MAC TR-141

# THE BCPL REFERENCE MANUAL

Martin Richards
Arthur Evans, Jr.
Robert F. Mabee

December 1974

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC
CAMBRIDGE MASSACHUSETTS 02139

# The BCPL Reference Manual

by

Martin Richards

revised by

Arthur Evans, Jr.
Robert F. Mabee

11/26/74

---

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006.

# The BCPL Reference Manual

by

Martin Richards

revised by

Arthur Evans, Jr.
Robert F. Mabee

11/26/74

## Abstract

BCPL is a language which is readable and easy to learn, as well as admitting of an efficient compiler capable of generating efficient code. It is made self consistent and easy to define accurately by an underlying 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 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 ten different systems.

---

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006.

# TABLE OF CONTENTS

# TABLE OF CONTENTS

## 1.0 Introduction

BCPL (Basic CPL) is a general purpose programming language which is particularly suitable for large nonnumerical problems in which machine independence is important. It was originally designed as a tool for compiler writing and has, so far, been used in at least three compilers. BCPL is currently implemented and running on the Honeywell 635 under GECOS III, on the Honeywell 645 and 6180 under Multics, on the IBM 360 under OS and CP/CMS, on the TX-2 at Lincoln Laboratory, on the CDC 6400, on the Univac 1108, and on the DEC PDP-9. There are also BCPL compilers on the KDF 9 at Oxford and on Atlas 2 at Cambridge. Other implementations are under construction.

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

The BCPL compiler is written in BCPL and is designed for fairly easy transfer to any other machine. Where possible the implementation dependent parts of the compiler have been separated out, so only a small proportion (about 1/5th) of the compiler needs to be rewritten for a new implementation. This part consists mostly of the code generator, which is entirely object-machine dependent. There is also the command interface, which is entirely operating-system dependent. In addition to modifying the compiler, it is necessary to design and write the interface with the new operating system; this usually includes several hundred lines of assembly language and ten or twenty BCPL routines.

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

## 1.1 Implementation Guides

This reference manual describes the BCPL language abstracted from any particular implementation. For each implementation there should be a specific implementation guide (possibly several documents) to describe in detail:

(1) The representation of a BCPL program in the particular character set, and other source file conventions such as ignoring columns 73-80 in card images. There should be a complete list of canonical symbols and their machine representations.

(2) The form and meaning of constructs left to the implementation. This includes the get directive, the external declaration, the call command, and finish, as well as possibly other constructs.

(3) The maximum lengths of names, section bracket tags, numbers and stringconstants, and the maximum number of cases in a

switchon, of elements in a table, of arguments to a function or routine. There may also be restrictions on the length and complexity of a program, on the depth of recursion, on the length of a single stack frame, and the number of global cells.

(4) The library. This consists of a number of routines written in BCPL or assembly language which can be called by ordinary BCPL calls. Usually a declaration for the library routines will be made available on-line in a form suitable for inclusion by the get directive.

(5) How a BCPL program is invoked from the command language or from another compiler language.

(6) How to invoke the compiler. Also its options, input and output files, temporary files, storage requirements, side effects, etc.

(7) All the error messages or codes that can be generated by the compiler or run-time routines.

(8) Extensions or restrictions in the canonical language. All departures from the standard BCPL described in this manual should be documented.

(9) Possibly some description of the object program, representation of strings, format of stack frame, etc.

(10) Several sample programs.

## 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 guide. In this chapter we give the machine independent canonical syntax of BCPL and provide guide lines 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:

let and "P3*n" 36 < + ; while

The symbols of a program are chosen from a finite set of tokens along with the following unbounded sets:

<name>
<number>
<stringconst>
<charconst>
<sectbra>
<sectket>

As the representations of the tokens may differ in different implementations because of character set limitations, this manual uses a canonical BCPL defined in the next section.

### 2.1 Canonical BCPL

The following are each a single canonical symbol with an associated character string part:

<name> A name is a single lower-case letter or a capital letter followed by any number of letters and digits. For example: i Abc TaxRate V3

<number> A number consists of one or more decimal digits; other forms are described in section 4.1.2.

<stringconst> A string constant consists of any number of string characters contained between two double quotes ("). An escape convention is described in section 4.1.3. For example: "abc"

<charconst> A character constant is a single string character enclosed between two single quotes (´). The same escape convention described in section 4.1.3 applies also to character constants. For example: ´p´ ´"´

<sectbra> A left section bracket consists of $( followed by any number of letters and digits. For example: $( $(Trans $(1

<sectket> A right section bracket consists of $) followed by any number of letters and digits. For example: $) $)xyz

These are all the other canonical symbols:

| | | | |
|---|---|---|---|
| and | be | break | by |
| call | case | default | do |
| endcase | external | false | finish |
| for | global | goto | if |
| ifnot | ifso | into | let |
| list | logand | logor | loop |
| lshift | lv | manifest | nil |
| not | or | rem | rep |
| repeat | repeatuntil | repeatwhile | resultis |
| return | rshift | rv | static |
| switchon | table | test | to |
| true | unless | until | valof |
| vec | while | | |

| | | | | | |
|---|---|---|---|---|---|
| + | .+ | - | .- | * | .* |
| / | ./ | = | .= | ≠ | .≠ |
| < | .< | > | .> | ≤ | .≤ |
| ≥ | .≥ | ≡ | ≢ | -> | , |
| := | : | ( | ) | [ | ] |
| ; | ! | | | | |

Throughout this manual syntax and programming examples will be given in the representation defined in this section.

## 2.2 Formal Syntax

### 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 zero or more repetitions of the bracketted entity, for example:

E {, E} means E | E, E | E, E, E | ... etc

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 BCPL

```
E ::= <name> | <stringconst> | <charconst> | <number>
      | true | false | nil | ( E ) | valof <block>
      | lv E | rv E | E ( <arg list> ) | E ! E | E [ E ]
      | E <diadic op> E | <monadic op> E | E -> E, E
      | vec <constant expression> | table <constant list>
      | list <E list>

<diadic op> ::= * | .* | / | ./ | rem | + | .+ | - | .-
      | = | .= | ≠ | .≠ | < | .< | > | .> | ≤ | .≤ | ≥ | .≥
      | lshift | rshift | logand | logor | = | ≠

<monadic op> ::= + | .+ | - | .- | not

<E list> ::= <E rep> {, <E rep>}

<E rep> ::= E | E rep <constant expression>

<arg list> ::= <E list> | <empty>

<constant expression> ::= E

<constant list> ::= <constant rep> {, <constant rep>}

<constant rep> ::= <constant expression>
      | <constant expression> rep <constant expression>

C ::= <E list> := <E list> | E ( <arg list> )
      | goto E | <name> : C | resultis E
      | if E do C | unless E do C | while E do C | until E do C
      | C repeat | C repeatuntil E | C repeatwhile E
```

```
      | loop | break | return | finish | endcase
      | test E then C or C | test E ifso C ifnot C
      | for <name> = E to E do C
      | for <name> = E to E by <constant expression> do C
      | switchon E into <block> | case <constant expression>: C
      | case <constant expression> to <constant expression>:  C
      |  default: C | call E ( <arg list> ) | <block> | <empty>

D ::= <name> ( <FPL> ) = E | <name> ( <FPL> ) be C
      | <name list> = <E list>

<FPL> ::= <name list> | <empty>

<name list> ::= <name> {, <name>}

<block> ::= $( <block body> $)

<block body> ::= <block item> {; <block item> }

<block item> ::= C | <declaration>

<declaration> ::= let D {and D}
      | manifest <decl body> | global <decl body>
      | external <decl body> | static <decl body>

<decl body> ::= $( <C def> {; <C def>}  $)

<C def> ::= <name> : <constant expression>
      | <name> = <constant expression>

<program> ::= <block body>
```

## 2.3 Hardware Representations

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

### 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 composed of letters and digits 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 two 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) A name is an identifier which is not a system word.

Thus on some implementations let and logor are system words while Let, LET, Logor and LOGOR may be used as names; but with a more restricted character set LET and LOGOR would be reserved system words and the programmer would have to represent the names in some other way, perhaps by:

F_LET, S_LET, F_LOGOR, S_LOGOR

### 2.3.2 Section Brackets

The preferred representation of a left section bracket consists of { followed by zero or more letters, digits, and other characters allowed in names. A right section bracket consists of } followed by zero or more letters, digits, etc. As the symbols { and } are used in this manual as meta-linguistic brackets, section brackets in the syntax and examples are represented using an alternate form also suitable for more limited character sets.

### 2.3.3 Equivalent Representations of Canonical Symbols

Several canonical symbols have alternate representations for clarity and compatibility. Thus by may be represented as step, and do may be represented as then. Many symbols ordinarily represented by non-alphabetic characters may also be represented by system words. For example, = may be represented as eq.

## 2.4 Preprocessor Conventions

Several functions which the compiler performs before syntactic analysis to improve readability and as a convenience to the programmer are collectively called preprocessor conventions.

### 2.4.1 Section Brackets

Section brackets are used to bracket blocks and commands. 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  until  i=0 do
     $(2    R (i)
            i := i + 1  $)1
```

is equivalent to:

```
$(1  until  i=0 do
     $(2    R (i)
            i := i + 1  $)2  $)1
```

### 2.4.2 Automatic Insertion of SEMICOLON

The canonical symbol SEMICOLON is inserted by the compiler between pairs of items if they appeared on different lines and if the first was from the set of items which may end a command or definition, namely:

```
loop  break  return  finish  endcase  repeat  true  false  nil
<name>  <number>  <stringconst>  <charconst>  <sectket>  )  ]
```

and the second is from the set of items which may start a command or declaration, namely:

```
test  for  if  unless  until  while  goto  resultis  call
switchon  case  default  endcase  loop  break  return
finish  valof  rv  lv  true  false  table  list  (  +  -  not
```

```
<name>  <number>  <stringconst>  <charconst>  <sectbra>
global  manifest  static  external  let
```

For example, the following two programs are equivalent:

```
x := x + 1                          x := x + 1;
if x > y do y := 0                  if x > y do y := 0;
R (x)                               R (x)
```

### 2.4.3 Automatic Insertion of DO

The canonical symbol DO is inserted by the compiler between pairs of items if they appeared on the same line and if the first is from the set of items which may end an expression, namely:

```
true  false  nil  <name>  <number>  <stringconst>
<charconst>  <sectket>  )  ]
```

and the second is from the set of items which must start a command, namely:

```
test  for  if  unless  until  while  goto  resultis  case
default  endcase  loop  break  return  finish  switchon  call
```

For example:

```
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
```

### 2.4.4 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   //  do it 200 times
                Readch (INPUT, lv Symb!i) $)   //  read a char
```

### 2.4.5 The Get Directive

A directive of the form

get <specifier>

may occur anywhere in a BCPL program; it directs the compiler to replace the characters of the directive by the text in the file referred to by the specifier. The syntactic form of the specifier is implementation dependent but will usually be a string constant.

## 3.0 Fundamental Concepts of BCPL

### 3.1 The Object Machine

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

The most important feature of the object machine is its store, which is represented diagrammatically in Figure 1.

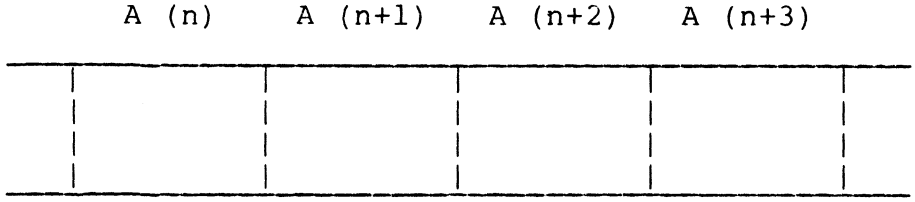



Figure 1 - The Machine's Store

It consists of a set of consecutive boxes (or storage cells) uniquely identified by arbitrary addresses. Some addressing function, A, places the consecutive integers in one-to-one correspondence with the addresses of consecutive cells. As is seen later, this property is important.

Each storage cell holds a binary 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. A large number of basic operations on Rvalues 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, or 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 are not defined and he would be losing machine independence if he performed

nonnumerical operations on Rvalues he knows 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 address and it locates the storage cell which is labelled by this address. This operation is assumed to be efficient and, as is seen later, the programmer may invoke it from within BCPL using the rv operator.

## 3.2 Variables, Manifest Constants, and Address Constants

Names in BCPL are associated either with storage cells or directly with Rvalues. 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. Variables are introduced by simple variable definitions, the for command, formal parameter lists, and the static and global declarations.

A manifest constant is a name which is directly associated with a constant Rvalue; this association takes place at compile time and remains the same throughout execution. Manifest constants are introduced only by the manifest declaration. There are many situations where manifest constants can be used to improve readability at no cost in run time efficiency.

An address constant is defined to be a name which is directly associated with an Rvalue representing in some way an address. The Rvalue cannot be determined until "load time" (just before execution) and remains the same during execution. Address constants cannot be used in constant expressions, which must be evaluated at compile time. Labels, the external declaration, and routine and function definitions introduce address constants.

## 3.3 Lvalues and Modes of Evaluation

As previously stated each storage cell is labelled by an address; this address 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 Figure 2.

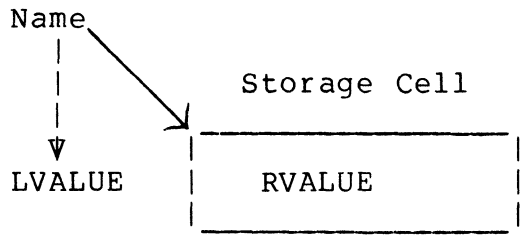



Figure 2 - The Form of a Variable

Within the machine an Lvalue is represented by a bit pattern of the same size as an Rvalue, and so an Rvalue can represent an Lvalue directly. The processes of finding the Lvalue and Rvalue of a variable are called Lmode and Rmode evaluation respectively. The idea of mode 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:

E1 := E2

where E1 and E2 are expressions. Loosely, the meaning of the assignment is to evaluate E2 and store its value in the storage cell referred to by E1. It is clear that the expressions E1 and E2 are evaluated in different ways and hence there is the classification into the two modes of evaluation. The expression E1 to the left of the := is evaluated in Lmode to yield the Lvalue of some storage cell and the right hand side E2 is evaluated in Rmode to yield an Rvalue; the contents of the storage cell is then replaced by the Rvalue. This process is shown diagrammatically in Figure 3.

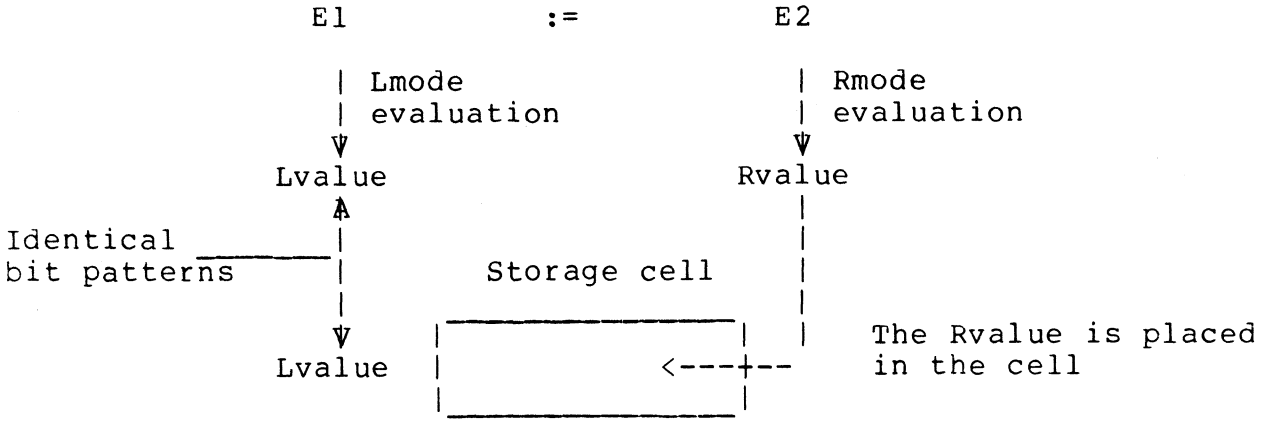



Figure 3 - The Process of Assignment

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

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

## 3.5 The Lv Operator

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

The syntactic form of an lv expression is:

lv E

where E is an Ltype expression. The evaluation process is shown in Figure 4.

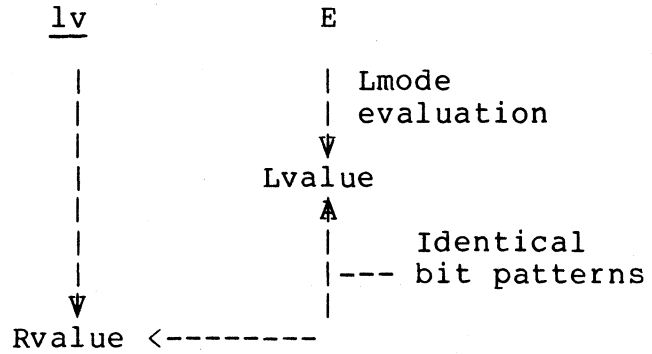



Figure 4 - The Evaluation of an lv Expression

The operand is evaluated in Lmode to yield an Lvalue and the result is a bit pattern identical to this Lvalue. Intuitively, lv x is the address in memory of the variable x. The lv operator is exceptional in that it is the only expression operator to invoke Lmode evaluation, and indeed in all other contexts, except the left hand side of the assignment, expressions are evaluated in Rmode.

## 3.6 The Rv Operator

The rv 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 Lvalue) is given.

The syntactic form of an rv expression is as follows:

```
rv E
```

and its process of evaluation is shown diagrammatically in Figure 5.

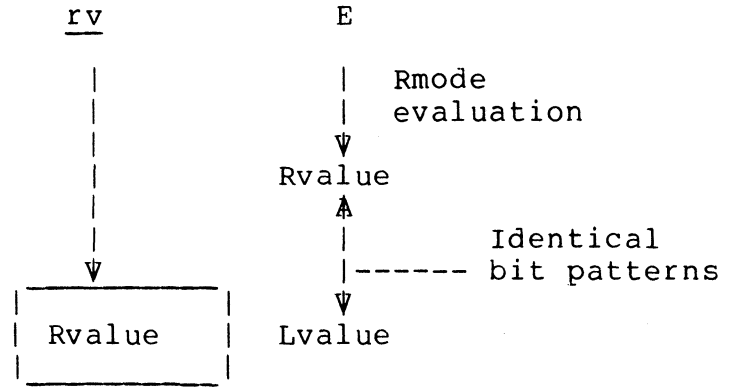



Figure 5 - The Evaluation of an rv Expression

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; it is also meaningful to evaluate it in Lmode, 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. Thus

```
rv 100 := 0
```

sets location 100 to zero.

## 3.7 The Vector Operator

The vector-application operator (represented here by !) takes advantage of the consecutive arrangement of storage cells. It finds the n'th successor to a given cell, as shown in Figure 6.

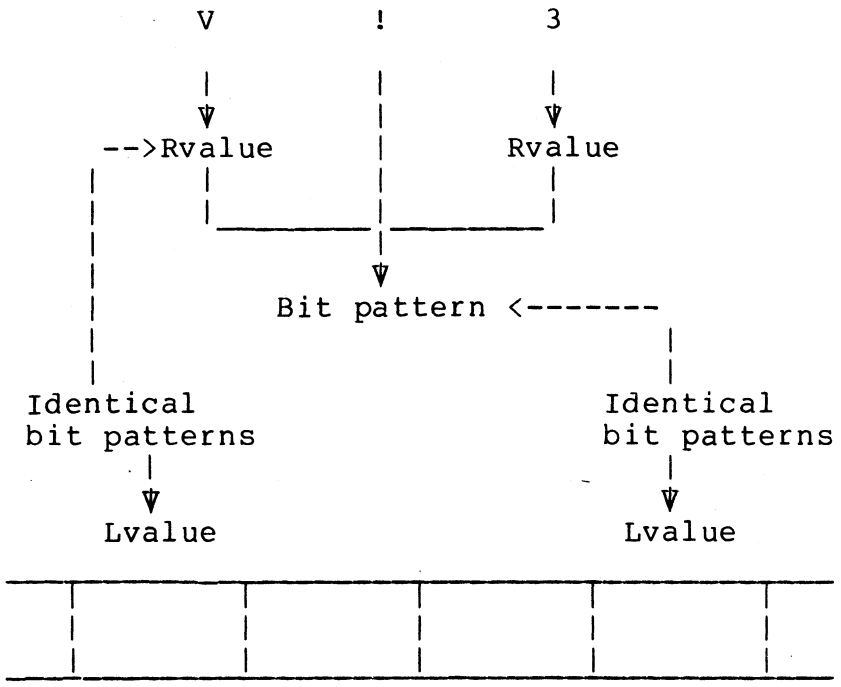



Figure 6 - An Interpretation of V ! 3

The diagram above 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. The cell at the right is the third successor of the one on the left. In terms of the addressing function A, if V = A (n) then the Lvalue of the cell on the right is A (n+3). Thus the expression:

V ! i

accurately models 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 by V. V can be thought of as the vector and i as the integer subscript.

A vector application is an Ltype expression; in Lmode evaluation it yields the address of the designated cell, and in Rmode evaluation it yields the contents.

Figure 7 shows how a vector application can be thought of as a data structure select operation. The variable Xpart acts as a named

selector applied to the data structure V. Manifest constants are commonly used to define structure selectors of this kind.

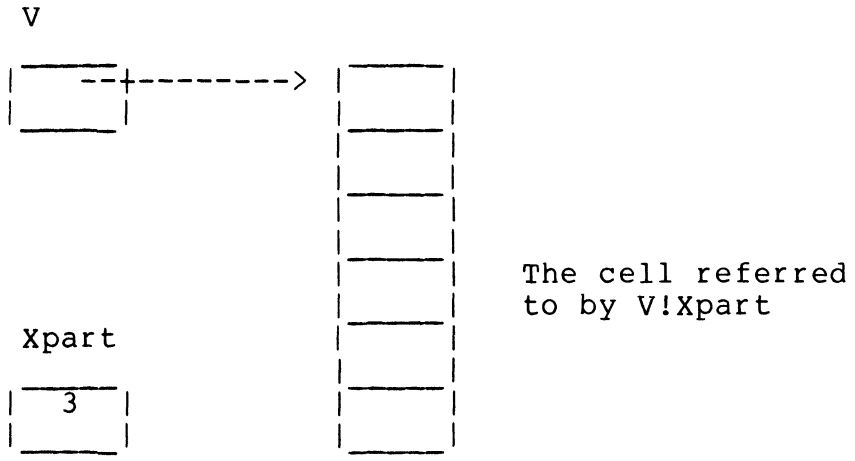



Figure 7 - An Interpretation of V ! Xpart

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

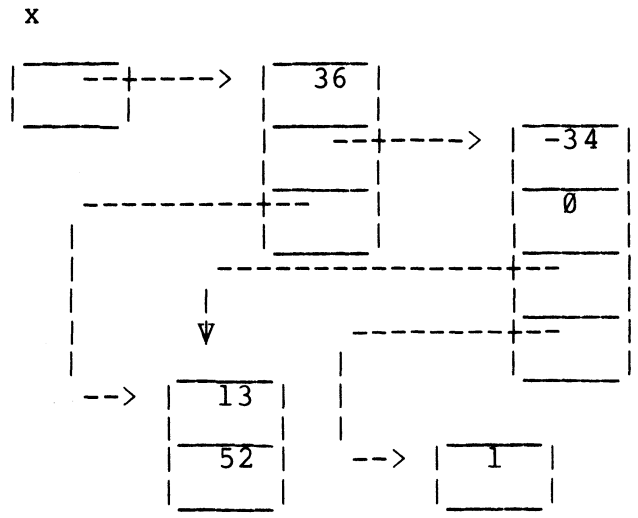



Figure 8 - A Structure of Integers and Pointers

## 3.8 Data Types

The unusual way in which BCPL 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

Much of the flavor of BCPL is the result of the conscious design decision to provide only one internal type, namely: the bit pattern (or Rvalue). In order to allow the programmer to model any conceptual type a large set of useful primitive operations has 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 summarized 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 BCPL much of the power of 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 V!i may depend on the value of i. One should note that, in languages (such as ALGOL [4] and CPL) 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 disadvantage is hopefully 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, nil, bracketted expressions, result blocks, lv expressions, rv expressions, vec expressions, table and list expressions, vector applications and function applications.

(b) Arithmetic expressions.

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

(c) Relational expressions.

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

(d) Shift expressions.

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

(e) Logical expressions.

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.

This section ends with descriptions of <constant expression> and <E list> although they are not syntactic subcategories of expressions.

## 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 BCPL 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:

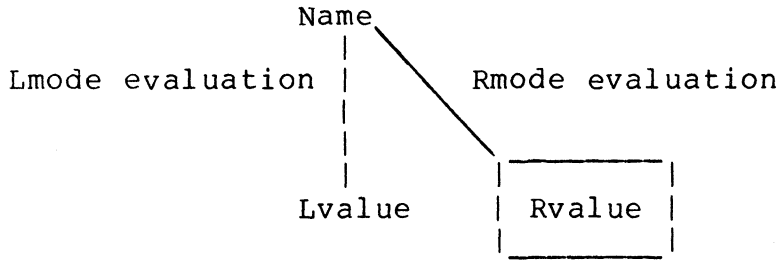
```
H3  Tax_rate  F  i
List4  StackP
```

Semantics:

A name may be associated directly with an Rvalue by means of a manifest declaration or by a label declaration, function or routine definition, or external declaration, or it may be associated with a storage cell to form a variable using any other kind of declaration. A variable, manifest constant, or address 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 or address constant can only be evaluated in Rmode 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:




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:

```
<digit>  ::= 0|1|2|3|4|5|6|7|8|9
<number> ::= <digit> {<digit>}
           | 8 <digit> {<digit>}
           | <digit> {<digit>} . {<digit>}
```

Examples: 132 43179 8377 3.14159 4.

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 octal digits. A decimal number is a sequence of digits not preceded by 8; its Rvalue is a bit pattern representing the integer in a way which depends on the implementation. A floating point number is a sequence of digits with a decimal point embedded or at the end. The Rvalue is implementation dependent.

Some implementations may admit of other number forms, such as hexadecimal.

### 4.1.3 String Constants

Syntactic form: " {<string character>} "

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 asterisk (*) 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 | * |

Some implementations may admit additional escapes in strings.

Examples: "End of test" "=" "*""
"*n*tTRA*tL1*n" " " ""

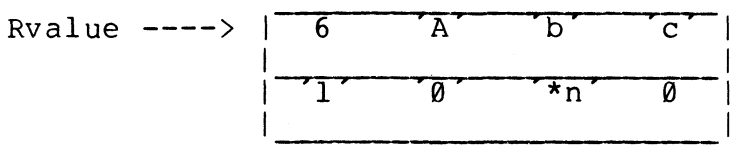
Semantics:

The Rvalue 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 implementation dependent. For an implementation which packs four characters per word, the string

"Abc1Ø*n"

might be represented as follows:




| 6 | 'A' | 'b' | 'c' |
|---|---|---|---|
| '1' | 'Ø' | '*n' | Ø |

The storage cells containing the string should not be updated; some implementations use a "memory-protect" hardware facility to prevent updating.

### 4.1.4 Character Constants

Syntactic form: ' <string character> '

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

Examples: 'x' '=' '*n' '"'
'**' 'A' '*'

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

Syntactic 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. (N.B. These are numerically equal in a ones-complement machine.)

### 4.1.6 Nil

Syntactic form: nil

Example: let x = nil

Semantics:

The Rvalue of nil is undefined. Its purpose is to avoid initializing a newly defined cell. In the example, the dynamic variable x is defined without an initial value.

### 4.1.7 Bracketted Expressions

Syntactic form: ( E )

Examples:
```
T rem ((x-y)/(x+y) + 2/z)
(B -> A, B) ! (i+1)
```

Semantics:

Parentheses may enclose any expression without changing its mode of evaluation or its value. Their sole purpose is to specify grouping.

### 4.1.8 Result Blocks

Syntactic form: valof <block>

Example:
```
valof $( for i=1 to n do
             if P (i, x) resultis false
         resultis 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 resultis statement is encountered; this causes execution of the block to cease and the Rvalue of the expression in the resultis command is the result. See section 5.14.

### 4.1.9 Lv Expressions

Syntactic form: lv E
where E is a primary expression.

Examples:
```
Readch (INPUT, lv Ch)
U := lv V!i
```

Semantics:

The Rvalue of an lv expression 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 lv operator in section 3.5.

### 4.1.10 Rv Expressions

Syntactic form: rv E
where E is a primary expression.

Example:
```
rv x := rv (f (i) + 2)
```

Semantics:

An rv expression is an Ltype expression and may be evaluated to yield either an Lvalue or an Rvalue. It is evaluated by evaluating its operand in Rmode to yield a bit pattern which is interpreted as the Lvalue 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 is described further in section 3.6.

### 4.1.11 Vector Expressions

Syntactic form: vec <constant expression>

Examples:
```
let v = vec 100
Word := vec Vmax / 4
```

Semantics:

Let the value of the constant expression be n. Then the Rvalue of the vector expression is the address (Lvalue) of the first word of a block of storage n + 1 words long. Thus there is both a zero'th word and an n'th word.

The storage is dynamic in class and is newly allocated by each evaluation of the expression. It remains allocated for as long as execution is dynamically between the reference and the end of the function or routine body, or the end of the smallest enclosing scope of any dynamic variable declaration. In the first example above, the

storage remains allocated as long as the cell v does. Repeated evaluation of the expression within a particular invocation of a function or routine results in allocating the same block of storage each time.

### 4.1.12 Table and List Expressions

Syntactic form: table <constant list>
list <E list>

Examples:
```
let T = table '0', '1', '2', '3',
            '4', '5', '6', '7', '8', '9',
            'A', 'B', 'C', 'D', 'E', 'F'
Cv := list "zero", "one", "two", "three",
          "four", "five", "six", "seven",
          "eight", "nine", "ten"
```

Semantics:

All the expressions which appear after table must have Rvalues which can be determined at compile time. The Rvalue 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' in the example above. The elements of a table should not be updated; some implementations use a "memory-protect" hardware facility to prevent updating.

The list expression is similar to table. The initial values can be any expressions. They are evaluated and stored in the list at the time the list expression is evaluated. The storage is allocated dynamically as for vectors. See section 4.1.11.

```
let L = list E0, E1, ... En
```

is equivalent to

```
let L = vec n
L!0, L!1, ... L!n := E0, E1, ... En
```

### 4.1.13 Vector Applications

Syntactic form: E1 ! E2 | E1 [ E3 ]

E1 and E2 are primary expressions and E3 is any expression. The operator is left associative and thus

x ! y ! z means (x ! y) ! z

Examples:

```
V ! (i+1) := V ! i + p ! Xpart

case SEQ: Trans (x[H2])
          Trans (x[H3])
          return
```

Semantics:

The expression E1!E2 is defined to take the Rvalue of the (E2)'th successor to the cell whose Lvalue is E1. Its purpose is explained in section 3.7.

The expression E1 [E2] is equivalent to E1!(E2).

### 4.1.14 Function Applications

Syntactic form: E0 (<E list>) | E0 ()

E0 is a primary expression and the <E list> may contain any expressions.

Examples:

```
f (x)
H (1, 2*t)
(x=0 -> f, P3)(1, "ZT", y+2)
Nextparam ()
```

Semantics:

The evaluation of a function application is explained in section 6.3.2.

## 4.2 Arithmetic Expressions

Syntactic form:

```
E * E | E / E | E rem E |
E + E | E - E |
+ E | - E |
E .* E | E ./ E |
E .+ E | E .- E |
.+ E | .- E
```

The operators * / rem .* and ./ are equally binding and associate

to the left; they are more binding than + - .+ or .- which also associate to the left.

Thus

x * y rem z means (x * y) rem z

x + y - z/t means (x+y) - (z/t)

Examples:

```
2*x*x + 6*x*y + 7*y*y
v ! (f (x) rem 13) + G (x)
x .* 2.3 .+ y .* 4.7
```

Semantics:

The arithmetic expressions evaluate their operands in Rmode. The integer operators then interpret the Rvalues as integers and yield Rvalues representing the integer results of the arithmetic. The floating point operators similarly interpret the Rvalues as floating point numbers and yield Rvalues representing the floating point results of the arithmetic.

The operators * and / denote integer multiplication and division respectively.

The operator 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 result of subtracting E2 from E1.

The expression - E1 has the same meaning as 0 - E1.

The operators .* and ./ denote floating point multiplication and division respectively.

The infixed operators .+ and .- denote floating point addition and subtraction.

The expression .+ E is the same as E.

The expression .- E is the same as 0.0 .- E.

## 4.3 Relational Expressions

Syntactic form: E <relop> E {<relop> E} where
<relop> ::= = | ≠ | < | > | ≤ | ≥ |
.= | .≠ | .< | .> | .≤ | .≥

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

Examples:
```
if 0 ≤ x < y goto L
A ! i := f (x) = g (x)
x .= 0.0 -> 0.0, y ./ x
```

Semantics:

For a simple relational expression defined by

E <relop> E

the operands are evaluated in Rmode; the Rvalues obtained are then interpreted as integers or floating point numbers according to the operator and if the particular relation is true then the result of the expression is true, otherwise the result is false. An extended relation such as

E1 <relop 1> E2 <relop 2> E3

is equivalent to the following expression:

(E1 <relop 1> E2) logand (E2 <relop 2> E3)

However, the number of times E2 is evaluated is undefined.

The correspondence between the operators and their meanings is given below.

| Integer Operator | Floating Point Operator | Meaning |
|---|---|---|
| = | .= | equal to |
| ≠ | .≠ | not equal to |
| < | .< | less than |
| > | .> | greater than |
| ≤ | .≤ | less than or equal to |
| ≥ | .≥ | greater than or equal to |

## 4.4 Shift Expressions

Syntactic form: E1 lshift E2 | E1 rshift E2

E2 is any relational, arithmetic, or primary expression and E1 is any shift, relational, arithmetic or primary expression; the shift operators are thus just less binding than the relations and associate to the left.

Examples:
```
let P (t) = t!3 rshift 10 logand 8377
x := x lshift Bytesize logor Ch
```

Semantics:

The operands are evaluated in Rmode 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 E1 lshift E2 is the bit pattern produced by shifting E1 to the left by E2 places. The operator rshift is similar to lshift, only it shifts to the right. Vacated positions are filled with zeros and the result is undefined if E2 is negative or greater than the number of bits in an Rvalue.

## 4.5 Logical Expressions

Syntactic form:
```
not E
| E logand E | E logor E
| E = E | E=/ E
```

The operator not is most binding; then, in decreasing order of binding power, there are:

logand, logor, =,=/

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

Examples:
```
B := not B
if x=0 logor y=0 resultis f(t)
x := x logand 8770077 logor y logand 87700
```

Semantics:

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

The application of the operator not yields the logical negation of its operand. The result of any other logical operator is a bit pattern whose nth bit depends only on the nth bits of the operands and can be determined from the following table.

| nth bits of operands | Operator logand | logor | = | ≠ |
|---|---|---|---|---|
| both ones | 1 | 1 | 1 | 0 |
| both zeros | 0 | 0 | 1 | 0 |
| otherwise | 0 | 1 | 0 | 1 |

The operators logand and logor are interpreted differently when an expression is being evaluated to control conditional execution, specifically in the if, while, test, and repeatwhile commands and the conditional expression. In most implementations one operand is evaluated first and if its value determines the result the other operand is not evaluated. This occurs when one operand of logand is false or when one operand of logor is true.

## 4.6 Conditional Expressions

Syntactic form: E1 -> E2, E3

E1, E2 and E3 may be any logical expressions or expressions of greater binding power. E2 and E3 may in addition be conditional expressions. Thus:

```
      B1 -> x, B2 -> y, z   means   B1 -> x, (B2 -> y, z)
and   B1 -> B2 -> x, y, z   means   B1 -> (B2 -> x, y), z
```

Example:

```
let f (x) = x < 0 -> 0,
            x > 10 -> 10,
            x
```

Semantics:

The Rvalue of a conditional expression is obtained by evaluating either E2 or E3 in Rmode depending on whether the value of E1 is true or false.

```
true  -> E2, E3  means  E2
false -> E2, E3  means  E3
```

If the value of E1 is neither true or 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 Constant Expressions

Syntactic form: <constant expression> ::= E

Example: 36 + 3 * Table_size

Semantics:

A constant expression is one whose Rvalue can be determined at compile time. It may be a number, a truth value, a character constant, a manifest constant, or a bracketted, relational, shift, logical, or conditional expression composed of constant expressions.

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

## 4.8 Expression lists

Syntactic form:
```
<E list> ::= <E rep> {, <E rep>}
<E  rep>  ::= E | E rep <constant expression>
```

Examples:
```
let T = table 0 rep 10   // Array of zeros.
a, b, c := a + 1, b + 1, c + 1
R (a, b, c rep 4)
```

Semantics:

Lists of expressions are useful in several contexts, such as argument lists and assignment commands. They are purely a syntactic feature.

E0 rep n

is equivalent to

E0, E0, ... E0

where the number of E0 terms is given by the value of n. Thus rep is merely a notation to avoid repetitive typing.

## 5.0 Commands

### 5.1 Simple Assignment Commands

Syntactic form: E1 := E2

Examples:
```
x := 1
V ! i := U ! i + W ! i
```

Semantics:

The assignment operation has already been discussed in section 3.4. E1 must be an Ltype expression and it is evaluated in Lmode to yield an Lvalue, and E2 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 rv expression.
(c) A vector application.
(d) A conditional expression whose alternatives are both Ltype expressions.

### 5.2 Assignment Commands

Syntactic form: <E list> := <E list>

There must be the same number of expressions in the list on the right of the := as there are on the left.

Example: x, V!i := 1, U!i + W!i

Semantics:

The assignment command is semantically equivalent to a sequence of simple assignment commands. The general form

L1, L2, ... Ln := R1, R2, ... Rn

is equivalent to the following set of simple assignments:

```
L1 := R1
L2 := R2
   :
Ln := Rn
```

The order of execution of the assignments is not defined and may not be relied on. 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 eliminating the need for section brackets in certain circumstances. For instance the following command

```
if x = y do $( V!3 := Ø
               B   := true $)
```

may be written

```
if x = y do V!3, B := Ø, true
```

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

```
Symb!i, i := Rch (), i + 1
```

may have different effects in different implementations.

## 5.3 Routine Commands

Syntactic form: E0 (<E list>) | E0 ()

E0 is any primary expression and the <E list> may contain any expressions.

Examples:

```
R (x)
Compjump (x!H2, false, L)
(C ! i)()
```

Semantics:

The execution of a routine application is explained in detail in section 6.3.2.

## 5.4 Labelled Commands

Syntactic form: <name> : C

Examples:

```
Next:  Rch ()
   L:  Chkind := Kind (Ch)
```

Semantics:

A labelled command is a form of declaration which associates the

name directly with the Rvalue representing the location of the command. The scope of the name is the smallest textually enclosing routine or function body.

The Rvalue of a label may be the operand of a goto command, as described in the next section. For an explanation of the term scope see section 6.1.

## 5.5 Goto Commands

Syntactic form:
```
goto E
where E is any expression.
```

Examples:
```
goto Next
goto S ! i
goto x = 0 -> Error, Tvec!x
```

Semantics:

E is evaluated to yield an Rvalue, and then execution jumps to the command whose label has 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. The effect of violating this rule is usually chaos.

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

## 5.6 If Commands

Syntactic form:
```
if E do C
unless E do C
```

Examples:
```
if x = 0 do x := 10
unless Symb=S_COMMA do Report (30)
unless S ! i = W ! i resultis false
```

Note the automatic insertion of do by the compiler in the third example. See section 2.4.3.

Semantics:

The command if E do C is executed by evaluating E to yield a truth value (see section 4.5). 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 unless E do C is equivalent to if not (E) do C.

## 5.7 While Commands

Syntactic form:

```
while E do C
until E do C
```

Examples:

```
while N > SSP do LoadT (S_LOCAL, SSP)
until T ! 0 = 0 do T := T ! 0
```

Semantics:

The command while E do C is equivalent to:

```
        goto L
M :     C
L :     if E goto M
```

where L and M are identifiers which do not occur elsewhere in the program.

The command until E do C is equivalent to while not (E) do C.

## 5.8 Test Commands

Syntactic form:

```
test E then C or C
test E ifso C ifnot C
```

Example:

```
test 2*n > (CaseK ! n - CaseK ! 1)/2 + 7
     then Lswitch (1, n, D)
     or Bswitch (1, n, D)
```

Semantics:

The command test E then C1 or C2 is equivalent to:

```
        if not (E) goto L
        C1
        goto M
L :     C2
M :
```

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

The command test E ifso C1 ifnot C2 is equivalent to test E then C1 or C2. The ifso and ifnot clauses may be interchanged.

## 5.9 Repeat Commands

Syntactic form:

```
C repeatwhile E
C repeatuntil E
C repeat
```

Examples:

```
Rch() repeatuntil Ch = '*n'
$( WP := WP + 1
   S ! WP := Ch
   Rch () $) repeatwhile 'A' ≤ Ch ≤ 'Z'
```

Semantics:

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

```
C repeatwhile E  is equivalent to  L: C; if E goto L
C repeatuntil E  is equivalent to  C repeatwhile not (E)
C repeat         is equivalent to  C repeatwhile true
```

where L is an identifier which is not used elsewhere in the program.

## 5.10 For Commands

Syntactic form:

```
for <name> = E to E do C
for <name> = E to E by <constant> do C
```

Example:

```
for i = 0 to 122 do V ! i := i
```

Semantics:

The for command can be defined by the following equivalent forms:

```
for N = E1 to E2 by E3 do C
```

is equivalent to

```
$( let N, Z = E1, E2
   while N ≤ Z do
         $( C
            N := N + E3  $)  $)
```

if E3 is positive, or

```
$( let N, Z = E1, E2
   while N ≥ Z do
         $( C
            N := N + E3  $)  $)
```

if E3 is negative. (The value of E3 is known at compile time.) Z is

an identifier not used elsewhere in the program. Also:

```
for N = E1 to E2 do C
```

is equivalent to

```
for N = E1 to E2 by 1 do C
```

The to and by clauses may be interchanged. Note that the initial value and end limit expressions E1 and E2 are evaluated only once. E3 must be a constant expression so that its sign is known at compile time.

## 5.11 Loop, Break, and Endcase Commands

Syntactic form:

```
loop
break
endcase
```

Examples:

```
for i = 1 to v!0 do
    $(  let x = v!i
        if x = 0 loop
        - - -
        - - -
        break
      L1:
    $)
L2:

switchon Op into
    $(  case SWITCHON:  Transswitch (x)
                        endcase
        case SEQ:       Trans (x!1)
                        Trans (x!2)
                        endcase
        - - -
    $)
L3:
```

Semantics:

Execution of the break command causes a jump to the point just after the smallest textually enclosing loop, introduced by one of the following key words:

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

In the example, this is the point labelled L2.

The loop command causes a jump to the end of the body of the smallest enclosing loop, so that the end condition is tested and the

loop repeated as required. In the example, this is the point labelled L1. In a for loop the loop command also causes the index to be incremented before the test is made (as usual).

The endcase command causes a jump to the point just after the smallest textually enclosing switchon block. In the third example, this is the point labelled L3.

## 5.12 Finish Commands

Syntactic form: finish

Example:

```
if Reportcount > Reportmax do
        $( WriteS ('*nToo many errors*n')
           Endwrite (OUTPUT)
           finish  $)
```

Semantics:

The finish command causes execution of the program to cease in an orderly manner. 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 x!H1 = S_COMMA do
                $( MapB (F, x!H3)
                   F (x!H2)
                   return $)
            F (x) $)1
```

Semantics:

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

## 5.14 Resultis Commands

Syntactic form: resultis E

Example:
```
valof $( for i = 0 to n do
             if V!i ≠ U!i resultis false
         resultis true $)
```

Semantics:

The execution of the command resultis E causes the execution of the smallest enclosing result block to cease and yield the value which is the Rvalue of E.

## 5.15 Switchon Commands

Syntactic form:
```
switchon E into <block>
where the block contains labels of the form:
case <constant>:
case <constant> to <constant>:
or default:
```

Example:
```
let Trans (x) be
    $(1 if x = 0 return
        switchon x ! H1 into
        $( default: Report (100); return
           case S_LET:  - - -
                        - - -
                        endcase
           - - -
           case S_SEQ:  Trans (x ! H2)
                        Trans (x ! H3)
                        endcase      $)1
```

Semantics:

The expression after switchon is evaluated to yield an Rvalue 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 the effect is undefined.

The case label

    case E1 to E2:

is equivalent to

    case E1: case E1 + 1: case E1 + 2: ... case E2:

where E2 must not be less than E1.

Note that the names S_LET 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 Call Commands

Syntactic form: call E0 (<E list>) | call E0 ()

Example: call Terminate (Name char 32, lv Code fixed)

Semantics:

In most implementations BCPL does not use the system standard call sequence. The call command provides a way of calling routines not written in BCPL. The nature of the argument list is extremely implementation dependent. In the implementation from which the example is drawn, the types of the arguments must usually be provided to the called program. This information is provided by infixed and postfixed operators which are not allowed in any other context.

## 5.17 Blocks

Syntactic form:

```
<block item> ::= C | <declaration>
<block body> ::= <block item> {; <block item>}
<block>      ::= $( <block body> $)
```

Example:

```
$( let List2 (x, y) = valof
       $( let P = Newvec (1)
          P ! 0, P ! 1 := x, y
          resultis P $)
   finish $)
```

Semantics:

A block body consists of a sequence of intermixed commands and declarations. It is executed by executing the declarations and commands in sequence.

The names declared by the declarations are local to the block and the dynamic storage cells allocated 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.Ø Definitions and Declarations

Before a name may be used in a BCPL 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, manifest constant, or address 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 name labelling a command is a form of declaration and it declares a name whose scope is the smallest enclosing routine or function body.

(c) A let declaration declares a name or set of names whose scope is the declaration itself and all succeeding commands and declarations within the smallest enclosing block body. A let declaration at the outer level of a program includes the rest of the program in its scope.

(d) A manifest, external, global, or static declaration declares a set of names whose scope is all succeeding commands and declarations within the smallest enclosing block body or program.

(e) 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 and the labels in the routine body must all 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 BCPL, 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 or longer.

(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

static declarations,
and global declarations.

Dynamic variables are declared by

simple variable definitions,
for commands,
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 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 to the segment in which they are declared.

Function and routine definitions, labels, and the manifest and external declarations do not introduce variables.

## 6.3 Let Declarations

Syntactic form: let D {and D}
where D denotes a definition

Example:
```
let x, y = Ø, 1
and f (t) = 2*t - 1
and ItermV = vec 22
```

Semantics:

A let declaration may occur in a block body or at the outer level of a program and may be used to declare simple variables, functions and routines. The scope of the names declared is the textual region of program consisting of the let declaration itself and the succeeding declarations and commands of the block. At the outer level of a program a let declaration may only declare functions and routines. The definitions between the ands 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 definitions are described below.

### 6.3.1 Simple Variable Definitions

Syntactic form: <name> {, <name>} = <E list>

All the names must be distinct and the number of names on the left of the = must be the same as the number of expressions on the right of the =.

Example:
```
let x = 1
and y, z = f (t) + 3, A!H2
and v = vec 5Ø
```

Semantics:

In the general form

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

dynamic data items with names N1, N2, ... Nn are first declared but not initialized, then the assignment command

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

is executed.

### 6.3.2 Function and Routine Definitions

Syntactic form:

```
<function definition> ::= <name> () = E |
                          <name> (<name list>) = E
<routine definition>  ::= <name> () be C |
                          <name> (<name list>) be C
```

The list of names in parentheses is called the formal parameter list.

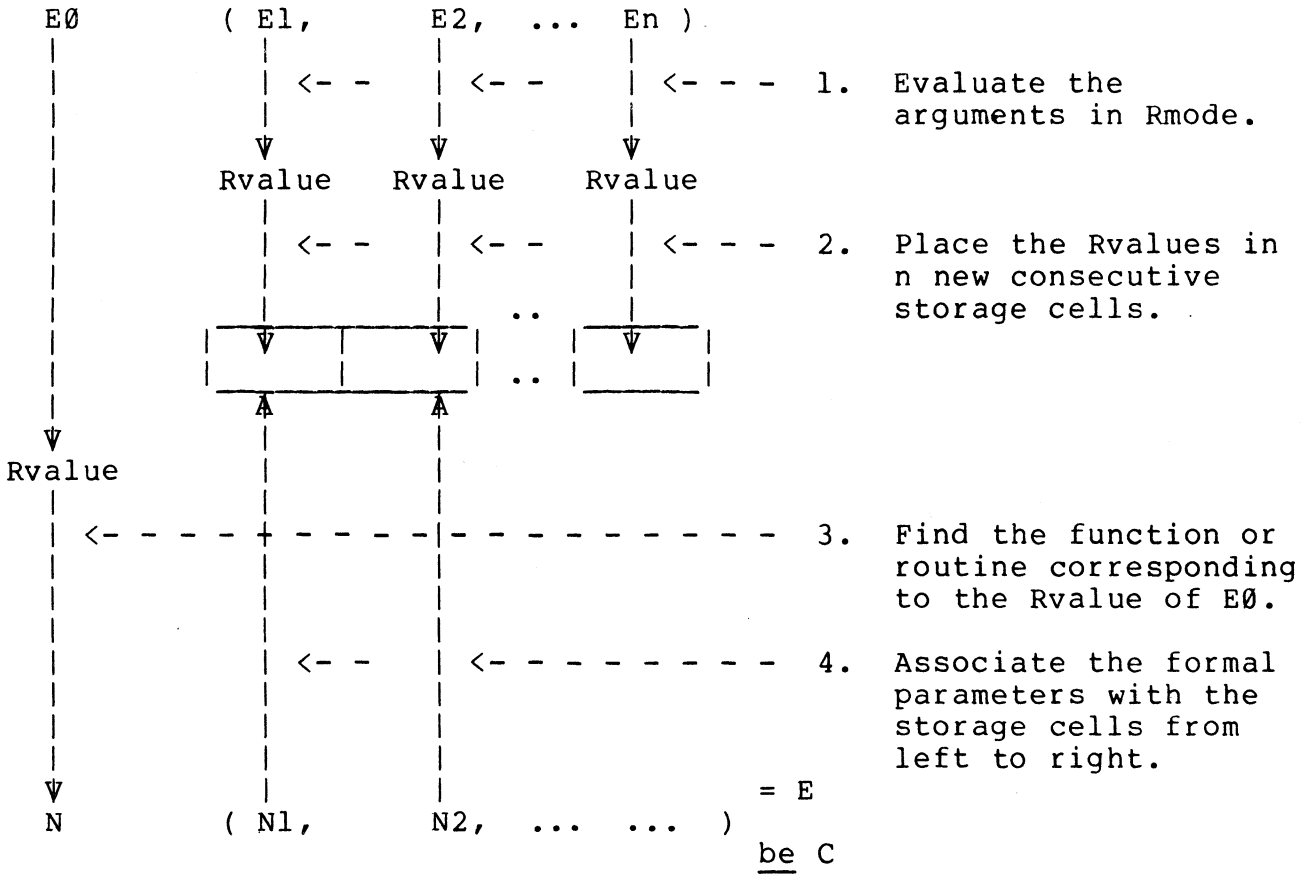
Example:

```
let Node (x) = valof
       $(  let P = Freelist
           Freelist := P + 3
           P!0, P!1, P!2 := x, 0, 0
           resultis P $)
and Put (x, t) be
       $(  if t!0 = x return
           t := t!0 < x -> t + 1, t + 2
           test rv t = 0
           then rv t := Node (x)
           or   Put (x, rv t)   $)
```

Semantics:

The purpose of a function or routine definition is to associate a name with an Rvalue 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 Figure 9.




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.

Figure 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 variadic 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 BCPL 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 = 0, 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.

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 a manifest constant or address constant or static variable (see section 6.2).

In terms of the implementation, this restriction states that either the Rvalue or the Lvalue of every free variable of a function or routine is known prior to execution (but not necessarily at compile time).

Note that the following program is illegal:

```
let a, b = 1, 2
let f (x) = a*x + b
```

However, it may be corrected as follows:

```
static $(  a = 1; b = 2 $)
let    f (x) = a*x + b
```

but this is not necessarily equivalent - e.g., if a or b is updated.

## 6.4 Manifest Declarations

Syntactic form:
```
manifest $( <decl item> {; <decl item>} $)
where  <decl item> ::= <name> = <constant>
```

Examples:
```
manifest $( H1=0; H2=1; H3=2 $)
manifest $( S_LET=74
            S_SEQ=73
            S_COMMA=38  $)
```

Semantics:

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. Any constant expression may be used.

## 6.5 Static Declarations

Syntactic Form:
```
static $( <decl item> {; <decl item>} $)
where  <decl item> ::= <name> = <constant>
```

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

Semantics:

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 initialization are performed prior to execution of the program. Thus the initialization is performed only once. Any constant expression may be used.

## 6.6 Global Declarations

Syntactic form:
```
global $( <decl item> {; <decl item>} $)
where  <decl item> ::= <name> : <constant>
```

Examples:
```
global $( Charcode:127; Option:128  $)
global $( Rdblockbody:140; Rdblock:141
          Rexp:144; Rdef: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.

## 6.7 External Declarations

Syntactic form: external $( <decl item> {; <decl item>} $)
where <decl item> ::= <name> = <constant>

Example: external $( Initiate = "hcs_$initiate" $)

Semantics:

The external declaration defines a set of names directly associated with Rvalues representing routines and functions in other separately compiled programs. The constant expression in the declaration is implementation dependent but will usually be a string constant representing the name of an "external reference".

The external declaration can also be used to make routines in this program known to other programs, as a result of the following rule:

> If a function or routine definition occurs within the scope of an external declaration with the same name, then the function or routine is defined as an "external symbol" with the name derived from the external declaration.

The connection between an external reference and the corresponding external symbol will be made by a loader (linker, binder) sometime before or during execution, the details depending on the operating system.

For example, the following segment will define an external function.

```
external $( F = "f$F" $)
let F (g,x) = g (x) + g (-x)
```

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

```
external
    $(  F = "f$F"
        Write = "library$Write"
    $)
let G (t) = t * t + t + 3
for i = 0 to 10 do  Write (F (G, i))
```

## References


[1] Barron, D. W. et al — "The Main Features of CPL" The Computer Journal, Vol. 6, 1963, p. 134.

[2] Strachey, C. — "CPL Working Papers" Cambridge University Mathematical Laboratory and London Institute of Computer Science (1965)

[3] IBM Reference Manual — 709/7094 FORTRAN Programming System, Form C28-6054-2

[4] Naur, P. (ed) — "Revised Report on the Algorithmic Language ALGOL 60" The Computer Journal, Vol. 5, January 1963, p. 349