Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'bc.1p' command

$ man bc.1p

BC(1P) POSIX Programmer's Manual BC(1P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

bc ? arbitrary-precision arithmetic language

SYNOPSIS

bc [-l] [file...]

DESCRIPTION

The bc utility shall implement an arbitrary precision calculator. It shall take input from any files given, then read from the standard in? put. If the standard input and standard output to bc are attached to a terminal, the invocation of bc shall be considered to be interactive, causing behavioral constraints described in the following sections.

OPTIONS

The bc utility shall conform to the Base Definitions volume of POSIX.1?2017, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

-l (The letter ell.) Define the math functions and initialize scale to 20, instead of the default zero; see the EXTENDED DESCRIPTION section.

OPERANDS

The following operand shall be supported:

file A pathname of a text file containing bc program statements. After all files have been read, bc shall read the standard input.

STDIN

See the INPUT FILES section.

INPUT FILES

Input files shall be text files containing a sequence of comments, statements, and function definitions that shall be executed as they are read.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of bc:

LANG Provide a default value for the internationalization vari? ables that are unset or null. (See the Base Definitions vol? ume of POSIX.1?2017, Section 8.2, Internationalization Vari? ables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard er? ror.

NLSPATH Determine the location of message catalogs for the processing of LC_MESSAGES.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The output of the bc utility shall be controlled by the program read, and consist of zero or more lines containing the value of all executed expressions without assignments. The radix and precision of the output shall be controlled by the values of the obase and scale variables; see the EXTENDED DESCRIPTION section.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

Grammar

The grammar in this section and the lexical conventions in the follow? ing section shall together describe the syntax for bc programs. The general conventions for this style of grammar are described in Section 1.3, Grammar Conventions. A valid program can be represented as the non-terminal symbol program in the grammar. This formal syntax shall take precedence over the text syntax description.

```
%token    EOF NEWLINE STRING LETTER NUMBER
%token    MUL_OP
/*        '*', '/', '%'                        */
%token    ASSIGN_OP
/*        '=', '+=', '-=', '*=', '/=', '%=', '^=' */
%token    REL_OP
/*        '==', '<=', '>=', '!=', '<', '>'        */
%token    INCR_DECR
/*        '++', '--'                           */
%token    Define    Break    Quit    Length
/*        'define', 'break', 'quit', 'length'     */
%token    Return    For    If    While    Sqrt
/*        'return', 'for', 'if', 'while', 'sqrt'  */
%token    Scale    Ibase    Obase    Auto
/*        'scale', 'ibase', 'obase', 'auto'       */
%start    program
```

```
%%
program             : EOF
                    | input_item program
                    ;
input_item          : semicolon_list NEWLINE
                    | function
                    ;
semicolon_list      : /* empty */
                    | statement
                    | semicolon_list ';' statement
                    | semicolon_list ';'
                    ;
statement_list      : /* empty */
                    | statement
                    | statement_list NEWLINE
                    | statement_list NEWLINE statement
                    | statement_list ';'
                    | statement_list ';' statement
                    ;
statement           : expression
                    | STRING
                    | Break
                    | Quit
                    | Return
                    | Return '(' return_expression ')'
                    | For '(' expression ';'
                        relational_expression ';'
                        expression ')' statement
                    | If '(' relational_expression ')' statement
                    | While '(' relational_expression ')' statement
                    | '{' statement_list '}'
                    ;
function            : Define LETTER '(' opt_parameter_list ')'
```

```
                '{' NEWLINE opt_auto_define_list
                statement_list '}'
        ;
opt_parameter_list   : /* empty */
        | parameter_list
        ;
parameter_list       : LETTER
        | define_list ',' LETTER
        ;
opt_auto_define_list : /* empty */
        | Auto define_list NEWLINE
        | Auto define_list ';'
        ;
define_list          : LETTER
        | LETTER '[' ']'
        | define_list ',' LETTER
        | define_list ',' LETTER '[' ']'
        ;
opt_argument_list    : /* empty */
        | argument_list
        ;
argument_list        : expression
        | LETTER '[' ']' ',' argument_list
        ;
relational_expression : expression
        | expression REL_OP expression
        ;
return_expression    : /* empty */
        | expression
        ;
expression           : named_expression
        | NUMBER
        | '(' expression ')'
```

```
                | LETTER '(' opt_argument_list ')'
                | '-' expression
                | expression '+' expression
                | expression '-' expression
                | expression MUL_OP expression
                | expression '^' expression
                | INCR_DECR named_expression
                | named_expression INCR_DECR
                | named_expression ASSIGN_OP expression
                | Length '(' expression ')'
                | Sqrt '(' expression ')'
                | Scale '(' expression ')'
                ;
named_expression    : LETTER
                | LETTER '[' expression ']'
                | Scale
                | Ibase
                | Obase
                ;
```

Lexical Conventions in bc

The lexical conventions for bc programs, with respect to the preceding grammar, shall be as follows:

1. Except as noted, bc shall recognize the longest possible token or delimiter beginning at a given point.
2. A comment shall consist of any characters beginning with the two adjacent characters "/*" and terminated by the next occurrence of the two adjacent characters "*/". Comments shall have no effect except to delimit lexical tokens.
3. The <newline> shall be recognized as the token NEWLINE.
4. The token STRING shall represent a string constant; it shall con? sist of any characters beginning with the double-quote character ("") and terminated by another occurrence of the double-quote character. The value of the string is the sequence of all charac?

ters between, but not including, the two double-quote characters. All characters shall be taken literally from the input, and there is no way to specify a string containing a double-quote character. The length of the value of each string shall be limited to {BC_STRING_MAX} bytes.

5. A <blank> shall have no effect except as an ordinary character if it appears within a STRING token, or to delimit a lexical token other than STRING.

6. The combination of a <backslash> character immediately followed by a <newline> shall have no effect other than to delimit lexical to? kens with the following exceptions:

   * It shall be interpreted as the character sequence "\<newline>" in STRING tokens.

   * It shall be ignored as part of a multi-line NUMBER token.

7. The token NUMBER shall represent a numeric constant. It shall be recognized by the following grammar:

```
NUMBER  : integer
        | '.' integer
        | integer '.'
        | integer '.' integer
        ;
integer : digit
        | integer digit
        ;
digit   : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
        | 8 | 9 | A | B | C | D | E | F
        ;
```

8. The value of a NUMBER token shall be interpreted as a numeral in the base specified by the value of the internal register ibase (de? scribed below). Each of the digit characters shall have the value from 0 to 15 in the order listed here, and the <period> character shall represent the radix point. The behavior is undefined if dig? its greater than or equal to the value of ibase appear in the to?

ken. However, note the exception for single-digit values being as? signed to ibase and obase themselves, in Operations in bc.

9. The following keywords shall be recognized as tokens:

   auto ibase length return while
   break if obase scale
   define for quit sqrt

10. Any of the following characters occurring anywhere except within a keyword shall be recognized as the token LETTER:

    a b c d e f g h i j k l m n o p q r s t u v w x y z

11. The following single-character and two-character sequences shall be recognized as the token ASSIGN_OP:

    = += -= *= /= %= ^=

12. If an '=' character, as the beginning of a token, is followed by a '-' character with no intervening delimiter, the behavior is unde? fined.

13. The following single-characters shall be recognized as the token MUL_OP:

    * / %

14. The following single-character and two-character sequences shall be recognized as the token REL_OP:

    == <= >= != < >

15. The following two-character sequences shall be recognized as the token INCR_DECR:

    ++ --

16. The following single characters shall be recognized as tokens whose names are the character:

    <newline> ( ) , + - ; [ ] ^ { }

17. The token EOF is returned when the end of input is reached.

Operations in bc

There are three kinds of identifiers: ordinary identifiers, array iden? tifiers, and function identifiers. All three types consist of single lowercase letters. Array identifiers shall be followed by square brack? ets ("[]"). An array subscript is required except in an argument or

auto list. Arrays are singly dimensioned and can contain up to {BC_DIM_MAX} elements. Indexing shall begin at zero so an array is indexed from 0 to {BC_DIM_MAX}-1. Subscripts shall be truncated to integers. The application shall ensure that function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict.

The following table summarizes the rules for precedence and associativity of all operators. Operators on the same line shall have the same precedence; rows are in order of decreasing precedence.

Table: Operators in bc

Operator	Associativity
++, --	N/A
unary -	N/A
^	Right to left
*, /, %	Left to right
+, binary -	Left to right
=, +=, -=, *=, /=, %=, ^=	Right to left
==, <=, >=, !=, <, >	None

Each expression or named expression has a scale, which is the number of decimal digits that shall be maintained as the fractional portion of the expression.

Named expressions are places where values are stored. Named expressions shall be valid on the left side of an assignment. The value of a named expression shall be the value stored in the place named. Simple identifiers and array elements are named expressions; they have an initial value of zero and an initial scale of zero.

The internal registers scale, ibase, and obase are all named expressions. The scale of an expression consisting of the name of one of these registers shall be zero; values assigned to any of these registers are truncated to integers. The scale register shall contain a

global value used in computing the scale of expressions (as described below). The value of the register scale is limited to 0 ? scale ? {BC_SCALE_MAX} and shall have a default value of zero. The ibase and obase registers are the input and output number radix, respectively.

The value of ibase shall be limited to:

2 ? ibase ? 16

The value of obase shall be limited to:

2 ? obase ? {BC_BASE_MAX}

When either ibase or obase is assigned a single digit value from the list in Lexical Conventions in bc, the value shall be assumed in hexa? decimal. (For example, ibase=A sets to base ten, regardless of the cur? rent ibase value.) Otherwise, the behavior is undefined when digits greater than or equal to the value of ibase appear in the input. Both ibase and obase shall have initial values of 10.

Internal computations shall be conducted as if in decimal, regardless of the input and output bases, to the specified number of decimal dig? its. When an exact result is not achieved (for example, scale=0; 3.2/1), the result shall be truncated.

For all values of obase specified by this volume of POSIX.1?2017, bc shall output numeric values by performing each of the following steps in order:

1. If the value is less than zero, a <hyphen-minus> ('-') character shall be output.
2. One of the following is output, depending on the numerical value:
   * If the absolute value of the numerical value is greater than or equal to one, the integer portion of the value shall be output as a series of digits appropriate to obase (as described be? low), most significant digit first. The most significant non-zero digit shall be output next, followed by each successively less significant digit.
   * If the absolute value of the numerical value is less than one but greater than zero and the scale of the numerical value is greater than zero, it is unspecified whether the character 0 is

output.

* If the numerical value is zero, the character 0 shall be out? put.

3. If the scale of the value is greater than zero and the numeric value is not zero, a <period> character shall be output, followed by a series of digits appropriate to obase (as described below) representing the most significant portion of the fractional part of the value. If s represents the scale of the value being output, the number of digits output shall be s if obase is 10, less than or equal to s if obase is greater than 10, or greater than or equal to s if obase is less than 10. For obase values other than 10, this should be the number of digits needed to represent a precision of 10s.

For obase values from 2 to 16, valid digits are the first obase of the single characters:

0 1 2 3 4 5 6 7 8 9 A B C D E F

which represent the values zero to 15, inclusive, respectively.

For bases greater than 16, each digit shall be written as a separate multi-digit decimal number. Each digit except the most significant fractional digit shall be preceded by a single <space>. For bases from 17 to 100, bc shall write two-digit decimal numbers; for bases from 101 to 1000, three-digit decimal strings, and so on. For example, the deci? mal number 1024 in base 25 would be written as:

01 15 24

and in base 125, as:

008 024

Very large numbers shall be split across lines with 70 characters per line in the POSIX locale; other locales may split at different charac? ter boundaries. Lines that are continued shall end with a <backslash>.

A function call shall consist of a function name followed by parenthe? ses containing a <comma>-separated list of expressions, which are the function arguments. A whole array passed as an argument shall be speci? fied by the array name followed by empty square brackets. All function

arguments shall be passed by value. As a result, changes made to the formal parameters shall have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function shall be the value of the expression in the parentheses of the return statement or shall be zero if no expression is provided or if there is no return statement.

The result of sqrt(expression) shall be the square root of the expres? sion. The result shall be truncated in the least significant decimal place. The scale of the result shall be the scale of the expression or the value of scale, whichever is larger.

The result of length(expression) shall be the total number of signifi? cant decimal digits in the expression. The scale of the result shall be zero.

The result of scale(expression) shall be the scale of the expression. The scale of the result shall be zero.

A numeric constant shall be an expression. The scale shall be the num? ber of digits that follow the radix point in the input representing the constant, or zero if no radix point appears.

The sequence (expression) shall be an expression with the same value and scale as expression. The parentheses can be used to alter the nor? mal precedence.

The semantics of the unary and binary operators are as follows:

-expression

The result shall be the negative of the expression. The scale of the result shall be the scale of expression.

The unary increment and decrement operators shall not modify the scale of the named expression upon which they operate. The scale of the re? sult shall be the scale of that named expression.

++named-expression

The named expression shall be incremented by one. The result shall be the value of the named expression after incrementing.

--named-expression

The named expression shall be decremented by one. The result

shall be the value of the named expression after decrementing.

named-expression++

The named expression shall be incremented by one. The result shall be the value of the named expression before incrementing.

named-expression--

The named expression shall be decremented by one. The result shall be the value of the named expression before decrementing.

The exponentiation operator, <circumflex> ('^'), shall bind right to left.

expression^expression

The result shall be the first expression raised to the power of the second expression. If the second expression is not an inte? ger, the behavior is undefined. If a is the scale of the left expression and b is the absolute value of the right expression, the scale of the result shall be:

if b >= 0 min(a * b, max(scale, a)) if b < 0 scale

The multiplicative operators ('*', '/', '%') shall bind left to right.

expression*expression

The result shall be the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result shall be:

min(a+b,max(scale,a,b))

expression/expression

The result shall be the quotient of the two expressions. The scale of the result shall be the value of scale.

expression%expression

For expressions a and b, a%b shall be evaluated equivalent to the steps:

1. Compute a/b to current scale.
2. Use the result to compute:

a - (a / b) * b

to scale:

max(scale + scale(b), scale(a))

The scale of the result shall be:

max(scale + scale(b), scale(a))

When scale is zero, the '%' operator is the mathematical remain?

der operator.

The additive operators ('+', '-') shall bind left to right.

expression+expression

The result shall be the sum of the two expressions. The scale of the result shall be the maximum of the scales of the expressions.

expression-expression

The result shall be the difference of the two expressions. The scale of the result shall be the maximum of the scales of the ex?

pressions.

The assignment operators ('=', "+=", "-=", "*=", "/=", "%=", "^=") shall bind right to left.

named-expression=expression

This expression shall result in assigning the value of the ex?

pression on the right to the named expression on the left. The scale of both the named expression and the result shall be the scale of expression.

The compound assignment forms:

named-expression <operator>= expression

shall be equivalent to:

named-expression=named-expression <operator> expression

except that the named-expression shall be evaluated only once.

Unlike all other operators, the relational operators ('<', '>', "<=", ">=", "==", "!=") shall be only valid as the object of an if, while, or inside a for statement.

expression1<expression2

The relation shall be true if the value of expression1 is strictly less than the value of expression2.

expression1>expression2

The relation shall be true if the value of expression1 is strictly greater than the value of expression2.

expression1<=expression2

The relation shall be true if the value of expression1 is less than or equal to the value of expression2.

expression1>=expression2

The relation shall be true if the value of expression1 is greater than or equal to the value of expression2.

expression1==expression2

The relation shall be true if the values of expression1 and ex? pression2 are equal.

expression1!=expression2

The relation shall be true if the values of expression1 and ex? pression2 are unequal.

There are only two storage classes in bc: global and automatic (local). Only identifiers that are local to a function need be declared with the auto command. The arguments to a function shall be local to the func? tion. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as auto shall be allocated on entry to the function and released on returning from the function. They there? fore do not retain values between function calls. Auto arrays shall be specified by the array name followed by empty square brackets. On entry to a function, the old values of the names that appear as parameters and as automatic variables shall be pushed onto a stack. Until the function returns, reference to these names shall refer only to the new values.

References to any of these names from other functions that are called from this function also refer to the new value until one of those func? tions uses the same name for a local variable.

When a statement is an expression, unless the main operator is an as? signment, execution of the statement shall write the value of the ex? pression followed by a <newline>.

When a statement is a string, execution of the statement shall write the value of the string.

Statements separated by <semicolon> or <newline> characters shall be executed sequentially. In an interactive invocation of bc, each time a <newline> is read that satisfies the grammatical production:

input_item : semicolon_list NEWLINE

the sequential list of statements making up the semicolon_list shall be executed immediately and any output produced by that execution shall be written without any delay due to buffering.

In an if statement (if(relation) statement), the statement shall be ex? ecuted if the relation is true.

The while statement (while(relation) statement) implements a loop in which the relation is tested; each time the relation is true, the statement shall be executed and the relation retested. When the rela? tion is false, execution shall resume after statement.

A for statement(for(expression; relation; expression) statement) shall be the same as:

```
first-expression
while (relation) {
    statement
    last-expression
}
```

The application shall ensure that all three expressions are present.

The break statement shall cause termination of a for or while state? ment.

The auto statement (auto identifier [,identifier] ...) shall cause the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers shall be specified by following the array name by empty square brackets. The ap? plication shall ensure that the auto statement is the first statement in a function definition.

A define statement:

```
define LETTER ( opt_parameter_list ) {
    opt_auto_define_list
    statement_list
```

}

defines a function named LETTER. If a function named LETTER was previ?
ously defined, the define statement shall replace the previous defini?
tion. The expression:

LETTER (opt_argument_list)

shall invoke the function named LETTER. The behavior is undefined if the number of arguments in the invocation does not match the number of parameters in the definition. Functions shall be defined before they are invoked. A function shall be considered to be defined within its own body, so recursive calls are valid. The values of numeric constants within a function shall be interpreted in the base specified by the value of the ibase register when the function is invoked.

The return statements (return and return(expression)) shall cause ter?
mination of a function, popping of its auto variables, and specifica?
tion of the result of the function. The first form shall be equivalent to return(0). The value and scale of the result returned by the func?
tion shall be the value and scale of the expression returned.

The quit statement (quit) shall stop execution of a bc program at the point where the statement occurs in the input, even if it occurs in a function definition, or in an if, for, or while statement.

The following functions shall be defined when the -l option is speci?
fied:

s(expression)

Sine of argument in radians.

c(expression)

Cosine of argument in radians.

a(expression)

Arctangent of argument.

l(expression)

Natural logarithm of argument.

e(expression)

Exponential function of argument.

j(expression1, expression2)

Bessel function of expression2 of the first kind of integer order expression1.

The scale of the result returned by these functions shall be the value of the scale register at the time the function is invoked. The value of the scale register after these functions have completed their execution shall be the same value it had upon invocation. The behavior is unde? fined if any of these functions is invoked with an argument outside the domain of the mathematical function.

EXIT STATUS

The following exit values shall be returned:

0 All input files were processed successfully.

unspecified

An error occurred.

CONSEQUENCES OF ERRORS

If any file operand is specified and the named file cannot be accessed, bc shall write a diagnostic message to standard error and terminate without any further action.

In an interactive invocation of bc, the utility should print an error message and recover following any error in the input. In a non-interac? tive invocation of bc, invalid input causes undefined behavior.

The following sections are informative.

APPLICATION USAGE

Automatic variables in bc do not work in exactly the same way as in ei? ther C or PL/1.

For historical reasons, the exit status from bc cannot be relied upon to indicate that an error has occurred. Returning zero after an error is possible. Therefore, bc should be used primarily by interactive users (who can react to error messages) or by application programs that can somehow validate the answers returned as not including error mes? sages.

The bc utility always uses the <period> ('.') character to represent a radix point, regardless of any decimal-point character specified as part of the current locale. In languages like C or awk, the <period>

character is used in program source, so it can be portable and unam? biguous, while the locale-specific character is used in input and out? put. Because there is no distinction between source and input in bc, this arrangement would not be possible. Using the locale-specific char? acter in bc's input would introduce ambiguities into the language; con? sider the following example in a locale with a <comma> as the decimal-point character:

```
define f(a,b) {
    ...
}
...
f(1,2,3)
```

Because of such ambiguities, the <period> character is used in input. Having input follow different conventions from output would be confus? ing in either pipeline usage or interactive usage, so the <period> is also used in output.

EXAMPLES

In the shell, the following assigns an approximation of the first ten digits of '?' to the variable x:

```
x=$(printf "%s\n" 'scale = 10; 104348/33215' | bc)
```

The following bc program prints the same approximation of '?', with a label, to standard output:

```
scale = 10
"pi equals "
104348 / 33215
```

The following defines a function to compute an approximate value of the exponential function (note that such a function is predefined if the -l option is specified):

```
scale = 20
define e(x){
    auto a, b, c, i, s
    a = 1
    b = 1
```

```
        s = 1
        for (i = 1; 1 == 1; i++){
            a = a*x
            b = b*i
            c = a/b
            if (c == 0) {
                return(s)
            }
            s = s+c
        }
    }
```

The following prints approximate values of the exponential function of the first ten integers:

```
    for (i = 1; i <= 10; ++i) {
        e(i)
    }
```

RATIONALE

The bc utility is implemented historically as a front-end processor for dc; dc was not selected to be part of this volume of POSIX.1?2017 because bc was thought to have a more intuitive programmatic interface. Current implementations that implement bc using dc are expected to be compliant.

The exit status for error conditions has been left unspecified for several reasons:

* The bc utility is used in both interactive and non-interactive situations. Different exit codes may be appropriate for the two uses.
* It is unclear when a non-zero exit should be given; divide-by-zero, undefined functions, and syntax errors are all possibilities.
* It is not clear what utility the exit status has.
* In the 4.3 BSD, System V, and Ninth Edition implementations, bc works in conjunction with dc. The dc utility is the parent, bc is the child. This was done to cleanly terminate bc if dc aborted.

The decision to have bc exit upon encountering an inaccessible input

file is based on the belief that bc file1 file2 is used most often when at least file1 contains data/function declarations/initializations. Having bc continue with prerequisite files missing is probably not use? ful. There is no implication in the CONSEQUENCES OF ERRORS section that bc must check all its files for accessibility before opening any of them.

There was considerable debate on the appropriateness of the language accepted by bc. Several reviewers preferred to see either a pure sub? set of the C language or some changes to make the language more compat? ible with C. While the bc language has some obvious similarities to C, it has never claimed to be compatible with any version of C. An inter? preter for a subset of C might be a very worthwhile utility, and it could potentially make bc obsolete. However, no such utility is known in historical practice, and it was not within the scope of this volume of POSIX.1?2017 to define such a language and utility. If and when they are defined, it may be appropriate to include them in a future version of this standard. This left the following alternatives:

1. Exclude any calculator language from this volume of POSIX.1?2017.
   The consensus of the standard developers was that a simple program? matic calculator language is very useful for both applications and interactive users. The only arguments for excluding any calculator were that it would become obsolete if and when a C-compatible one emerged, or that the absence would encourage the development of such a C-compatible one. These arguments did not sufficiently ad? dress the needs of current application developers.
2. Standardize the historical dc, possibly with minor modifications.
   The consensus of the standard developers was that dc is a fundamen? tally less usable language and that that would be far too severe a penalty for avoiding the issue of being similar to but incompatible with C.
3. Standardize the historical bc, possibly with minor modifications.
   This was the approach taken. Most of the proponents of changing the language would not have been satisfied until most or all of the in?

compatibilities with C were resolved. Since most of the changes considered most desirable would break historical applications and require significant modification to historical implementations, al? most no modifications were made. The one significant modification that was made was the replacement of the historical bc assignment operators "=+", and so on, with the more modern "+=", and so on. The older versions are considered to be fundamentally flawed be? cause of the lexical ambiguity in uses like a=-1.

In order to permit implementations to deal with backwards-compati? bility as they see fit, the behavior of this one ambiguous con? struct was made undefined. (At least three implementations have been known to support this change already, so the degree of change involved should not be great.)

The '%' operator is the mathematical remainder operator when scale is zero. The behavior of this operator for other values of scale is from historical implementations of bc, and has been maintained for the sake of historical applications despite its non-intuitive nature.

Historical implementations permit setting ibase and obase to a broader range of values. This includes values less than 2, which were not seen as sufficiently useful to standardize. These implementations do not in? terpret input properly for values of ibase that are greater than 16. This is because numeric constants are recognized syntactically, rather than lexically, as described in this volume of POSIX.1?2017. They are built from lexical tokens of single hexadecimal digits and <period> characters. Since <blank> characters between tokens are not visible at the syntactic level, it is not possible to recognize the multi-digit ``digits'' used in the higher bases properly. The ability to recognize input in these bases was not considered useful enough to require modi? fying these implementations. Note that the recognition of numeric con? stants at the syntactic level is not a problem with conformance to this volume of POSIX.1?2017, as it does not impact the behavior of conform? ing applications (and correct bc programs). Historical implementations also accept input with all of the digits '0'-'9' and 'A'-'F' regardless

of the value of ibase; since digits with value greater than or equal to ibase are not really appropriate, the behavior when they appear is un? defined, except for the common case of:

```
ibase=8;
    /* Process in octal base. */
...
ibase=A
    /* Restore decimal base. */
```

In some historical implementations, if the expression to be written is an uninitialized array element, a leading <space> and/or up to four leading 0 characters may be output before the character zero. This be? havior is considered a bug; it is unlikely that any currently conform? ing application relies on:

```
echo 'b[3]' | bc
```

returning 00000 rather than 0.

Exact calculation of the number of fractional digits to output for a given value in a base other than 10 can be computationally expensive. Historical implementations use a faster approximation, and this is per? mitted. Note that the requirements apply only to values of obase that this volume of POSIX.1?2017 requires implementations to support (in particular, not to 1, 0, or negative bases, if an implementation sup? ports them as an extension).

Historical implementations of bc did not allow array parameters to be passed as the last parameter to a function. New implementations are en? couraged to remove this restriction even though it is not required by the grammar.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 1.3, Grammar Conventions, awk

The Base Definitions volume of POSIX.1?2017, Chapter 8, Environment Variables, Section 12.2, Utility Syntax Guidelines