BCPL

Technical Memorandum No 69/1

The BCPL Reference Manual

by

Martin Richards

January 1969

The University Mathematical Laboratory
Corn Exchange Street,
Cambridge.

Abstract

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

Contents

1.0 Introduction

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 three compilers. BCPL is currently implemented and running on CTSS at Project MAC, the GE 635 under GECOS and on MULTICS. 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 (or 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, and so only a small proportion (about 1/5th) of the compiler needs to be rewritten for a new implementation. In addition to modifying the compiler, it is necessary to design and write the interface with the new operating system; this is usually written in assembly language and its length is likely to be between 200 and 1000 instructions.

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

2.0 Hardware Representations and Syntax

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

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

let LET let .LET

2.1 Table of Canonical Symbols

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

Name of Canonical Symbol	Hardware Examples
NUMBER	103 2 6000
NAME	Abc H2 i Tax_rate
STRINGCONST	"A" "" "36*t"
CHARCONST	'p' ':' '*n'
TRUE	true
FALSE	false
OCT	8 oct $8
VALOF	valof
VECAP	!
LV	lv
RV	rv
DIV	/
REM	rem
MULT	*
PLUS	+
MINUS	-
EQ	= eq
NE	≠ ne
LS	< ls
GR	> gr
LE	≤ le
GE	≥ ge
NOT	¬ not
LSHIFT	lshift
RSHIFT	rshift
LOGAND	∧ &
LOGOR	∨ logor
EQV	≡ eqv
NEQV	≠ neqv
COND	→ ->
COMMA	,
TABLE	table
AND	and
ASS	:=

Name of Canonical Symbol	Hardware Examples
GOTO	goto
RESULTIS	resultis
COLON	:
TEST	test
FOR	for
IF	if
UNLESS	unless
UNTIL	until
REPEAT	repeat
REPEATWHILE	repeatwhile
REPEATUNTIL	repeatuntil
BREAK	break
RETURN	return
FINISH	finish
SWITCHON	switchon
CASE	case
DEFAULT	default
LET	let
MANIFEST	manifest
GLOBAL	global
STATIC	static
BE	be
SECTBRA	$($(6
SECTKET	$) $)Trans
RBRA	(
RKET	)
SEMICOLON	;
INTO	into
TO	to
DO	do then
OR	or
VEC	vec

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

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

2.2.1 Syntactic Notation

The syntax given in this manual is Bachus 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 repetion, for example:

$$E \left\{ , \; E \right\}^{\infty} \text{ means}$$
$$E \mid E , E \mid E , E , E \mid \ldots \text{ etc}$$

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

Expression

E ::= <name> | <stringconst> | <charconst> | <number> |
true | false | (E) | valof <block> | lv E | rv E |
E(<E list>) | E() | E <diadic op> E | <monadic op> E |
E → E, E | table <constant> { , <constant> }$^{\infty}_{0}$

<diadic op> ::= ↓ | * | / | rem | + | - |
= | ≠ | ls | gr | le | ge |
lshift | rshift | ∧ | ∨ | ≡ | ≢

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

<E list> ::= E { , E }$^{\infty}$

<constant> ::= E

Statement

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

declaration

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

<name list> = <E list> | <name> = vec <constant>

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

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

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

<block body> ::= C { ; C }* | { <declaration> }+ { ; C }+

<declaration> ::= let D { and D }* | static <decl body>

manifest <decl body> | global <decl body>

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

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

<program> ::= <block body>

2.3 Hardware Representations

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

2.3.1 Names and System Words

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

For character sets with capital and small letters:

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

(2) A name is either

  (a) a single small letter

  (b) a capital letter followed by any sequence of letters, digits and possibly other suitable characters (e.g. #. #)

For character sets with only capital letters:

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

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

(3) A name is an identifier which is not a system word.

Thus on some implementations let and logor are system words while Let, 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

nur . ergänzen!

2.3.2 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
```

2.3.3 Automatic Insertion of SEMICOLON

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

```
R(x, y)(B → f, g)(1, 2)   // possible omission of ';' before (B
```

A simple rule which is guaranteed to be safe is:

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

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

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

Example: the following two programs are equivalent

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

2.3.4 Automatic Insertion of DO

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

Example: the following programs

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

is equivalent to

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

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

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

2.3.5 Comments

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

```
let R() be //  this is a routine which refills Symb
    $( for i = 1 to 200 do
                Readch (INPUT, lv Symb ! i)  $)
```

2.3.6 The "Get" Directive

A directive of the form

get <specifier>

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

nicht implementiert !

2.3.7 An Example Program

```
//  This is an example of a hardware representation of BCPL using
// both capital and small letters.

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

3.0 Fundamental Concepts of BCPL

3.1 The Object Machine

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

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

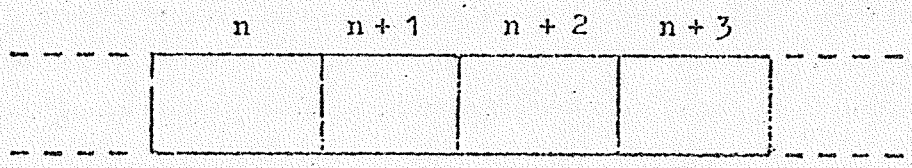


Fig. 1 - The Machines Store

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

Each storage cell holds a binary bit pattern called an Rvalue (or Right hand value). All storage cells are of the same size and the length of Rvalues is a constant of the implementation which is usually between 24 and 36 bits. An Rvalue is the only kind of object which can be manipulated directly in BCPL and the value of every variable and expression in the language will always be an Rvalue.

Rvalues are used by the programmer to model abstract objects of many different kinds such as truth values, strings and functions, and there are a large number of basic operations on Rvalues which have been provided in order to help the programmer model the transformation of his abstract objects. In particular, there are the usual arithmetic operations which operate on Rvalues in such a way that they closely model integers. One can either think of these operations as ones which interpret their operands as integers, perform the integer arithmetic and convert the result back into the Rvalue form, alternatively one may think of them as operations which work directly on bit patterns and just happen to be useful for representing integers. This latter approach is closer to the BCPL philosophy. Although the BCPL programmer has direct access to the bits of an Rvalue, the details of the binary representation used to represent integers is not defined and he would be losing machine independence if he performed nonnumerical operations on Rvalues he 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 integer and it locates the storage cell which is labelled by this integer. This operation is assumed to be efficient and, as will be seen later, the programmer may invoke it from within BCPL using the rv operator.

3.2 Variables and Manifest Constants

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

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

3.3 Lvalues and Modes of Evaluation

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

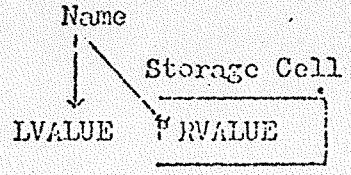


Fig. 2. - The form of a variable

Within the machine an Lvalue is represented by a binary 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 ideas 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 left hand expression E1 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 fig. 3.

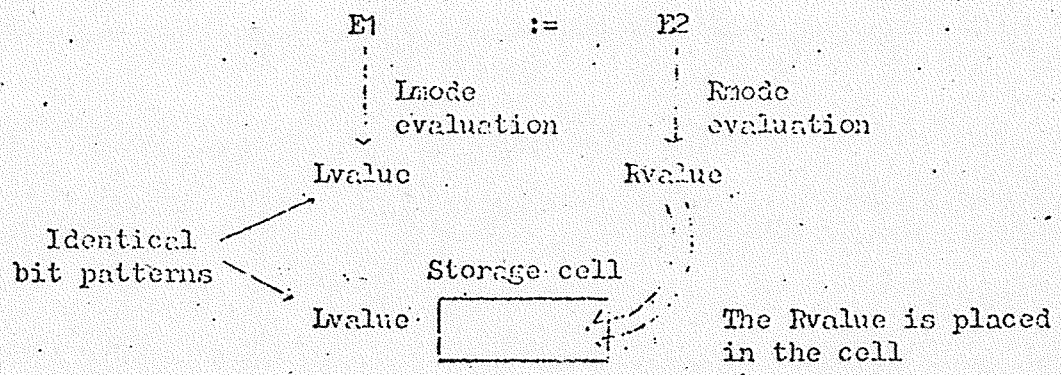


Fig. 3. - The process of assignment

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

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

3.5 The Lv Operator

As previously stated an Lvalue is represented by a binary 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 expressions is:

lv E

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

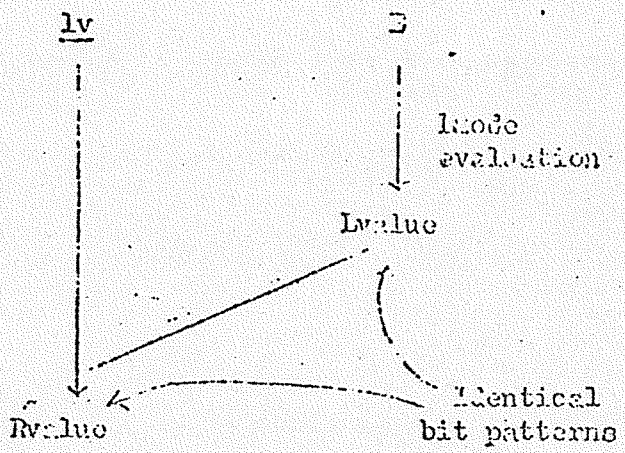


Fig. 4. - The evaluation of an lv expression

The operand is evaluated in Lmode to yield an Lvalue and the result is a bit pattern identical to this Lvalue. 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 fig. 5.

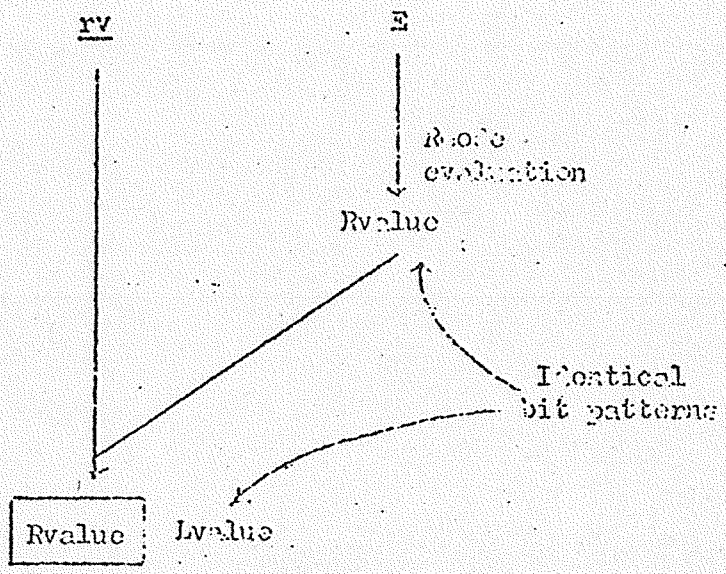


Fig. 5. - The evaluation of an rv expression

The operand is evaluated in Rmode and then the storage cell whose Lvalue is the identical bit pattern is found. If the rv expression is being evaluated in Rmode, then the contents of the cell is the result; however, it is also meaningful to evaluate it in 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.

3.7 Data Structures

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

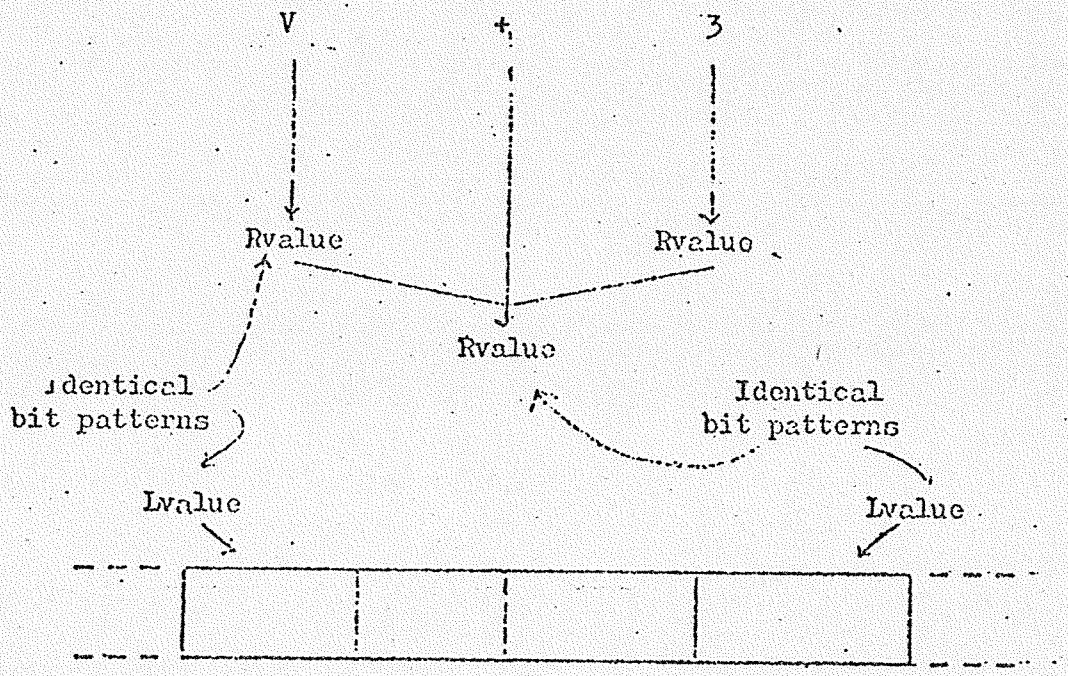


Fig. 6. - An interpretation of V + 3

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

Thus the expression:

rv (V + i)

acts very like a vector application, since, as i varies from zero to three, the expression refers to the different elements of the set of four cells pointed to be V. V can be thought of as the vector and i as the integer subscript.

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

E1↓E2 is equivalent to rv (E1 + E2)

A simple example of its use is the following command:

V↓(i + 1) := V↓i + U↓i

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

V↓3 ≡ rv (V + 3) by definition
≡ rv (3 + V) since + is commutative
≡ 3↓V

Thus V↓3 and 3↓V are semantically equivalent; however, it is useful to attach different interpretations to them. We have already seen an interpretation of V↓3 so let us consider the other expression. If we rewrite 3↓V as Xpart↓V where Xpart has value 3, we can now conveniently think of this expression as a selector (Xpart) applied to a structure (V). This interpretation is shown in fig. 7.

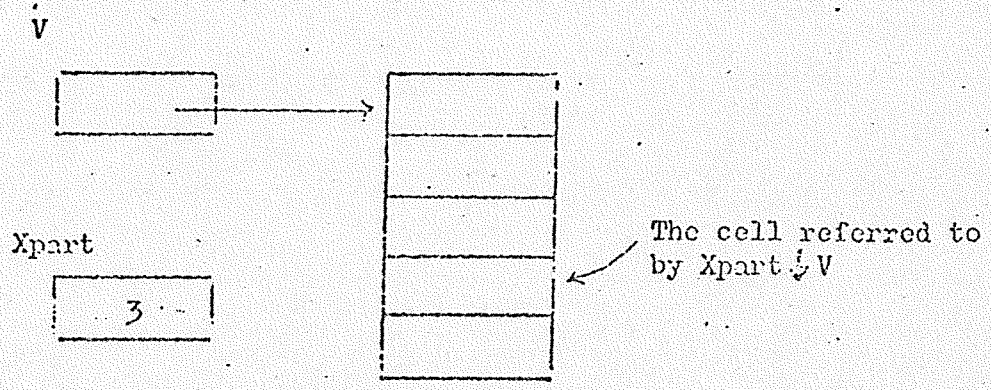


Fig. 7. - An interpretation of Xpart V

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

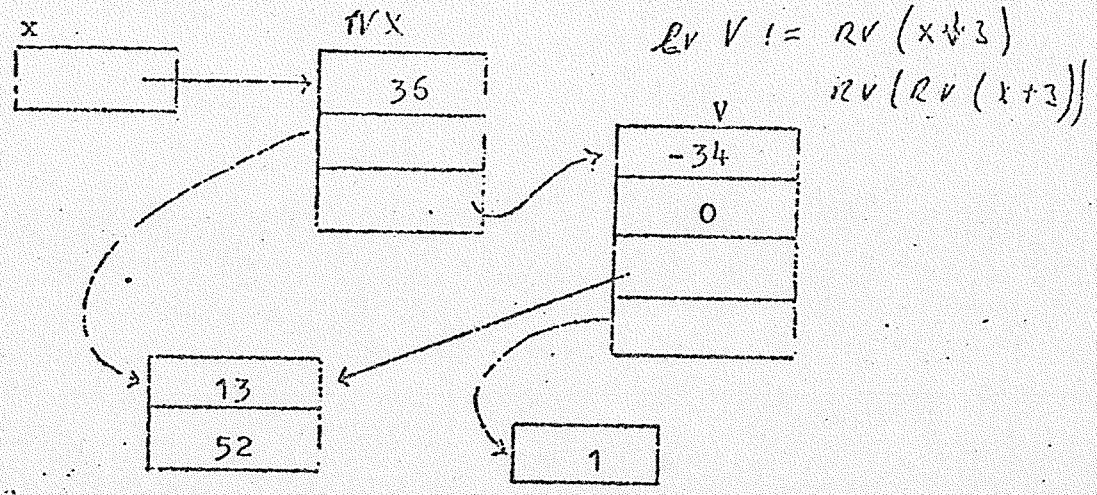


Fig. 8. - A structure of integers and pointers

3.8 Data Types

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

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

1. There is no need for type declarations in the language, since the type of every variable is already known. This helps to make programs concise and also simplifies such linguistic problems as the handling of actual parameters and separate compilation.

2. It gives BCPL much of the power as a language with dynamically varying types and yet retains the efficiency of a language (like FORTRAN [3] with manifest types; for, although the internal type of an expression is always known by the compiler, its conceptual type can never be and, indeed, it may depend on the values of variables within the expression. For instance, the conceptual type of 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 slight disadvantage is easily outweighed by the simplicity, power and efficiency that this treatment of types makes possible.

4.0 Expressions

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

(a) Primary expressions

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

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

(b) Arithmetic expressions.

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

(c) Relational expressions.

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

(d) Shift expressions.

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

(e) Logical expression.

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

be used on boolean data.

(f) Conditional expressions.

A conditional expression allows for conditional evaluation of one of two expressions.

(g) Tables.

A table is a static vector whose elements are preset with specified values.

4.1 Primary Expressions

All the primary expressions are described in this section.

4.1.1 Names

Syntactic form:

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

Examples:

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

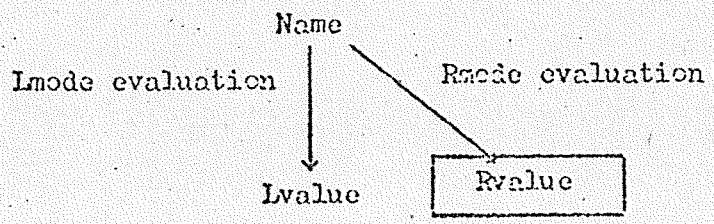
Semantics:

A name may be associated directly with an Rvalue by means of a manifest declaration or it may be associated with a storage cell to form a variable using any other kind of declaration. A variable or manifest constant can be referred to by its name throughout the

scope of its declaration (see section 6.0 on scopes and extents of definitions).

A manifest constant can only be evaluated in 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> }*

Examples: 132 43179 8377

Semantics:

A number is an Rtype expression and may only be evaluated in Rmode. The symbol 8 introduces an octal constant whose Rvalue is the right justified bit pattern specified by the sequence of digits.

A decimal number is a sequence of digits not preceded by 8; its Rvalue is a bit pattern representing the integer in a way which depends on the implementation.

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 character (*) is used as an escape character with the following conventions:

*n	represents	newline
*s	represents	space
*b	represents	backspace
*t	represents	tab
*"	represents	"
*'	represents	'
**	represents	*

Examples:

"End of test" "=" "*""

"*n*tTRA*tL1*n" " "

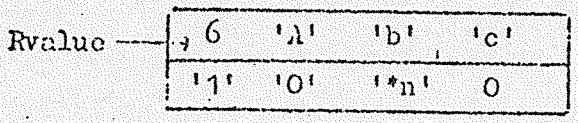
Semantics:

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 not defined. For an implementation which packs four characters per word, the string

"Abc10*n"

might be represented as follows:



4.1.4 Character Constants

Syntactic form: '<string characters>'

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.

4.1.6 Bracketted Expressions

Syntactic form: (E)

Examples: T rem ((x-y)/(x+y) + 2/z)

(B -> A, B) ↓ (i+1)

Semantics:

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

4.1.7 Result blocks

Syntactic form: 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.

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

4.1.9 Rv Expressions

Syntactic form: rv E where E is 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 in section 3.6.

4.1.10 Vector and Structure Applications

Syntactic form: E1 ↓ E2

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

x↓y↓z means (x↓y)↓z

Examples:

```
V↓(i+1) := V↓i + Xpart↓p

case SEQ: Trans( M2↓x)
          Trans( M3↓x)
          return
```

Semantics:

The expression E1↓E2 is defined to be equivalent to rv (E1 + E

Its purpose is explained in section 3.7.

4.1.11 Function Applications

Syntactic form: E0(E1, E2, ... En)

where E0 is a primary expression and E1 to E are any expressions. The list of expressi may be empty.

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 6.3.3.

4.2 Arithmetic Expressions

Syntactic form:

E * E | E/E | E rem E |
E + E | E - E |
+ E | - E

The operators * / and rem 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)

Semantics:

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

The operator * and / denotes integer mutiplication 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.

4.3 Relational Expressions

Syntactic form: $E \left\{ \text{<relop>}\ E \right\}_1^{\infty}$ where

<relop> ::= = | ≠ | ls | gr | le | ge

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

Examples: if 0 le x ls y goto L

A↓i := f(x)=g(x)

Semantics:

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

Operator	Meaning
=	equal to
≠	not equal to
ls	less than
gr	greater than
le	less than or equal to
ge	greater than or equal to

4.4 Shift Expressions

Syntactic form: E1 lshift E2 | E1 rshift E2

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

Examples: let P(t) = t↑3 rshift 10^8377

x := x lshift BytesizeVCh

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 if 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 ∧ E | E ∨ E |

E ≡ E | E ≢ E

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

$$\wedge \ , \vee \ , \equiv \ , \not\equiv$$

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

Examples:

```
B := not B
if x=0 ∨ y=0 resultis f(t)
x := x ∧ 8 770077 ∨ y ∧ 8 7700
```

Semantics:

The operands of all the logical operators are interpreted as binary 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 n^{th} bit depends only on the n^{th} bits of the operands and can be determined from the following table.

n^{th} bits of operands	Operator			
	$\wedge$	$\vee$	$\equiv$	$\not\equiv$
both ones	1	1	1	0
both zeros	0	0	1	0
otherwise	0	1	0	1

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 evaluatio either E2 or E3 in Rmode depending on whether the value of E1 is tru or false.

true → E2, E3 means E_2

false → E2, E3 means E3

If the value of E1 is neither true nor false the result of the conditional expression is undefined.

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

4.7 Tables

Syntactic form: table <constant> [, <constant>]...

Example: - let T = table '0', '1', '2', '3', '4', '5', '6', '7'
'8', '9', 'A', 'B', 'C', 'D', 'E', 'F'

Semantics:

All the expressions in the list must have Rvalues which can be determined at compile time. They may consist of manifest constants, numbers, character constants or arithmetic expressions with constant operands. 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 allocative of the storage cells and the initialization are performed prior to execution of the program.

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

4.8 Constant Expressions

Syntactic form: <constant> ::= 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 character constant, a manifest constant, or and expression composed of these, brackets

and the operators * / + and -.

Constant expressions are used in

(a) case labels

(b) vector definitions

(c) manifest, static and global declarations

and (d) tables.

5.0 Commands

5.1 Simple Assignment Commands

Syntactic form: E1 := E2

Examples: x := 1

Vii := Uii + Wii

Semantics:

The assignment operation has already been discussed in section 3.4. E1 must be on 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: L1, L2, ... Ln := R1, R2, ... Rn

Example: x, V\i := 1, Uii + Wii

Semantics:

The assignment command is semantically equivalent to a sequence of simple assignment commands. The general form given above is equivalent to the following set of simple assignments:

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

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

```
x, y := y, x
```

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

```
if x = y do  $( V↓3 := 0
               B   := true $)
```

may be written

```
if x = y do  V↓3, B := 0, 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 on different implementations.

5.3 Routine Commands

Syntactic form: E0(E1, E2, ... En)
where E0 is any primary expression and E1 to En are any expressions. The list of expressions may be empty.

Examples:
```
R(x)
Compjump (H2↓x, false, L)
(C↓i)()
```

Semantics:

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

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 declares the name as a static variable with a defined initial value. The scope of the name is the smallest textually enclosing

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

or program.

The Rvalue of a label may be the operand of a goto command, see the next section. For an explanation of the terms static variable and scope see section 6.2.

5.5 Goto Commands

Syntactic form: goto E

where E is any expression.

Examples:

```
goto Next
goto S!i
goto x = 0 -> L, f(x)
```

Semantics:

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

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

5.6 If Commands

Syntactic form:

```
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 2.3.4.

Semantics:

The command if E do C is executed by evaluating E to yield a

truth value, then, if the result is false execution is complete, if the resultis 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
```

Example:

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

Semantics:

The command if 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.

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 N = E1 to E2 do C
where N is a name.

Example: for i = 0 to 122 do V↓i := i

Semantics:

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

```
$( let N, Z = E1, E2
   until  N > Z  do
          $(  C
              N := N + 1  $)  $)
```

where Z is an identifier not used elsewhere in the program. Note that the initial value and end limit expressions E1 and E2 are evaluated only once, and that the control variable N moves in steps of plus one.

5.11 Break Commands

Syntactic form: break

Example:

```
until j = 0 do
      $( if A > CaseK↓j break
         CaseK↓(j+1)  := CaseK↓j
         CaseL↓(j+1)  := CaseL↓j
         j  :=  j - 1  $)
```

Semantics:

Execution of the break command causes ~~the cause~~ a jump to the point just after the smallest textually enclosing loop command.

The loop commands are these with the following key words:

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

5.12 Finish Commands

Syntactic form: finish

Example:

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

Semantics:

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

5.13 Return Commands

Syntactic form: return

Example:

```
let MapB(F, x) be
        $(1  if  x=0 return
             if H1!x = S_COMMA do
                  $( MapB( F, H3  x)
                     F(H2 + x)
                     return    $)
             F(x)  $)1
```

Semantics:

The return command causes the execution of the smallest enclosing routine body to cease and 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 fals
         resultis true $)
```

Semantics:

The execution of resultis E causes the execution of the smallest enclosing result block to cease and yield as a 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>: or
default:

Example:
```
let Trans(x) be
    $(1 if x = 0 return
        switchon H1!x into
        $( default: Report(100); return
           case S_LST: - - -
                       - - -
                       return
           - - -
           case S_SEQ: Trans(H2!x)
                       Trans(H3!x)
                       return      $)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 control passes to the point just after the switchon command.

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 Blocks

Syntactic form: $({ <declaration> ; }* C ; C ... $)

$(C ; C ... $)

Example:

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

Semantics:

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

The names declared by the declarations are local to the block and the storage cells allocated often only remain in existence as

long as execution is dynamically within the block. For a detailed discussion of scopes and extents see sections 6.1 and 6.2.

6.0 Definitions and Declarations

Before a name may be used in a 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 or manifest constant. The scope of a name depends on its declaration as follows:

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

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

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

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

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

6.2 Extent and Space Allocation

The extent of a variable is the time through which it exists and has a storage cell (with its associated Lvalue). Throughout the extent of a variable it remains associated with the same storage cell and so the Lvalue remains constant; however, the contents of the cell (or Rvalue) may be replaced by the execution of an assignment command. In 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.

(b) Dynamic variables

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

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

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

Dynamic variables are declared by

simple variable definitions,
for commands,
vector definitions,
and formal parameters.

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

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

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

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

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

6.3 Let Declarations

Syntactic form: let D { and D }*

where D denotes a definition

Example:

```
let x,y = 0, 1
and f(t) = 2*t - 1
and ItermV = vec 22
```

Semantics:

A let declaration may occur in the declaration sequence of a block and may be used to declare simple variables, vectors, functions and routines. The scope of the variables declared is the textual region of program consisting of the let declaration itself, the succeeding declarations and the commands of the block. The definitions between the 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 definition are described below.

6.3.1 Simple Variable Definitions

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

where N1 to Nn are different names and E1 to En are any expressions.

Example:

```
let x = 1
and y, z = f(t) + 3, H2!A
```

Semantics:

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

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

6.3.2 Vector definitions

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

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

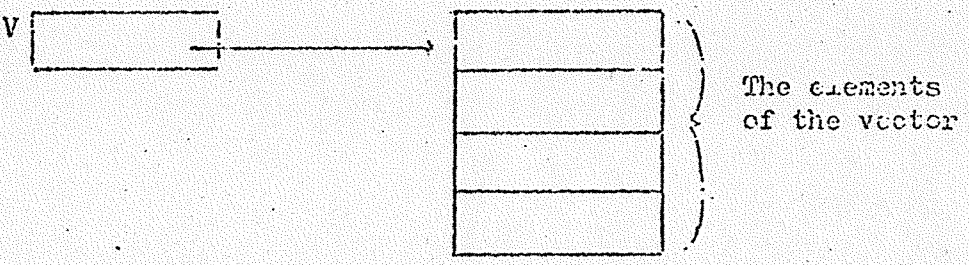
Semantics:

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

start with 0

let V = vec 3

can be shown diagrammatically as follows:



All the storage cells are allocated from the stack and so both the vector variable and the vector elements are dynamic data items. The elements of the vector are not initialised.

6.3.3 Function and Routine Definitions

Syntactic form:

<function definition> ::= N() = E |
N(N1, N2, ... Nn) = E

<routine definition> ::= N() be C |
N(N1, N2, ... Nn) be C

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

Example:

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

parameters of the call.

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

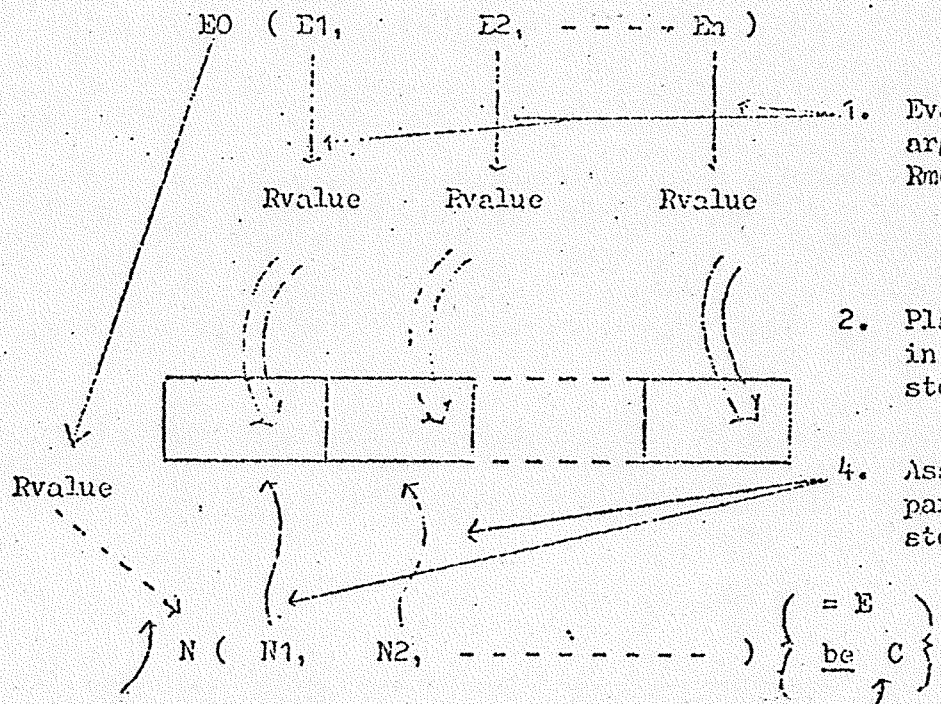


Fig. 9 - The process of calling a function or routine

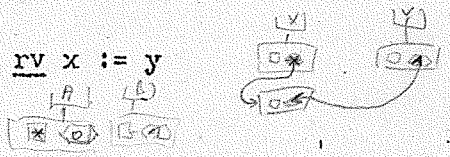
The number of formal parameters need not equal the number of actual parameters and so it is possible to define a variadic routine. Consider:

```
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.

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

All functions and routines may be defined and used recursively.

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

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

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

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
```

6.4 Manifest Declarations

Syntactic form: manifest <decl body>

where

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

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

Although both : and = are allowed in manifest declarations, by convention = should be preferred.

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 connot thereafter be changed. The names so declared are not variables and may not appear in a left hand context.

6.5 Static Declarations

Syntactic Form: <u>static</u> <decl body>
where <decl body> is defined in section 6.4. Although both : and = may be used in <u>static</u> declaration, by convention = should be preferred.

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 initialisation are performed prior to execution of the program.

6.6 Global Declarations

Syntactic form: global <decl body>

where <decl body> is defined in section 6.4.

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

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. Some global variables are initialised prior to execution of the program as a result of the following rule:

If a function or routine definition occurs within the scope of a global variable with the same name, then the function or routine

variable defined shares the storage cell of the global variable and it is initialised to the value of the function or routine prior to execution of the program.

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

```
global  $(  F:7   $)
let     F(g,x) = g(x) + g(-x)
```

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

```
global  $(  F:7;  Write:67   $)
let     g(t) = t + 3
for     i = 0 to 10 do  Write (F(g, i))
```

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

7.0 Example Programs

THE THREE EXAMPLES GIVEN HERE ARE TAKEN FROM THE BCPL COMPILER, AND THUS GIVE A REALISTIC DEMONSTRATION OF THE LANGUAGE IN THE APPLICATION FOR WHICH IT WAS DESIGNED.

THE SECOND EXAMPLE USES A HARDWARE REPRESENTATION WHICH HAS NO SMALL LETTERS; IT IS INCLUDED SO THAT THE READABILITY OF THE REPRESENTATIONS MAY BE COMPARED.

IT IS NOT PRACTICAL TO GIVE AN EXPLANATION OF THE FUNCTIONS AND ROUTINES DEFINED IN THE EXAMPLES AND, INDEED, IT IS NOT NECESSARY TO UNDERSTAND THEM IN DETAIL SINCE THEY ARE ONLY INTENDED TO GIVE A GENERAL IDEA OF WHAT BCPL PROGRAMS LOOK LIKE.

```
// THIS IS THE PART OF THE BCPL COMPILER THAT
// COMPILES SWITCHES INTO ATLAS MACHINE CODE.

GET 'HEAD3'    // INSERT THE HEADING FILE

STATIC $( CASEK=0; CASEL=0 $)

LET CGSWITCH() BE

$(1 LET A = VEC 200
    LET B = VEC 200
    LET N = READN()    // READ THE NUMBER OF CASES
    LET D = READL()    // READ THE DEFAULT LABEL

    CASEK, CASEL := A, B

    FOR I = 1 TO N DO
        $( LET K = READN()    // READ THE CASES AND
           LET L = READL()    // SORT INTO ASCENDING ORDER
           LET J = I-1

           UNTIL J=0 DO    // PLACE THE LATEST CASE IN
                           // ITS PROPER POSITION
              $( IF K > CASEK!J BREAK
                 CASEK!(J+1), CASEL!(J+1) := CASEK!J, CASEL!J
                 J := K - 1    $)

           CASEK!(J+1), CASEL!(J+1) := K, L    $)

    SIMPLIFY()
    STORE(0, SSP-2)
    MOVETOR(AREG, ARG1)    // COMPILE CODE TO PUT THE CONTROL
                           // EXPRESSION INTO THE A REGISTER
    TEST 3*N > 7 + (CASEK!N-CASEK!1)/2    // TEST WHETHER TO
                                          // COMPILE
        THEN DSWITCH(1, N, D)             // A DIRECT SWITCH
          OR BSWITCH(1, N, D)             // OR A TREE SWITCH

    COMPS(F121, 127, 0, 0, D)  // COMPILE A JUMP TO DEFAULT
    INITSTACK(SSP-1)    $)1
```

```
AND BSWITCH(P, Q, D) BE   // COMPILE A TREE SWITCH

$( IF Q-P < 6 DO
       $( FOR I = P TO Q DO   // COMPILE A SEQUENTIAL
                              // SEARCH
            $( COMPS(F172, AREG, 0, CASEK!I, 0)
               COMPS(F224, 127, 0, 0, CASEL!I)   $)
          RETURN $)

$( LET L = NEXTPARAM()
   LET T = (P+Q)/2     // FIND THE CASE WITH AVERAGE VALUE

   COMPS(F172, AREG, 0, CASEK!T, 0)  // COMPILE A
   COMPS(F226, 127, 0, 0, L)         // CONDITION JUMP

   BSWITCH(P, T-1, D) // COMPILE SWITCH FOR THE SMALLER CASES
   COMPS(F121, 127, 0, 0, D)  // COMPILE A JUMP TO DEFAULT

   COMPLAB(L)
   COMPS(F224, 127, 0, 0, CASEL!T)
   BSWITCH(T+1, Q, D) // COMPILE SWITCH FOR THE LARGER CASES
   RETURN $) $)   // ALL DONE

AND DSWITCH(P, Q, D) BE   // COMPILE A DIRECT SWITCH

$( LET L = NEXTPARAM()   // FOR THE TABLE OF LABELS

   COMPS(F172, AREG, 0, CASEK!P, 0)
   COMPS(F227, 127, 0, 0, D)
   COMPS(F170, AREG, 0, CASEK!Q, 0)
   COMPS(F227, 127, 0, 0, D)
   COMPS(F124, AREG, AREG, 0, 0)
   COMPS(F124, AREG, AREG, 0, 0)
   COMPS(F101, 127, AREG, -4*CASEK!P, L)

   COMPDATALAB(L)   // WE NOW COMPILE THE TABLE
   FOR K = CASEK!P TO CASEK!Q DO
         TEST CASEK!P = K
              THEN $( COMPD(0, CASEL!P)
                      P := P + 1  $)
                OR COMPD(0, D)

   CONDHW()  $)  // COMPILE A HALF WORD IF NECESSARY
```

```
GET '/TRNHEAD'                          // THIS IS THE PART OF
                                        // THE BCPL COMPILER THAT
                                        // TRANSLATES CONDITIONAL
LET JUMPCOND(X, B, L) BE                // JUMPS INTO OCODE.

$(JC LET SW = B

     SWITCHON H1!X INTO

     $( CASE S.FALSE: B := NOT B
        CASE S.TRUE:   IF B DO COMPJUMP(L)
                       RETURN

        CASE S.NOT:  JUMPCOND(H2!X, NOT B, L)
                     RETURN

        CASE S.LOGAND: SW := NOT SW
        CASE S.LOGOR:
         TEST SW THEN $( JUMPCOND(H2!X, B, L)
                         JUMPCOND(H3!X, B, L)  $)

                 OR $( LET M = NEXTPARAM()
                       JUMPCOND(H2!X, NOT B, M)
                       JUMPCOND(H3!X, B, L)
                       COMPLAB(M)  $)

         RETURN

        DEFAULT: LOAD(X)
                 OUT2P(B -> S.JT, S.JF, L)
                 SSP := SSP - 1
                 RETURN  $)JC
```

```
GET  'HEAD2'      // THIS IS THE PART OF THE BCPL COMPILER
                  // THAT COMPILES THE ASSIGNMENT COMMAND

LET ASSIGN(X, Y) BE  // X IS THE LEFT SIDE, Y THE RIGHT

$(1 IF X=0 LOGOR Y=0 DO
      $( UNLESS X=0 & Y=0 DO REPORT(110, CURRENTBRANCH)
         RETURN   $)

    SWITCHON H1!X INTO  // SWITCH ON THE LEADING OPERATOR
                        // OF THE LEFT HAND SIDE
    $( CASE COMMA: UNLESS H1!Y = COMMA DO
                      $( REPORT(112, CURRENTBRANCH)
                         RETURN   $)
                   ASSIGN(H2!X, H2!Y)
                   ASSIGN(H3!X, H3!Y)
                   RETURN

       CASE NAME:
          $( LET T = CELLWITHNAME(X)
             LET K, N = DVEC!(T+1), DVEC!(T+2)
             IF T=0 DO REPORT(115, X)
             IF T<DVECP & K=LOCAL DO REPORT(116, X)

             LOAD(Y)
             SSP := SSP - 1

             SWITCHON K INTO

          $( DEFAULT: REPORT(117, X)
                      N := 0

             CASE LOCAL: OUT2(SP, N); RETURN

             CASE GLOBAL: OUT2(SG, N); RETURN

             CASE LABEL: OUT2(SL, N); RETURN   $)  $)

       CASE RV:
       CASE VECAP:
       CASE COND: LOAD(Y)
                  LOADLV(X)
                  OUT1(STIND)
                  SSP := SSP - 2
                  RETURN

       DEFAULT: REPORT(100, CURRENTBRANCH)        $)1
```

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 68" The Computer Journal, Vol. 5, January 1963, p.349.