MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Memorandum-M-352
July 21, 1967

To: Project MAC Participants

From: Martin Richards

Subject: The BCPL Reference Manual

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.

0.0 Index

1.0 Introduction

BCPL is the heart of the BCPL Compiling System; It is a language which looks much like true CPL [1] but is, in fact, a very simple language which is easy to compile into efficient code. The main differences between BCPL and CPL are:

(1) A simplified syntax.

(2) All data items have Rvalues which are bit patterns of the same length and the type of an Rvalue depends only on the context of its use and not on the declaration of the data item. This simplifies the compiler and improves the object code efficiency but as a result there is no type checking.

(3) BCPL has a manifest named constant facility.

(4) Functions and routines may only have free variables which are manifest named constants or whose Lvalues are manifest constants (i.e., explicit functions or routines, labels or global variables).

(5) The user may manipulate both L and Rvalues explicitly.

(6) There is a scheme for separate compilation of segments of a program.

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 necessarily implementation dependent since it depends on the character set that is available. To simplify the description of any implementation of BCPL a canonical syntax has been defined and this is given in the section 2.2. The canonical representation of a BCPL program consists of a sequence of symbols from the following set.

2.1.1 BCPL Canonical Symbols

NUMBER NAME STRINGCONST TRUE FALSE

VALOF LV RV DIV REM PLUS MINUS

EQ NE LS GR LE GE NOT LSHIFT RSHIFT

LOGAND LOGOR EQV NEQV COND COMMA

AND ASS GOTO RESULTIS COLON TEST

FOR IF UNLESS WHILE UNTIL REPEAT REPEATWHILE

REPEATUNTIL BREAK RETURN FINISH SWITCHON CASE

DEFAULT LET MANIFEST GLOBAL

BE SECTBRA SECTKET RBRA RKET SBRA SKET

SEMICOLON INTO TO DO OR VEC STAR

The symbols NUMBER, NAME, STRINGCONST, SECTBRA and SECTKET denote composite items which each have an associated sequence of characters.

2.1.2 Hardware Conventions and Preprocessor Rules

(a) If the implementation character set contains both capital and small letters then the following conventions hold:

(1) A name is either a single small letter or a sequence of letters and digits starting with a capital letter. The character immediately following a name may not be a letter or a digit.

(2) A sequence of two or more small letters which is not part of a NAME, SECTBRA, SECTKET or STRINGCONST is a reserved system word and may be used to represent a canonical symbol. For example:

let and logor could be used to represent LET and LOGOR but Let and Logor are names.

(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.

(d) The canonical symbol SEMICOLON is inserted 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:

BREAK RETURN FINISH REPEAT SKET RKET

SECTKET NAME STRINGCONST NUMBER TRUE FALSE

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

RBRA VALOF LV RV NAME

(e) The canonical symbol DO is inserted between pair 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:

SKET RKET SECTKET NAME NUMBER

STRINGCONST TRUE FALSE

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

(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 implementation dependent but it will usually be either a string constant or an integer.

Example:

The following is a complete program segment for separate compilation; it is written in a fictitious hardware representation and exhibits some of the preprocessor rules. Note that it was not necessary to write a single 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=S do  //  The symbol $( represents a SECTBRA

                  $( let p = E + 4
                    and N = Dvec*[E]
                    while p ls S do    // Note that ls is a
                                       //  system word, p is a name.
                         $( if Dvec*[p]=N do Report [142, N]
                             p := p + e $)
                    E := E + e    $)1   // Note that this closes
                                        // two sections.
```

2.2 Canonical Syntax

The syntax given in this section defines all the legal constructions in BCPL without specifying either the relative binding powers or association rules of the command and expression operators. These are given later in the manual in the descriptions of each construction.

To improve the readability of the syntax a hardware representation of the canonical symbols has been used. For instance:

( ) [ ] § <u>§</u> are used to denote RBRA RKET SBRA SKET SECTBRA and SECTKET.

```
E               ::= <name> | <stringconst> } <number> | true | false
                    ( E ) | valof <block> | lv E | rv E | E [ E ] | E*[ E ]
                    E <diadic op> | ~E | +E | -E | E→ E, E } E < , E >_0

<diadic op>   ::= * | / | rem | + | - | = | ≠ | < | > | ≥ | lshift + rshift | ∧ | ∨ | ≡
```

[also, less-or-equal and notequivalent; unrepresented in available font-- DMR]

```
<constant>        ::=  E

D                 ::=  D < and D>_0 | <name> [ <namelist>_1- ] = E |
                       <name> [ <namelist>_1- ] be <block> |
                       <namelist> = E | <name> = vec <constant>

<namelist>        ::= <name> <, <name> >_0

C                ::=  E :=  E | E[ E ] | E [ ] | goto E | <name> : < C >_1-
                     if E do C | unless E do C | while E do C | until E do C |
                     C repeat | C repeatwhile E | C repeatuntil E |
                     test E then C or C | for <name> = E to E do C |
                     break | return | finish | resultis E |
                     switchon E into <block> | case <constant> : < C >_1- |
                     default : < C >_1- | <block>

<block>          ::= § C < ; C >_0 §  | § < let D | <constdef> >_1 <; C >_0 §

<constdef        ::= < manifest | global > § <name> < = | : > <constant>
                         < ; <name> < = | : > <constant> >_0 §
```

3.0 Data Items

3.1 Rvalues, Lvalues and Data Items

An RVALUE is a binary bit pattern of a fixed length (which is implementation dependent), it is usually the size of a computer word. Rvalues may be used to represent a variety of different kinds of objects such as integers, truth values, vectors or functions. The actual kind of object represented is called the TYPE of the Rvalue.

A BCPL expression can be evaluated to yield an Rvalue but its type remains undefined until the Rvalue is used in some definitive context and it is then assumed to represent an object of the required type. For example, in the following function application

```
(B*[i] → f, g) [1, Z[i]]
```

the expression (B*[i] → f, g) is evaluated to yield an Rvalue which is then interpreted as the Rvalue of a function since the expression occurred in the operator position of a function application; whether f and g are in fact functions is not explicitly checked. Similarly the expression B*[i] (which is a vector application) occurs where are boolean is expected and so its Rvalue is interpreted as a truth value.

There is no explicit check to ensure that there are no type mismatches.

An LVALUE is a bit pattern representing a storage location containing an Rvalue. An Lvalue is the same size as an Rvalue and is a type in BCPL. There is one context where an Rvalue is interpreted as an Rvalue and that is as the operand of the monadic operator <u>rv</u>. For example, in the expression

<u>rv</u> f[i]

the expression f[i] is evaluated to yield an Rvalue which is then interpreted as an Lvalue since it is the operand of <u>rv</u>. The application of <u>rv</u> on this Lvalue yields the Rvalue which is contained in the location represented (or referred to) by the Lvalue.

An Lvalue may be obtained by applying the operator <u>lv</u> to an identifier, a vector application or an <u>rv</u> expression, see section 4.9.

A DATA ITEM is composed of an Lvalue, the referenced Rvalue and possibly an associated identifier. The term is used loosely to mean the Lvalue, Rvalue or identifier depending on context.

In BCPL a data item may have at most one identifier. The following diagram shows a data item for a six bit machine.

[Diagram with broken line from x to box with 0 0 0 1 0 1,
and also to 001101; unbroken arrow from that to the 0 0 0 1 0 1 box -- DMR]

The broken line indicates the semantic association between an identifier x and the storage location which is represented by the box; the unbroken arrow shows the correspondence between the Lvalue 001101 and the Rvalue 000101, and the broken arrow shows the correspondence between the identifier x and the Lvalue 001101.

It is meaningful to say that the Lvalue of the data item x is the bit pattern 001101, that the Rvalue of the data item x is the bit pattern 000101 and that the Lvalue 001101 refers to the data item x.

3.2 Types

An Rvalue may represent an object of one of the following types:

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

Although the bitpattern representations of each type is implementation dependent certain relations between types is not.

(1) The Rvalue of a vector v, say, is identical to the Lvalue of its zeroth element:

```
v = lv v*[0]
```

and hence

```
rv v = v*[0]
```

(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] = 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*[0] = x
    v*[1] = y
and v*[n] = 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 are Rvalues of

```
v*[0], v*[1], v*[2] and v*[3] are 126, 36, 18 and 99
```

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.

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

4.0 Primary expressions

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.
One hardware representation is as follows:
The name character alphabet contains the letters A .... Z and a .... z and the digits 0 .... 9 and those are

all represented directly by the same hardware characters. A name either starts with a capital letter and is terminated by the first non-letter or digit, or it is a single small 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 7.4) 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:

<u>lv</u> N

4.2 String Constants

Syntactic form: ' <string alphabet character>$_0$ '

The hardware representation of characters in the string alphabet is implementation dependent. One hardware representation is as follows:

The string character alphabet contains all the characters except * and ' are represented directly. These two exceptions are represented by

** and *' respectively.

In addition

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

Semantics: A string constant of length one has an Rvalue which is the bit pattern representation of the character; this is right justified and filled with zeros.

A string constant with length other than one is represented as a BCPL vector; the length and the string characters are packed in successive words of the vector.

Example:

If characters are packed 4 per word then the string:
'Abc10*n'
is represented as follows:

[picture with Rvalue pointing at box with
6 'A' 'b' 'c'
*'1' '0' '*n' 0*
--DMR]

4.3 Numerical Constants

Syntactic form: <digit>$_1$ or 8 <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.

4.4 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

4.5 Bracketted Expressions

Syntactic form: (E)

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

4.6 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, this causes execution of the block to cease and returns the value of the expression in the resultis command.

4.7 Vector Applications

Syntactic form: E1 * [E2]

The asterisk is necessary to distinguish a vector application from a function application. E1 is a primary expression.

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 evaluate a vector application 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

<u>lv</u> E1 * [E2]

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

E1*[E2] = <u>rv</u> (E1+E2)

lv E1 * [E2] = E1 + E2

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 7.7.

4.9: 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 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 4.7.

(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, global or label then its Lvalue is a manifest constant (but its Rvalue is not), see section 7.2.

4.10 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. The result is the Rvalue of this data item.

5.0 Compound Expressions

5.1 Arithmetic Expressions

Syntactic form: **E1 * E2 or E1 / E2 or E1 rem E2 or**
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 of *[the division of E1 by E2 if]* **El 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 El divided by E2; its exact specification is Implementation dependent.

The operators + and − are self−explanatory.

5.2 Relational Expressions

Syntactic form: **El <relop> E2 .... <relop> En**
where <relop> ::= = | ≠ | < | > | ≤ | >=
and n >= 2 *[Lack of real >= character is a font problem]*

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 El ... En are interpreted as signed integers and the relational operators have their usual mathematical meanings.**

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: ~ E1 or E1 ∧ E2 or E1 ∨ E2 or
E1 ≡ E2 or E1 NEQV E2
[NEQV was a crossed triple-bar]

The operator ~ is most binding; then, in decreasing order of binding power are:

∧ ∨ ≡ 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 ~ yields the logical negation of its operand. The result of the application 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 by the following table.

The values of the		Operator		
nth bits	∧	∨	≡	NEQV
both ones	1	1	1	0
both zeros	0	0	1	0
otherwise	0	1	0	1

5.5 Conditional Expressions

Syntactic form: $E1 \rightarrow E2, E3$

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

Semantics: The value of the conditional expression $E1 \rightarrow E2, E3$ 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.

6.0 Commands

6.1 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.

6.2 Simple Assignment Commands

Syntactic form: E1 :   E2

Semantics: E1 may either be an identifier, a vector application or an rv 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.

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 refers to a position in the global vector (see section 7.3) if and only if the labelled command occurs within the scope of a global 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. In BCPL, **false** is represented by zero and true by the complement of **false**.

~ **false**

If the value of E represents **false** then the command C is not executed; if it represents **true** then it is executed and if it represents 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

Semantics: This statement is equivalent to

while ~ (E) **do** C

6.10 Test Commands

Syntactic form: test B then C1 or C2

Semantics: 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 B 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 > 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, then if a case exists which has a constant with the same value then execution is resumed at that label; otherwise if there is a default label then execution is continued from there; otherwise execution is resumed just after the end of the switchon command.

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

6.18 Blocks

Syntactic form: § C < ; C $>_0$ § or
§ < let D | <constdef> $>_1$.< ; C $>_0$ §

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

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 of 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 definition.

There are three semantic restrictions concerning scope rules; these are:

(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 a global, an explicit function or routine, or as a label.

(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 a global.

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 extents 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 a global declaration 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.

A data item is initialized at most once at the start of its extent; static data items are intialized prior to execution of the program and a dynamic data item at the time when its declaration is executed. Both static and dynamic data items may have their Rvalues changed by assignment.

During the execution of a recursive function or routine, a single textural declaration may give rise to more than one activation of its definee. The only declarations which can give rise to multiple activations are those that declare dynamic data items namely:

simple definitions, vector definitions and formal parameters.

7.3 Global Declarations

Syntatic form: global § <name> : <constant>
< ; <name> : <constant> >$_0$~ §

Semantics: The global vector is the sole means of communication between separately compiled segments of a program. To call a function or routine which is declared in one segment from a position in another it is necessary to declare it as a global in each of the two segments.

The above declaration declares a set of names to be global and allocates the positions in the global vector as defined by the manifest constants.

A global variable is a static data item and has an Lvalue which is a manifest constant.

7.4 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.

7.5 Simple Definitions

Syntactic form: 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.

7.6 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 4.7.

7.7 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 a position in the global vector if and only if the function definition is in the scope of a global definition of the same name.

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.

7.8 Routine Definitions

•Syntactic form: N [<namelist>$_{1-}$] <u>be</u> <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.

7.9 Simultaneous Definitions

Syntactic form: D < <u>and</u> 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.

8.0 Example Program

The program given in this section is part of the BCPL library used in the BCPL compiler itself.

The hardware representation was specially designed to suit an IBM 1050 typewriter with a 938 golf ball. Note that the symbols '(' and ')' represent the canonical symbols SBRA and SKET respectively.

In the example (which was run on the Project MAC 7094 computer), two files were printed out and then the former was compiled into relocatable binary.
It is hoped that this example exhibits the readability of BCPL as well as some of its features.

[but, regrettably, the example doesn't appear in the copy I have -- DMR]