Users' Reference to B

Ken Thompson

ABSTRACT

B is a computer language intended for recursive, primarily non-numeric applications typified by system programming. B has a small, unrestrictive syntax that is easy to compile. Because of the unusual freedom of expression and a rich set of operators, B programs are often quite compact.

This manual contains a concise definition of the language, sample programs, and instructions for using the PDP-11 version of B.

[*This is a rendition, after scanning, OCR, and editing, of an internal Bell Labs Technical Memorandum dated January 7, 1972. It is Ken's original manual for the B language on the PDP-11. An image of the original document is available in PDF, but it is big: just under 1MB.*

Nearby is CSTR #8, which is a report by Steve Johnson and Brian Kernighan describing the B implementation on Honeywell equipment.

The language being described was really the same, but it's interesting to look at the differences in description and environment. To describe the BNF syntax, Ken used a variant that depends on super- and subscripts to say "how many", which in the original were stacked above each other when they occurred simultaneously. This doesn't look too awful in the HTML rendering, but was probably better in the original.

*The other thing that is observable is the degree to which the GCOS version had to struggle to work in that environment. Its library description is full of odd concepts like "the AFT" the *SRC file, and other peculiarities.*

--DMR, July 1997]

Introduction

B is a computer language directly descendant from BCPL [1,2]. B 1s running at Murray Hill on the DEC PDP-11 computer under the UNIX-11 time sharing system [3,4]. B is good for recursive, non-numeric, machine independent applications, such as system and language work.

B, compared to BCPL, is syntactically rich in expressions and syntactically poor in statements. A look at the examples in section 9 of this document will give a flavor of the language.

B was designed and implemented by D. M. Ritchie and the author.

Syntax

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

1. The metacharacters < and > are removed. Literals are underlined [in typewriter font here -- DMR] to differentiate them from syntactic variables.
2. The metacharacter | is removed. Each syntactic alternative is placed on a separate line.
3. The metacharacters { and } denote syntactic grouping.
4. A syntactic group followed by numerical sub- and superscripts denote repetition of the group as follows:

$\{..\}_m$ m,m+1,...

$\{..\}_m^n$ m,m+1,...,n

Canonical Syntax

The syntax given in this section defines all the legal constructions in B without specifying the association rules. These are given later along with the semantic description of each construction.

program ::=
{definition}$_0$

definition ::=
name {[{constant}$_0^1$]}$_0^1$ {ival {, *ival*}$_0$}$_0^1$;
name ({name {, *name*}$_0$}$_0^1$) *statement*

ival ::=
constant
name

statement ::=
auto *name* {constant}$_0^1$ {, *name* {constant}$_0^1$}$_0$; *statement*
extrn *name* {, *name*}$_0$; *statement*
name : *statement*
case constant : *statement*
{ {statement}$_0$ }
if (*rvalue*) *statement* {else *statement*}$_0^1$

```
        while ( rvalue ) statement
        switch rvalue statement
        goto rvalue ;
        return {( rvalue )}0^1 ;
        {rvalue}0^1 ;

rvalue ::=
        ( rvalue )
        lvalue
        constant
        lvalue assign rvalue
        inc-dec lvalue
        lvalue inc-dec
        unary rvalue
        & lvalue
        rvalue binary rvalue
        rvalue ? rvalue : rvalue
        rvalue ( {rvalue {, rvalue}0 }0^1 )

assign ::=
        = {binary}0^1

inc-dec ::=
        ++
        --

unary ::=
        -
        !

binary ::=
        |
        &
        ==
        !=
        <
        <=
        >
        >=
        <<
        >>
        -
        +
        %
        *
        /

lvalue ::=
        name
        * rvalue
        rvalue [ rvalue ]

constant ::=
        {digit}1
        ' {char}1^2 '
        " {char}0 "
```

```
name ::=
        alpha {alpha-digit}0^7

alpha-digit ::=
        alpha
        digit
```

Comments and Character Sets

Comments are delimited as in PL/I by /* and */.

In general, B requires tokens to be separated by blanks, comments or newlines, however the compiler infers separators surrounding any of the characters `(){}[],;?:` or surrounding any maximal sequence of the characters `+-*/<>&|!`.

The character set used in B is ANSCII.

The syntactic variable 'alpha' is not defined. It represents the characters `A` to `Z`, `a` to `z`, _, and backspace.

The syntactic variable 'digit' is not defined. It represents the characters `0`, `1`, `2`, ... `9`.

The syntactic variable 'char' is not defined. It is essentially any character in the set plus the escape character `*` followed by another character to represent characters not easily represented in the set. The following escape sequences are currently defined:

```
*0      null
*e      end-of-file
*(      {
*)      }
*t      tab
**      *
*'      '
*"      "
*n      new line
```

All keywords in the language are only recognized in lower case. Keywords are reserved.

3.0 values and Lvalues

An rvalue is a binary bit pattern of a fixed length. On the PDP-11 it is 16 bits. Objects are rvalues of different kinds such as integers, labels, vectors and functions. The actual kind of object represented is called the type of the rvalue.

A B expression can be evaluated to yield an rvalue, but its type is undefined until the rvalue is

used in some context. It is then assumed to represent an object of the required type. For example, in the following function call

```
(b? f:g[i] )(1, x>1)
```

The expression (b?f:g[i]) is evaluated to yield an rvalue which is interpreted to be of type function. Whether f and g[i] are in fact functions is not checked. Similarly, b is assumed to be of type truth value, x to be type integer etc.

There is no check to insure that here are no type mismatches. Similarly, there are no wanted, or unwanted, type conversions.

An lvalue is a bit pattern representing a storage location containing an rvalue. An lvalue is a type in B. The unary operator * can be used to interpret an rvalue as an lvalue. Thus

```
*X
```

evaluates the expression x to yield an rvalue, which is then interpreted as an lvalue. If it is then used in an rvalue context, the application of * yields the rvalue therein stored. The operator * can be thought of as indirection.

The unary operator & can be used to interpret an lvalue as an rvalue. Thus

```
&x
```

evaluates the expression x as an lvalue. The application of & then yields the lvalue as an rvalue. The operator & can therefore be thought of as the address function.

The names lvalue and rvalue come from the assignment statement which requires an lvalue on the left and an rvalue on the right.

4.0 Expression Evaluation

Binding of expressions (lvalues and rvalues) is in the same order as the sub-sections of this section except as noted. Thus expressions referred to as operands of + (section 4.4) are expressions defined in sections 4.1 to 4.3. The binding of operators at the same level (left to right, right to left) is specified in each sub-section.

4.1 Primary Expressions

1. A name is an lvalue of one of three storage classes (automatic, external and internal).
2. A decimal constant is an rvalue. It consists of a digit between 1 and 9 followed by any number of digits between O and 9. The value of the constant should not exceed the maximum value that can be stored in an object.
4.

An octal constant is the same as a decimal constant except that it begins with a zero. It is then interpreted in base 8. Note that 09 (base 8) is legal and equal to 011. A character constant is represented by ' followed by one or two characters (possibly escaped) followed by another '. It has an rvalue equal to the value of the characters packed and right adjusted.

5. A string is any number of characters between " characters. The characters are packed into adjacent objects (lvalues sequential) and terminated with the character '*e'. The rvalue of the string is the lvalue of the object containing the first character. See section 8.0 for library functions used to manipulate strings in a machine independent fashion.

6. Any expression in () parentheses is a primary expression. Parentheses are used to alter order of binding.

7. A vector is a primary expression followed by any expression in [] brackets. The two expressions are evaluated to rvalues, added and the result is used as an lvalue. The primary expression can be thought of as a pointer to the base of a vector, while the bracketed expression can be thought of as the offset in the vector. Since E1[E2] is identical to *(E1+E2), and addition is commutative, the base of the vector and the offset in the vector can swap positions.

8. A function is a primary expression followed by any number of expressions in () parentheses separated by commas. The expressions In parentheses are evaluated (in an unspecified order) to rvalues and assigned to the function's parameters. The primary expression is evaluated to an rvalue (assumed to be type function). The function is then called. Each call is recursive at little cost in time or space.
Primary expressions are bound left to right.

4.2 Unary Operators

1. The rvalue (or indirection) prefix unary operator * is described in section 3.0. Its operand is evaluated to rvalue, and then used as an lvalue. In this manner, address arithmetic may be performed.

2. The lvalue (or address) prefix unary operator & is also described in section 3.0. Note that &*x is identically x, but *&x is only x if x is an lvalue.

3. The operand of the negate prefix unary operator - is interpreted as an integer rvalue. The result is an rvalue with opposite sign.

4. The NOT prefix unary operator ! takes an integer rvalue operand. The result is zero if the operand is non-zero. The result is one if the operand is zero.

5.

The increment ++ and decrement -- unary operators may be used either in prefix or postfix form. Either form requires an lvalue operand. The rvalue stored in the lvalue is either incremented or decremented by one. The result is the rvalue either before or after the operation depending on postfix or prefix notation respectively. Thus if x currently contains the rvalue 5, then ++x and x++ both change x to 6. The value of ++x is 6 while x++ is 5.
Similarly, --x and x-- store 4 in X. The former has rvalue result 4, the latter 5.

Unary operators are bound right to left. Thus -!x++ is bound -(!(x++)).

4.3 Multiplicative Operators

The multiplicative binary operators *, /, and %, expect rvalue integer operands. The result is also an integer.

1. The operator * denotes multiplication.
2. The operator / denotes division. The result is correct if the first operand is divisible by the second. If both operands are positive, the result is truncated toward zero. Otherwise the rounding is undefined, but never greater than one.
3. The operator % denotes modulo. If both operands are positive, the result is correct. It is undefined otherwise.

The multiplicative operators bind left to right.

4.4 Additive Operators

The binary operators + and - are add and subtract. The additive operators bind left to right.

Shift Operators

The binary operators << and >> are left and right shift respectively. The left rvalue operand is taken as a bit pattern. The right operand is taken as an integer shift count. The result is the bit pattern shifted by the shift count. Vacated bits are filled with zeros. The result is undefined if the shift count negative or larger than an object length. The shift operators bind left to right.

4.6 Relational Operators

The relational operators < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to) take integer rvalue operands. The result is one if the operands are in the given relation to one another. The result is zero otherwise.

4.7 Equality Operators

The equality operators == (equal to) and != (not equal to) perform similarly to the relation operators.

4.8 AND operator

The AND operator & takes operands as bit patterns. The result is the bit pattern that is the bit-wise AND of the operands. The operator binds and evaluates left to right.

The OR operator

The OR operator | performs exactly as AND, but the result is the bit-wise inclusive OR of the operands. The OR operator also binds and evaluates left to right.

4.10 Conditional Expression

Three rvalue expressions separated by ? and : form a conditional expression. The first expression (to the left of the ?) is evaluated. If the result is non-zero, the second expression is evaluated and the third ignored. If the value is zero, the second expression is ignored and the third is evaluated. The result is either the evaluation of the second or third expression.

Binding is right to left. Thus a?b:c?d:e is a?b:(c?d:e).

4.11 Assignment Operators

There are 16 assignment operators in B. All have the form

```
lvalue op rvalue
```

The assignment operator = merely evaluates the rvalue and stores the result in the lvalue. The assignment operators =|, =&, ===, =|=, =<, =<=, =>, =>=, =<<, =>>, =+, =-, =%, =*, and =/ perform a binary operation (see sections 4.3 to 4.9) between the rvalue stored in the assignment's lvalue and the assignment's rvalue. The result is then stored in the lvalue. The expression x=*10 is identical to x=x*10. Note that this is not x= *10. The result of an assignment is the rvalue. Assignments bind right to left, thus x=y=0 assigns zero to y, then x, and returns the rvalue zero.

5.0 Statements

Statements define program execution. Each statement is executed by the computer in sequence. There are, of course, statements to conditionally or unconditionally alter normal sequencing.

5.1 Compound Statement

A sequence of statements in {} braces is syntactically a single statement. This mechanism is provided so that where a single statement is expected, any number of statements can be placed.

5.1 Conditional Statement

A conditional statement has two forms. The first:

```
if(rvalue) statement1
```

evaluates the rvalue and executes $statement_1$, if the rvalue is non-zero. If the rvalue is zero, $statement_1$ is skipped. The second form:

```
if(rvalue) statement1 else statement2
```

is defined as follows in terms of the first form:

```
if(x=(rvalue)) statement1 if(!x) statement2
```

Thus, only one of the two statements is executed, depending on the value of rvalue. In the above example, x is not a real variable, but just a demonstration aid.

5.3 While Statement

The while statement has the form:

```
while(rvalue) statement
```

The execution is described in terms of the conditional and goto statements as follows:

```
x: if(rvalue) { statement goto x; ]
```

Thus the statement is executed repeatedly while the rvalue is non-zero. Again, x is a demonstration aid.

5.4 Switch Statement

The switch statement is the most complicated statement in B. The switch has the form:

```
switch rvalue statement1
```

Virtually always, $statement_1$ above is a compound statement. Each statement in $statement_1$ may be preceded by a case as follows:

```
case constant:
```

During execution, the rvalue is evaluated and compared to each case constant in undefined order. If a case constant is equal to the evaluated rvalue, control is passed to the statement following the case. If the rvalue matches none of the cases, $statement_1$ is skipped.

5.5 Goto Statement

The goto statement is as follows:

```
goto rvalue ;
```

The rvalue is expected to be of type label. Control is then passed to the corresponding label. Goto's cannot be executed to labels outside the currently executing function level.

5.6 Return Statement

The return statement is used in a function to return control to the caller of a function. The first form simply returns control.

```
return ;
```

The second form returns an rvalue for the execution of the function.

```
return ( rvalue ) ;
```

The caller of the function need not use the returned rvalue.

3.7 Rvalue Statement

Any rvalue followed by a semicolon is a statement. The two most common rvalue statements are assignment and function call.

5.8 Null Statement

A semicolon is a null statement causing no execution. It is used mainly to carry a label after the last executable statement in a compound statement. It sometimes has use to supply a null body to a while statement.

6.0 Declarations

Declarations in B specify storage class of variables. Such declarations also, in some circumstances, specify initialization.

There are three storage classes in B. Automatic storage is allocated for each function invocation. External storage is allocated before execution and is available to any and all functions. Internal storage is local to a function and is available only to that function, but is available to all invocations of that function.

6.1 External Declaration

The external declaration has the form:

```
extrn name1 , name2 ... ;
```

The external declaration specifies that each of the named variables is of the external storage class. The declaration must occur before the first use of each of the variables. Each of the variables must also be externally defined.

6/2 Automatic Declaration

The automatic declaration also constitutes a definition:

```
auto name1 {constant}0^1 , name2 {constant]0^1 ... ;
```

In absence of the constant, the automatic declaration defines the variable to be of class automatic. At the same time, storage is allocated for the variable. When an automatic declaration is followed by a constant, the automatic variable is also initialized to the base of an automatic vector of the size of the constant. The actual subscripts used to reference the vector range from zero to the value of the constant less one.

6.3 Internal Declaration

The first reference to a variable not declared as external or automatic constitutes an internal declaration, All internal variables not defined as labels are flagged as undefined within a function. Labels are defined and initialized as follows:

```
name :
```

7.0 External Definitions

A complete B program consists of a series of external definitions. Execution is started by the hidden sequence

```
main(); exit();
```

Thus, it is expected that one of the external definitions is a function definition of main. (Exit is a predefined library function. See section 8.0)

7.1 Simple Definition

The simple external definition allocates an external object and optionally initializes it:

```
name {ival , ival ...}0 ;
```

If the external object is defined with no initialization, it is initialized with zero. A single initialization with a constant initializes the external with the value of the constant. Initialization with a name initializes the external to the address of that name. Many such initializations may be accessed as a vector based at &name.

7.2 Vector Definitions

An external vector definition has the following form:

```
name [ {constant}0^1 ] {ival , ival ...}0 ;
```

The name is initialized with the lvalue of the base of an external vector. If the vector size is missing, zero is assumed. In either case, the vector is initialized with the list of initial values. Each initial value is either a constant or a name. A constant initial value initializes the vector element to the value of the constant. The name initializes the element to the address of the name. The actual size is the maximum of the given size and the number of initial values.

7.3 Function Definitions

Function definitions have the following form:

```
name ( arguments ) statement
```

The name is initialized to the rvalue of the function. The arguments consist of a list of names separated by commas. Each name defines an automatic lvalue that is assigned the rvalue of the corresponding function call actual parameters. The statement (often compound) defines the execution of the function when invoked.

8.0 Library Functions

There is a library of B functions maintained in the file /etc/libb.a. The following is a list of those functions currently in the library. See section II of [4] for complete descriptions of the functions marked with an *.

c = char(string, i);

The i-th character of the string is returned.

error = chdir(string) ;

The path name represented by the string becomes the current directory. A negative number returned indicates an error.

error = chmod(string, mode);

The file specified by the string has its mode changed to the mode argument. A negative number returned indicates an error. (*)

error = chown(string, owner);

The file specified by the string has its owner changed to the owner argument. A negative number returned indicates an error. (*)

error = close(file) ;

The open file specified by the file argument is closed. A negative number returned indicates an error. (*)

file = creat(string, mode);

The file specified by the string is either truncated or created in the mode specified depending on its prior existence. In both cases, the file is opened for writing and a file descriptor is returned. A negative number returned indicates an error. (*)

```
ctime(time, date);
```

The system time (60-ths of a second) represented in the two-word vector time is converted to a 16-character date in the 8-word vector date. The converted date has the following format: "Mmm dd hh:mm:ss".

```
execl(string, arg0, arg1, ..., 0);
```

The current process is replaced by the execution of the file specified by string. The arg-i strings are passed as arguments. A return indicates an error. (*)

```
execv(string, argv, count);
```

The current process is replaced by the execution of the file specified by string. The vector of strings of length count are passed as arguments. A return indicates an error. (*)

```
exit( ) ;
```

The current process is terminated. (*)

```
error = fork( ) ;
```

The current process splits into two. The child process is returned a zero. The parent process is returned the process ID of the child. A negative number returned indicates an error. (*)

```
error = fstat(file, status);
```

The i-node of the open file designated by file is put in the 20-word vector status. A negative number returned indicates an error. (*)

```
char = getchar( ) ;
```

The next character form the standard input file is returned. The character `*e' is returned for an end-of-file.

```
id = getuid();
```

The user-ID of the current process is returned. (*)

```
error = gtty(file, ttystat);
```

The teletype modes of the open file designated by file is returned in the 3-word vector ttstat. A negative number returned indicates an error. (*)

```
lchar(string, i, char);
```

The character char is stored in the i-th character of the string.

```
error = link(string1, string2);
```

The pathname specified by string2 is created such that it is a link to the existing file specified by string1. A negative number returned indicates an error. (*)

error = mkdir(string, mode);

The directory specified by the string is made to exist with the specified access mode. A negative number returned indicates an error. (*)

file = open(string, mode);

The file specified by the string is opened for reading if mode is zero, for writing if mode is not zero. The open file designator is returned. A negative number returned indicates an error. (*)

printf(format, argl, ...);

See section 9.3 below.

printn(number, base);

See section 9.1 below.

putchar(char) ;

The character char is written on the standard output file.

nread = read(file, buffer, count);

Count bytes are read into the vector buffer from the open file designated by file. The actual number of bytes read are returned. A negative number returned indicates an error. (*)

error = seek(filet offset, pointer);

The I/O pointer on the open file designated by file is set to the value of the designated pointer plus the offset. A pointer of zero designates the beginning of the file. A pointer of one designates the current I/O pointer. A pointer of two designates the end of the file. A negative number returned indicates an error. (*)

error = setuid(id);

The user-ID of the current process is set to id. A negative number returned indicates an error. (*)

error = stat(string, status);

The i-node of the file specified by the string is put in the 20-word vector status. A negative number returned indicates an error. (*)

error = stty(file, ttystat);

The teletype modes of the open file designated by file is set from the 3-word vector ttystat. A negative number returned indicates an error. (*)

time(timev);

The current system time is returned in the 2-word vector timev. (*)

error = unlink(string);

The link specified by the string is removed. A negative number returned indicates an

error. (*)

```
error = wait( );
```

The current process is suspended until one of its child processes terminates. At that time, the child's process-ID is returned. A negative number returned indicates an error. (*)

```
nwrite = write(file, buffer, count);
```

Count bytes are written out of the vector buffer on the open file designated by file. The actual number of bytes written are returned. A negative number returned indicates an error. (*)

Besides the functions available from the library, there is a predefined external vector named argv included with every program. The size of argv is argv[0]+1. The elements argv[1]...argv[argv[0]] are the parameter strings as passed by the system in the execution of the current process. (See shell in II of [4])

9.0 Examples

The examples appear exactly as given to B.

9.1

```
/* The following function will print a non-negative number, n, to
  the base b, where 2<=b<=10,  This routine uses the fact that
  in the ANSCII character set, the digits 0 to 9 have sequential
  code values.  */

printn(n,b) {
        extrn putchar;
        auto a;

        if(a=n/b) /* assignment, not test for equality */
                printn(a, b); /* recursive */
        putchar(n%b + '0');
}
```

9.2

```
/* The following program will calculate the constant e-2 to about
   4000 decimal digits, and print it 50 characters to the line in
```

```
   groups of 5 characters.  The method is simple output conversion
   of the expansion
     1/2! + 1/3! + ... = .111....
   where the bases of the digits are 2, 3, 4, . . . */

main() {
        extrn putchar, n, v;
        auto i, c, col, a;

        i = col = 0;
        while(i<n)
                v[i++] = 1;
        while(col<2*n) {
                a = n+1 ;
                c = i = 0;
                while (i<n) {
                        c =+ v[i] *10;
                        v[i++]  = c%a;
                        c =/ a--;
                }

                putchar(c+'0');
                if(!(++col%5))
                        putchar(col%50?' ': '*n');
        }
        putchar('*n*n');
}

v[2000];
n 2000;
```

9.3

```
/* The following function is a general formatting, printing, and
   conversion subroutine.  The first argument is a format string.
   Character sequences of the form `%x' are interpreted and cause
   conversion of type 'x' of the next argument, other character
   sequences are printed verbatim.   Thus

    printf("delta is %d*n", delta);

    will convert the variable delta to decimal (%d) and print the
    string with the converted form of delta in place of %d.   The
    conversions %d-decimal, %o-octal, *s-string and %c-character
    are allowed.

    This program calls upon the function `printn'. (see section
    9.1) */

printf(fmt, x1,x2,x3,x4,x5,x6,x7,x8,x9) {
        extrn printn, char, putchar;
        auto adx, x, c, i, j;

        i= 0;   /* fmt index */
```

```
        adx = &x1;      /* argument pointer */
loop :
        while((c=char(fmt,i++) ) != `%') {
                if(c == `*e')
                        return;
                putchar(c);
        }
        x = *adx++;
        switch c = char(fmt,i++) {

        case `d': /* decimal */
        case `o': /* octal */
                if(x < 0) {
                        x = -x ;
                        putchar('-');
                }
                printn(x, c=='o'?8:10);
                goto loop;

        case 'c' : /* char */
                putchar(x);
                goto loop;

        case 's': /* string */
                while(c=char(x, j++)) != '*e')
                        putchar(c);
                goto loop;
        }
        putchar('%') ;
        i--;
        adx--;
        goto loop;
}
```

10.0 Usage

Currently on UNIX, there is no B command. The B compiler phases must be executed piecemeal. The first phase turns a B source program into an intermediate language.

```
/etc/bc source interm
```

The next phase turns the intermediate language into assembler source, at which time the intermediate language can be removed.

```
/etc/ba interm asource
rm interm
```

The next phase assembles the assembler source into the object file a.out. After this the a.out file can be renamed and the assembler source file can be removed.

```
as asource
mv a.out object
rm asource
```

The last phase loads the various object files with the necessary libraries in the desired order.

```
ld object /etc/brt1 -lb /etc/bilib /etc/brt2
```

Now a.out contains the completely bound and loaded program and can be executed.

```
a.out
```

A canned sequence of shell commands exists invoked as follows:

```
sh /usr/b/rc x
```

It will compile, convert, assemble and load the file x.b into the executable file a.out.

12.0 Implementation and Debugging

A B program is implemented as a reverse Polish threaded code interpreter: The object code consists of a series of addresses of interpreter subroutines. Machine register 3 is dedicated as the interpreter program counter. Machine register 4 is dedicated as the interpreter display pointer. The display pointer points to the base of the current stack frame. The first word of each stack frame is a pointer to the previous stack frame (prior display pointer.) The second word in each frame is is the saved interpreter program counter (return point of the call creating the frame.) Automatic variables start at the third word of each frame. Machine register 5 is dedicated as the interpreter stack pointer. The machine stack pointer plays no role in the interpretation. An example source code segment, object code and interpreter subroutines follow:

```
        automatic = external + 100.;
        va; 4           / lvalue of first automatic on stack
        x; .external    / rvalue of external on stack
        c; 100.         / rvalue of constant on stack
        b12             / binary operator #12 (+)
        b1              / binary operator #1 (=)
        ...
va:
        mov     (r3)+,r0
        add     r4,r0           / dp+offset of automatic
        asr     r0              / lvalues are word addresses
        mov     r0,(r5)+
        jmp    *(r3)+           / linkage between subroutines
x:
        mov     *(r3)+,(r5)+
        jmp     *(r3)+
c:
        mov     (r3)+,(r5)+
        jmp     *(r3)+

b12:
        add     -(r5),-2(r5)
        jmp     *(r3)+

b1:
```

```
mov     -(r5),r0        / rvalue
mov     -(r5),r1        / lvalue
asl     r1              / now byte address
mov     r0,(r1)         / actual assignment
mov     r0,(r5)+        / = returns an lvalue
jmp     *(r3)+
```

The above code as compared to the obvious 3 instruction directly executed equivalent gives the approximate 5:1 speed and 2:1 space penalties one pays in using B.

The salient features for debugging are then:

1. Machine r4 is the display pointer and can be used to trace function calls and determine automatic variable values at each call.
2. Machine r3 is the current program counter and can be used to determine the current point of execution. All externals are globals with their variable names prefixed by `.'. Thus the debugger [4] can be used directly to give values of external variables.
4. All data lvalues are word addresses and therefore not directly examinable by the debugger. (See ` request in I of [4]) [I don't really know what this might refer to. It is probably some command in the very early Unix debugger -- DMR]

13.0 Nasties

This section describes the 'glitches' found in all languages, but rarely reported.

1. The compiler makes sense of certain expressions with operators in ambiguous cases (e.g. a+++b) but not others even in unambiguous cases (e.g. a+++++b).
2. The B assembler /etc/ba does not correctly handle all possible combinations of intermediate language. The symptom is undefined symbols in the assembly of the output from /etc/ba. This is rare.
3. The B interpreter /etc/bilib is really a library of threaded code segment. The following code segments have not yet been written:

```
b103    =&
b104    ===
b105    =!=
b106    =<=
b107    =<
b110    =>=
b111    =>
b120    =/
```

4.

Initialization of external variables with addresses of other externals is not possible due to a loader deficiency.

5. Since externals are implemented as globals with names preceded by '.', the external names `byte', `endif', `even' and `globl' conflict with assembler pseudooperations and should be avoided.

Diagnostics

Diagnostics consist of two letters, an optional name, and a source line number. Due to the free format of the source, the number might be high. The following is a list of the diagnostics.

```
error   name    meaninq

$)      --      {} imbalance
()      --      () imbalance
*/      --      /* */ imbalance
[]      --      [] imbalance
>c      --      case table overflow (fatal)
>e      --      expression stack overflow (fatal)
>i      --      label table overflow (fatal)
>s      --      symbol table overflow (fatal)
ex      --      expression syntax
lv      --      rvalue where lvalue expected
rd      name    name redeclaration
sx      keyword statement syntax
un      name    undefined name
xx      --      external syntax
```

[signature line]

References

1. Richards, M. The BCPL Reference Manual. Multics repository M0099.
2. Canaday, R.H. and Ritchie, D.M. Bell Laboratories BCPL. MM 69-1371/1373-7/12.
3. Ritchie, D.M. The UNIX Time Sharing System. MM 71-1273-4.
4. Thompson, K. and Ritchie, D.M. UNIX Programmer's Manual. Available by arrangement.

[*Two of these references--the one to Richards's early BCPL manual, and to the first edition of the UNIX Programmer's manual, are available under my main home page. The ultimate content of*

"The UNIX Time Sharing System evolved into the C.ACM paper, there (in further revised form) into the BSTJ paper available there as well.]