MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LINCOLN LABORATORY

TO: TX-2 Users **DATE:** 6 May 1969

FROM: Henry Ancona

SUBJECT: The BCPL Reference Manual

This is the revised (hopefully correct) version of the TX-2 BCPL Manual. Please see me if you have any comments on the language or if the manual is unclear.

The next memo will describe the usage of BCPL on TX-2. The BCPL team consists of Tom Barkalow, Carl Ellison and myself.

HA:cn

The BCPL Reference Manual*

by

M. Richards
M.I.T. Project MAC
Cambridge, Massachusetts 02139

E.I. Ancona
M.I.T. Lincoln Laboratory
Lexington, Massachusetts 02173

ABSTRACT

BCPL is a simple recursive programming language designed for compiler writing and system programming; it was derived from true CPL (Combined Programming Language) by removing those features of the full language which make compilation difficult, namely, the type and mode matching rules and the variety of definition structures with their associated scope rules. BCPL on TX-2 differs from BCPL on CTSS (developed by M. Richards) by the addition of the external storage class, subword expressions, and the generalization of static storage. Global declarations have been removed.

*Work reported herein was supported (in part) by Project MAC, and MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract No. Nonr-4102 (01).

0.0 Index

---

* not implemented yet

1.0 Introduction

BCPL is a general purpose recursive programming language which is particularly suitable for large non-numerical problems in which machine independence is an important factor. It was originally designed as a vehicle for compiler construction and has, so far, been used in three compilers. BCPL is currently implemented and running on CTSS at Project MAC, the GE 635 under GE COS and on a KDF 9 at Oxford. Other implementations are under construction for MULTICS, the ICT 1900 series, Atlas, the System 360, and the TX-2. The language was orginally developed and implemented by M. Richards at Project MAC.

2.0 BCPL Syntax

The syntactic notation used in this manual is basically BNF with the following extensions:

(1) The symbols E, D, and C are used as shorthand for <expression> <definition> and <command>.

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

2.1 Hardware Syntax

The hardware syntax is the syntax of an actual implementation of the language and is, therefore, implementation dependent since it depends on the character set that is available. To simplify the transfer of BCPL from one machine to another, a set of canonical symbols has been developed. All compilers have a preprocessor which translates the symbols dictated by the hardware into the canonical symbols.

On TX-2, the character set which is used is the set of all capital letters and digits, in both black and red, together with the operators as

described in the next section, in black only. In this manual, the syntax is described in terms of large and small letters. Note that large letters correspond to black letters on TX-2, and small letters correspond to red letters on TX-2.

Non-printing characters, such as space, may be either black or red.

The canonical representation of a BCPL program consists of a sequence of symbols from the following set:

2.1.1 BCPL Canonical Symbols

Throughout the rest of this section words composed entirely of capital letters will be used as the names of canonical symbols. The names of all these symbols are given below together with corresponding examples of how they may be represented using the hardware representation adopted by this manual for TX-2. The list of black (large) words under 'hardware example' is the list of reserved words.

Canonical Symbol	Hardware Example	Described in Section
NUMBER	103 #777	4.4
NAME	abc i h2	4.1
STRINGCONST	'xyz*n' 'p'	4.2
CHARCONST	Σ a Σ 3	4.3
TRUE	TRUE	4.5
FALSE	FALSE	4.5
VALOF	VALOF	4.7
LV	LV	3.1, 4.10
RV	RV	3.1, 4.11
MULT	* or x	5.1
DIV	/	5.1
REM	REM	5.1
PLUS	+	5.1
MINUS	-	5.1
EQ	EQ	5.2
NE	NE	5.2
LS	LS	5.2
GR	GR	5.2
LE	LE	5.2
GE	GE	5.2
NOT	~ or NOT	5.4
LSHIFT	LSHIFT	5.3
RSHIFT	RSHIFT	5.3
LOGAND	∧ or LOGAND	5.4
LOGOR	∨ or LOGOR	5.4
EQV	EQV	5.4
NEQV	NEQV	5.4
COND	→	5.6
COMMA	,	5.6
TABLE	TABLE	5.7
AND	AND	7.10
ASS	=	6.1

Canonical Symbol	Hardware Example	Described in Section
GOTO	GOTO	6.5
RESULTIS	RESULTIS	6.16
COLON	←	6.4
TEST	TEST	6.10
FOR	FOR	6.12
IF	IF	6.6
UNLESS	UNLESS	6.7
WHILE	WHILE	6.8
UNTIL	UNTIL	6.9
REPEAT	REPEAT	6.11
REPEATWHILE	REPEATWHILE	6.11
REPEATUNTIL	REPEATUNTIL	6.11
BREAK	BREAK	6.13
RETURN	RETURN	6.15
FINISH	FINISH	6.14
SWITCHON	SWITCHON	6.17
CASE	CASE	6.17
DEFAULT	DEFAULT	6.17
LET	LET	7.2
MANIFEST	MANIFEST	7.5
BE	BE	7.2
SECTBRA	{6	2.1.2
SECTKET	}1	2.1.2
RBRA	(	4.6
RKET	)	4.6
SEMICOLON	‖	2.1.2
INTO	INTO	6.17
TO	TO	6.12
DO	DO or THEN	2.1.2, 6.12
OR	OR	6.10
VEC	VEC	7.7
VECAP	|	4.10

Note that the symbols NUMBER, NAME, STRINGCONST, SECTBRA and SECTKET denote composite symbols which have associated strings of characters.

2.1.2 Hardware Conventions and Preprocessor Rules

The Preprocessor is the name of the part of the BCPL compiler which transforms the raw source text of a program into canonical symbols. The hardware conventions in the TX-2 version are as follows:

(a)

A name is any sequence of red or black letters and digits, starting with a letter, which is not a reserved word. The character immediately following a name may not be a letter or a digit.

All reserved words are strings of black letters and digits.

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

```
LET R() BE // This routine refills the vector symb
{ FOR i = 1 TO 200 DO readch (input, LV symb! i)}
```

(c) Section brackets may be tagged with a sequence of letters and digits and two section brackets are said to match if their tags are identical. More than one section may be closed by a single closing section bracket since, on encountering a closing section bracket, if the current opening section bracket is found not to match then the current section is automatically closed by the insertion of an extra closing bracket. The process is repeated until the matching open section bracket is found. For example:

```
{1  UNTIL i EQ 0 DO
  {2  R(i)
    i = i - 1  }1
```

The final section bracket }1 does not match }2 and is, therefore, equivalent to }2 }1.

(d) The canonical symbol SEMICOLON is inserted by the Preprocessor between pairs of items if they appear on different lines and if the first

is from the set of items which may end a command or definition, namely:

BREAK RETURN FINISH REPEAT RKET

SECTKET NAME STRINGCONST NUMBER TRUE FALSE CHARCONST

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

TEST FOR IF UNLESS UNTIL WHILE GOTO RESULTIS

CASE DEFAULT BREAK RETURN FINISH SECTBRA SWITCHON

RBRA VALOF RV NAME RH LH Q1 Q2 Q3 Q4

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

RKET SECTKET NAME NUMBER

STRINGCONST TRUE FALSE CHARCONST

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 BREAK RETURN FINISH SWITCHON

(f) A directive of the form:

GET <specifier>

may be used anywhere in a BCPL program; it directs the compiler to replace the directive with the file or input stream of text referred to by the specifier. The exact syntactic form of the specifier is a string constant.

Example:

The following is a complete program segment for separate compilation: it is written in the TX-2 hardware representation (smalls = red, capitals = black) and exhibits some of the preprocessor rules. Note that it was not necessary to write a single double vertical bar (canonical semicolon) since they will all be inserted automatically.

```
GET 'head2' // This 'gets' the file called head2 which presumably
     //declares   checkdistinct, report and dvec

LET checkdistinct (E,S) BE
      {1 UNTIL E EQ S DO // The symbol { represents a SECTBRA

            { LET p = E + 4

              AND N = dvec|E
              WHILE P LS S DO
```

```
        { IF dvec|p EQ N DO report (142, N)
            p = p + 4 }
    E = E + 4}1 // Note that this closes
                // two sections.
```

3.0 Data Items

3.1 Rvalues, Lvalues and Conceptual Memory

The conceptual machine on which BCPL assumes it is implemented has a memory which is a vector of fixed length memory words (36 bits on TX-2).

Each memory word has a name, which is commonly known as its address, and in BCPL is known as an Lvalue. The contents of a memory word is known as an Rvalue.

3.2 Types

There is only one actual data type in BCPL. This is a bit pattern of a certain fixed length (36 bits in TX-2).

Of course, a programmer will use the bit patterns in different ways. These correspond to the conceptual data types. These include:

integer, logical, Boolean, function, routine, label, string, vector and Lvalue.

However, BCPL will never check, either at compile or run-time, whether the variable used has the correct conceptual type. To BCPL, it is simply a bit pattern and its context determines how it is to be interpreted.

(1) The Rvalue of a variable of conceptual type vector is a 36-bit pattern which is interpreted as the Lvalue of its zeroth element

i.e., v and LV v|o have the same Rvalue and also

RV v and v|o have the same Rvalue.

(2) The Lvalue of the n^{th} element of a vector v may be obtained by adding the integer n to v; thus LV v|n is equal to v + n

(3) If x, y and t are the first, second and n^{th} parameters of a function or routine and if v = LV x, then

v|o = x
v|1 = y
and v|(n - 1) = t

This property may be used to define functions and routines with a variable number of actual parameters. In the definition of such a function or routine

it is necessary to give a formal parameter list which is at least as long as the longest actual parameter list of any call for it.

Example:

The following definition

```
LET R(a, b, c, d, e, f) BE
   { LET v = LV a
     - - - -
     - - - -
           }
```

defines the routine R which may be called with 6 or less actual parameters. During the execution of the routine, the variable v may be used as a vector whose first n elements are the first n actual parameters of the call: thus during the following call

```
R( 126, 36, 18, 99 )
```

the initial Rvalues of

```
v|0, v|1, v|2, v|3 are 126, 36, 18, 99
```

(4) The Rvalue of a label is a bit pattern representing the program position of the labelled command. Note that it does not contain information about the activation level of the function or routine in which the label occurred.

(5) The Rvalue of a function or routine is a representation of the entry point of the function or routine.

3.3 Modes of Evaluation

In the assignment statement E1 = E2, where E1 and E2 are expressions, we assign E2 to E1.

E2 is evaluated in right hand mode, and E1 is evaluated in left hand mode. Both E1 and E2 have associated Lvalues and Rvalues. The result of an evaluation in right hand mode is used as an Rvalue, while an evaluation in left hand mode is used as an Lvalue.

Example:

```
X = Y    Lvalues: Xaddr    Yaddr
         Rvalues: Xcontent Ycontent
```

Y is evaluated in right hand mode, and yields Ycontent.

X is evaluated in left hand mode, and yields Xaddr.

Both Ycontent and Xaddr are 36-bit patterns and the effect is to assign Ycontent to the contents of the memory location Xaddr.

There exist two operators which change the mode of evaluation. These are LV and RV.

Example:

RV X = LV Y

Normally we would evaluate the right hand side to yield Ycontent. However, LV changes the mode to left hand evaluation to yield Yaddr.

Similarly, the left hand side is evaluated in right hand mode to yield Xcontent.

We still use Yaddr and Xcontent in their "normal" modes.

Thus, the bit pattern Yaddr is assigned to the contents of the memory location Xcontent.

4.0 Primary Expressions

A primary expression is any expression described in this section.

4.1 Names

Syntactic form:

A name is a sequence of one or more characters from a restricted alphabet called the name character alphabet. The hardware representation of characters in this alphabet and the rules for recognizing the starts and ends of names are implementation dependent.

The TX-2 hardware representation is as follows:

The name character alphabet contains the letters A. . . . Z (red) and a. . . . z (black) and the digits 0. . . . 9 (red or black) and these are all represented directly by the corresponding hardware characters. A name must start with a letter.

Semantics:

Two names are equal if they have the same sequence of name alphabet characters. A name may always be evaluated to yield an Rvalue. If the name was declared to be a manifest constant (see section 7.5) then the Rvalue will be the same on every evaluation· if the name was declared in any other way then it is a variable and its Rvalue may be changed dynamically by an assignment command. If N is a variable then its Lvalue is the Rvalue of the expression:

LV N

4.2 String Constants

Syntactic form:

'< string alphabet character >$_0$ '

The hardware representation of characters in the string alphabet is implementation dependent.

The TX-2 hardware representation is as follows:

The string character alphabet contains

all the hardware characters with the two exceptions of ' and * which are represented by *' and ** respectively.

In addition

*n	represents	newline
*s	"	space
*b	"	backspace
*t	"	tab

Semantics:

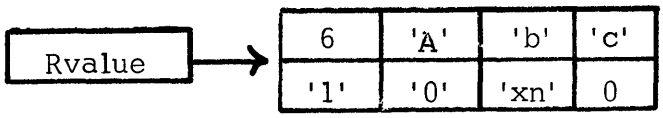
A string constant is represented as a BCPL vector: the length and the string characters are packed in successive words of the vector.

Example:

Characters are packed 4 per word so the string:

'Abc10*n'

is represented as follows:



4.3 Character Constants

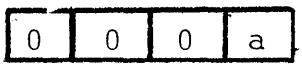
Syntactic form: Σ<character>

Semantics:

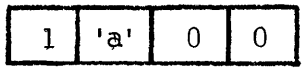
A character constant is a single character whose Rvalue is the bit pattern representation of the character; this is right justified and the word is filled with zeros.

Example:

The RV of Σa is



Note that the RV of the string constant 'a' is a pointer to the word:



4.4 Numerical Constants

Syntactic form: <digit>$_1$ or #<digit>$_1$

Semantics: The sequence of digits is interpreted as a decimal integer in the former case, and as a right justified octal number in the latter.

4.5 TRUE and FALSE

Syntactic form: TRUE or FALSE

Semantics: The Rvalue of TRUE is a bit pattern entirely composed of ones; the Rvalue of FALSE is zero.
Note that TRUE = ~ FALSE

4.6 Bracketted Expressions

Syntactic form: (E)

Semantics: Parentheses may enclose any expression; their sole purpose is to specify grouping.

4.7 Result Blocks

Syntactic form: VALOF <block>

Semantics: A result block is a form of BCPL expression; it is evaluated by executing the block until a RESULTIS statement is encountered, which causes execution of the block to cease and returns the Rvalue of the expression in the RESULTIS command.

4.8 Function Applications

Syntactic form: E1 (E2, E3, . . . En)

E1 is a primary expression.

Semantics: The function application is evaluated by evaluating the expressions E1, E2, . . . En and assigning the Rvalues of E2 . . . En to the first n - 1 formal parameters of the function whose Rvalue is the value of E1; this function is then entered. The result of the application is the Rvalue of the expression in the function

definition, see section 7.8. The precedence of a function application is higher than that of a vector application, i.e., y | f(x) is y | (f(x)).

4.9 Vector Applications

Syntactic form:

E1 | E2

where both E1 and E2 are primary expressions.

Semantics:

A vector is represented by a pointer to a consecutive group of words which are the elements of the vector. The pointer points to the zeroth element. To obtain the Rvalue of a vector application, the E1 and E2 are evaluated to yield two Rvalues, the first is interpreted as a vector pointer and the second as the subscript; the element is then accessed to yield the result.

The Lvalue of an element may be obtained by evaluating the expression

LV E1 | E2

The representations of Vectors, Lvalues and integers is such that the following relations are true:

E1 | E2 = RV (E1 + E2)

LV E1 | E2 = E1 + E2

Note that E1 | E2 | E3 | E4 is calculated as

((E1 | E2) | E3) | E4)

4.10 LV Expressions

Syntactic form:

LV E

E is a primary expression.

Semantics:

The Lvalue of some expressions may be obtained by applying the operator LV; it is only meaningful to apply LV to a vector application, an RV expression or an identifier which is not a manifest constant. The precedence of LV expressions is lower than that of vector applications, e.g., LV F(Y) | X is LV((F(Y)) | (X))

The result of the application depends on the leading operator of the operand as follows:

(a) A vector application.

The result is the Lvalue of the element referenced, see section 4.9 .

(b) An RV expression.

The result is the value of the operand of RV. The following relation is always true:

LV RV E = E

(c) A name.

The result is the Lvalue of the data item with the given name (which must not be a manifest constant). If the name was declared explicitly as a function, routine, static, external or label then its Lvalue is a manifest constant (but its Rvalue is not). See section 7.2.

4.11 RV expressions

Syntactic form: RV E

E is a primary expression.

Semantics: The value of an RV expression is obtained by evaluating its operand to yield an Rvalue which is then interpreted as the Lvalue of a data item. When evaluated in right hand mode, the result is the Rvalue of this data item. For left hand mode, see section 6.1. The precedence of RV expressions is lower than that of vector applications

4.12 Half word Expressions

Syntactic form: RH E or LH E

where E is a primary expression.

Semantics: When evaluated in right hand mode, the value of an RH or LH expression is the 36-bit pattern consisting of the right half or left half of E, respectively, with sign extension into the other half. (Configs 11 and 12) For left hand mode, see section 6.1. The precedence of half word expressions is lower than that of vector applications.

4.13 Quarter word Expressions

Syntactic form: Q1 E or Q2 E or Q3 E or Q4 E

where E is a primary expression.

Semantics:

When evaluated in right hand mode, the value of a quarter expression is the 36-bit pattern consisting of the appropriate quarter of E in quarter 1, with sign extension. (Configs 13-16.) For left-hand mode, see section 6.1. The precedence of quarter word expressions is lower than that of vector applications.

5.0 Compound Expressions

5.1 Arithmetic Expressions

Syntactic form:

E1 * E2 or E1 / E2 or E1 REM E2 or E1 x E2

E1 + E2 or + E1 or E1 - E2 or -E1

The operators * / and REM are more binding than + and - and associate to the right. The operators + and - associate to the left.

Semantics:

All these operators interpret the Rvalues of their operands as signed integers, and all yield integer results.

The operator * denotes integer multiplication.

The division operator / yields the correct result if E1 is divisible by E2; it is otherwise implementation dependent but the rounding error is never greater than 1.

The operator REM yields the remainder of E1 divided by E2; its exact specification is implementation dependent.

The operators + and - are self-explanatory.

5.2 Relational Expressions

Syntactic form:

E1 <relop> E2 . . . . <relop> En

where <relop> ::= EQ | NE | LS | GR | LE | GE
and n≥ 2

The relational operators are less binding than the arithmetic operators.

Semantics:

The result of evaluating an extended relation is true if and only if all the individual relations are true. The order of evaluation is undefined. The Rvalues of the expressions E1 . . . En are interpreted as signed integers and the relational operators have their usual mathematical meanings. Note, therefore that the value of an expression such as x EQ TRUE is implementation dependent.

5.3 Shift Expressions

Syntactic form: E1 LSHIFT E2 or E1 RSHIFT E2

E2 is any primary or arithmetic expression and E1 is any shift, relational, arithmetic or primary expression. Thus the shift operators are less binding than the relations on the left and more binding on the right.

Semantics: The Rvalue of E1 is interpreted as a logical bit pattern and that of E2 as an integer. The result of E1 LSHIFT E2 is the bit pattern E1 shifted to the left by E2 places. E1 RSHIFT E2 is as for LSHIFT but shifts to the right. Vacated positions are filled with zeros and the result is undefined if E2 is negative or greater than the data item size.

5.4 Logical Expressions

Syntactic form: $\sim$ E1 or E1 $\wedge$ E2 or E1 $\vee$ E2 or

E1 EQV E2 or E1 NEQV E2

The operator $\sim$ is most binding; then, in decreasing order of binding power are:

$\wedge$, $\vee$, EQV, NEQV.

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

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

The application of the operator $\sim$ yields the logical negation of its operand. The result of the application of any other logical operator is a bi pattern whose n^{th} bit depends only on the n^{th} bits of the operands and can be determined by the following table.

The values of the n^{th} bits	Operator $\wedge$	$\vee$	EQV	NEQV
both ones	1	1	1	0
both zeros	0	0	1	0
otherwise	0	1	0	1

5.5 Subword Expressions

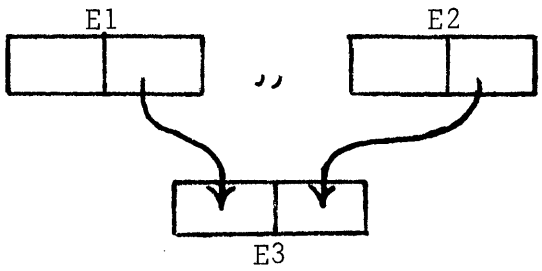
Syntactic form: E1,, E2

E1 and E2 may be any logical expression or exps of greater binding power

i.e., E1 → E2, E3,, E4 ∧ E5

parses as E1 → E2, (E3,, (E4 ∧ E5))

Semantics: E1,, E2 produces a 36-bit pattern whose left half is the same as the right half of E1, and whose right half is the same as the right half of E2.



5.6 Conditional Expressions

Syntactic form: E1 → E2, E3

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

Semantics: The value of the conditional expression E1 → E2, E3 is the Rvalue of E2 or E3 depending on whether the value of E1 represents true or false respectively. In either case only one alternative is evaluated. If the value of E1 does not represent either true or false then the result of the conditional expression is undefined.

5.7 Tables

Syntactic form: TABLE E0, E1, ... , En

where all the expressions are more binding than comma; however, only expressions composed of constants and the operators

+ - * / Σ ' TABLE VEC

are semantically allowable.

Semantics:

A table is a static vector whose elements are initialized prior to execution to the values of the expressions E0 to En; all these expressions must have values which can be computed at compile time. The Rvalue of a table is a pointer to its zeroth element. When used in this context, VEC denotes a static vector.

6.0 Commands

6.1 Simple Assignment Commands

Syntactic form: E1 = E2

Semantics: E1 may either be an identifier, a vector application, an RV expression, a half word expression, or a quarter word expression, and its effect is as follows:

(a) If E1 is an identifier:

The identifier must refer to a data item which has an Lvalue (i.e., it must not be declared as a manifest named constant). The assignment replaces the Rvalue of this data item by the Rvalue of E2.

(b) If E1 is a vector application:

The element referenced by E1 is updated with the Rvalue of E2.

(c) If E1 is an RV expression:

The operand of RV is evaluated to yield a value which is then interpreted as an Lvalue; the Rvalue of E2 then replaces the Rvalue of the data item referred to by the Lvalue.

(d) If E1 is a half word expression (RH or LH):

RH E3 = E4 is syntactic sugar for
E3 = LH E3,, E4

LH E3 = E4 is syntactic sugar for
E3 = E4,, E3

See section 5.5.

(e) IF E1 is a quarter word expression:

Q3 E3 = E4 is semantically equivalent to

E3 = (E3∧#777000777777) ∨ ((E4∧#777)LSHIFT 18)

i.e., put Q1 of E4 in Q3 of E3 without changing the rest of E3.

6.2 Assignment Commands

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

Semantics: The semantics of the assignment command is defined in terms of the simple assignment command; the command given above is semantically equivalent to the following sequence:

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

Note that the individual assignments are executed from left to right and not simultaneously.

6.3 Routine Commands

Syntactic form: E1 (E2, E3, . . . En)

where E1 is a primary expression.

Semantics: The above command is executed by assigning the Rvalues of E2, E3, . . ., En to the first n - 1 formal parameters of the routine whose Rvalue is the value of E1; this routine is then entered. The execution of this command is complete when the execution of the routine body is complete.

6.4 Labelled Commands

Syntactic form: N ⌐ C where N is a name.

Semantics: This declares a data item with name N; its scope is the smallest textually enclosing routine body or result block and its initial Rvalue is a bit pattern representing the program position of the command C. Its Lvalue is a manifest constant, and may be referenced from a separately compiled program if and only if the labelled command occurs within the scope of an external with the same name as the label. The Rvalue of a label is initialized prior to execution of the program.

6.5 Goto Commands

Syntactic form: GOTO E

Semantics: E is evaluated to yield an Rvalue, then execution is resumed at the statement whose

label had the same initial Rvalue.

6.6 If Commands

Syntactic form: IF E DO C

Semantics: E is evaluated to yield an Rvalue which is then interpreted as a truth value. See section 4.5 for the representation of Boolean values. If the value of E represent neither TRUE nor FALSE then the effect is implementation dependent.

6.7 Unless Commands

Syntactic form: UNLESS E DO C

Semantics: This statement is exactly equivalent to the following:

IF ~ (E) DO C

6.8 While Commands

Syntactic form: WHILE E DO C

Semantics: This is equivalent to the following sequence:

GOTO L

M ← C

L ← IF E GOTO M

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

6.9 Until Commands

Syntactic form: UNTIL E DO C

This statement is equivalent to

WHILE ~ (E) DO C

6.10 Test Commands

Syntactic form: TEST E THEN C1 OR C2

This statement is equivalent to the

following sequence:

```
        IF ~ (E) GOTO L
        C1
        GOTO M
    L ← C2
    M ←
```

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

6.11 Repeated Commands

Syntactic form:

C REPEAT or

C REPEATWHILE E or C REPEATUNTIL E

Where C is any command other than an IF, UNLESS UNTIL, WHILE, TEST or FOR command.

Semantics:

C REPEAT is equivalent to:

```
L ← C
    GOTO L
```

C REPEATWHILE E is equivalent to:

```
L ← C
    IF E GOTO L
```

C REPEATUNTIL E is equivalent to:

```
L ← C
    IF ~ (E) GOTO L
```

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

6.12 For Commands

Syntactic form:

FOR N = E1 TO E2 DO C

where N is a name.

Semantics:

The above statement is equivalent to:

```
{ LET N = E1
    UNTIL N GR E2 DO
      {C
        N = N + 1 } }
```

6.13 Break Commands

Syntactic form: BREAK

Semantics: When this statement is executed it causes execution to be resumed at the point just after the smallest textually enclosing loop command. The loop commands are those with the following key words:

UNTIL, WHILE, REPEAT, REPEATWHILE, REPEATUNTIL, and FOR.

6.14 Finish Commands

Syntactic form: FINISH

Semantics: This causes the execution of the program to cease.

6.15 Return Commands

Syntactic form: RETURN

Semantics: This causes a return from a routine body to the point just after the routine command which made the routine call.

6.16 Resultis Commands

Syntactic form: RESULTIS E

Semantics: This causes execution of the smallest enclosing result block to cease and return the Rvalue of E.

6.17 Switchon Commands

Syntactic form: SWITCHON E INTO <block>

where the block contains labels of the form:

CASE <constant> ☛ or

DEFAULT☛

Semantics: The expression is first evaluated and if a case exists which has a constant with the

same arithmetic value then execution is resumed at that label; otherwise, if there is a default label then execution is continued from there, and if there is not, execution is resumed just after the end of the switchon command.

The switch is implemented as direct switch, a sequential search or a tree search depending on the number and range of the case constants.

6.18 Blocks

Syntactic form: {<declaration>$_1$ <|| C>$_0$ } or
{C <|| C>$_0$ }

Semantics: A block is executed by executing the declarations (if any) and then executing the commands of the block in sequence.

The scope of the definee of a declaration is the region of program consisting of the declaration itself, the succeeding declarations and the command sequence.

7.0 Definitions

7.1 Scope Rules

The SCOPE of a name N is the textual region of program throughout which N refers to the same data item. Every occurrence of a name must be in the scope of a declaration of the same name.

There are three kinds of declaration:

(1) A formal parameter list of a function or routine: its scope is the function or routine body.

(2) The set of labels set by colon in a routine or result block: its scope is the routine or result block body.

(3) Each declaration in the declaration sequence of a block: its scope is the region of program consisting of the declaration itself, the succeeding declarations and the command sequence of the block.

Two data items are said to be declared at the same level of definition if they were declared in the same formal parameter list, as labels of the same routine or result block, or in the same definitions.

There are three semantic restrictions concerning scope rules, namely:

(a) Two data items with the same name may not be declared in the same level of definition.

(b) If a name N is used but not declared within the body of a function or routine, then it must either be a manifest named constant or a data item with a manifest constant Lvalue, that is it must have been declared as an external, an explicit function or routine, or as a label. Thus the following program is illegal:

```
LET  x = 1
LET  f(y) = x + y
```

since x is a dynamic data item, see sections 7.6 and 7.2.

(c) A label set by colon may not occur within the scope of a data item with the same name if that data item was declared within the scope of the label and was not an external.

7.2 Space Allocation and Extent of Data Items

The EXTENT of a data item is the time through which it exists and

has an Lvalue. Throughout the extent of a data item, its Lvalue remains constant and its Rvalue is changed only by assignment.

In BCPL data items can be divided into two classes

(1) Static data items:

Those data items whose extent lasts as long as the program execution time; such data items have manifest constant Lvalues. Every static data item must have been declared either in a function or routine definition, in an external or static declaration, in a TABLE expression, or as a label set by colon.

(2) Dynamic data items:

Those data items whose extent is limited; the extent of a dynamic data item starts when its declaration is executed and continues until execution leaves the scope of the declaration. Every dynamic data item must be declared either by a simple definition, a vector definition or as a formal parameter. The Lvalue of such a data item is not a manifest constant.

7.3 External Declarations

Syntactic form: EXTERNAL { <name > <// <name>>$_0$ }

Semantics: The external declaration declares a set of names to be used in common by separately compiled segments of a program. This storage must be initialized, i.e., one segment must declare the storage for the variables. Within the program segment where the name is declared, it must still appear in the external declaration. This declaration should be used by all function, routine, label or static definitions which are used by separately compiled program segments.

7.4 Static Declarations

Syntactic form: STATIC {<name> = <constant>}
{ , ‖<name> = <constants>>$_0$}

Semantics: This declaration declares each name to have a value equal to the value of its associated expression. Only expressions composed of constants and the operators

+ - * / Σ ' TABLE VEC

are allowable. When used in this context, VEC denotes a static vector.

7.5 Manifest Declarations

Syntactic form: MANIFEST {<name> = <constant>
< ,‖<name> = <constant> >$_0$ }

Semantics: This declaration declares each name to be a manifest constant with a value equal to the value of its associated constant expression. The meaning of a program would remain unchanged if all occurrences of manifest named constants were textually replaced by their corresponding values.

This facility has been provided to improve the readability of programs and to give the programmer greater flexibility in the choice of internal representations of data.

7.6 Simple Definitions

Syntactic form: LET N1, N2, . . . Nn = E1, E2, . . . En

Semantics: Data items with names N1 . . . Nn are first declared, but not initialized, and then the following assignment command is executed

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

A simple definition declares dynamic data items.

Note that all definitions must occur at the beginning of blocks.

7.7 Vector Definitions

Syntactic form: N = VEC <constant>

where N is a name.

Semantics: The value of the constant expression must be a manifest constant and it defines the maximum allowable subscript value of the vector N. The minimum subscript value is always zero. The initial Rvalue of N is the Lvalue of the zeroth element of the vector; both N and the elements of the vector are dynamic data items.

The use of a vector is described in section 4.9.

7.8 Function Definitions

Syntactic form: N(<namelist>$_1$_) = E

where N is a name.

Semantics: This defines a function with name N; the data item defined is static and has its Rvalue initialized prior to execution of the program. The Lvalue of N is a manifest constant, and refers to an external if it is in the scope of an external definition.

The names in the name list are called formal parameters and their scope is the body of the function E. The extent of a formal parameter lasts from the moment of its initialization in a call until the time when the evaluation of the body is complete.

All functions and routines may be defined and used recursively.

Function applications are described in section 4.8.

7.9 Routine Definitions

Syntactic form: N(<namelist>$_1$_) BE <block>

where N is a name.

Semantics: This defines a routine with name N. The semantics of a routine definition is exactly as for a function definition except that the body of a routine is a block and therefore its application yields no result. A routine should therefore only be called in the context of a command.

Routine commands are described in section 6.3.

7.10 Simultaneous Definitions

Syntactic form: D <AND D>$_0$

Semantics: All the definitions are effectively executed simultaneously and all the defined data items have the same scope which, by the scope rules given in 7.1, includes the simultaneous definition itself; a set of mutually recursive functions and routines may thus be declared.

REFERENCES

[1] Strachey, C. (Editor) "CPL Working Papers" a technical report, London Institute of Computer Science and the University Mathematical Laboratory, Cambridge (1966).

[2] Richards, M. "The BCPL Reference Manual", Project MAC Memo - M-352-1, M.I.T. Cambridge, Mass. (Feb. 1968).

[3] Richards, M. "BCPL: A Tool for Compiler Writing and System Programming", 1969 Spring Joint Computer Conference.