BCPL MANUAL

James E. Curry

Xerox Palo Alto Research Center
3180 Porter Drive
Palo Alto, California 94304
(415) 493-1600

February 15, 1973

Variables

BCPL is a vaguely ALGOL-like language (it is block-structured; it allocates procedure space dynamically, so recursion is permissable; and most BCPL statements correspond roughly to ALGOL statements, although there are syntactic differences). The major difference between BCPL and ALGOL is that all ALGOL variables are declared with data-types (integer, real, boolean, string, array, procedure, label, pointer, etc.), whereas all BCPL variables have the same data-type: a 16-bit value. In ALGOL, the meaning of an expression is dependent both on its context and on the data-types of the entities involved, and only expressions with certain data-types may appear in a given context. In BCPL, any expression may be used in any context; the context alone determines how the 16-bit value of the expression is interpreted. BCPL never checks that a value is "appropriate" for use in a given way. For example, an expression which appears in a 'goto' statement is assumed to have as its value the address of someplace which is reasonable to jump to; the thing following a 'goto' need not be a label. The advantages of this philosophy about data-types are that it allows the programmer to do almost anything, and that it makes the language conceptually simple. The disadvantages are that the user can make errors which would have been caught by data-type checking, and that some things must be done explicitly which ALGOL-type languages would do automatically (implicit indirection on pointer variables, operations on multi-word values such as real numbers and strings, type conversion, etc.).

Although BCPL has only one data-type, it does distinguish between two kinds of variables: static and dynamic. They differ as to when and where the cells to which they refer are allocated. A static variable refers to a cell which is allocated at the beginning of program execution (i.e., by the BCPL loader); it refers to the same memory cell for as long as the program runs. A dynamic variable refers to a cell which is (conceptually) allocated when the block in which it is defined is entered, and exists only until execution of that block terminates. The space from which the dynamic variable is allocated is created dynamically when the procedure containing its defining block is called.

As in ALGOL, variable names (and other names) are defined in declarations. The lexical scope of a declared name (the portion of the source text in which the name is defined) is governed by BCPL's block structure.

Scope Rules and Blocks

At the outermost level, a BCPL source file consists of a sequence of global declarations followed by a multiple procedure declaration. The possible global declarations are:

```
external [NAME; ...; NAME]
static [NAME = CONST; ...; NAME = CONST]
manifest [NAME = CONST; ...; NAME = CONST]
```

The 'external' and 'static' declarations define static variables; the 'manifest' declaration defines literals.

A multiple procedure declaration has the form

```
let NAME(ARG, ..., ARG) BODY
and NAME(ARG, ..., ARG) BODY
...
and NAME(ARG, ..., ARG) BODY
```

where BODY is either 'be STAT' or '= EXP'.

The NAMEs in 'external', 'static', and 'manifest' declarations at the outermost level are defined from the point of declaration to the end of the source file; all of the NAMEs in the "let ... and ..." sequence at the outermost level are defined in all of the BODYs. These are the only names which are globally defined. All other names are defined either as ARGs in the procedure declarations, or in local declarations within compound statements in the BODYs.

A compound statement is a sequence of statements and declarations, separated by semicolons, and enclosed within the brackets '[' and ']'. (If a carriage return separates two statements, the semicolon can be omitted.) The brackets have a function similar to that of the words 'begin' and 'end' in ALGOL. A compound statement may be used wherever a simple statement can be; in this manual, 'STAT' always means either a simple statement or a compound statement. Compound statements are used when two or more statements are needed in a context in which BCPL expects a single statement (e.g., as the body of a procedure, or as one of the arms of a conditional statement). Compound statements also delimit the scope of locally declared names.

The brackets delimiting a compound statement may be labeled with a sequence of letters and digits immediately following the '[' or ']'. When a labeled ']' is seen by the compiler, each unmatched '[' (whether it is labeled or not) is implicitly matched until the '[' with the same label is matched. Thus, in:

```
if n gr 0 do [1 i = 1
                until i gr n do
```

```
[ 2 x!i = Ø; i = i + 1 ]1
```

the "]1" closes both compound statements. Note that a space or tab must be present between an unlabeled '[' and a statement that starts with a name. Usually some error will be detected quickly if no space is left (as in "if n gr Ø do [i = 0 ... "). But sometimes the resulting statement will be legal (as in "if n gr Ø do [rv x = 0 ... "). In such cases, the error may not be detected until the end of the source text; this is often the cause of a non-obvious "unmatched section bracket" syntax error.

As mentioned above, a compound statement may contain local declarations. These declarations may be intermixed with statements (unlike ALGOL, in which declarations may appear only at the beginning of a compound statement). "Declaration" here includes dynamic variable declarations ('let NAME1, ..., NAMEn = EXP1, ..., EXPn'), as well as the 'external', 'static', 'manifest', and procedure declarations mentioned above. The following rules govern the scope of local names:

1) A local declaration may appear in a compound statement only in the following contexts: at the beginning of a statement, or after a semicolon (including a semicolon implicitly inserted by the compiler between statements on different lines), or following a statement label that follows a semicolon. The effect of this rule is to disallow things like "if x eq Ø then let y = Ø (although "if x eq 0 then [let y = Ø ...] is perfectly legal). A declaration may be labeled.

2) A declaration starts a <u>block</u>; the block ends at the end of the compound statement containing the declaration. A name defined in the declaration is known only within the block introduced by the declaration, and in sub-blocks contained within that block if the name is not redeclared.

3) (Exception to rule (2).) A <u>dynamic</u> variable is not known in any procedure body other than the one in which it was declared. Thus, if the procedure "g" is declared inside of the body of procedure "f", the dynamic variables defined in "f" are <u>not</u> known to "g". (This is because the dynamic variables of "f" reside in space which is dynamically allocated when "f" is called. When "g" is called, it does not know where this space is; in fact, there might be more than one execution of "f" in progress when "g" is called, or there might not be any active execution of "f".)

4) A statement label ('NAME: ...') appearing within a block is treated as if it were a static variable declared immediately after the declaration which begins the block. So a label is known throughout its enclosing block, but not outside that block.

5) A name which is declared external may not be redeclared within the scope of the external declaration. (A static declaration of an external name has a special meaning. See the section on static and external variables.) Any other name may be redeclared in a sub-block of a block in which it is defined, as in ALGOL.

Manifest Declarations

The declaration

manifest [NAME1 = CONST1; ...; NAMEn = CONSTn]

defines NAME1 through NAMEn as literals. The expressions CONST1 through CONSTn must be constant expressions; that is, their values must be computable by the compiler. The meaning of a program would be unchanged if each manifest name were replaced by a string of digits representing its value. In particular, manifest names do not have addresses.

Static and External Declarations

Static variables may be declared in four ways: by a 'static' or 'external' declaration, by a procedure declaration, or by a statement label assignment.

The declaration

static [NAME1 = CONST1; ...; NAMEn = CONSTn]

defines NAME1 through NAMEn as static variables, and causes them to be initialized with the values CONST1 through CONSTn at the beginning of program execution (i.e., in the "save file" created by the loader). The CONSTs must be expressions whose values are computable by the compiler. The expression 'nil' should be used if it doesn't matter what the variable is initialized to.

If a carriage return separates two "NAME = CONST" phrases, the semicolon between them may be omitted.

Any of the NAMEs that are preceded by an '@' will be allocated by the loader in page zero. Such variables are called "common" variables. They can be addressed directly by the compiled code, whereas normal static variables must be addressed by indirection through a literal; so common variables are more efficient. However, there is room in page zero for at most 150 (decimal) common variables; the loader will complain bitterly if too many common variables are assigned.

The procedure declarations

```
let NAME(ARG, ..., ARG) be STAT
let NAME(ARG, ..., ARG) = EXP
```

declare NAME as a static variable which is to be initialized by the loader to the address of the code compiled for the procedure. (This is not quite accurate for code which is to be dynamically loaded -- see the loader documentation for details.) If NAME is preceded by an '@', it will be defined as a common variable.

The procedure declaration is discussed fully in the sections on procedure and dynamic variable declarations.

A statement label assignment

```
NAME:  STAT
```

declares NAME as a static variable which is to be initialized by the loader to the address of the code compiled for STAT (unless the code is to be dynamically loaded -- see the loader documentation). A label assignment does not begin a block; the name is treated as if it were declared immediately after the declaration which begins the smallest enclosing block. Thus, a label is defined throughout the block in which it appears.

The declaration

```
external [NAME1; ...; NAMEn]
```

declares NAME1 through NAMEn as external static variables. The purpose of the 'external' declaration is to allow separately compiled pieces of a program to reference the same variables. Within a given source file, the scope of an external variable is the same as that of other types of variables; but if two or more separately compiled source files declare a given name 'external', the loader will make each refer to the same cell. In (exactly) one of the source files in which a given name is declared external, the name should also be declared as a static variable (by a 'static' declaration, a procedure declaration, or a statement label assignment) someplace within the scope of the

'external' declaration. (Note that the static declaration must follow the external declaration.) This is not a re-definition of the name (external names cannot be redefined -- see below), but rather tells the loader how to initialize the external static variable. The loader will complain (although not fatally) about an external variable which is not declared static someplace, or about one which is declared static more than once. (If the initial value of such a variable doesn't matter, everything is fine, since the variable is allocated anyway; but undefined procedures and labels should probably be investigated.)

Unlike other names, a name which is declared external may not be redeclared anywhere within the scope of the external declaration, except for a static declaration with the above meaning. That is, it has the same definition throughout the block in which it is declared external. It may be redeclared in a parallel block.

NAMEs that are preceded by an '@' will be defined as common variables. A NAME that is declared both external and static may be designated as common in either or both declarations.

If a carriage return separates two of the NAMEs, the semicolon between them may be omitted.

Note that only static variables may be external (not manifest names or dynamic variables).

Procedure Declarations

There are two kinds of BCPL procedures: "functions", which return a value upon completion, and "routines", which do not. A function is defined by a declaration of the form

```
let NAME (ARG1, ..., ARGn) = EXP
```

A routine is defined by

```
let NAME(ARG1, ..., ARGn) be STAT
```

NAME is the name of the function or routine being defined. (Actually, NAME becomes a static variable which will be initialized with the address of the procedure, as noted in the section on static variables.) ARG1 through ARGn are the formal parameters (dummy arguments) of the procedure. They are either NAMEs, or the special symbol 'nil', indicating an unnamed argument. ARG1 through ARGn become the first n dynamic variables declared in the procedure body. If there are no dummy arguments, the declaration is of the form 'let NAME( ) be STAT' or 'let NAME( ) = EXP'.

In the function declaration, EXP is the expression whose value is returned when the function is called. EXP may be a simple BCPL expression; but for most functions it will be an expression of the form 'valof STAT', where STAT may be a compound statement. The STAT in a 'valof' expression should contain at least one 'resultis' statement. The STAT is executed until a statement of the form 'resultis EXP' is encountered; then EXP becomes the value of the 'valof' expression, and therefore the result of the function. The 'valof' expression will also terminate when control would otherwise pass to the statement following STAT. If this happens, the value of the 'valof' expression is garbage.

In the routine declaration, STAT is the statement which is executed when the routine is called. STAT may be a compound statement. STAT may contain one or more 'return' statements; the routine returns when a 'return' statement is executed, or when control would otherwise pass to the statement following STAT.

A multiple procedure declaration has the form

```
let NAME1(ARG, ..., ARG) be STAT (=EXP)
and NAME2(ARG, ..., ARG) be STAT (=EXP)
...
and NAMEn(ARG, ..., ARG) be STAT (= EXP)
```

This declares the procedures NAME1 through NAMEn "simultaneously'; that is, all of the NAMEi's are known in each of the procedure bodies. (So, for example, NAME1 can call NAME2 and NAME2 can call NAME1.) The ARGs, of course, are defined only in their corresponding procedure bodies.

A procedure body may contain procedure declarations; the names of such procedures will be local to the defining body (unless they are declared external). But remember rule (3) in the section on the scope of dynamic variables.

Procedure Execution

A procedure is called by a statement or expression of the form

```
EXP(EXP1, EXP2, ..., EXPn)
```

(EXP determines the procedure to be executed; EXP1 through EXPn are the actual parameters. If there are no actual parameters, the form is 'EXP( )'. A procedure call is an expression if it appears in a context in which a value is expected (e.g., in the right-hand side of an assignment statement); otherwise, it is a statement. The calling mechanism is the same in either case. The only difference is that in the context of an expression, the

procedure is expected to return a value; if it doesn't (because it is a "routine" rather than a "function"), a garbage value will be used. A value which is returned by a function called in the context of a statement is discarded.

EXP will normally be a name declared as a procedure in some BCPL source file. (Recall that the value of this name will normally be the address at which the code compiled for the procedure resides.) But EXP may be any BCPL expresssion. A "JSR" is done to 'rv EXP' (see the section on rv-expressions). All bets are off if EXP does not point at a procedure body.

When the procedure is entered, it first allocates some "frame" space from someplace in memory. This "frame" is a block of memory which the current execution of the procedure will use for the actual parameter values, for any dynamic variables and vectors declared within the procedure, and for any temporary storage needed by the procedure. The size of the frame is determined by how much space the procedure needs. The space is de-allocated when the procedure executes the 'return' or 'resultis' corresponding to the call that allocated the frame.

After the frame space is allocated, the values of EXP1 through EXPn are stored in the first n words of the frame. These n words are those referenced by the n formal parameters ARG1, ..., ARGn in the procedure declaration, assuming that the procedure is called with exactly the number of actual parameters as it was declared to have. (No check is made to see if actual and formal parameters match. If there are fewer actual parameters, the formal parameters with no corresponding actual parameters will have garbage values. If there are more actual parameters than formal parameters, the actual parameters with no corresponding formal parameters will be lost; but this may create havoc by clobbering memory words which are not part of the newly created frame.)

Note that each formal parameter takes on the <u>value</u> of its corresponding actual parameter at the beginning of the procedure call. This implies that procedure calls are implemented by the "call by value" mechanism (in the ALGOL sense); assigning a value to a formal parameter within a procedure does not affect the value of the corresponding actual parameter in the calling routine, although it does change the value of the formal parameter for the remainder of the procedure execution. Suppose the function "next" is defined by:

```
let next(x) = valof [x = x + 1; resultis x]
```

and called as follows:

```
a = Ø; b = next (a)
```

After the call of next, "a" will still be Ø, but "b" will be 1. (We can write "next" in such a way as to allow it to change the value of "a" by using the address-manipulation primitives of BCPL:

```
let next (xaddr) = valof
     [rv xaddr = rv xaddr + 1; resultis rv xaddr]
```

Then calling "next" as follows:

```
a = Ø; b = next (lv a)
```

will cause both "a" and "b" to have the value 1.)

After the procedure frame has been allocated and the actual parameters have been stored in the frame, the procedure body is executed. When the procedure terminates, the frame space is de-allocated and control returns to the caller.

Dynamic Declarations

A dynamic variable refers to a cell at some fixed position in the frame associated with the current execution of the procedure in which it is defined. This cell is only allocated to the variable while the block defining the variable is active (e.g., while the block is being executed, or while a procedure called from within the block is being executed). Outside of the block, the cell may be used for something else.

Dynamic variables are declared in two ways: by a declaration of the form

```
let NAME1, ..., NAMEn = EXP1, ..., EXPn
```

and as formal parameters in a procedure declaration.

The declaration

```
let NAME1, ..., NAMEn = EXP1, ..., EXPn
```

allocates n consecutive frame cells to NAME1 through NAMEn, and compiles code to assign the values of EXP1 through EXPn to NAME1 through NAMEn. Unlike other declarations, this declaration is executable; NAME1 through NAMEn always refer to the same frame cells, but the values stored in these cells are recomputed each time the declaration is executed. The assignment is done left-to-right.

The EXPs may be any BCPL expression. In addition, there are two special cases: 'nil' and 'vec CONST'.

If EXPi is the symbol 'nil', the variable NAMEi is declared, but no value is assigned to NAMEi. Thus, "let x = nil" declares x, but compiles no code; "x" will have some garbage value until something is assigned to it.

If EXPi is the special expression 'vec CONST' (where CONST is an expression that can be evaluated by the compiler), the value assigned to NAMEi will be the address of the first word of a block of CONST+1 consecutive frame cells. This "vector" of CONST+1 cells is allocated from the frame space, and NAMEi is initialized to point to that vector. These cells exist as long as NAMEi exists; they may be used for something else outside of the block in which the declaration appears.

In a procedure declaration

```
let NAME(ARG1, ..., ARGn) be STAT (=EXP)
```

ARG1 through ARGn are declared as dynamic variables; their scope is the entire procedure body. This declaration is equivalent to

```
let NAME( ) be
    [let ARG1, ..., ARGn = nil, ..., nil; STAT]
```

or to

```
let NAME( ) = valof
    [let ARG1, ..., ARGn = nil, ..., nil; resultis EXP]
```

That is, ARG1 through ARGn are the first n dynamic variables declared in the procedure body, and therefore refer to the first n cells in the frame. The procedure call 'NAME(EXP1, ..., EXPm)' stores the values of the m actual arguments in the first m cells of the newly created frame. So if m > n, cells m + 1 through n will be clobbered; this is undesirable if the frame is less than n cells long. If m = n, all is well. If m < n, ARGs m + 1 through n will have garbage values. This permits procedures to be called with a variable number of actual arguments, as long as enough formal arguments are declared to provide space for the largest actual argument list. For example, if we define a procedure something like

```
let f(xØ, x1, x2, ..., x20) be
    [   let arg = lv xØ
        ... arg!i ...
    ]
```

then the expression 'arg!i' references the ith argument.

The ARGs are usually NAMEs, but the special symbol 'nil' is also legal as an ARG. The 'nil' has the effect of leaving space for an argument, but not declaring a name for that argument. So the procedure "f" above might also have been defined as

let f(xØ, nil, nil, ..., nil) ...

Argument i can still be referenced by 'arg!i'.

Memory References

There are three kinds of BCPL expressions which refer to memory cells: variable names, rv-expressions, and vector references. These are the only things that can appear as the left-hand side of an assignment statement 'REF = EXP' or as the argument of an lv-expression 'lv REF'. In an assignment statement, REF specifies the cell to be modified. The value of an lv-expression is the address of the cell specified by REF. (These two contexts are the only ones in which the form of the expression is restricted.) In all other contexts, the value of a memory-reference expression is the value contained in the specified cell.

Memory reference expressions are described below in terms of the Nova instructions compiled. There are six Nova op-codes that reference memory: LDA ac, STA ac, JMP, JSR, ISZ, DSZ. The symbol "OP" in the description below designates one of these op-codes; the address of the op-code is in standard Nova form (@ displacement, index). In general, an assignment statement generates a STA; a 'goto' generates a JMP; a procedure call generates a JSR; and other contexts generate a LDA. Statements of the form 'REF = EXP ± 1' generate ISZ or DSZ (followed by a NOP).

dynamic variable names:

Dynamic variables are allocated cells in the first 400 (octal) words of the frame for the procedure in which they are declared. While a procedure is being executed, AC2 always points at the 200th (octal) word of its frame; so dynamic variables are referenced by "OP n,2", where $-200 \leq n \leq 177$. (Because the frame is allocated dynamically when a procedure is called, dynamic variables cannot be accessed directly from any procedure other than the one in which they are declared, as noted in scope rule (3).)

static variable names:

Static variables are allocated space by the loader, either in "common" (page zero) or in another area of memory which is fixed during loading. Common variables are accessed by "OP n,Ø", where 0 ≤ n ≤ 377. Other static variables are not directly addressable, since they are in some arbitrary area of core, so they are addressed through indirection by "OP @n,1", where n is the PC-relative offset (-200 ≤ n ≤ 177) of a word containing the address of the static variable.

vector references: EXP1 ! EXP2

This expression references a memory cell whose address is given by the value of (EXP1 + EXP2). The reason for calling an expression like "A!I" a "vector reference" is illustrated by the following: Suppose that the value of the variable "A" is the address of the first word of a zero-origin one-dimensional array (a "vector"). Then the expression "A!I" references the Ith word of the vector A, since the value of the expression "A+I" is the address of this word.

In general, vector references generate code to compute the sum of EXP1 and EXP2 in AC3 (e.g., "LDA 0,EXP1; LDA 3,EXP2; ADD 0,3"), and then reference the vector element with "OP Ø,3". In the case where EXP2 (or EXP1) is a small constant (-200 ≤ n ≤ 177), EXP1 (or EXP2) is loaded into AC3, and the vector element is accessed by "OP n,3". In any case, a vector reference always uses indexing through AC3. See the note on rv-expressions below.

rv-expressions: rv EXP

This expression references a memory cell via indirect addressing through EXP. In general, the value of EXP is computed and stored in a temporary cell in the frame, and the reference is done by "OP @n,2", where n is the offset of the temp cell. There are several special cases: If EXP is a dynamic variable name, "OP @n,2" is used, where n is the frame offset of the variable. If EXP is a common variable name, "OP @n,Ø" is used, where n is the page zero address of the variable. If EXP is a static variable name, "OP @n,1" is used,

where n is the PC-relative offset of a word containing the address of the static variable with the indirect bit (bit Ø) set. If EXP is a vector reference, "OP @n,3" is used, after loading AC3 appropriately.

The expression 'rv EXP' may also be written '@EXP'.

An rv-expression always generates an indirect reference through a memory cell. A vector reference always generates an instruction which is indexed by AC3. Therefore, "rv EXP" is <u>not</u> necessarily equivalent to "EXP1!EXP2" when the values of (EXP) and (EXP1 + EXP2) are the same: the rv-expression will always cause a multiple indirection if EXP has bit Ø set; a vector reference will never do so, since indexing ignores bit Ø.

Constants

BCPL recognizes the following constructs as constants:

- A name which is declared 'manifest' is treated as if it had been replaced by its value.
- A string of digits is interpreted as a decimal integer. It may not exceed 2**15-1 (32767 decimal, 77777 octal).
- A string of digits preceded by a '#' is interpreted as an octal integer. It must be less than 2**16-1 (177777 octal, 65535 decimal).
- A string of digits immediately followed by 'B' or 'b' is also interpreted as an octal integer. If the 'B' or 'b' is immediately followed by a (decimal) number n, the octal value is shifted left n bits. Thus, #1230, 1230B, and 123B3 all represent the same value. One-bits may not be shifted out of bit Ø.
- The reserved words 'true' and 'false' are constants with values #177777 and 0 respectively.
- A '$' followed by any printing character other than '*' represents a constant whose value is the ASCII code of the character. '*' is an escape character; the following escapes are recognized:

S	space (#40)
T	tab (#11)
N	carriage return (#15)
L	line feed (#12)
"	double quote (#42) ($" is also O.K.)
nnn	the octal number 'nnn'. (There must be exactly three octal digits.)

'*' followed by anything else gives an error.

The compiler evaluates most expressions that involve only constants, and treats the resulting value as a single constant. (The exceptions are 'selecton' and 'valof' expressions; memory references; and function calls. Conditional expressions like 'CONST ? CONST1, CONST2' are evaluated; the value is CONST2 if CONST is Ø, and CONST1 otherwise.) Throughout this manual, the symbol 'CONST' (described as "an expression which can be evaluated by the compiler") means either one of the constant constructs above, or an expression involving only constants.

Expressions

In order of decreasing precedence, the legal BCPL expressions are:

NAME; constant; (EXP)

EXP(EXP1, ..., EXPn)

EXP1!EXP2

lv EXP; rv EXP; ± EXP

EXP1 <mul> EXP2 (<mul>: *, /, rem, lshift, rshift)

EXP1 + EXP2; EXP1 - EXP2

vec CONST

EXP1 <rel> EXP21(<rel>: eq, ne, ls, le, gr, ge)

not EXP

EXP1&EXP2

EXP1%EXP2

EXP1 xor EXP2; EXP1 eqv EXP2

EXP ? EXP1, EXP2

selecton EXP into ...

valof STAT

Operators with the same precedence are left-associative, except for '<mul>', '&', '%', 'xor', and 'eqv', which are right-associative. Precedence and associativity can be changed by parenthesizing. Some cases to note:

"a/b*c" is "a/(b*c)"
"rv v!i" is "rv(v!i)"
"v!i+j" is "(v!i)+j"
"a%b&c" is "a%(b&c)"
"a & b eq c" is "a & (b eq c)"

Precedence only determines the way in which an expression is parsed; nothing is implied about order of evaluation. In general, the order in which the sub-expressions of an expression are computed is unspecified. So, although "f(x) + g(y) * h(z)" means "f(x) + (g(y) + h(z)", no assumption should be made about which function is executed first.

string literals

A sequence of characters enclosed in double quotes (") is a string literal. Its value is the address of the first word of a block of memory containing the string. A BCPL string is stored two bytes per word, left-hand byte first, with the left-hand byte of the first word containing the number of characters in the string. If the string has an even number of characters, the right-hand byte of the last word is Ø; but if it has an odd number of characters, the last word of the string contains the last two characters, not two Ø bytes. Note that BCPL strings are quite different from Nova strings.

Strings have a maximum length of 255 characters. The character '*' appearing in a string is an escape character, as described for character constants.

EXP ( )
EXP (EXP1, EXP2, ..., EXPN)

The value of EXP is assumed to be the address of a BCPL function. This function is called with the

values of EXP1, ..., EXPN as arguments. The value of the function call is the value returned by the function via a 'result is' statement. See the section on procedure execution for details.

The call is implemented by a Nova JSR instruction (a memory reference op-code) to 'rv EXP'. So if EXP has bit Ø set, a multiple indirection will take place. If bit Ø is zero, the value of EXP is the address of the first instruction executed.

The empty argument list "( )" is necessary if there are no arguments. "x = f( )" calls a function, but "x = f" puts the address of the function in "x". Forgetting the "( )" is a common error; be careful.

lv REF

REF must be a variable name, a vector reference, or an rv-expression; anything else gives an error message. The value of the lv-expression is the address of the cell which REF references (but see the note cn l"lv(rv EXP)" below.

The value of "lv NAME", if NAME is a dynamic variable, is the sum of the current frame pointer (which is in AC2) and the offset of the variable in the frame (a constant). This address is valid only while the block in which the variable was declared is active.

The value of "lv NAME", where NAME is a static variable, is the address of the static variable. This is a constant throughout the execution of the program, since static variables never move.

The value of "lv(EXP1!EXP2)" is the sum of the values of EXP1 and EXP2.

The value of "lv (rv EXP)" is the value of EXP. Note that this is nct the address of the cell that "rv EXP" references if EXP has bit Ø set. In this case, "rv EXP" would cause a multiple indirection.

rv EXP
EXP1 ! EXP2

See the section on memory references.

+EXP

The value is the value of EXP.

-EXP

The value is the two's-complement of the value of EXP.

EXP1 * EXP2

The value is the low-order 16 bits of the 32-bit signed product. If one of the EXPs is a constant whose value is a power of 2, a left shift is done; otherwise the standard Nova multiply sequence is done. There is currently no way to get at the high-order part of the product, or to detect overflow.

EXP1 / EXP2
EXP2 rem EXP2

The standard Nova signed integer divide sequence is done. (Division by a power of two is <u>not</u> done by shifting.) The "/" expression gives the 16-bit signed quotient; the "rem" expression gives the 16-bit remainder, which has the same sign as EXP1. If EXP2 is zero, the results are undefined. There is currently no way to detect this.

EXP1 lshift EXP2
EXP1 rshift EXP2

The value is the value of EXP1 shifted left or right EXP2 bits. Vacated positions are filled with <u>Ø's</u>. Bits shifted off either end of the 16-bit word are lost. The shifts are logical, not arithmetic, in that the sign bit may be changed. There are currently no arithmetic- or circular-shift operators.

EXP1 + EXP2
EXP1 - EXP2

The value is the sum (difference) EXP1 and EXP2. The statement 'EXP = EXP ± 1' generates an ISZ or DSZ followed by a NOP. There is currently no way to detect overflow.

EXP1 eq EXP2
EXP1 ne EXP2
EXP1 ls EXP2
EXP1 le EXP2
EXP1 gr EXP2
EXP1 ge EXP2

The (signed) values of EXP1 and EXP2 are compared, and the value of the expression is always either 'true' (#177777) or 'false' (Ø).

not EXP

The value is the logical complement (one's-complement) of the value of EXP. But see the note on "&" and "%" below.

EXP1 & EXP2
EXP1 % EXP2

In most contexts, the value is the logical-and or logical-or of EXP1 and EXP2. However, in the context of the boolean part of an 'if', 'unless', 'test', 'while', 'until', 'repeatwhile', or 'repeatuntil' statement, or of a conditional expression, the evaluation of an expression involving 'not', "&", or "%" is optimized. This optimization can change the meaning of the expression. For example, the sequence "if a & b then ..." is not always the same as the sequence "x = a&b; if x then ...", even if the evaluation of "a" and "b" do not involve side effects. See the section on conditional statements.

EXP1 xor EXP2
EXP1 eqv EXP2

The value of the "xor" expression is the logical exclusive-or of EXP1 and EXP2. The value of the "eqv" expression is the logical complement of this value.

EXP ? EXP1, EXP2

The value is the value of EXP1 if EXP is non-zero, or the value of EXP2 if EXP is zero. EXP is optimized if it involves 'not', "&", or "%"; see the section on conditional statements.

valof STAT

This expression causes the statement STAT to be executed until a 'resultis EXP' statement is encountered or until control would otherwise pass to the statement following STAT. If a 'resultis EXP' is executed, EXP becomes the value of the 'valof STAT' expression. If execution of STAT terminates, the expression has a garbage value. The 'valof' expression is usually used as a function body; but it may be used anyplace an expression can be.

selecton EXP into

```
[    case CONST1:  EXP1
     ...
     case CONSTn:  EXPn
     default:  EXPØ
]
```

This expression is equivalent to

```
[    case CONST1:  resultis EXP1
     ...
     case CONSTn:  resultis EXPn
     default:  resultis EXPØ
]
```

That is, its value is EXPi if the value of EXP is CONSTi, or EXPØ if EXP is not equal to any of the CONSTs. If no 'default' label appears, the 'selecton' expression will have a garbage value in none of the cases is matched.

Statements

Assignment statements:

REF = EXP

The value of EXP is stored into the memory cell referenced by REF. See the section on memory references.

Multiple assignments:

REF1, ..., REFn = EXP1, ..., EXPn

This statement is equivalent to the sequence "REF1 = EXP1; ...; REFn = EXPn". The assignments are made left-to-right.

Routine calls:

EXP ( )
EXP(EXP1, EXP2, ..., EXPn)

A routine call differs from a function call only in that a routine call occurs in a context where a statement is expected, whereas a function call occurs in a context where an expression (a value) is expected. The calling sequence for routines is identical to that for functions.

Conditionals and iterative statements:

The evaluation of EXP in an 'if', 'unless', 'test', 'while', 'until', 'repeatwhile', or 'repeatuntil' statement is optimized if EXP involves 'not', '&', or "%". In general, EXP "succeeds" if it is non-zero, "fails" if it is 0. But 'EXP1&EXP2' is tested by first testing one of the EXPs; if it "fails", the &-expression "fails", and the other expression is not evaluated. Simialrly, in 'EXP1%EXP2', one of the EXPs is tested; if it "succeeds", 'EXP1%EXP2' succeeds. A 'not EXP' ",succeeds" if EXP "fails", and "fails" if EXP "succeeds".

This optimization has two significant consequences:

a) In a statement such as "if f(x) & g(x) do ...", it is not guaranteed that both functions will be executed; so any side-effects of "f" and "g" cannot be depended on.

b) The statement "if x & y do ..." is not necessarily equivalent to the sequence "z = x&y; if z do ...". For example, if "x" has the value 1 and "y" has the value 2, "z = #x&y" would assign the value 0 to "z", because "1&2" is zero; so "if z do ..."

will consider "z" to "fail". But both "x" and "y" are nonzero, so "if x&y do ..." will consider "x&y" to "succeed". In general, '&' should be used in conditional statements only when its operands are known to take on only the values 'true' (#177777) or 'false' (Ø). Note that this is the case for relations; so "if x ne Ø & y ne Ø" does the right thing.

```
if EXP do STAT
unless EXP do STAT
```

The 'if' statement executes STAT if EXP succeeds. The 'unless' statement executes STAT if EXP fails. The word 'do' may be replaced by the word 'then', but (unlike ALGOL) no 'else' clause is allowed; use the 'test' statement for two-armed conditionals. The 'do' or 'then' may be omitted if STAT appears on the same line as the 'if' or 'unless' clause, and if STAT is one of the following types of statements:

'if' 'unless' 'test' 'while' 'until' 'for'
'goto' 'return' 'resultis' 'switchon' 'break'
'loop' 'endcase'

```
test EXP then STAT1 or STAT2
test EXP ifso STAT1 first STAT2
text EXP ifnot STAT2 ifso STAT1
```

Each of the above 'test' statements executes STAT1 if EXP succeeds, or STAT2 if EXPfails. Both clauses must be present; use the 'if' statement or the 'unless' statement for one-armed conditionals. If 'then' and 'or' are used, they must appear in that sequence; the STAT following 'then' is the true branch. If 'ifso' and 'ifnot' are used, they may appear in either order; the STAT following 'ifso' is the true branch.

```
while EXP do STAT
until EXP do STAT
```

The 'while' statement executes STAT as long as EXP succeeds. The 'until' statement executes STAT as long as EXP fails. The test on EXP is done before the first execution of STAT. The word 'do' may be omitted in the same contexts as for the 'if' statement.

The 'while' statement is equivalent to:

"goto M; L: STAT; M: if EXP goto L"

The 'until' statement is equivalent to

"goto M; L: STAT; M: unless EXP goto L"

STAT repeatwhile EXP
STAT repeatuntil EXP

The 'repeatwhile' statement executes STAT as long as EXP succeeds. The 'repeatuntil' statement executes STAT as long as EXP fails. STAT is executed once before the text on EXP is done. STAT may be a single statement or a block.

The 'repeatwhile' statement is equivalent to:

"L: STAT; if EXP goto L"

The 'repeatuntil' statement is equivalent to:

"L: STAT; unless EXP goto L"

STAT repeat

The 'repeat' statement executes STAT repeatedly (until terminated by a 'break', 'return', 'resultis', or 'goto' statement). It is equivalent to:

"L:STAT; goto L"

for NAME = EXP1 to EXP2 by CONST do STAT

NAME is a legal variable name; EXP1 and EXP2 may be arbitrary expressions; "by CONST" may be missing (1 is assumed), but if present, it must be a constant expression. The 'for' statement is (logically) equivalent to the following block:

```
[   let NAME, lim, inc = EXP1, EXP2, CONST
    goto M
    L:  STAT
        NAME = NAME + inc
    M:  test inc ge 0
        ifso if NAME ge lim goto L
        ifnot if NAME le lim goto L
]
```

Several things about the 'for statement should be noted:

1) The controlled variable is implicitly declared as a new dynamic variable; it is defined only in STAT, and not accessible after the loop terminates.

2) EXP2 is evaluated only once, at the beginning of the 'for' statement.

3) As noted, CONST (if present) must be a constant expression. If it is negative, the termination test is reversed.

4) STAT is nott executed if the initial condition fails the termination test (like ALGOL, unlike FORTRAN).

5) STAT# is executed when the controlled variable is equal to the limit.

break
loop

These are single-word BCPL statements which are legal only in the context of an iterative statement. The effect of 'break' is to jump to the statement immediately following the smallest textually enclosing iterative statement. The effect of 'loop' is to jump to the point at which the next iteration starts: to the test in a 'while', 'until', 'repeatwhile', or 'repeatuntil' statement; to the increment of NAME in a 'for' statement; or to the beginning of a 'repeat' statement.

Labels:

NAME: STAT

Any BCPL statement may be labeled. A label is effectively a declaration of a static variable which is initialized with the address of the labeled statement. It differs from other declarations in that it does not implicitly start a new block. Instead, it is treated as if it appeared at the beginning of the smallest textually enclosing block. See the section on static declarations for details.

Goto:

goto EXP

A Nova JMP is done to 'rv EXPx'. (One way to think of this is that it is equivalent to "pc = rv EXP", where "pc" is the Nova's program counter.) So if "lab1" is a variable whose value is #1000, "goto lab1" will jump to absolute location #1000. If "lab2" has the value "101000, and absolute location #1000 contains #2000, "goto lab2" will jump to absolute location #2000.

Returns:

return
resultis EXP

These statements cause a return from the procedure in which they appear. 'return' is only legal in a routine body; 'resultis EXP' is only legal in a function body.

Switches:

switchon EXP into CASEBLOCK

CASEBLOCK is a BCPL block which contains labels of the form "case CONSTi:", where the CONSTi are constant expressions. CASEBLOCK may also contain a label of the form "default:". The effect of a 'switchon' statement is as follows: If the CASEBLOCK contains a 'case' label whose constant CONSTi is equal to the value of EXP, a jump is done to that label. If no CONSTi matches the value of EXP, a jump is done to the 'default' label if there is one, or to the statement immediately following the CASEBLOCK if there is no default label.

The appearance of a 'case' label does not terminate the preceding case. That is, in

```
switchon Char into
[    case $A:  x = 1
     case $B:  x = 2
     default:  x = 0
]
```

"x" will be Ø no matter what "Char" contains. The statements "x = 1" and "x = 2" should be followed by a jump to the end of the CASEBLOCK. The single-word BCPL statement 'endcase' would accomplish this.

Case labels are legal only in CASEBLOCKs, and not in any sub-blocks of a CASEBLOCK. In connection with this, recall that a declaration implicitly begins a new block. Therefore the sequence

```
switchon x into
    [   case Ø:   let temp = Ø
                  ...
        case 1:   ...
    ]
```

will cause the compiler to complain that "case 1:" does not appear in a CASEBLOCK. The code which uses "temp" must be enclosed in a block of its own which does not span other case labels.

Switches are implemented by grouping the case values into one or more value ranges in which listed values are fairly dense, and doing an indexed branch on each of these ranges. Case values which do not fall into these clusters are checked individually if all of the indexed branches fail.

endcase

This single-word statement is legal only within the scope of a 'switchon' statement. It causes a transfer to the end of the smallest enclosing 'switchon' statement.