The BCPL Programming Manual

by

M. Richards

The Computer Laboratory,
University of Cambridge,
Corn Exchange Street,
Cambridge   CB2 3QG

November 1974

TABLE OF CONTENTS

1. INTRODUCTION

BCPL is a programming language designed primarily for non-numerical applications such as compiler-writing and general systems programming. It has been used successfully to implement compilers, interpreters, text editors, game playing programs and operating systems. The BCPL compiler is written in BCPL and runs on several machines including the IBM 370/165 at Cambridge.

Some of the distinguishing features of BCPL are:

> The syntax is rich, allowing a variety of ways to write conditional branches, loops, and subroutine definitions. This allows one to write quite readable programs.
>
> The basic data object is a word (32 bits on the 370) with no particular disposition as to type. A word may be treated as a bit-pattern, a number, a subroutine entry or a label. Neither the compiler nor the run-time system makes any attempt to enforce type restrictions. In this respect BCPL has both the flexibility and pitfalls of machine language.
>
> Manipulation of pointers and vectors is simple and straightforward.
>
> All subroutines may be called recursively.

This manual is not intended as a primer; the constructs of the language are presented with scant motivation and few examples. To use BCPL effectively on the 370 one must have a good understanding of how the machine works and be familiar with its operating system. To the experienced and disciplined programmer it is a powerful and useful language but there are few provisions for the protection of naive users.

The main body of this manual describes the official standard subset of BCPL which will be supported at most BCPL installations. Many of the larger installations provide extensions to the language and a summary of the extensions available on the 370 implementation can be found in Appendix D. Users are strongly recommended to remain within the standard subset unless there are exceptionally strong reasons for not doing so.

Acknowledgments

The overall layout and organisation of this manual is based on a manual written by J.H. Morris of the University of California, Berkeley, which itself was based on a well-written memorandum by R.H. Canaday and D.M. Richie of Bell Telephone Laboratories.

The initial design and implementation of BCPL was done on CTSS at Project MAC in 1967 and since then the language has developed and been transferred to many machines around the world.

The machine code library was implemented for the 370 by J.K.M. Moody and many of the language extensions for the 370 were implemented with the assistance of H.C.M. Meekings. Many of the extensions were first designed and implemented by J.L. Dawson.

The language design was strongly influenced by the author's experience with CPL. This language is described by D.W. Barron et al in "The Main Features of CPL", The Computer Journal, Vol. 6, p.134.

2. LANGUAGE DEFINITION

2.1 Program

At the outermost level, a BCPL program consists of a sequence of declarations. To understand the meaning of a program, it is necessary to understand the meaning of the more basic constructs of the language from which it is made. We, therefore, choose to describe the language from the inside out starting with one of the most basic constructs, namely the 'element'.

2.2 Elements

```
        <element>  ::=  <identifier> | <number> |
                        TRUE | FALSE | ? |
                        <string constant> | <character constant>
```

An <identifier> consists of a sequence of letters, digits and underlines, the first character of which must be a letter.

A <number> is either an integer consisting of a sequence of decimal digits, or an octal constant consisting of the symbol '#' followed by octal digits, or a hexadecimal constant consisting of the character pair #X followed by hexadecimal digits. The reserved words TRUE and FALSE denote -1 and 0 respectively (on a 2's complement machine) and are used to represent the two truth values. The symbol '?' may be used anywhere in an expression when no specific value is required, as in:

```
                LET OP, A = ?, ?
```

A <string constant> consists of up to 255 characters enclosed in string quotes ("). The internal character set is EBCDIC (on the 370). The character " may be represented only by the pair *" and the character * can only be represented by the pair **. Other characters may be represented as follows:

*N	is newline
*C	is carriage return
*T	is horizontal tab
*S	is space
*B	is backspace
*P	is newpage

In a string the sequence

```
        * <newline> [ <space> | <tab> ] *
```

is skipped. Thus, the string

```
                "THIS STRING *
                *CONTAINS NEWLINES*
                * AND SPACES"
```

is equivalent to

"THIS STRING CONTAINS NEWLINES AND SPACES"

The machine representation of a string is the address of the region of store where the length and characters of the string are packed. The packing and unpacking of strings may be done using the machine dependent library routines PACKSTRING and UNPACKSTRING, and individual characters in a string can be accessed and updated using the library routines GETBYTE and PUTBYTE, see section 2.8.2.

A <character constant> consists of a single character enclosed in character quotes ('). The character ' can be represented in a character constant only by the pair *'. Other escape conventions are the same as for a string constant. A character constant is right justified in a word. Thus 'A' = 193 (on the 370).

2.3 Expressions

Because an identifier has no type information associated with it, the type of an element (and hence an expression) is assumed to match the type required by its context.

All expressions are listed below. E1, E2 and E3 represent arbitrary expressions except as noted in the descriptions which follow the list, and K0, K1 and K2 represent constant expressions (whose values can be determined at compile time, see section 2.3.8).

Primary	element	
	(E1)	
function call	E1()	
	E1(E2,E3,...)	
addressing	E1!E2	subscripting
	@E1	address generaton
	!E1	indirection
arithmetic	E1 * E2	
	E1 / E2	
	E1 REM E2	integer remainder
	E1 + E2	
	+ E1	
	E1 - E2	
	- E1	
relational	E1 = E2	
	E1 ~= E2	not equal
	E1 < E2	
	E1 <= E2	
	E1 > E2	
	E1 >= E2	
shift	E1 << E2	left shift by E2(>=0) bits
	E1 >> E2	right shift by E2(>=0) bits

logical	~ E1	not (complement)
	E1 & E2	and
	E1 | E2	inclusive or
	E1 EQV E2	bitwise equivalence
	E1 NEQV E2	bitwise not-equivalence (exclusive or)
conditional	E1 -> E2, E3	
table	TABLE K0,K1,K2,...	
valof	VALOF command	

The relative binding power of the operators is as follows:

(highest, most binding)	function call
	! (subscripting)
	@ !
	* / REM
	+ -
	relationals
	shifts (see section 2.3.4)
	~
	&
	|
	EQV NEQV
	->
	TABLE
(lowest, least binding)	VALOF

Operators of equal binding power associate to the left. For example,

```
X + Y - Z
```

is equivalent to

```
(X + Y) - Z
```

In order that the rule allowing the omission of most semicolons should work properly, a diadic operator may not be the first symbol on a line.

The function call will be described with the function definition in section 2.6.6, and the valof expression will be described with the resultis command in section 2.5.5.

2.3.1 Addressing operators

A powerful pair of operators in BCPL are those which allow one to generate and use addresses. An address may be manipulated using integer arithmetic and is indistinguishable from an integer until it is used in a context which requires an address. If the value of a variable X is the address of a word in storage, then X+1 is the address of the next word.

If V is a variable, then associated with V is a single word of memory, which is called a cell. The contents of the cell is called the value of V and the address of the cell is called the address of V.

An address may be used by applying the indirection operator (!). The expression

```
                !E1
```

has, as value, the contents of the cell whose address is the value of the expression E1. Only the low-order 22 bits of E1 are used (on the 370).

An address may be generated by means of the operator @. The expression

```
                 @E1
```

is only valid if E1 is one of the following:

(1) An identifier (not declared by a manifest declaration), in which case @V is the address of V.

(2) A subscripted expression, in which case the value of @E1!E2 is E1+E2.

(3) An indirection expression, in which case the value of @!E1 is E1.

Case (1) is self-explanatory. Case (2) is a consequence of the way vectors are defined in BCPL. A vector of size n is a set of n + 1 contiguous words in memory, numbered 0, 1, 2, ..., n. The vector is identified by the address of word 0. If V is an identifier associated with a vector, then the value of V is the address of word 0 of the vector.

```
                  -------              -------
              V  |   *---|-----------> |       |  0
                  -------              |-------|
                                       |       |  1
                                       |-------|
                                       |       | ...
                                       |-------|
                                       |       |  n
                                        -------
```

The value of the expression

```
                V!E1
```

is the contents of cell number E1 of vector V, as one would expect. The address of this cell is the value of

```
                V + E1
```

hence

```
                @(V!E1)  =  V + E1
```

This relation is true whether or not the expression

```
                V!E1
```

happens to be valid, and whether or not V is an identifier.

Case (3) is a consequence of the fact that the operators @ and ! are inverse.

The interpretation of

```
!E1
```

depends on context as follows:

(1) If it appears as the left-hand side of an assignment statement, e.g.

```
!E1 := E2
```

E1 is evaluated to produce the address of a cell and E2 is stored in it

(2) @(!E1) = E1 as noted above.

(3) In any other context E1 is evaluated and the contents of that value, treated as an address, is taken.

Thus, ! forces one more contents-taking than is normally demanded by the context.

As a summarising example, consider the four variables A, B, C and D with initial values @C, @D, 5 and 7, respectively. Then, after the assignment

```
A := B
```

their values will be @D, @D, 5, 7.

If, instead, the assignment

```
A := !B
```

had been executed, then the values would have been 7, @D, 5, 7.

And if the assignment

```
!A := B
```

had been executed, then the values would have been @C, @D, @D, 7.

Note that

```
@A := B
```

is not meaningful, since it would call for changing the address associated with A, and that association is permanent.

2.3.2 Arithmetic operators

The arithmetic operators *, /, REM, + and - act on 32 bit quantities (on the 370) interpreted as integers.

The operators * and / denote integer multiplication and division. The operator REM yields the integer 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 operators + and - may be used in either a monadic or diadic context and perform the appropriate integer arithmetic operations.

The treatment of arithmetic overflow is undefined.

2.3.3 Relations

A relational operator compares the integer values of its two operands and yields a truth-value (TRUE or FALSE) as result. The operators are as follows:

=	equal
~=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

The operators = and ~= make bitwise comparisons of their operands and so may be used to determine the equality of values regardless of the kind of objects they represent.

An extended relational expression such as

```
        'A' <= CH <= 'Z'
```

is equivalent to

```
        'A' <= CH  &  CH <= 'Z'
```

2.3.4 Shift operators

In the expression E1 << E2 (E1 >> E2), E2 must evaluate to yield a non-negative integer. The value is E1, taken as a bit-pattern, shifted left (or right) by E2 places. Vacated positions are filled with 0 bits.

Syntactically, the shift operators have lower precedence on the left than relational operators but greater precedence on the right.

Thus, for example,

```
        A << 10 = 14
```

is equivalent to

```
        (A<<10) = 14
```

but

```
        14 = A << 10
```

is equivalent to

```
        (14=A) << 10
```

2.3.5 Logical operators

The effect of a logical operator depends on context. There are two logical contexts: 'truth-value' and 'bit'. The truth-value context exists whenever the result of the expression will be interpreted immediately as true or false. In this case each subexpression is interpreted, from left to right, in truth-value context until the truth or falsehood of the expression is determined. Then evaluation stops. Thus, in a truth-value context, the evaluation of

```
        E1 | E2 & ~E3
```

is as follows.

E1 is evaluated, and if true the whole expression is true,

otherwise E2 is evaluated, and if false the whole expression is false,
otherwise E3 is evaluated, and if false the whole expression is true,
otherwise the whole expression is false.

In a 'bit' context, the operator ~ causes bit-by-bit complementation of its operand.   The other operators combine their operands bit-by-bit according to the following table:

Operands		Operator			
		&	|	NEQV	EQV
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1

2.3.6   Conditional operator

The expression

```
        E1 -> E2, E3
```

is evaluated by evaluating E1 in truth-value context.   If it yields true, then the expression has value E2, otherwise E3.   E2 and E3 are never both evaluated.

2.3.7   Table

The value of the table expression

```
        TABLE  K0, K1, K2, ...
```

is the address of a static vector of cells initialised to the values of K0, K1, K2, ... which must be constant expressions.

2.3.8   Constant expression

A constant expression is any expression involving only numbers, character constants, names declared by manifest declaration, TRUE, FALSE and the operators *, /, REM, + and - .

2.4   Section brackets

Blocks, compound commands and some other syntactic constructions use the symbols $(and $) which are called opening and closing section brackets.

A section bracket may be tagged with a sequence of letters, digits and underlines (the same characters as are used in identifiers). A section bracket immediately followed by a space or newline is, in effect, tagged with null.

An opening section bracket can be matched only by an identically tagged closing bracket. When the compiler finds a closing section bracket with a non-null tag, if the nearest opening bracket (smallest currently open section) does not match, that section is closed and the process repeats until a matching opening section bracket is found.

Thus it is impossible to write sections which are overlapping (not nested).

2.5 Commands

The complete set of commands is shown here, with E, E1, E2 and K denoting expressions, C, C1 and C2 denoting commands, and D1 and D2 denoting declarations.

routine call	E(E1, E2, ...)
	E()

assignment	<left hand side list> := <expression list>
conditional	IF E THEN C UNLESS E THEN C TEST E THEN C1 OR C2
repetitive	WHILE E DO C UNTIL E DO C C REPEAT C REPEATWHILE E C REPEATUNTIL E FOR N = E1 TO E2 BY K DO C FOR N = E1 TO E2 DO C
resultis	RESULTIS E
switchon	SWITCHON E INTO <compound command>
transfer	GOTO E FINISH RETURN BREAK LOOP ENDCASE
compound	$(C1; C2; ... $)
block	$(D1; D2; ...; C1; C2; ... $)

Discussion of the routine call is deferred until section 2.6.6 where function and routine declarations are described.

2.5.1 Assignment

The command

```
                E1 := E2
```
causes the value of E2 to be stored into the cell specified by E1. E1 must have one of the following forms:

(1)	The identifier of a variable	<identifier>
(2)	A subscripted expression	E3!E4
(3)	An indirection expression	!E3

In case (1) the cell belonging to the identifier is updated. Cases (2) and (3) have been described in section 2.3.1.

A list of assignments may be written thus:
```
                E1, E2, ..., En := F1, F2, ..., Fn
```
where Ei and Fi are expressions. This is equivalent to
```
                E1 := F1
                E2 := F2
                   ...
                En := Fn
```

2.5.2 Conditional commands

```
                IF  E  THEN  C1
                UNLESS  E  THEN  C2
                TEST  E  THEN  C1  OR  C2
```

Expression E is evaluated in truth-value context. Command C1 is executed if E is true, otherwise the command C2 is executed.

2.5.3 For command

```
                FOR  N = E1  TO  E2  BY  K  DO  C
```

N must be an identifier and K must be a constant expression. This command will be described by showing an equivalent block.

```
                $( LET  N, t = E1, E2
                   UNTIL N > t DO $(  C
                                      N := N + K  $)
                $)
```

If the value of K is negative the relation N > t is replaced by N < t. The declaration
```
                LET  N, t = E1, E2
```
declares two new cells with identifiers N and t; t being a new identifier that does not occur in C. Note that the control variable N is not available outside the scope of the command.

The command
```
                FOR  N = E1  TO  E2  DO  C
```
is equivalent to
```
                FOR  N = E1  TO  E2  BY  1  DO  C
```

2.5.4 Other repetitive commands

```
WHILE  E  DO  C
UNTIL  E  DO  C
C  REPEAT
C  REPEATWHILE  E
C  REPEATUNTIL  E
```

Command C is executed repeatedly until condition E becomes true or false as implied by the command. If the condition precedes the command (WHILE, UNTIL) the test will be made before each execution of C. If it follows the command (REPEATWHILE, REPEATUNTIL), the test will be made after each execution of C, and so C is executed at least once. In the case of

```
C  REPEAT
```

there is no condition and termination must be by a transfer or resultis command in C. C will usually be a compound command or block.

Within REPEAT, REPEATWHILE and REPEATUNTIL C is taken as short as possible. Thus, for example

```
IF E THEN C REPEAT
```

is the same as

```
IF E THEN $( C REPEAT $)
```

and

```
E := VALOF C REPEAT
```

is the same as

```
E := VALOF $( C REPEAT $)
```

2.5.5 Resultis command and valof expression

The expression

```
VALOF  C
```

where C is a command (usually a compound command or block) is called a valof expression. It is evaluated by executing the commands (and declarations) in C until a resultis command

```
RESULTIS  E
```

is encountered. The expression E is evaluated, its value becomes the value of the valof expression and execution of the commands within C ceases.

A valof expression must contain one or more resultis commands and one must be executed.

In the case of nested valof expressions, the resultis command terminates only the innermost valof expression containing it.

2.5.6 Switchon command

```
SWITCHON  E  INTO  <compound command>
```

where the compound command contains labels of the form

```
    CASE  <constant expression>:
or  DEFAULT:
```

The expression E is first evaluated and, 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, and if there is not, execution is resumed just after the end of the switchon command.

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

2.5.7 Transfer of control

```
GOTO  E
FINISH
RETURN
BREAK
LOOP
ENDCASE
```

The command GOTO E interprets the value of E as an address, and transfers control to that address, see section 2.6.7. The command FINISH causes an implementation dependent termination of the entire program. RETURN causes control to return to the caller of a routine. BREAK causes execution to be resumed at the point just after the smallest textually enclosing repetitive command. The repetitive commands are those with the following key words:

UNTIL, WHILE, REPEAT, REPEATWHILE, REPEATUNTIL and FOR

LOOP causes execution to be resumed at the point just before the end of the body of a repetitive command. For a for command it is the point where the control variable is incremented, and for the other repetitive commands it is where the condition (if any) is tested. ENDCASE causes execution to be resumed at the point just after the smallest enclosing switchon command.

2.5.8 Compound command

A compound command is a sequence of commands enclosed in section brackets.

```
$(  C1; C2; ... $)
```

The commands C1, C2, ... are executed in sequence.

2.5.9 Block

A block is a sequence of declarations followed by a sequence of commands enclosed together in section brackets.

```
$(  D1; D2; ... ; C1; C2; ... $)
```

The declarations D1, D2, ... and the commands C1, C2, ... are executed in sequence. The scope of an identifier (i.e. the region of program where the identifier is known) declared in a declaration is the declaration itself (to allow recursive definition), the subsequent declarations and the commands of the block. Notice that the scope does not include earlier declarations or extend outside the block.

2.6 Declarations

Every identifier used in a program must be declared explicitly. There are 10 distinct declarations in BCPL:

global, manifest, static, dynamic, vector, function, routine, formal parameter, label and for-loop control variable.

The declaration of formal parameters is covered in sections 2.6.6 and 2.6.7, and the for-loop is described in section 2.5.3.

The scope of identifiers declared at the head of a block is described in the previous section.

2.6.1 Global

A BCPL program need not be compiled in one piece. The sole means of communication between separately compiled segments of program is the global vector. The declaration

```
        GLOBAL $( Name : constant-expression $)
```

associates the identifier Name with the specified location in the global vector. Thus Name identifies a static cell which may be accessed by Name or by any other identifier associated with the same global vector location.

Global declarations may be combined.

```
        GLOBAL $( N1:K1; N2:K2; ...; Nn:Kn  $)
```

is equivalent to

```
        GLOBAL  $( N1:K1  $)
        GLOBAL  $( N2:K2  $)
        ...
        GLOBAL  $( Nn:Kn  $)
```

2.6.2 Manifest

An identifier may be associated with a constant by the declaration

```
        MANIFEST  $(  Name = constant-expression  $)
```

An identifier declared by a manifest declaration may only be used in contexts where a constant would be allowable. It may not, for instance, appear on the left hand side of an assignment. Like global declarations, manifest declarations may be combined.

```
        MANIFEST  $(  N1=K1; N2=K2; ...; Nn=Kn  $)
```

is equivalent to

```
        MANIFEST  $(  N1=K1  $)
        MANIFEST  $(  N2=K2  $)
        ...
        MANIFEST  $(  Nn=Kn  $)
```

2.6.3 Static

A variable may be declared and given an initial value by the declaration

```
                STATIC  $(  Name = constant-expression  $)
```

The variable that is declared is static, that is it has a cell permanently allocated to it throughout the execution of the program (even when control is not dynamically within the scope of the declaration). Like global declarations, static declarations may be combined.

```
                STATIC  $(  N1=K1; N2=K2; ...; Nn=Kn  $)
```

is equivalent to

```
                STATIC  $(  N1=K1  $)
                STATIC  $(  N2=K2  $)
                ...
                STATIC  $(  Nn=Kn  $)
```

2.6.4 Dynamic

The declaration

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

creates dynamic cells and associates with them the identifiers N1, N2, ..., Nn. These cells are initialised to the values of E1, E2, ..., En. The space reserved for these cells is released when the block in which the declaration appears is left.

2.6.5 Vector

The declaration

```
                LET  N = VEC  K
```

where K is a constant expression, creates a dynamic vector by reserving K + 1 cells of contiguous storage in the stack, plus one cell which is associated with the identifier N. Execution of the declaration causes the value of N to become the address of the K + 1 cells. The storage allocated is released when the block is left.

2.6.6 Function and routine

The declaration

```
                LET  N(P1, P2, ..., Pm) = E
```

declares a function named N with m parameters. The parentheses are required even if m = 0. A parameter name has the same syntax as an identifier, and its scope is the expression E. A routine declaration is similar to a function declaration except that its body is a command.

```
                LET N(P1, P2, ..., Pm) BE C
```

If the declaration is within the scope of a global declaration for N, then the global cell will be initialised to the entry address of the function (or routine) before execution of the program. Thus the function may be accessed from anywhere. Otherwise, a static cell is created, is associated with the identifier N, and is initialised to the entry address.

The function or routine is invoked by the call

```
                E0(E1, E2, ..., Em)
```

where expression E0 evaluates to the entry address. In particular, within the scope of the identifier N, the function or routine may be invoked by the call

```
N(E1, E2, ..., Em)
```

provided the value of N has not been changed during the execution of the program.

Each value passed as a parameter is copied into a newly created cell which is then associated with the corresponding parameter name. The cells are consecutive in store and so the argument list behaves like an initialised dynamic vector. The space allocated for the argument list is released when evaluation of the call is complete. Notice that arguments are always passed by value; however, the value passed may, of course, be an address.

A function call is a call in the context of an expression. If a function is being called, the result is the value of E, and if a routine is being called, the result is undefined. A routine call is a call in the context of a command and may be used to call either a function or a routine. A routine call has no result.

No dynamic (or vector or formal) variable that is declared outside the function may be referred to from within E.

2.6.7 Label

A label may be declared by

```
Name:
```

A label declaration may precede any command or label declaration, but may not precede any other form of declaration. Exactly as in the case of a function or routine, a label declaration creates a static cell if it is not within the scope of a global declaration of the same identifier. The local or global cell is initialised before execution with the address of the point in the program labelled, so that the command

```
GOTO Name
```

has the expected effect.

The scope of a label depends on its context. It is the smallest of the following regions of program:

(1) the command sequence of the smallest textually enclosing block,
(2) the body of the smallest textually enclosing valof expression or routine,
(3) the body of the smallest enclosing for command.

Labels may be assigned to variables and passed as parameters. It is, in general, not useful for them to be declared global, but they can be assigned to global variables.

Using a goto command to transfer to a label which is outside the current function or routine will produce undefined (chaotic) results. Such transfers can only be performed using the library functions LEVEL and LONGJUMP which are described in section 2.8.2.

2.6.8 Simultaneous declaration

Any declaration of the form

```
        LET ...
```

may be followed by one or more declarations of the form

```
        AND ...
```

where any construct which may follow LET may follow AND. As far as scope is concerned, such a collection of declarations is treated like a single declaration. This makes is possible, for example, for two routines to know each other without recourse to the global vector.

2.7 Miscellaneous features

2.7.1 GET

It is possible to include a file in the source text of a program using a get directive of the form:

```
        GET  "string"
```

On the 370, text of the get directive is replaced by the text of the file whose DDNAME is string. A get directive should appear on a line by itself.

2.7.2 Comments and spaces

The character pair // introduces a comment. All characters from (and including) // up to but not including the character 'newline' will be ignored by the compiler. The character pair /* introduce a comment which is terminated by the pair */, such a comment may extend over several lines.

Blank lines are also ignored.

Space and tab characters may be inserted freely except inside a basic symbol, but Space or tab characters are required to separate identifiers or system words from adjoining identifiers or system words.

2.7.3 Optional symbols and synonyms

The reserved words DO and THEN are synonyms in BCPL. Most implementations of BCPL also allow other synonyms and a list of the synonyms for the 370 implementation can be found in Appendix A.

In order to make BCPL programs easier to read and to write, the compiler allows the syntax rules to be relaxed in certain cases. The word DO (or THEN) may be omitted whenever it is immediately followed by the keyword of a command (e.g. RESULTIS). Any semicolon occurring as the last symbol of a line may be omitted. As an example, the following two programs are equivalent:

```
IF A = 0  DO GOTO X;        |        IF  A = 0  GOTO X
A := A - 1;                 |        A := A - 1
```

2.8 The run-time library

This section summarises the library functions and routines that are available on the 370 implementation of BCPL.

2.8.1 Input-Output routines

The input/output facilities of BCPL are quite primitive and simple, and are always invoked by means of function or routine calls.

FINDINPUT(ddname) is a function taking a string for the ddname of a data-set as argument and returning a stream-pointer to be used by the input routines. If the data-set is not already open for reading it is opened. If the data-set does not exist, the result is zero.

SELECTINPUT(stream) is a routine which selects the specified input stream for future reading.

RDCH() is a function whose result is the next character from the currently selected input stream. If the stream is exhausted, it yields ENDSTREAMCH(=-1).

UNRDCH() is a routine that will cause the next call of RDCH to yield the same character that it returned on its last call for the currently selected input stream.

REWIND() repositions the currently selected input stream to point to the first record.

ENDREAD() closes the currently selected input stream.

FINDOUTPUT(ddname) is a function taking a string for the ddname of a data-set as argument and returning a stream-pointer to be used by the output routines. If the data-set is not already opened for writing it is opened. If the data-set does not exist, the result is zero.

SELECTOUTPUT(stream) is a routine which selects the specified output stream for future writing.

WRCH(CH) will write the character CH to the currently selected output stream.

ENDWRITE() closes the currently selected output stream.

ENDTOINPUT() closes the currently selected output stream and reopens it for reading.

INPUT() is a function that will return with the currently selected input stream.

OUTPUT() is a function that will return with the currently selected output stream.

TRIMINPUT(SW) sets the control that specifies the treatment of trailing blanks in records read from the current input stream. If SW is true trailing blanks will be skipped, if SW is false they will not.

READREC(V) is a function that will read a record from the current input stream into the vector V packing four characters per word. The result is the number of characters read, or -1 if the stream is exhausted.

WRITEREC(V,N) will write N characters from the vector V to the current output stream followed by a newline. The characters in V are packed four per word.

WRITESEG(V,N) will write N characters from the vector V to the current output stream. The characters in V are packed four per word.

2.8.2 Other useful subroutines

PACKSTRING(V,S) is a function which packs the characters V!1 to V!N into S, where N = V!0 & 255. The result is the subscript of the highest element of S used (i.e. N/4 on the 370).

UNPACKSTRING(S,V) is a routine to unpack characters from the string S into V!1 to V!N when N is the length of the string, and set V!0 = N.

GETBYTE(S, I) is a function whose result is the Ith character of the string S. By convention the zeroth character of a string is its length.

PUTBYTE(S, I, CH) is a routine which will update the Ith character of the string S with CH.

WRITES(S) writes the string S to the current output stream.

NEWLINE() writes a newline to the current output stream.

WRITED(N,D) writes the integer N to the current output stream right justified in a field of width D places. If D is too small the number is written correctly using as many characters as necessary.

WRITEN(N) is equivalent to WRITED(N,0).

WRITEOCT(N,D) writes the D least significant octal digits of N to the current output stream.

WRITEHEX(N,D) writes the D least significant hexadecimal digits of N to the current output stream.

WRITEF(FORMAT,A,B, ...) is a routine to output A,B, ... to the current output stream according to FORMAT. The FORMAT string is copied to the stream until the end is reached or the warning character '%' is encountered. The character following the '%' defines the format of the next value to be printed as follows:

```
%%      print '%'
%S      print as a string
```

%C	print as a character
%N	print as a integer (minimum width)
%In	print as a integer width n
%On	print as an octal number width n
%Xn	print as a hexadecimal number width n

In the last three cases the width n is represented by a single hexadecimal digit. The routine takes the format and a maximum of 11 additional arguments.

MAPSTORE() prints a map of the program area including function and routine names, and the values of all global variables used.

BACKTRACE() prints a summary of the dynamic stack giving the names of all functions and routines currently active and the values of the first few local variables of each.

ABORT(CODE,ADDR) is called automatically by the system after most faults. It calls BACKTRACE and MAPSTORE in order to provide the user with some postmortem information.

STOP(N) will terminate the job step, returning a completion code N.

LEVEL() is a function whose result is the current value of the run-time stack pointer for use with LONGJUMP. The stack pointer changes only when a function or routine is entered or left.

LONGJUMP(P,L) will cause a non-local jump to the label L at the activation level given by the stack pointer P.

TIME() is a function whose result is the computation time used in units of 26 micro-seconds.

APTOVEC(F,N) is a function which will apply F to two arguments V and N where V is a vector of size N. APTOVEC could (illegally) be defined in BCPL as follows:

```
LET APTOVEC(F,N) = VALOF
       $(  LET V = VEC N
           RESULTIS F(V,N) $)
```

2.8.3 Library variables

STACKBASE points to the base of the runtime stack.

STACKEND points to the end of the runtime stack.

PARMS holds a string representing the PARM field for the current job step.

TERMINATOR holds the character following the last digit of the most recent number read in by READN.

ENDSTREAMCH is a manifest constant (=-1) which is produced by RDCH when the input stream is exhausted.

3. USING BCPL ON THE 370

Files relating to the BCPL system are catalogued in the file directory BCPLIB. These include the compiler itself BCPLIB.SYS2, the library modules which are held in a partitioned data set BCPLIB.LIB and the standard header file BCPLIB.LIBHDR which contains global declarations for all the library routines. There is, also, an information document BCPLIB.INFO which contains recent news about the BCPL system.

There are four catalogued procedures BCPLCLG, BCPLCL, BCPLC and BCPLLG to simplify the use of the compiler.

3.1 Compilation

The BCPL compiler is usually invoked by means of one of the catalogued procedures. The following complete job illustrates the use of BCPLCLG. It will compile and run the routine START.

```
JOB ...
LIMSTORE 150K
// EXEC BCPLCLG
//BCPL.SYSIN DD DATA

GET "LIBHDR"

LET START(PARM) BE $(1
...
...
$)1
/*
```

The compiler runs as one job step and currently requires a region size of 150K, hence the need for LIMSTORE 150K. The example given above will cause the program to be compiled, link-edited and run with the standard library. The catalogued procedure BCPLCLG is as follows:

```
//BCPLCLG PROC TRKS='5,5',PRINTC='SYSOUT=C',REGC=150K,
//             MINC=,SECC=20,CONDC=,PARMC='L12000/K',
//             LIBC='BCPLIB.SYS2',LIBHDR='BCPLIB.LIBHDR',
//             SYSLIBL='BCPLIB.LIB',IOSPACE=54K,
//             PRINTL='SYSOUT=C',REGL=98K,MINL=,SECL=20,
//             CONDL='4,LT,BCPL',
//             LISTL=LIST,MAPL=NOMAP,XREFL=NOXREF,LETL=LET,
//             ATTL=,CAL=CALL,DCBL=DCBS,BLKL=4096,
//             SIZEL='(999999,8K)',NAMEL='&GOSET(BCPLGO)',
//             DISPL='(NEW,PASS),SPACE=(TRK,(8,3,1),RLSE)',
//             PARMG=,PRINTG='SYSOUT=A',REGG=120K,MING=1439,
//             SECG=,CONDG='(4,LT,BCPL),(4,LT,LKED)'
//*
//BCPL    EXEC PGM=BCPL,REGION=&REGC,PARM='$&IOSPACE.$&PARMC',
//             TIME=(&MINC,&SECC),COND=(&CONDC)
//STEPLIB DD   DISP=SHR,DSN=&LIBC
//SYSPRINT DD  &PRINTC
//LIBHDR   DD  DISP=SHR,DSN=&LIBHDR
//SYSGO    DD  DSN=&LOADSET,DISP=(MOD,PASS),UNIT=DISC,
//             SPACE=(TRK,(&TRKS),RLSE),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=2480)
//LKED    EXEC PGM=IEWL,REGION=&REGL,
// PARM='SIZE=&SIZEL,&LISTL,&MAPL,&XREFL,&LETL,&ATTL,&CAL,&DCBL',
//             COND=(&CONDL),TIME=(&MINL,&SECL)
//SYSLIB   DD  DSN=&SYSLIBL,DISP=SHR
//SYSLMOD  DD  DSN=&NAMEL,DISP=&DISPL,DCB=BLKSIZE=&BLKL,UNIT=DISC
//SYSPRINT DD  &PRINTL
//SYSUT1   DD  UNIT=DISC,SPACE=(TRK,(&TRKS)),SEP=SYSLMOD,
//             DSN=&SYSUT1
//SYSLIN   DD  DSN=&LOADSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//GO      EXEC PGM=*.LKED.SYSLMOD,COND=(&CONDG),REGION=&REGG,
//             PARM='&PARMG',TIME=(&MING,&SECG)
//SYSPRINT DD  &PRINTG
```

3.1.1 Library declarations

The directive

```
        GET "LIBHDR"
```

will insert the standard library declarations from the data-set whose DDNAME is LIBHDR. The default setting of this data-set is BCPLIB.LIBHDR. The items declared in this file are shown below. By convention library variables are given global numbers in the range 1 to 99 and so users should avoid allocating globals in this region for their own purposes.

```
GLOBAL
$(  START:1              // The main routine
    ABORT:3              // Calls BACKTRACE and MAPSTORE
    BACKTRACE:4          // Summarise the run-time stack
    SELECTINPUT:11       // Select input stream
    SELECTOUTPUT:12      // Select output stream
    RDCH:13              // Read a character
    WRCH:14              // Write a character
    UNRDCH:15            // 'Unread' a character
    INPUT:16             // Find current input
    OUTPUT:17            // Find current output
    TRIMINPUT:20         // Set trailing blank control
    READREC:23           // Read a record
    WRITEREC:24          // Write record
    WRITESEG:25          // Write part-record
    TIME:28              // Find computation time
    STOP:30              // Terminate job step
    LEVEL:31             // Find activation pointer
    LONGJUMP:32          // Make non-local jump
    REWIND:35            // Rewind input stream
    APTOVEC:40           // Apply function to dynamic vector
    FINDOUTPUT:41        // Find specified output stream
    FINDINPUT:42         // Find specified input stream
    ENDREAD:46           // Close input stream
    ENDWRITE:47          // Close output stream
    ENDTOINPUT;51        // Close and reposition output stream
    STACKBASE:54         // Base of stack pointer
    STACKEND:55          // End of stack pointer
    WRITES:60            // Write a string
    WRITEN:62            // Write a number (minimum width)
    NEWLINE:63           // Write a newline
    PACKSTRING:66        // Pack characters
    UNPACKSTRING:67      // Unpack characters
    WRITED:68            // Write a number
    READN:70             // Read a number
    TERMINATOR:71        // Terminator character of READN
    WRITEHEX:75          // Write a hexadecimal number
    WRITEF:76            // Write with format
    WRITEOCT:77          // Write an octal number
    MAPSTORE:78          // Map the store
    GETBYTE:85           // Obtain a character from a string
    PUTBYTE:86           // Update a character in a string
$)

MANIFEST
$(  ENDSTREAMCH= -1      // End of stream character
$)
```

3.1.2 Diagnostics

The BCPL compiler has three passes: parse, translate and code-generate. There are correspondingly three kinds of error diagnostic.

A parse diagnostic occurs when a relatively simple syntactic error is detected during the first pass of compilation. The message includes a portion of the source program to give the context and a brief description of the probable error. The compiler usually skips to the end of the line before continuing the parse. Later error messages should be viewed with suspicion since the automatic recovery is often not very successful.

Translation phase diagnostics occur in the second pass of compilation and report errors such as the use of an undeclared identifier. Each error is briefly described and a representation of the relevant portion of the parse tree is printed.

Code-generation diagnostics are rare and usually result from table overflows or compiler errors.

3.1.3 Compilation options

The compilation of a program can be controlled by various compilation options passed to the compiler by the PARM field of the EXEC card that invoked the compiler. The options for the code-generator are separated from those for the first phase of the compiler by a slash. Most options are specified by single letters and some are primarily debugging aids for the implementer of the compiler.

The first phase options are as follows:

Ln	Set the size of work-space area used during compilation. The best value of n is usually between 6000 and 12000.
Dn	Set the size of the work-space area used to hold declared names. The default setting is D1800 which allows for 600 names to be declared at any one time.
N	Disable the GET directive.
S	List the source program with line numbers.
T	Print the parse tree of the source program.
O	Print the intermediate object code form of the program.

The code generator options are as follows:

K	Compile instructions with each function and routine to count the number of times they are executed. The counts can be printed using MAPSTORE.

P   Compile instructions after labels and conditional jumps to accumulate execution counts.   These counts can be printed using MAPSTORE and allow one to make a detailed analysis of the execution of the program.

G   Execute the compiled program as a subroutine of the compiler and thus save the overhead of the 'link-edit' and 'go' steps. This option is particularly useful when running BCPL under the BATCH monitor, but one is limited to the region size and library used by the compiler.

H   Construct a symbol table that gives the identifiers of local variables. The extended post-mortem will then print the values of such variables together with source names (specify NEEDS "PM" to obtain the special post-mortem).

L   Output an assembly listing of the compiled program.

N   Do not generate an object module for the program.

The default setting of the PARM field when using BCPLCLG is 'L12000/K'.

3.2 Execution

3.2.1 Loading

Each compiled segment of a BCPL program has an external reference to BCPLMAIN which is the entry point of the standard machine code library and this, in turn, has a reference to BLIB which is the portion of the standard library written in BCPL.   Thus, when the compiled segments are link-edited together, the library modules are automatically incorporated.   The standard library modules are held in BCPLIB.LIB.

When the complete program is executed, the machine code library initialises the run-time system and obtains space for the global vector and stack.   The globals are initialised to their appropriate values and then control is passed to the BCPL program by calling the routine START (global 1) which must have been defined by the programmer.   START is passed a string representing the PARM field of the job control card that caused the program to be executed.

The size of the global vector is the smallest multiple of 100 words large enough to accommodate the highest global number actually used in any segment of the loaded program.   The size of the run-time stack depends on the space available in the region in which the program is run.   Some space is returned for input/output buffers and system use.   The limits of the stack are held in STACKBASE and STACKEND.

If the PARM string starts with a sequence of the form:

```
$kKgGiItTD$
```

then the standard initialisation is modified as follows:

kK specifies the size of the I/O space.
gG specifies that the global vector will be initialised in blocks of g (default 100).
iI specifies that the first iK bytes (default 250K) of the stack will be initialised to '* STCK *'.
tT specifies the amount of time in centi-seconds to be made available for post mortem routines (default 0.75 seconds).
D disable the standard trapping of faults (so that a SYSUDUMP can be generated).

All these directives are optional and may be omitted. The remainder of the string following the second '$' is passed to the program (as the argument of START).

When START is called, the initial output selection is to SYSPRINT, if it exists; and the initial input selection is from SYSIN, if it exists.

3.2.2   Execution faults

In the event of an execution fault such as division by zero or a protection exception the routine ABORT is called.   This will print the fault number and the program address when the fault was detected, followed by a summary of the runtime stack (printed out by BACKTRACE) and a map of the program store and globals (printed out by MAPSTORE). This information is output to SYSPRINT which should therefore always be provided.

3.2.3   A complete job

The following is an example of a complete BCPL job to compile and execute a BCPL program using the catalogued procedure BCPLCLG.

```
JOB ...
LIMSTORE 150K
// EXEC BCPLCLG
//BCPL.SYSIN DD DATA

// THIS IS A DEMOSTRATION BCPL PROGRAM

GET "LIBHDR"
    // THIS INSERTS THE STANDARD GLOBAL DECLARATION
LET START(PARM) BE $(1  // START(GLOBAL 1) IS THE MAIN ROUTINE

GLOBAL $( TREE:100; TREEP:101; CH:102  $)

STATIC $( COUNT=0; MIN=0; MAX=0  $)

MANIFEST $(  // THE FOLLOWING NAMES WILL
             // BE USED AS SELECTORS
VAL=0; LEFT=1; RIGHT=2
$)
```

```
// THE FUNCTIONS PUT, LIST AND SUM(DEFINED BELOW)
// OPERATE ON A TREE STRUCTURE WHOSE ROOT IS HELD
// IN TREE.  IF T IS A BRANCH IN THIS TREE THEN
// EITHER T=0
// OR  T POINTS TO A TREE NODE AND VAL!T IS AN
//     INTEGER(K SAY), LEFT!T IS A BRANCH CONTAINING
//     NUMBERS <K AND RIGHT!T IS A BRANCH CONTAINING
//     NUMBERS >=K.
LET PUT(K, P) BE  // THE ROUTINE PUT WILL ADD A NODE TO THE
                  // TREE WHOSE ROOT IS POINTED TO BY P.

   $(P UNTIL !P=0 DO
         $( LET T = !P
            P := K<VAL!T -> @LEFT!T, @RIGHT!T  $)

        VAL!TREEP, LEFT!TREEP, RIGHT!TREEP := K, 0, 0
        !P := TREEP
        TREEP := TREEP + 3  $)P

AND LIST(T) BE  // LIST THE NUMBERS HELD IN THE TREE T
     UNLESS T=0 DO $( LIST(LEFT!T)
                      IF COUNT REM 10 = 0 DO NEWLINE()
                      COUNT := COUNT + 1
                      WRITEF(" %I6", VAL!T)
                      LIST(RIGHT!T)  $)
AND SUM(T) = T=0 -> 0,
             VAL!T<MIN -> SUM(RIGHT!T),
             VAL!T>MAX -> SUM(LEFT!T),
             VAL!T+SUM(LEFT!T)+SUM(RIGHT!T)

LET V = VEC 600

TREE, TREEP := 0, V

NXT: CH := RDCH()  // THIS IS A CONVENIENT WAY
                   // TO ORGANISE A TEST PROGRAM
SW: SWITCHON CH INTO

$(S CASE 'Q': CASE ENDSTREAMCH:
              WRITES("*NEND OF TEST*N")
              FINISH

    CASE 'P':  PUT(READN(), @TREE)  // PUT A NUMBER
               CH := TERMINATOR     // IN THE TREE
               GOTO SW

    CASE 'L': NEWLINE()  // LIST THE NUMBERS IN THE TREE
              COUNT := 0
              LIST(TREE)
              NEWLINE()
              GOTO NXT

    CASE 'S': MIN := READN()
              MAX := READN()
```

```
                WRITEF("*NSUM OF NUMBERS BETWEEN %N AND %N IS %N*N",
                       MIN, MAX, SUM(TREE))
                CH := TERMINATOR
                GOTO NXT

      CASE 'M':  MAPSTORE(); GOTO NXT  // PRINT A STORE MAP

      CASE 'Z': TREE := 0; WRITES("*NTREE CLEARED*N"); GOTO NXT

      CASE '*S': CASE '*N': GOTO NXT  // IGNORE SPACE AND NEWLINE

      DEFAULT: WRITEF("*NBAD CH '%C'*N", CH); GOTO NXT  $)S

  $)1   // END OF PROGRAM
  /*
  //GO.SYSIN DD *
  P24 P13 P96 P46 P-12 P0 P45
  L S10 50
  Q
  /*
```

When this job is run, its GO step will output the following:

```
    -12      0     13     24     45     46     96

SUM OF NUMBERS BETWEEN 10 AND 50 IS 128

END OF TEST
```

Appendix A: Basic symbols and synonyms

The following list of words and symbols are treated as atoms by the syntax analyser. The name of the symbol or its standard representation on the 370 is given in the first column, and examples or synonyms are given in the second.

Basic symbol	Examples and synonyms
identifier	A H1 PQRST TAX_RATE
number	126 7249 #3771
string constant	"A" "*NTEST"
character constant	'X' ')' '*N' '"'
TRUE	
FALSE	
(	
)	
@	LV
!	RV
*	
/	
REM	
+	
-	
=	EQ
~=	NE
<=	LE
>=	GE
<	LS
>	GR
<<	LSHIFT
>>	RSHIFT
~	NOT
&	/\ LOGAND
|	\/ LOGOR
EQV	
NEQV	
->	
,	
TABLE	
VALOF	
;	
:	
$(	$(AB $(1
$)	$)AB $)1
VEC	
BE	
LET	
AND	
:=	
BREAK	
LOOP	
ENDCASE	
RETURN	

FINISH	
GOTO	
RESULTIS	
SWITCHON	
INTO	
REPEAT	
REPEATUNTIL	
REPEATWHILE	
DO	THEN
UNTIL	
WHILE	
FOR	
TO	
BY	
TEST	
THEN	DO
OR	ELSE
IF	
UNLESS	
CASE	
DEFAULT	

Appendix B: BNF of BCPL

This appendix presents the Backus Naur Form of the syntax of BCPL. The whole syntax is given, with the following exceptions:

1. Comments are not included, and the space character is not represented even where required.

2. The section bracket tagging rule is not included, since it is impossible to represent in BNF.

3. The graphic escape sequences allowable in string and character constants are not represented.

4. No account is made of the rules which allow dropping of semicolon and DO in most cases. It seemed that these rules unnecessarily complicate the BNF syntax yet are easy to understand by other means.

5. BCPL has several synonymous system words and operators: for example, DO and THEN. Only a standard form of these symbols is shown in the syntax; a list of synonyms can be found in Appendix A.

6. Certain constructions can be used only in specific contexts. Not all these restrictions are included: for example, CASE and DEFAULT can only be used in switches, and RESULTIS only in VALOF expressions. Finally, there is the necessity of declaring all identifiers that are used in a program.

7. There is a syntactic ambiguity relating to <repeated command> which is resolved in section 2.5.4.

The brackets [ ] imply arbitrary repetition of the categories enclosed.

1. Identifiers, Strings, Numbers

```
<letter>  ::=  A | B | ... | Z

<octal digit>  ::=  0 | 1 | ... | 7

<hex digit>  ::=  0 | 1 | ... | F

<digit>  ::=  0 | 1 | ... | 9

<string constant>  ::=   "<255 or fewer characters>"

<character constant>  ::=  '<one character>'

<octal number>  ::=  # <octal digit> [ <octal digit> ]

<hex number>  ::=  #X <hex digit> [ <hex digit> ]

<number>  ::=  <octal number> | <hex number> | <digit> [ <digit> ]
```

```
<identifier>  ::=  <letter> [ <letter> | <digit> | _ ]
```

2. Operators

```
<address op>  ::=  @ | !

<mult op>  ::=  * | / | REM

<add op>  ::=  + | -

<rel op>  ::=  = | ~= | <= | >= | < | >

<shift op>  ::=  << | >>

<and op>  ::=  &

<or op>  ::=  |

<eqv op>  ::=  EQV | NEQV
```

3. Expressions

```
<element>  ::=  <character constant> | <string constant> |
                <number> | <identifier> | TRUE | FALSE

<primary E>  ::=  <primary E> (<expression list>) |
                  <primary E> ( ) |
                  (<expression>) | <element>

<vector E>  ::=  <vector E> ! <primary E> | <primary E>

<address E>  ::=  <address op> <address E> | <vector E>

<mult E>  ::=  <mult E> <mult op> <address E> | <address E>

<add E>  ::=  <add E> <add op> <mult E> |
              <add op> <mult E> |
              <mult E>

<rel E>  ::=  <add E> [ <rel op> <add E> ]

<shift E>  ::=  <shift E> <shift op> <add E> | <rel E>

<not E>  ::=  ~ <shift E> | <shift E>

<and E>  ::=  <not E> [ <and op> <not E> ]

<or E>  ::=  <and E> [ <or op> <and E> ]

<eqv E>  ::=  <or E> [ <eqv op> <or E> ]

<conditional E>  ::= <eqv E> -> <conditional E> , <conditional E> |
                     <eqv E>
```

```
<expression>  ::=  <conditional E> |
                   TABLE <constant expression>
                     [ , <constant expression> ] |
                   VALOF <command>
```

4. <u>Constant Expressions</u>

```
<C element>  ::=  <character constant> | <number> | <identifier> |
                  TRUE | FALSE | (<const expression>)

<C mult E>  ::=  <C mult E> <mult op> <C element> | <C element>

<constant expression>  ::=  <constant expression> <add op> <C mult E> |
                            <add op> <C mult E> | <C mult E>
```

5. <u>Lists of Expressions and Identifiers</u>

```
<expression list>  ::=  <expression> [ , <expression> ]

<name list>  ::=  <name> [ , <name> ]
```

6. <u>Declarations</u>

```
<manifest item>  ::=  <identifier> = <constant expression>

<manifest list>  ::=  <manifest item> [ ; <manifest item> ]

<manifest declaration>  ::=  MANIFEST $( <manifest list> $)

<static declaration>  ::=  STATIC $( <manifest list> $)

<global item>  ::=  <identifier> : <constant expression>

<global list>  ::=  <global item> [ ; <global item> ]

<global declaration>  ::=  GLOBAL $( <global list> $)

<simple definition>  ::=  <name list> = <expression list>

<vector definition>  ::=  <identifier> = VEC <constant expression>

<function definition>  ::=  <identifier> (<name list>) = <expression> |
                            <identifier> ( ) = <expression>

<routine definition>  ::=  <identifier> (<name list>) BE <command> |
                           <identifier> ( ) BE <command>

<definition>  ::=  <simple definition> | <vector definition> |
                   <function definition> | <routine definition>

<simultaneous declaration>  ::=  LET <definition> [ AND <definition> ]

<declaration>  ::=  <simultaneous declaration> |
                    <manifest declaration> |
                    <static declaration> |
                    <global declaration>
```

7.   <u>Left hand side Expressions</u>

```
<LHSE>  ::=  <identifier> | <vector E> ! <primary E> |
             ! <primary E>

<left hand side list>  ::=  <LHSE> [ , <LHSE> ]
```

8.   <u>Unlabelled Commands</u>

```
<assignment>  ::=  <left hand side list> := <expression list>

<simple command>  ::=  BREAK | LOOP | ENDCASE | RETURN | FINISH

<goto command>  ::=  GOTO <expression>

<routine command>  ::=  <primary E> (<expression list>) |
                        <primary E> ( )

<resultis command>  ::=  RESULTIS <expression>

<switchon command>  ::=  SWITCHON <expression> INTO <compound command>

<repeatable command>  ::=  <assignment> | <simple command> |
                           <goto command> | <routine command> |
                           <resultis command> | <repeated command> |
                           <switchon command> | <compound command> |
                           <block>

<repeated command>  ::=  <repeatable command> REPEAT |
                         <repeatable command> REPEATUNTIL <expression> |
                         <repeatable command> REPEATWHILE <expression>

<until command>  ::=  UNTIL <expression> DO <command>

<while command>  ::=  WHILE <expression> DO <command>

<for command>  ::=  FOR <identifier> = <expression> TO <expression>
                        BY <constant expression> DO <command> |
                    FOR <identifier> = <expression> TO <expression>
                        DO <command>

<repetitive command>  ::=  <repeated command> | <until command> |
                           <while command> | <for command>

<test command>  ::=  TEST <expression> THEN <command> OR <command>

<if command>  ::=  IF <expression> THEN <command>

<unless command>  ::=  UNLESS <expression> THEN <command>
```

```
<unlabelled command>  ::=  <repeatable command> | <repetitive command>
                           <test command> | <if command> |
                           <unless command>
```

9. Labelled Commands

```
<label prefix>  ::=  <identifier> :

<case prefix>  ::=  CASE <constant expression> :

<default prefix>  ::=  DEFAULT :

<prefix>  ::=  <label prefix> | <case prefix> | <default prefix>

<command>  ::=  <unlabelled command> |
                <prefix> <command> |
                <prefix>
```

10. Blocks and Compound Commands

```
<command list>  ::=  <command> [ ; <command> ]

<declaration part>  ::=  <declaration> [ ; <declaration> ]

<block>  ::=  $( <declaration part> ; <command list> $)

<compound command>  ::=  $( <command list> $)

<program>  ::=  <declaration part>
```

Appendix C: The EBCDIC character set

The following table gives the decimal values and graphics of the characters available on the 370 implementation of BCPL.

0		1		2		3		4		5	HT	6		7	
8		9		10		11		12	NP	13	CR	14		15	
16		17		18		19		20		21	NL	22	BS	23	
24		25		26		27		28		29		30		31	
32		33		34		35		36		37	LF	38		39	
40		41		42		43		44		45		46		47	
48		49		50		51		52		53		54		55	
56		57		58		59		60		61		62		63	
64	SP	65		66	[	67	]	68		69		70		71	
72		73		74		75	.	76	<	77	(	78	+	79	|
80	&	81		82		83		84		85		86		87	
88		89		90	!	91	$	92	*	93	)	94	;	95	~
96	-	97	/	98	\	99	┃	100	┤	101	^	102	_	103	
104		105		106		107	,	108	%	109	_	110	>	111	?
112		113		114		115		116		117		118		119	
120		121		122	:	123	#	124	@	125	'	126	=	127	"
128		129	a	130	b	131	c	132	d	133	e	134	f	135	g
136	h	137	i	138		139		140		141		142		143	
144		145	j	146	k	147	l	148	m	149	n	150	o	151	p
152	q	153	r	154		155		156		157		158		159	
160		161		162	s	163	t	164	u	165	v	166	w	167	x
168	y	169	z	170		171		172		173		174		175	
176		177		178		179		180		181		182		183	
184		185		186		187		188		189		190		191	
192		193	A	194	B	195	C	196	D	197	E	198	F	199	G
200	H	201	I	202		203		204		205		206		207	
208		209	J	210	K	211	L	212	M	213	N	214	O	215	P
216	Q	217	R	218		219		220		221		222		223	
224		225		226	S	227	T	228	U	229	V	230	W	231	X
232	Y	233	Z	234		235		236		237		238		239	
240	0	241	1	242	2	243	3	244	4	245	5	246	6	247	7
248	8	249	9	250		251		252		253		254		255	

Where	HT	is	horizontal tab
	NL	is	newline
	CR	is	carriage return
	LF	is	line feed
	NP	is	newpage
	BS	is	backspace
	SP	is	space

and the end-of-stream character ENDSTREAMCH is minus one.

Appendix D: Common extensions

The extensions given here are available on the 370 version of BCPL and, although they are not in the standard language, they should be considered by other implementers of BCPL planning to extend the language. This appendix is provided in the hope that it will reduce needless incompatibilities between different implementations.

It must be stressed that these extensions should only be used where absolutely necessary, and then as sparingly as possible. They tend to decrease the efficiency and the understandability of the program and often indicate bad programming style.

On machines with suitable wordlengths, floating-point arithmetic and field selection are appropriate extensions. These are described in the next two sections The last section describes other miscellaneous extensions[

Floating-point arithmetic

A floating-point constant may have one of the following forms:

```
i.jEk
i.j
iEk
```

where i and j are unsigned integers and k is a (possibly signed) integer. The value is represented on the 370 as a 32 bit floating-point number.

The arithmetic and relational operators for floating-point quantities are as follows:

```
#*  #/
#+  #-
#=  #~=  #<=  #>=  #<  #>
```

They have the same precedence as the corresponding integer operators. There are, also, two monadic operators FIX and FLOAT for conversions between integers and floating-point numbers. They have the same precedence as @.

Field selectors

Field selectors allow quantities smaller than a whole word to be accessed with reasonable convenience and efficiency. A selector is applied to a pointer using the operator OF(or ::). It has three components: the size, the shift and the offset. The size is the number of bits in the field, the shift is the number of bits between the right-most bit of the field and the right hand end of the word containing it, and the offset is the position of the word containing the field relative to the pointer.

The precedence of OF is the same as that of the subscription operator(!), but its left operand (the selector) must be a constant expression.   A convenient way to specify a selector is to use the operator SLCT whose syntax is as follows:

```
        <constant expression>  ::=  SLCT <size>:<shift>:<offset> |
                                    SLCT <size>:<shift> |
                                    SLCT <size>
```

where <size>, <shift> and <offset> are constant expressions.   Unless explicitly specified the shift and offset are assumed to be zero by default.   A size of zero indicates that the field extends to the left hand end of the word.

Selectors are best defined using manifest declarations.

A selector application may be used on the left hand side of an assignment and in any other context where an expression may be used, except as the operand of @.   In the assignment

```
                F OF P := E
```

the appropriate number of bits from the right hand end of E are assigned to the specified field.   When

```
                F OF P
```

is evaluated in any other context, the specified field is extracted and shifted so as to appear at the right hand end of the result.

On the 370, fields corresponding to half-words and bytes are treated efficiently.

<u>Miscellaneous extensions</u>

a)   Identifiers

Capital and small letters, dots and underlines may be used in identifiers and section bracket tags.   System words must still be spelt in capitals.

b)   Constants

Binary, octal and hexadecimal constants may be written using the warning sequences #B, #O(or just #) and #X.   They denote right justified values.

c)   Comments

The sequence || (like //) introduces a comment which extends to the end of the line.   The sequence |* ...   *| (like /* ...   */) is a bracketed comment (possibly containing newlines).

d)   Operators

The monadic operators ABS and #ABS obtain the absolute value of an integer or floating-point number. They have the same precedence as @.

The operator <> has a similar meaning to semicolon but is syntactically more binding than DO, OR, REPEAT etc. For example

```
IF E DO C1 <> C2
```

is equivalent to

```
IF E DO $( C1 ; C2 $)
```

The operator _ is a synonym for the assignment operator :=, and both may be preceded by one of the following diadic operators:

```
*  /  REM  #*   #/
+  -  #+  #-
&
|
EQV  NEQV
```

The meaning of

```
E1 <op>:= E2
```

is the same as

```
E1 := E1 <op> E2
```

e) Segment headings

A segment of BCPL program may start with a directive of the following form:

```
SECTION "name"
```

where name is a module name acceptable to the linkage-editor. It defines the section name given to the object module corresponding to the segment of program.

This directive may be followed by one or more directives of the form:

```
NEEDS "name"
```

where name is a module name acceptable to the linkage-editor. It causes an external reference to be set up in the object module, with the result that the specified object will be retrieved automatically by the linkage-editor.