of the rightmost box shown in the diagram. If the operator rv is applied to $V + 3$, then the contents of that cell will be accessed.

3/8

15/1/69
Thus the expression:
\[ rv \ (V + i) \]
acts very like a vector application, since, as \( i \) varies from zero to three, the expression refers to the different elements of the set of four cells pointed to be \( V \). \( V \) can be thought of as the vector and \( i \) as the integer subscript.

Since this facility is so useful, the following syntactic sugaring is provided:

\[ E_1, E_2 \text{ is equivalent to } rv \ (E_1 + E_2) \]

A simple example of its use is the following command:

\[ V[i+1] := V[i] + U[i] \]

One can see how the \( rv \) operation can be used in data structures by considering the following:

\[ V[3] \overset{\text{def}}{=} rv \ (V + 3) \quad \overset{\text{def}}{=} \quad rv \ (3 + V) \quad \overset{\text{since } + \text{ is commutative}}{=} \quad 31V \]

Thus \( V[3] \) and \( 31V \) are semantically equivalent; however, it is useful to attach different interpretations to them. We have already seen one interpretation of \( V[3] \), so let us consider the other expression. If we rewrite \( 31V \) as \( X \text{part}!V \) where \( X \text{part} \) has value \( 3 \), we can now conveniently think of this expression as a selector (\( X \text{part} \)) applied to a structure (\( V \)). This interpretation is shown in fig. 7.
Fig. 7. - An interpretation of 'part V

By letting the elements of structures themselves be structures, it is possible to construct compound data structures of arbitrary complexity. Fig. 8 shows a structure composed of integers and pointers.

Fig. 8. - A structure of integers and pointers
3.8 Data Types

The unusual way in which ECPL treats data types is fundamental to its design and thus some discussion of types is in order here. It is useful to introduce two classes:

(a) Conceptual types

(b) Internal types

The Conceptual type of an expression is the kind of abstract object the programmer had in mind when he wrote the expression. It might be, for instance, a time in milliseconds, a weight in grams, a function to transform feet per second to miles per hour, or it might be a data structure representing a parse tree. It is, of course, impossible to enumerate all the possible conceptual types and it is equally impossible to provide for all of them individually within a programming language. The usual practice when designing a language is to select from the conceptual types a few basic ones and provide a suitable internal representation together with an adequate set of basic operations. The term internal type refers to any one of these basic types and the intention is that all the conceptual types can be modelled effectively using the internal types.

A few of the internal types provided in a typical language, such as CPL, are listed below

real
integer
label
integer function
(real, boolean) vector

3/11 15/1/69
Much of the flavour of PCPL is the result of the conscious
design decision to provide only one internal type, namely: the
binary bit pattern (or Rvalue). In order to allow the programmer
to model any conceptual type a large set of useful primitive operations
have been provided. For instance, the ordinary arithmetic operators
+,-, *, and / have been defined for Rvalues in such a way as to model
the integer operations directly. The six standard relational
operators have been defined and a complete set of bit manipulating
operations provided. In addition, there are some stranger bit
pattern operations which provide ways of representing functions,
labels and, as we have already seen, vectors and structures. All the
operations provided are uniformly efficient and they have not been
overdefined. For instance, the effect of adding a number to a label,
or a vector to a function is not defined even though it is possible
for a programmer to cause it to take place.

The most important effects of designing a language in this way
can be summarised as follows:

1. There is no need for type declarations in the language, since the
type of every variable is already known. This helps to make
programs concise and also simplifies such linguistic problems as
the handling of actual parameters and separate compilation.
2. It gives BCFL much of the power as a language with dynamically varying types and yet retains the efficiency of a language (like FORTRAN [3]) with manifest types; for, although the internal type of an expression is always known by the compiler, its conceptual type can never be and, indeed, it may depend on the values of variables within the expression. For instance, the conceptual type of $W_i$ may depend on the value of $i$.

One should note that, in languages (such as ALGOL [4] and CIL) where the elements of vectors must all have the same type, one needs some other linguistic device in order to handle more general data structures.

3. Since there is only one internal type there can be no automatic type checking and it is possible to write nonsensical programs which the compiler will translate without complaint. This slight disadvantage is easily outweighed by the simplicity, power and efficiency that this treatment of types makes possible.
4.0 Expressions

All BCPL expressions are described in this section. They are grouped into syntactic classes of decreasing binding power as follows:

(a) Primary expressions

These are the most binding and most primitive expressions, they are:

Names, numbers, truth values, string constants, character constants, bracketed expressions, result blocks, lv expressions, rv expressions, vector applications and function applications.

(b) Arithmetic expressions.

These expressions provide the standard integer operations of multiplication, division, remainder, addition and subtraction. They are less binding than the primary expressions.

(c) Relational expressions.

A relational expression takes integer arguments and yields a boolean value depending on the truth of the relation.

(d) Shift expressions.

The shift operations allow one to shift a binary bit pattern to the left or right by a specified number of places.

(e) Logical expression.

These expressions allow one to manipulate bits of an Rvalue directly. They may be used in conjunction with the shift operators to pack and unpack data. The standard BCPL representations of true and false are chosen so that the logical operators may also
be used on boolean data.

(f) Conditional expressions.
A conditional expression allows for conditional evaluation of one of two expressions.

(g) Tables.
A table is a static vector whose elements are preset with specified values.

4.1 Primary Expressions

All the primary expressions are described in this section.

4.1.1 Names

Syntactic form:
A name is a canonical symbol of ECPL and its hardware representation is implementation dependent. If there are sufficient hardware characters available it consists of any sequence of letters, digits and underlines starting with a capital letter. A single small letter may also be used as a name.

Examples:

H3 Tax_rate F i
List4 StackP

Semantics:
A name may be associated directly with an Rvalue by means of a manifest declaration or it may be associated with a storage cell to form a variable using any other kind of declaration. A variable or manifest.constant can be referred to by its name throughout the
scope of its declaration (see section 6.0 on scopes and extents of definitions).

A manifest constant can only be evaluated in Emode and its result is the Rvalue which was associated with it by its declaration.

A variable is the association of a name with a storage cell and it may be represented as follows:

```
\begin{tikzpicture}
  \node (name) {Name};
  \node (rmode) {Rmode evaluation};
  \node (lmode) {Lmode evaluation};
  \node (rvalue) {Rvalue};
  \node (lvalue) {Lvalue};
  \draw [->] (name) -- (lmode);
  \draw [->] (name) -- (rmode);
  \draw [->] (lmode) -- (lvalue);
  \draw [->] (rmode) -- (rvalue);
\end{tikzpicture}
```

It may be evaluated in Lmode to yield the Lvalue of the storage cell, or it may be evaluated in Rmode to yield the contents of the cell; in either case the result is a bit pattern of standard Rvalue length.

4.1.2 Numbers

Syntactic form: \( <\text{digit}> ::= 0|1|2|3|4|5|6|7|8|9 \)

\( <\text{number}> ::= <\text{digit}> | <\text{number}> <\text{digit}> | 8 <\text{number}> <\text{digit}> \)

Examples: 132 43179 8377

Semantics:

A number is an Rtype expression and may only be evaluated in Rmode. The symbol 8 introduces an octal constant whose Rvalue is the right justified bit pattern specified by the sequence of digits.
A decimal number is a sequence of digits not preceded by \( \overline{0} \); its value is a bit pattern representing the integer in a way which depends on the implementation.

### 4.1.3 String Constants

**Syntactic form:** 
\[
\text{" } \text{<string character>} \text{ "}
\]

A string constant is a canonical symbol of BCPL and its hardware representation is implementation dependent. Where possible it is a sequence of characters enclosed in double quotes (""). The character (\( * \)) is used as an escape character with the following conventions:

- \( *n \) represents newline
- \( *s \) represents space
- \( *b \) represents backspace
- \( *t \) represents tab
- \( *" \) represents ""
- \( *' \) represents 
- \( *\* \) represents \( * \)

**Examples:**

"End of test" "n" "*""
"n*t*RA*tL*n" "*"

**Semantics:**

The R-value of a string constant is a pointer to a set of consecutive storage cells containing the length and characters of the string in some packed form. The number of bits per character
and the number of characters per storage cell are not defined. For an implementation which packs four characters per word, the string

"Abc10n"

might be represented as follows:

<table>
<thead>
<tr>
<th>Rvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>'a'</td>
</tr>
<tr>
<td>'b'</td>
</tr>
<tr>
<td>'c'</td>
</tr>
<tr>
<td>'1'</td>
</tr>
<tr>
<td>'0'</td>
</tr>
<tr>
<td>'n'</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**4.1.4 Character Constants**

*Syntaxic form:* `'<string characters>'`

The same escape conventions that are used in string constants may be used in character constants.

*Examples:* `'
\x1f 0
1
'`

*Semantics:*

Every string alphabet character has an integer code and the Rvalue of a character constant is the Rvalue of its corresponding integer code. The character code is implementation dependent.

**4.1.5 Truth values**

*Syntaxic form:* `true` or `false`

*Semantics:*

The Rvalue of `false` is a bit pattern entirely composed of zeros and the Rvalue of `true` is the complement of `false`, namely a bit pattern entirely composed of ones.
4.1.6 Bracketed Expressions

Syntactic form: \(( E )\)

Examples:
\[ \begin{align*}
& T \text{ rem } ((x-y)/(x+y) + 2/\pi) \\
& (B \rightarrow A, B) \downarrow (i+1)
\end{align*} \]

Semantics:
Parenthesis may enclose any expression without changing its mode of evaluation or its value and their sole purpose is to specify grouping.

4.1.7 Result Blocks

Syntactic form: \( \text{valof } <\text{block}> \)

Example:
\[ \text{valof } \{ \text{for } i=1 \text{ to } n \text{ do} \\
\quad \text{if } P(i,x) \text{ results false} \\
\quad \text{results true } \} \]

Semantics:
A result block is a form of BCPL expression in which commands can be executed before the value of the expression is found. It is evaluated by executing the block until a results statement is encountered; this causes execution of the block to cease and the value of the expression in the results command is the result. See section 5.14.
4.1.8 \textbf{Iv Expressions}

**Syntax form:** \[ \text{Iv} \ E \]
where \( E \) is a primary expression

**Examples:**
- \text{Readch (INPUT, Iv Ch)}
- \( V := \text{Iv V}i \)

**Semantics:**
The Rvalue of an Iv expressions is the bit pattern obtained by evaluating the operand (which must be an Ltype expression) in Lmode. See the discussion of left and right hand values in section 3.3, and of the Iv operator in section 3.5.

4.1.9 \textbf{Rv Expressions}

**Syntax form:** \[ \text{Rv} \ E \]
where \( E \) is primary expression.

**Example:**
- \( \text{Rv x := Rv f(i) + 2} \)

**Semantics:**
An Rv expression is an Ltype expression and may be evaluated to yield either an Ivalue or an Rvalue. It is evaluated by evaluating its operand in Rmode to yield a bit pattern which is interpreted as the Ivalue of a storage cell. In Lmode evaluation this bit pattern is the result, but for Rmode evaluation the contents of the storage cell is the result. The Rv expression in section 3.6.

[4/7] 15/1/69
4.1.10 Vector and Structure Applications

**Syntactic form:** \[ \mathbf{E}_1 \downarrow \mathbf{E}_2 \]

\( \mathbf{E}_1 \) and \( \mathbf{E}_2 \) are primary expressions. The operator is left associative and thus

\[ x \downarrow y \downarrow z \; \text{means} \; (x \downarrow y) \downarrow z \]

**Examples:**

\[ \mathbf{V}_1(i+1) := \mathbf{V}_1 + \mathbf{X}@ip \]

\[ \text{case } \mathbf{E}_2: \begin{align*}
\text{Trans}(\mathbf{H}_2; \mathbf{x}) \\
\text{Trans}(\mathbf{H}_3; \mathbf{x}) \\
\text{return}
\end{align*} \]

**Semantics:**

The expression \( \mathbf{E}_1 \downarrow \mathbf{E}_2 \) is defined to be equivalent to \[ x \downarrow (\mathbf{E}_1 + \mathbf{E}_2) \]

Its purpose is explained in section 3.7.

4.1.11 Function Applications

**Syntactic form:** \[ \mathbf{E}_0(\mathbf{E}_1, \mathbf{E}_2, ..., \mathbf{E}_n) \]

where \( \mathbf{E}_0 \) is a primary expression and \( \mathbf{E}_1 \) to \( \mathbf{E}_n \) are any expressions. The list of expressions may be empty.

**Examples:**

\[ f(x) \]

\[ H(1, 2*t) \]

\[ (x:=0; f, P3)(1, "ZT", y:=2) \]

NextPoint()

**Semantics:**

The evaluation of a function application is explained in section 6.3.5.
4.2 Arithmetic Expressions

Syntactic form: 

\[ E \cdot E \mid E/E \mid E \text{ rem } E \mid E + E \mid E - E \mid + E \mid - E \]

The operators \(*\) / and \(\text{rem}\) are equally binding and associate to the left; they are more binding than + or - which also associate to the left.

Thus

\[ x \cdot y \text{ rem } z \quad \text{means} \quad (x \cdot y) \text{ rem } z \]
\[ x + y \text{ - } z/t \quad \text{means} \quad (x \cdot y) - (z/t) \]

Examples:

\[ 2 \cdot x^2 + 6 \cdot x \cdot y + 7 \cdot y \cdot y \]
\[ V \div (f(x) \text{ rem } 13) + G(x) \]

Semantics:

The arithmetic expressions evaluate their operands in Fmode, the operator then interprets the Rvalues as integers and yields an Rvalue representing the integer result of the arithmetic.

The operator \(*\) and \(/\) denotes integer multiplication and division respectively.

The operator \(\text{rem}\) yields the remainder after dividing the left hand operand by the right hand one. If both operands are positive the result will be positive, it is otherwise implementation dependent.

The expression \(E1 + E2\) yields an Rvalue representing the integer summation of \(E1\) and \(E2\).

The Rvalue of \(+E1\) is the Rvalue of \(E1\).

The expression \(E1 - E2\) yields an Rvalue representing the \(4/9\)
result of subtracting E2 from E1.

The expression -E1 has the same meaning as O-E1.

4.3 Relational Expressions

Syntactic form:

\[ E \{ <\text{relop}> \} E \]

where

\[ <\text{relop}> ::= \{ = \mid \neq \mid \lt \mid \leq \mid \geq \mid \gt \} \]

The relational operators are just less binding than the arithmetic operators.

Examples:

\[ \text{if } 0 \text{ le } x \text{ le } y \text{ goto L} \]

\[ A \text{ i } := f(x)=g(x) \]

Semantics:

All the operands of a relational expression are evaluated in Rmode; the Rvalues obtained are then interpreted as integers and if all the dyadic relations are true then the result of the whole expression is true, otherwise the result is false. The correspondence between the operators and their meanings is given below.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>#</td>
<td>not equal to</td>
</tr>
<tr>
<td>\lt</td>
<td>less than</td>
</tr>
<tr>
<td>\leq</td>
<td>greater than</td>
</tr>
<tr>
<td>\geq</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>\gt</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

4/10

15/1/69
4.4 Shift Expressions

Syntactic form: \[ E_1 \text{ lshift } E_2 \mid E_1 \text{ rshift } E_2 \]

\( E_2 \) is any primary or arithmetic expression and \( E_1 \) is any shift, relational, arithmetic or primary expression; the shift operators are thus less binding than the relations on the left and more binding on the right.

Examples:
\[
\text{let } P(t) = 4|3 \text{ rshift } 10|8377
\]
\[
x := x \text{ lshift } \text{ByteSize} \oplus \text{Ch}
\]

Semantics:

The operands are evaluated in Node to yield Rvalues. The left hand one is interpreted directly as a bit pattern and the right hand one as an integer to indicate the number of places to shift.

The result of \( E_1 \text{ lshift } E_2 \) is the bit pattern produced by shifting \( E_1 \) to the left by \( E_2 \) places. The operator \( \text{rshift} \) is similar to \( \text{lshift} \), only if it shifts to the right. Vacated positions are filled with zeros and the result is undefined if \( E_2 \) is negative or greater than the number of bits in an Rvalue.

4.5 Logical Expressions

Syntactic form:
\[
\begin{align*}
\text{not } & E \mid \\
E \land & E \mid E \lor E \\
E \equiv & E \mid E \neq E
\end{align*}
\]

4/11

15/1/69
The operator \texttt{not} is most binding; then, in decreasing order of binding power, there are:

\[ \land, \lor, \equiv, \not \]

All the logical operators are less binding than the shift operators.

Examples:

\begin{align*}
B & := \texttt{not} \; B \\
\text{if } x=0 \lor y=0 & \quad \text{results } f(t) \\
x & := x \land \not y \quad 770077 \lor y \land \not y \quad 7700
\end{align*}

Semantics:

The operands of all the logical operators are interpreted as binary bit patterns of ones and zeros.

The application of the operator \texttt{not} yields the logical negation of its operand. The result of any other logical operator is a bit pattern whose \textit{n}\textsuperscript{th} bit depends only on the \textit{n}\textsuperscript{th} bits of the operands and can be determined from the following table.

| \text{nth bits of operands} | \land & \lor & \equiv & \not |
|-----------------------------|------|------|------|------|
| both ones                   | 1    | 1    | 1    | 0    |
| both zeros                  | 0    | 0    | 1    | 0    |
| otherwise                   | 0    | 1    | 0    | 1    |
4.6 Conditional Expressions

Syntactic form: \( E_1 \rightarrow E_2, E_3 \)

\( E_1, E_2 \) and \( E_3 \) may be any logical expressions or expressions of greater binding power. \( E_2 \) and \( E_3 \) may in addition be conditional expressions. Thus:

\[
E_1 \rightarrow x, E_2 \rightarrow y, z \quad \text{means} \quad E_1 \rightarrow x, (E_2 \rightarrow y, z)
\]

and

\[
E_1 \rightarrow E_2 \rightarrow x, y, z \quad \text{means} \quad E_1 \rightarrow (E_2 \rightarrow x, y), z.
\]

Example: let \( f(x) = x < 0 \rightarrow 0, \)

\( x > 10 \rightarrow 10, \)

\( x \)

Semantics:

The \text{rvalue} of a conditional expression is obtained by evaluating either \( E_2 \) or \( E_3 \) in order depending on whether the value of \( E_1 \) is \text{true} or \text{false}.

\[
\text{true} \rightarrow E_2, E_3 \quad \text{means} \quad E_2
\]

\[
\text{false} \rightarrow E_2, E_3 \quad \text{means} \quad E_3
\]

If the value of \( E_1 \) is neither \text{true} nor \text{false} the result of the conditional expression is undefined.

A conditional expression is an Ltype expression if both its alternatives are Ltype expressions.
4.7 Tables

Syntactic form: \[ \text{table} \langle \text{constant} \rangle \{, \langle \text{constant} \rangle \}_n \]

Example: \[ \text{let T = table '0', '1', '2', '3', '4', '5', '6', '7, '8', '9', 'A', 'B', 'C', 'D', 'E', 'F} \]

Semantics:

All the expressions in the list must have \( \text{Ivalues} \) which can be determined at compile time. They may consist of manifest constants, numbers, character constants or arithmetic expressions with constant operands. The \( \text{Ivalue} \) of a table is a pointer to a set of consecutive storage cells whose initial values are given by the list of constant expressions; the allocation of the storage cells and the initialization are performed prior to execution of the program.

A table may be used as a vector; for instance, \( T.15 \) is equal to 'F'. The elements of a table may be updated.

4.8 Constant Expressions

Syntactic form: \[ \langle \text{constant} \rangle ::= \text{E} \]

Example: \[ 35 + 3 \times \text{Table_size} \]

Semantics:

A constant expression is one whose \( \text{Ivalue} \) can be determined at compile time. It may be a number, a character constant, a manifest constant, or any expression composed of these, brackets
and the operators \( \cdot / + \) and \( - \).

Constant expressions are used in
(a) case labels
(b) vector definitions
(c) manifest, static and global declarations
and (d) tables.
5.0 Commands

5.1 Simple Assignment Commands

Syntactic form: \[ E_1 := E_2 \]

Examples: \[ x := 1 \]
\[ v_{i,j} := u_{i,j} + w_{i,j} \]

Semantics:

The assignment operation has already been discussed in section 3.4. \( E_1 \) must be an Ltype expression and it is evaluated in Lmode to yield an Lvalue and \( E_2 \) is evaluated in Rmode to yield an Rvalue. The contents of the storage cell referred to by the Lvalue is then replaced by the Rvalue.

An Ltype expression may be of one of the following four kinds:

(a) A name referring to a storage cell.
(b) An L2 expression.
(c) A vector application
(d) A conditional expression whose alternatives are both Ltype expressions.

5.2 Assignment Commands

Syntactic form: \[ L_1, L_2, \ldots, L_n := R_1, R_2, \ldots, R_n \]

Example: \[ x, v_{i,j} := 1, u_{i,j} + w_{i,j} \]

Semantics:

The assignment command is semantically equivalent to a sequence of simple assignment commands. The general form given above is equivalent to the following set of simple assignments:
L1 := R1
L2 := R2
:
Lm := Rm

The order of execution of the assignments is not defined although in practice almost all implementations will work from left to right. Note that the assignment:

x, y := y, x

will not interchange the values of x and y. The main advantage of the general assignment command is the syntactic one of elimination the need for action brackets in certain circumstances. For instance the following command

\begin{verbatim}
if x = y do $
\begin{cases}
V \downarrow 3 := 0 \\
\text{B} := \text{true}
\end{cases}$
\end{verbatim}

may be written

\begin{verbatim}
if x = y do \begin{tabular}{l}
V \downarrow 3, \text{B} := 0, \text{true}
\end{tabular}
\end{verbatim}

Since the order of evaluation is not defined, some commands are strictly incorrect. For example, the command:

\begin{verbatim}
Symb[1, i := Rch(), i + 1
\end{verbatim}

may have different effects on different implementations.
5.3 Routine Commands

Syntactic form: \(EO(E_1, E_2, \ldots, E_n)\)

where \(EO\) is any primary expression and \(E_1\) to \(E_n\) are any expressions. The list of expressions may be empty.

Examples:

\(R(x)\)

\(\text{compjump (H2, x, false, L)}\)

\(\text{(C 'i')()}\)

Semantics:

The execution of a routine application is explained in section 6.3.3.

5.4 Labelled Commands

Syntactic form: \(<\text{name}> : C\)

Examples:

\(\text{Next: Rch()}\)

\(L: \text{Chkind := Kind(Ch)}\)

Semantics:

A labelled command is a form of declaration which declares the name as a static variable with a defined initial value. The scope of the name is the smallest textually enclosing

- body of a block,
- body of a routine,
- body of a result block,
- body of a for loop,

or program.

The R-value of a label may be the operand of a goto command, see the next section. For an explanation of the terms static variable and scope see section 6.2.
5.5 Goto Commands

Syntactic form: \[ \text{goto } E \]
where \( E \) is any expression.

Examples:
\begin{align*}
goto & \text{ Next} \\
goto & \text{ x!i} \\
goto & x = 0 \rightarrow \text{ L, } f(x)
\end{align*}

Semantics:

\( E \) is evaluated to yield an RValue, and then execution jumps to the command whose label had (initially) the same value. The point where execution is resumed must be at the same activation level as that of the goto command, or, in other words, the label and the goto command must both be in the same function or routine body.

As a general rule, it is a good policy to try to minimise the number of labels in a program as this will tend to improve its readability.

5.6 If Commands

Syntactic form: \[ \text{if } E \text{ do } C \text{ unless } E \text{ do } C \]

Examples:
\begin{align*}
\text{if } x=0 \text{ do } x := 10 \\
\text{unless } \text{Symb=S_COMMA do } \text{Report(30)} \\
\text{unless } \text{x!i=x!i resulting false}
\end{align*}

Note the automatic insertion of \text{do} by the compiler in the third example. See section 2.3.4.

Semantics:

The command \text{if } E \text{ do } C \text{ is executed by evaluating } E \text{ to yield a}
truth value, then, if the result is false execution is complete, if the result is true the command \( C \) is executed, and if the result is neither true nor false the effect is undefined.

The command \( \text{unless } E \text{ do } C \) is equivalent to \( \text{if not } (E) \text{ do } C \).

5.7 While Commands

Syntactic form:

\[
\text{while } E \text{ do } C \\
\text{until } E \text{ do } C
\]

Examples:

\[
\text{while } N \geq \text{ SSP do } \text{ LoadT}(S, \text{ LOCAL, SSP}) \\
\text{until } T \cdot 0 = 0 \text{ do } T := T \cdot 0
\]

Semantics:

The command \( \text{while } E \text{ do } C \) is equivalent to:

\[
\text{goto } L \\
N : C \\
L : \text{ if } E \text{ goto } H
\]

where \( L \) and \( H \) are identifiers which do not occur elsewhere in the program.

The command \( \text{until } E \text{ do } C \) is equivalent to \( \text{while not } (E) \text{ do } C \).

5.8 Test Commands

Syntactic form:

\[
\text{test } E \text{ then } C \text{ or } C
\]

Example:

\[
\text{test } 2^n > (\text{CaseK } n - \text{ CaseK } 1)/2 + 7 \\
\text{then } \text{LoSwitch}(1, n, D) \\
\text{or } \text{BoSwitch}(1, n, D)
\]
Semantics:

The command if $E$ then $C_1$ or $C_2$ is equivalent to:

\[
\begin{align*}
\text{if not } (E) & \text{ goto } L \\
& \text{ goto } N \\
L: & \text{ } C_2 \\
M: & \\
\end{align*}
\]

where $L$ and $N$ are identifiers which are not used elsewhere in the program.

5.9 Repeat Commands

Syntactic form:

\[
C \text{ repeatwhile } E \\
C \text{ repeatuntil } E \\
C \text{ repeat}
\]

Examples:

\[
\begin{align*}
\text{Reh()} & \text{ repeatuntil } Ch = 'n' \\
& \text{ $( WP := WP + 1 )$} \\
& \text{ $( WP := Ch )$} \\
\text{Reh()} & \text{ $S$ } \text{ repeatwhile } 'A' \leq Ch \leq 'Z'
\end{align*}
\]

Semantics:

The repeat commands are defined in terms of other equivalent commands, as follows:

\[
C \text{ repeatwhile } E \text{ is equivalent to } L: C; \text{ if } E \text{ goto } L
\]

\[
C \text{ repeatuntil } E \text{ is equivalent to } C \text{ repeatwhile not } (E)
\]

\[
C \text{ repeat} \text{ is equivalent to } C \text{ repeatwhile true}
\]

where $L$ is an identifier which is not used elsewhere in the program.
5.10 For Commands

Syntactic form: \[ \text{for } N = E_1 \text{ to } E_2 \text{ do } C \]
where \( N \) is a name.

Example: \[ \text{for } i = 0 \text{ to } 122 \text{ do } V \cdot i := i \]

Semantics:
The for command can be defined by the following equivalent form:

\[
\begin{aligned}
\text{let } n, z &= E_1, E_2 \\
\text{until } n > z \text{ do } C
\end{aligned}
\]

where \( z \) is an identifier not used elsewhere in the program. Note that the initial value and end limit expressions \( E_1 \) and \( E_2 \) are evaluated only once, and that the control variable \( N \) moves in steps of plus one.

5.11 Break Commands

Syntactic form: \[ \text{break} \]

Example: \[ \text{until } j = 0 \text{ do } C \]

Semantics:
Execution of the break command causes the current a jump to the point just after the smallest textually enclosing loop command.

5/7 15/1/69
The loop commands are those with the following key words:

until, while, repeat, repeatwhile, repeatuntil and for.

5.12 Finish Commands

Syntactic form: finish

Example:

```
if Reportcount > Reportmax do
  $( Write("Too many errors")
     EndWrite(OUTPUT)
     finish $)
```

Semantics:

The finish command causes execution of the program to cease.
Its exact effect is implementation dependent.

5.13 Return Commands

Syntactic form: return

Example:

```
let MapB(F, x) be
  $(1 if x=0 return
     if H1x = S_CONST do
       $( MapB(F, H3 x)
        F(H2 + x)
        return $)
    F(x) $)1
```

Semantics:

The return command causes the execution of the smallest enclosing routine body to cease and control returns to the point just after the routine call that invoked the current activation of the body.
5.14 Resultis Command

Syntactic form: \texttt{resultis } E

Example: \texttt{value } $(\text{for } i = 0 \text{ to } n \text{ do} \texttt{if } V \downarrow i \neq U \uparrow i \texttt{ resultis false resultis true } \)}$

Semantics:

The execution of \texttt{resultis } E causes the execution of the smallest enclosing result block to cease and yield and value which is the R-value of E.

5.15 Switchen Command

Syntactic form: \texttt{switchen } E \texttt{ into } \langle \text{block} \rangle

where the block contains labels of the form:

\texttt{case } \langle \text{constant} \rangle; \text{ or }
\texttt{default:}

Example: \texttt{let Trans}(x) \texttt{ be}

\texttt{$(1 \text{ if } x = 0 \text{ return switchen } H1 \downarrow x \texttt{ into})$
\texttt{$(\text{ default: Report(100); return case } S \_\_\_T: \text{ return case } S \_S3Q: \text{ Trans}(H2 \downarrow x) \text{ Trans}(H3 \downarrow x) \text{ return } 1)$}
Semantics:

The expression after switch is evaluated to yield an `Int` and then, if a case label exists which has a case constant of the same value then execution jumps to that point, otherwise if there is a default label execution resumes there. If the switch has no default label and if no case constant matches the switch expression then control passes to the point just after the `switch` command.

Note that the names S_INT and S_SEQ in the example above must have been declared to be manifest constants.

The switch is implemented by any one of a number of methods (e.g. direct switch, sequential search, hash table, binary tree) depending on the number and range of the case constants.

5.16 Blocks

Syntactic form:

\[
\{( \text{declaration} ) \, ^n \, _{1} \, ( ; \, ^n \, _{2} \, \text{seq} )
\]

\[
\{( \, _{2} \, ; \, ^m \, _{2} \, \text{seq} )
\]

Example:

\[
\{( \text{let List2}(x, y) = \text{valof}
\]

\[
\{( \text{let P = Newvec (1)}
\]

\[
P \downarrow 0, P \downarrow 1 := x, y
\]

\[
\text{result is P}
\]

\[
\text{finish}
\]

Semantics:

A block is executed by first performing the declarations (if any) and then executing the commands of the body in sequence.

The names declared by the declarations are local to the block and the storage cells allocated often only remain in existence as
long as execution is dynamically within the block. For a detailed discussion of scopes and extents see sections 6.1 and 6.2.
6.0 Definitions and Declarations

Before a name may be used in a JOPL program it must be
declared by the programmer in order to specify its scope, extent
and, possibly, its initial value.

6.1 Scope and Scope Rules

The SCOPE of a name N is the textual region of program
throughout which N refers to the same variable or manifest constant.

The scope of a name depends on its declaration as follows:

(a) A formal parameter list of a function or routine definition
    declares a list of names whose scope is the body of the
    function or routine defined.

(b) A declaration in the declaration sequence of a block declare
    a name or set of names whose scope is the succeeding
    declarations (if any) and the command sequence of the block.
    If the declaration is a let declaration the scope also
    includes the declaration itself.

(c) A name labelling a command is a form of declaration
    and it declares a variable whose scope is the smallest
    enclosing block body, function body, routine body,
    result block body, for loop body or program.

(d) The scope of the control variable of a for command is the
    body of the command.

If two variables have identical scopes then they must have
distinct names and so, for instance, the names in a formal
parameter list or the labels in a block must be different.
6.2 Extent and Space Allocation

The extent of a variable is the time through which it exists and has a storage cell (with its associated lvalue). Throughout the extent of a variable it remains associated with the same storage cell and so the lvalue remains constant; however, the contents of the cell (or rvalue) may be replaced by the execution of an assignment command. In ECPL, variables can be divided into two classes:

(a) Static variables

These are variables whose extents last as long as the program is running. The storage cell of a static variable is allocated prior to execution and continues to exist until the program has finished.

(b) Dynamic variables

The extent of a dynamic variable starts when its declaration is executed and continues until execution leaves its scope. Dynamic variables are useful when one needs some working space for a short period (perhaps during the execution of a routine) and it is too wasteful to use static storage. Dynamic variables are particularly useful when using functions and routines recursively.

The class of a variable depends only on its declaration. Static variables are declared by

- function or routine definitions,
- static declarations,
- global declarations,
- and labels set by colon.

6/2 15/1/69
Dynamic variables are declared by

simple variable definitions,

for commands,

vector definitions,

and formal parameters.

During the execution of a program there are three separate
areas of storage in which variables may reside; these are:

(a) the global vector,
(b) the stack,
(c) miscellaneous static cells.

The global vector provides a facility rather similar to
COMMON in FORTRAN and is used as a means of communication between
separately compiled segments of program. The programmer may use
a global declaration to associate names with particular cells in the
global vector. The variables so declared can hold functions and
routines declared in other segments, but may, of course, hold values
of any other conceptual type.

The stack is needed for the implementation of recursion and is
used to hold dynamic variables (such as vectors and function arguments
and anonymous results needed during the evaluation of expressions.

The miscellaneous static cells hold non-global static
variables which are local the segment in which they are declared.
6.3 Let Declarations

Syntactic form: \( \text{let } D_1 \text{ and } D_2 \text{ and } \ldots \text{ and } D_n \text{ where } D \text{ denotes a definition} \)

Example:
\[
\text{let } x, y = 0, 1 \\
\text{and } f(t) = 2t - 1 \\
\text{and } \text{Interv} = \text{vec} \ 22
\]

Semantics:

A let declaration may occur in the declaration sequence of a block and may be used to declare simple variables, vectors, functions and routines. The scope of the variables declared is the textual region of program consisting of the let declaration itself, the succeeding declarations and the commands of the block. The definitions between the and are at the same level and are effectively executed simultaneously, and by this means a let declaration may be used to declare a set of mutually recursive functions and routines.

The various kinds of basic definition are described below.

6.3.1 Simple Variable Definitions

Syntactic form: \( N_1, N_2, \ldots, N_n = E_1, E_2, \ldots, E_n \)

where \( N_1 \) to \( N_n \) are different names and \( E_1 \) to \( E_n \) are any expressions.

Example:
\[
\text{let } x = 1 \\
\text{and } y, z = f(t) + 3, H2 \downarrow A
\]
Semantics:

Dynamic variables with names N1, N2 ... Nn are first declared but not initialised, and then the following assignment command is executed:

N1, N2, ... Nn := E1, E2, ... En

6.3.2 Vector definitions

Syntactic form: N = vec <constant>
where N is a name

Examples:
let V = vec 64
and U = vec 10
Semantics:

The constant expression must have a value which can be determined at compile time; its value gives the maximum allowable subscript value for the vector declared. The initial value of the vector variable is a pointer to the zeroth element of set of cells allocated. The effect of the declaration:

\[
\text{let } V = \text{vec 3}
\]

can be shown diagrammatically as follows:

\[
\begin{align*}
V & \quad \rightarrow \quad \{\text{The elements of the vector}\} \\
& \quad \rightarrow \quad \text{All the storage cells are allocated from the stack and so both the vector variable and the vector elements are dynamic data items.} \\
& \quad \rightarrow \quad \text{The elements of the vector are not initialised.}
\end{align*}
\]

6.3.3 Function and Routine Definitions

Syntactic form:

\[
<\text{function definition}> ::= N() = E \mid N(N1, N2, \ldots, Nn) = E
\]
<routine definition> ::= N() be C | N(N1, N2, ... Nn) be C

where N and N1 to Nn are names. The list of names in parentheses is called the formal parameter list.

Example:

\[
\text{let Node(x) = val}
\]

\[
\text{let P = Freeman}
\]

\[
\text{Freeman := P + 3}
\]

\[
P10, P11, P12 := x, 0, 0
\]

\[
\text{result P}
\]

\[
\text{and Put(x, t) be}
\]

\[
\text{if t10 = x return}
\]

\[
t := t10 < x \rightarrow t + 1, t + 2
\]

\[
\text{test rv t = 0}
\]

\[
\text{then rv t := Node(x)}
\]

\[
\text{or Put(x, rv t)}
\]

Semantics:

The purpose of a function or routine definition is to define a variable with an initial value which may be used in a function or routine call. The heading of the definition consists of the name of the function or routine being defined, followed by a list of formal parameters (possibly empty) enclosed in parentheses. The formal parameter list is a form of declaration which declares a set of variables with the specified names and they all have the same scope, namely, the body of the function or routine. Formal parameters are dynamic variables whose storage cells are allocated at the moment of call. The initial values are given by the actual
parameters of the call.

The process of calling a function or routine is shown diagrammatically in fig 9.

1. Evaluate the arguments in Rmode

2. Place the Rvalues in a new consecutive storage cells

3. Find the function or routine corresponding to the Rvalue of EO

4. Associate the formal parameters with the storage cells from left to right

5. Evaluate or execute the body of the function or routine in the environment of the definition extended by the new variables

6a. For a routine call, return to the point just after the call

6b. For a function application, yield as result the Rvalue of the body of the function

Fig. 9 - The process of calling a function or routine
The number of formal parameters need not equal the number
of actual parameters and so it is possible to define a variable
routine. Consider:

```
let R(a, b, c, d, e, f) be
  $( let v = lv a
    ...
    v<0
    v<3
  )$,
R(4, 32, -14, 63)
```

Within the body of R, the variable v may be thought of as a vector
whose elements are the arguments of the call, and thus in this
example v<0 equals 4 and v<3 equals 63.

Note that the parameters of a LC.L call are passed by value;
however, it is still possible to achieve the effect of a call by
reference using the lv and rv operators. Consider:

```
let S(x, y) be
  rv x := y
let A, B = O, 1
S(lv A, B)
```

The effect of the call for S is to assign the current value of
B (namely 1) to the variable pointed to by lv A (namely A), thus after
the call A has value 1.

Note that the name declared by a function or routine definition
is a static variable, see section 6.2. To simplify the combination
of separately compiled segments of program the storage cell allocated
for a function or routine may be in the global vector, see section
6.6.
All functions and routines may be defined and used recursively.

There is one important restriction on functions and routines which has been imposed in order to achieve a very efficient recursive call. This restriction is as follows:

Every name which is used in the body of a function or routine but which is not declared there must be either a manifest constant or a static variable (see section 6.2).

In terms of the implementation, this restriction states that either the R-value or the L-value of every free variable of a function or routine is known prior to execution.

Note that the following program is illegal:

\[ \texttt{let } a, b = 1, 2 \]
\[ \texttt{let } f(x) = a\times x + b \]

however, it may be corrected as follows:

\[ \texttt{static } \$( a = 1; b = 2 \$) \]
\[ \texttt{let } f(x) = a\times x + b \]

6.4 Manifest Declarations

Syntactic form: \[ \text{manifest } <\text{decl body}> \]

where

\[ <\text{decl body}> ::= \$( <\text{C def}> i ; <\text{C def}> j , \$) \]
\[ <\text{C def}> ::= <\text{name}> : <\text{constant}> | <\text{name}> = <\text{constant}> \]
Although both : and = are allowed in manifest declarations, by convention = should be preferred.

Examples:

```
manifest $( M1=0; M2=1; M3=2 )$
manifest $( S_LET=74
        S_SEQ=73
        S_COMMA=38 )$
```

Semantics: \textit{manifest}.

A manifest declaration associates Rvalues directly with the declared names; the association takes place at compile time and cannot thereafter be changed. The names so declared are not variables and may not appear in a left hand context.

6.5 Static Declarations

Syntactic Form: \begin{verbatim}
static <decl body>
\end{verbatim}
where \textit{<decl body>} is defined in section 6.4. Although both : and = may be used in \textit{static} declaration, by convention = should be preferred.

Example:

```
static $( P = 0; Q = 0
        Reportmax = 10 )$
```

Semantics: \begin{verbatim}
\textit{static} \textit{M} \textit{P} \textit{Q} \textit{Reportmax} \end{verbatim}

A static declaration declares a set of static variables (see section 6.2) whose initial values are given. Both the allocation of storage cells and the initialisation are performed prior to execution of the program.
6.6 Global Declarations

Syntactic form: \[ \text{global } <\text{decl body}> \]
where \(<\text{decl body}>\) is defined in section 6.4.

Although both \(\) and \(=\) may be used in a global declaration, by convention \(=\) should be preferred.

Examples:
\[
\text{global } $(\text{Charcode:127; Opticn:128 })$
\text{global } $(\text{Rblock:body:140; Rblock:141})$
\text{Rexp:144; Rref:145; Rcom:146 })$
\]

Semantics:

A global declaration declares variables whose storage cells are in the global vector (see section 6.2). The main purpose of the global vector is to provide a means of communication between separately compiled segments of program. Each name in a global declaration is associated with a constant expression whose value specifies which storage cell in the global vector belongs to the name. The same global storage cell may be associated with variables in many separate segments and hence may be used to pass values from one segment to another. Some global variables are initialised prior to execution of the program as a result of the following rule:

If a function or routine definition occurs within the scope of a global variable with the same name, then the function or routine
variable defined shares the storage cell of the global variable and it is initialized to the value of the function or routine prior to execution of the program.

For example, the following segment will declare and initialize the seventh global variable to be a function.

\[
\text{global } \$ ( \text{ F:7 } \$ )
\text{let } F(g,x) = g(x) + g(-x)
\]

The following program is a segment which uses the function defined in the last example.

\[
\text{global } \$ ( \text{ F:7; Write:67 } \$ )
\text{let } g(t) = t + 3
\text{for } i = 0 \text{ to } 10 \text{ do Write } (F(g, i))
\]

The interface between BCPL and the operating system is provided by a machine code segment of basic functions and routines which may be called by ordinary BCPL calls. This interface is necessarily very implementation dependent.
7.0 Example Programs

The three examples given here are taken from the BCPL compiler, and thus give a realistic demonstration of the language in the application for which it was designed.

The second example uses a hardware representation which has no small letters; it is included so that the readability of the representations may be compared.

It is not practical to give an explanation of the functions and routines defined in the examples and, indeed, it is not necessary to understand them in detail since they are only intended to give a general idea of what BCPL programs look like.
This is the part of the BCPL compiler that compiles switches into ATLAS machine code.

GET 'HEAD3' // insert the heading file

static $( casek=0; casel=0 $)

let cgswitch() be

$(1 let a = vec 200
let b = vec 200
let n = readn() // read the number of cases
let d = readl() // read the default label

casek, casel := a, b

for i = 1 to n do
$(1 let k = readn() // read the cases and
let l = readl() // sort into ascending order
let j = i-1

until j=0 do // place the latest case in

$(1 if k > casek!j break
 casek!(j+1), casel!(j+1) := casek!j, casel!j
 j := k - 1 $)

casek!(j+1), casel!(j+1) := k, l $)

simplify()

store(0, ssp-2)

movetor(areg, arg1) // compile code to put the control

expression into the A register

test 3 n > 7 + (casek!n-casel!1)/2 // test whether to

then dswitch(1, n, d) // a direct switch

or bswitch(1, n, d) // or a tree switch

compes(f121, 127, 0, 2, d) // compile a jump to default

initstack(ssp-1) $) $)
AND BSWCETH(P, Q, D) BE  // Compile a tree switch

$(( IF Q-P < 6 DO
    $( FOR I = P TO Q DO  // Compile a sequential search
        $( COMPS(F172, AREG, O, CASEKIP, O)
        COMPS(F224, 127, O, O, CASELIP) )
    RETURN $)

$( LET L = NEXTPARAM()  // Find the case with average value
    LET T = (P+Q)/2
    COMPS(F172, AREG, O, CASEKIP, O)  // Compile a condition jump
    COMPS(F226, 127, O, O, L)
    BSWCETH(P, T-1, D)  // Compile switch for the smaller cases
    COMPS(F121, 127, O, O, D)  // Compile a jump to default

    COMPLAD(L)
    COMPS(F224, 127, O, O, CASELIP)
    BSWCETH(T+1, Q, D)  // Compile switch for the larger cases
    RETURN $ ) )  // All done

AND BSWCETH(P, Q, D) BE  // Compile a direct switch

$( LET L = NEXTPARAM()  // For the table of labels
    COMPS(F172, AREG, O, CASEKIP, O)
    COMPS(F227, 127, O, O, D)
    COMPS(F172, AREG, O, CASEKIP, O)
    COMPS(F227, 127, O, O, D)
    COMPS(F124, AREG, AREG, O, O)
    COMPS(F124, AREG, AREG, O, O)
    COMPS(F101, 127, AREG, -4*CASEKIP, L)

    COMPDATALAB(L)  // We now compile the table
    FOR K = CASEKIP TO CASEKIP DO
        TEST CASEKIP = K
        THEN $( COMPD(O, CASELIP)
            P := P + 1 $)
        OR COMPD(O, D)
    ENDI

    CONDHW() $)  // Compile a half word if necessary

7/3  29/1/69
GET '/TRNHEAD'

LET JUMPCOND(X, B, L) BE

$(JC LET SW = B

SWITCH ON H1:X INTO

$( CASE S.FALSE: B := NOT B
    CASE S.TRUE: IF B DO COMPJUMP(L)
    RETURN
    CASE S.NOT: JUMPCOND(H2:X, NOT B, L)
    RETURN
    CASE S.LOGAND: SW := NOT SW
    CASE S.LOGOR:
      TEST SW THEN $( JUMPCOND(H2:X, B, L)
      JUMPCOND(H3:X, B, L) $)
      OR $( LET M = NEXTPARAM()
      JUMPCOND(H2:X, NOT B, M)
      JUMPCOND(H3:X, B, L)
      COMPLAB(M) $)

    RETURN

    DEFAULT: LOAD(X)
    OUT2P(B -> S.JT, S.JF, L)
    SSP := SSP - 1
    RETURN $)JC
GET 'HEAD2' // This is the part of the BCPL compiler
// that compiles the assignment command

LET ASGN(x, y) BE // x is the left side, y the right
$(1 IF x=3 LOGOR y=0 DO
  $( UNLESS x=3 \ y=0 DO REPORT(110, CURRENTBRANCH)
  RETURN $(
SWITCH ON H1X INTO // Switch on the leading operator
  $( CASE COMMA: UNLESS H1Y = COMMA DO
    $( REPORT(112, CURRENTBRANCH)
    RETURN $(
    ASSIGN(H21X, H21Y)
    ASSIGN(H31X, H31Y)
    RETURN
CASE NAME:
  $( LET T = CELLYTHNAME(x)
  LET K, N = DDEC(T+1), DDEC(T+2)
  IF Y=0 DO REPORT(115, x)
  IF T<DDEC K=LOCAL DO REPORT(116, x)
LOAD(Y)
SSP := SSP - 1
SWITCH ON K INTO
  $( DEFAULT: REPORT(117, x)
  N := 0
  CASE LOCAL: OUT2(SP, N); RETURN
  CASE GLOBAL: OUT2(SG, N); RETURN
  CASE LABEL: OUT2(SL, N); RETURN $(
CASE RV:
CASE VECAP:
CASE COND: LOAD(Y)
  LOAD.Y(x)
  OUT1(ST1RD)
  SSP := SSP - 2
  RETURN
DEFAULT: REPORT(112, CURRENTBRANCH) $1
References


