bc – Arbitrary-Precision Arithmetic Language

Originally section 3 of chapter 4 (Execution Environment Utilities) of part 2 (Shell and Utilities) of Draft 11 of the IEEE P1003.2 standard (Information Technology – POSIX) – or P1003.2/D11.2 – 1991 September

1. Invocation

1.1. Command Line

bc [**-l**] *file ...*

The bc utility shall implement an arbitrary precision calculator and shall conform to the utility argument syntax guidelines described in 2.10.2.

Its input shall consist of the files named, with further input taken from the standard input. If the standard input and standard output to bc are attached to a terminal, the invocation of bc shall be considered to be *interactive*.

The following is the only option required

-l Define the math functions and initialize scale to 20, instead of the default zero (see 6 below),

and the following are the operands

file Pathnames of 0, 1, 2 or more text files containing bc program statements. After all *file*s have been read, bc shall read the standard input.

The bc utility shall exit with the value 0 if all input files were processed successfully, and with a non-zero value if an error occurred anywhere during the processing of the input files.

A diagnostic message will be issued if one of the files named in the command line cannot be found, and bc will end without carrying out any further action. The behavior of bc following an error encountered during the processing of a file is undefined. Following an error encountered during the reading of interactive input, bc will print a diagnostic and attempt to recover and process further input.

1.2. Input

Input files shall be text files containing sequences of comments, statements, and function definitions that will all be processed or executed as they are read.

1.3. Environment

The following environment variables shall affect the execution of bc

LANG	This variable shall determine the locale to use for the locale categories when both LC_ALL and the corresponding environment variable (beginning with LC_) do not specify a locale. See 2.6.
LC_ALL	This variable shall determine the locale to be used to override any values for locale categories specified by the settings of LANG or any environment variables beginning with LC_.
LC_CTYPE	This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).
LC_MESSAGES	This variable shall determine the language in which messages should be written.

1.4. Output

The output of the bc utility shall be controlled by the program read, and shall consist of zero or more lines containing the values of all expressions that were executed but not assigned. The radix and precision of the output shall be controlled by the values of the **obase** and **scale** variables. See 4.3.7.

Diagnostic output, as for most utilities, shall go to whatever file is designated *stderr*.

2. Extended Description

The syntax for BC is that described by the following grammar and lexical conventions. The description provided by this grammar takes precedence over any other description, in case of discrepancies.

2.1. bc Lexical Grammar

At the lexical level, the following tokens are recognized:

EOF	the end of file marker or **quit** keyword
EOL	the end of line marker or **;** (semicolon). An end-of-line marker is specifically required following the opening brace, **{**, of a function definition.
Str	character strings, given by the following syntax: **str** → **"***C****"**; with characters *C*: [^**"**].
X	identifiers – single letters in the ranges **a**-**z** only
num	numeric values, given by the following syntax **num** → *D*$^+$ [**.** *D**] | **.** *D*$^+$; with digits *D*: [**0**-**9**,**A**-**F**]
Op	any of +, **-**, *****, /, **%**, ^
assOp	any of =, +=, **-=**, ***=** , /=, **%=**, ^=
relOp	any of **==**, <=, >=, **!=**, <, >
incOp	any of ++, **--**

Also included as tokens as the keywords

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

and symbols

( ) , ; [ ] { }.

The processing at the lexical level is subject further to the following provisions:

(1) Wherever there is overlap ambiguity at a given point, the longest token recognized from that point wins out.

(2) The comments are exclusively of the form /* … */, and serve only as token separators.

(3) The actual character **'\n'** shall be recognized as an end-of-line marker.

(4) The character sequence enclosed between the double quotes in a **str** may include any character except the double-quote and may be of any length sufficient to fit within BC_STRING_MAX bytes.

(5) The space character **' '** shall serve only as a token separator, except within strings, where it shall be counted literally as a space character.

(6) A backslash character \ within a **str** shall be interpreted as a literal end-of-line character; within a **num**, it shall be ignored and the portions of the number immediately preceding and following it shall be concatenated. Outside a **str** or multi-line **num**, the combination of a backslash followed immediately by an end-of-line marker shall serve as nothing more than a token separator.

(7) The numeral represented by a **num** shall be taken in the base given by **ibase** in positional notation with **.** denoting the radix point. Other than the case of single-digit numerals being assigned to **ibase** or **obase**, the digits within a number must be less than the value given by **ibase**, or the behavior is undefined.

(8) The sequence =- with no intervening delimiter yields undefined behavior.

2.2. bc Syntax

A program consists of a list of 0, 1, 2 or more items, each representing a statement or the definition of a function. A function is defined with the keyword **define** following by a name, a list of 0, 1, 2 or more parameters enclosed in parentheses, and a sequence of declarations and statements enclosed in curly brackets. Further details on the syntax of declarations, statements and the expressions contained therein are given in the following grammar.

Program → *Item** **EOF**
Item → [*S*] **EOL**
Item → **define x (** [(*Dec* **,**)* **x**] **) { EOL** [**auto** *Dec* (**,** *Dec*)* **EOL**] ([*S*] **EOL**)* [*S*] **}**

Dec → **x**
Dec → **x [ ]**

S → *E*
S → **str**
S → **break**
S → **quit**

S → **return** [**(** [*E*] **)**]
S → **for (** *E* **;** *C* **;** *E* **)** *S*
S → **if (** *C* **)** *S*
S → **while (** *C* **)** *S*
S → **{** ([*S*] **EOL**)* [*S*] **}**

C → *E* **relOp** *E*

E → *L*
E → **num**
E → **(** *E* **)**
E → **x (** [*E* (**,** *E*)*] **)**
E → **-** *E*
E → *E* **Op** *E*
E → **incOp** *L*
E → *L* **incOp**
E → *L* **assOp** *E*
E → **length (** *E* **)**
E → **sqrt (** *E* **)**
E → **scale (** *E* **)**

L → **x**
L → **x [** *E* **]**
L → **scale**
L → **ibase**
L → **obase**

2.3. bc Operations

Identifiers, in bc, may denote values, arrays or functions. Each has the form, as described above, consisting only of a single letter. Arrays are further distinguished by the presence of square brackets **[ ]** following the identifier. When present as the argument to a function, or within a declaration, the brackets must follow the array name but contain no expression within.

Arrays may only have a single dimension, and have a maximum size of BC_DIM_MAX, with integer indices drawn from the interval [0, BC_DIM_MAX). Numeric subscripts are rounded down to integers, as need be.

Function names are distinguished by the presence of parentheses following the name and 0, 1 or 2 arguments contained within.

Therefore, these 3 types of identifiers occupy separate namespaces and the same name may be used, in a given context, once for each type of entity without conflict.

The following are the operators in bc, listed in decreasing order of precedence, with their associativity listed alongside. Operators on the same line have the same precedence.

Table 4.3 – bc Operator Precedence and Associativity

Operator	Associativity
++ --	--
unary -	--
^	leftward
* / %	rightward
+ binary -	rightward
= += -= *= /= %= ^=	leftward
== <= >= != < >	--

Associated with each expression is a *scale*, which is the number of decimal digits contained in its fractional part.

A distinguished subset of expressions, those designated *L* in the syntax above represent *lvalues*, which denote objects that refer to other values. Other expressions are designated *rvalues*. An *lvalue* is required as the left operand of an assignment operator (**assOp**) or the operand of an increment operator (**incOp**). In other contexts, an *lvalue* shall be used to denote the expression referred to by the object the *lvalue* denotes. All *lvalue*s are initialized to 0 with an initial scale of 0.

Other *lvalues* include the pre-defined variables **scale**, **ibase** and **obase**. Their respective scales are permanently set to 0. Any expression assigned to them will therefore be truncated to an integer value.

The variable **scale** is limited to an integer value in the interval [0, BC_SCALE_MAX), and has an initial value of 0. It represents the global scale used during the computation of expressions. Internal computations are all in base 10, and the value of **scale** pertains to the number of digits in this base only. Any necessary rounding, throughout computation, is done by truncation.

The variables **ibase** and **obase** represent respectively the input and output radix, used for reading and writing numbers. They are both limited to integer values, respectively in the intervals [2,16] and [2,BC_BASE_MAX] and have default values of 10.

A single digit assigned to either **ibase** or **obase** is taken with radix 16, independently of **ibase**. Thus, the statement **ibase=A** always assigns the value 10 to **ibase**. Otherwise, digits larger than **ibase** yield undefined behavior for the program containing them.

For all values of **obase** specified by this standard, the output, denoted here as *{N}*, of a numeric value *N*, and the output, denoted here *<N:s>* of the decimal part of *N* in a given precision *s*, shall comprise a sequence of digits as follows

(1) *{N}* = **-** *{-N}*, if $N < 0$.

(2) *{N}* = **d** *{N – d* **obase**^{k} *}*, if *d* **obase**$^k \le N <$ *(d+1)* **obase**^{k} for some integer $k \ge 0$. The numeral **d** corresponding to the value *d* is specified below.

(3) *{N}* = [**0**] **.** *<N:s>*, if $0 < N < 1$ and **scale** > 0. The presence of absence of the leading 0 is entirely at the discretion of the implementation. The number of digits, *s*, required in the output of the fractional part of a number shall be **scale**, itself, if **obase** is 10. Otherwise, it shall be whatever number of digits is required of a numeral in the radix given by **obase** to represent a base 10 precision of 10^{-s}.

(4) *{0}* = **0** and *{N}* = **0**, if $0 < N < 1$ and **scale** $= 0$.

(5) *<N:0>* = (empty string) and *<N:s+1>* = **d** *<e: s>* where d and e are respectively the integer and fractional parts of the product of **obase** and *N*. and **d** the numeral corresponding to the value *d*.

For integer values *d* in the interval [0, **obase**), when **obase** ≤ 16, the corresponding numeral **d** is drawn from the following list of characters

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

representing, respectively, the values zero through fifteen.

For **obase** > 16, the numeral **d** corresponding to the integer value *d* in the interval [0, **obase**) shall just be its base 10 number preceded by a space, unless it immediately follows the radix point. Each such numeral shall be the same size, prefixing them with 0's as need be. **Example**: the decimal number 1024, for **obase** = 25, would be written as

01 15 24 (i.e., space, 0, 1, space, 1, 5, space, 2, 4)

and for **obase** = 125, as

008 024 (i.e., space, 0, 0, 8, space, 0, 2, 4).

If necessary, a number shall be split onto separate lines, with 70 characters per line in the POSIX Locale, each line ending in a backslash. Other locals may split at different character boundaries.

A function call shall consist of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument shall be specified 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 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(***E***)** shall be the square root of the indicated number, *E*, truncated in the least significant decimal place. The scale of the result shall be the scale of *E*, or shall be set to the value of **scale**, if that is larger.

The result of **length(***E***)** shall, itself, have scale 0, and shall represent the total number of significant decimal digits in *E*.

The result of **scale(***E***)** shall, also, have scale 0, and shall represent the scale of the expression *E*.

A numeric constant is an expression, its scale is 0 if there is no radix point, otherwise its scale is the number of digits that follow the radix point in the input representing the constant.

The sequence **(***E***)**, where *E* is an expression, is an expression with the same value and scale as *E*. The parentheses can be used to alter the normal precedence.

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

-*A*	The additive inverse of *A*. Its scale is that of *A*.
++*L*, **--***L*	The value referred to by *L* is, respectively, increased or decreased by 1. The scale is not changed. The value of the expression is that of *L* after the change
*L***++**, *L***--**	The value referred to by *L* is, respectively, increased or decreased by 1. The scale is not changed. The value of the expression is that of *L* before the change
*A***^***B*	The value is *A* raised to tbe power *B*. If *B* is not an integer, the result is undefined. The scale of the result is scale, except that if $B \geq 0$ the scale may be no less than that of A, and no larger than the product of *B* and the scale of *A*.
*A****** *B*	The value is the product of *A* and *B*. The scale of the result is scale, except that it may be no smaller than either the scales of *A* or *B*, and no larger than the sum of their scales.
A/*B*	The value is the quotient of *A* and *B*. The scale of the result is scale.
*A***%***B*	The value is that obtained by $A - (A/B)*B$, except for the occurrence of side-effects in *B*.
*A***+***B*	The value is the sum of *A* and *B*. The scale is the larger of the scales of *A* and *B*.
*A***–***B*	The value is the difference of *A* and *B*. The scale is the larger of the scales of *A* and *B*.
L **=** *A*	Assigns the value *A* to *L*. The value of the expression is that of *L* after the change.
L op**=***A*	Equivalent to *L* = *L* op *A*, except that the expression *L* is evaluated only once.

The following are relational expressions and may only occur as the conditionals of an **if**, **while** or **for** statement.

A **<** *B*	Indicates that *A* is less than *B*.
A **>** *B*	Indicates that *A* is greater than *B*.
A **<=** *B*	Indicates that *A* is less than or equal to *B*.
A **>=** *B*	Indicates that *A* is greater than or equal to *B*.
A **==** *B*	Indicates that *A* is equal to *B*.
A **!=** *B*	Indicates that *A* is not equal to *B*.

There are only two storage classes in bc, global and automatic.

The parameters of a function and variables declared within a function with the **auto** command all have automatic storage. No two parameters, auto variables shall have the same name in a given function, and no parameter to a function shall have the same name as an auto variable. All auto variables are initially set to 0, each time the function is called. The initial value of a parameter to a function is that of the corresponding argument when the function is called.

Identifiers not declared within a function are global, all initially set to 0.

Variables have dynamic scope. An identifier inside the body of a function not declared there either as an auto variable or parameter refers to the same variable as it did in whatever called this function. Outside the body of a function, it refers to the corresponding global variable.

In particular, if a variable is declared in a function, it remains active until that function returns, even during calls to other functions, but is temporarily overridden if another function is called that has a variable of the same name within its either as an auto variable or parameter.

This is in marked contrast to a statically scoped language, like C, in which a variable inside a function body not declared as a local variable or parameter is considered always to be the global variable.

A statement of the form *E*, where *E* is an expression, evaluates the expression (carrying out any of its side-effects) and – if it is not an assignment or increment – print out the resulting value, followed by an end-of-line marker.

A statement of the form *S*, where *S* is a string, prints out the string.

Statements separated by **EOL** are to be executed in the order listed. If the statements appear outside the body of a function, they are executed as soon as an end-of-line marker is encountered.

A statement of the form **if (***C***)** *S* is executed by determining the condition *C* and, if true, executing *S*.

A statement of the form **while (***C***)** *S* is executed by determining the condition *C* and, if true, executing *S* and then repeating the execution of **while (***C***)** *S*. The following equivalence applies

if (*C***) { *S*; while (***C***)** *S* **}** ↔ **while (***C***)** *S*

In effect, the **while** statement is equivalent to the infinite **if** statement

if (*C***) { *S*; if (***C***) { *S*; if (***C***) {** … **}}}** ↔ **while (***C***)** *S*.

A statement of the form **for (** *A* **;** *C* **;** *B* **)** *S* is subject to the following equivalence

for (*A* **;** *C* **;** *B* **)** *S* ↔ **{***A***; while (***C***) { *S*;***B* **}}**

In contrast to C, all three expressions *A*, *B* and *C* must be present. In effect, it is equivalent to the following infinitary statement

{*A***; if (***C***) { *S*;***B*; **if (***C***) { *S*;***B*; **if (***C***) {** … **}}}}** ↔ **for (** *A* **;** *C* **;** *B* **)** *S*.

The **break** statement causes termination of a **for** or **while** statement.

A statement of the form **auto** D_1**,** …**,**D_n declares the indicated items as auto variables. Each declaration, D_1**,** …**,**D_n is either an identifier or an identifier followed by **[ ]**. A declaration of the form *a* **[ ]** declares the identifier *a* as an array variable. A declaration of the form *a*, without the following square brackets, declares the identifier *a* as an ordinary variable. An **auto** statement may only be used in a function definition, only one such statement may appear in the definition of any function, and it must precede all other statements in the body of the function.

A statement of the form **define** *f***(***D***,**…**,***D***) {** *S*…*S* **}** defines a function named by the identifier *f* with 0, 1, 2 or more parameters enclosed within parentheses, following by a sequence of statements between curly brackets. A function definition overrides any other definition previously issued to the same identifier *f*. The **define** statement may not be issued inside the body of a function.

The expression *f***(***E***,**…**,***E***)** invokes the function named by the identifier *f*. The result is undefined if the number of arguments appearing within the parentheses differs from number of parameters listed in the (most recently issued) declaration of the function. The values of the numeric constants inside the function are to be interpreted subject to the base given by **ibase** at the time the function was called.

The **define** statement also declares the function. The declaration is considered issued immediately following the listing of the parameters, whereas the definition is issued only following the listing of the statements in the body of the function.

A function must be declared before it is invoked. Since it is declared within its own statement body, a function may recursively invoke itself.

The statements **return** *A* may only be issued within the body of a function. Their execution brings to an end the invocation of the function containing them. The statements **return** and **return ( )** are both equivalent to **return 0**.

A function invocation shall have, as its value and scale, those of the expression in the first **return** statement encountered during the execution of the statements in the body of the function.

The **quit** statement brings to an immediate end the execution of bc, as soon as it is encountered. If the statement is encountered during the definition of a function, then bc ends in mid-definition. If it occurs inside a if, while or for statement, bc ends regardless of whether the body of the statement is executed or not.

When the –l option is specified, the following functions shall also be defined

s(*x***)**	The sine of *x*, with *x* in radians
c(*x***)**	The cosine of *x*, with *x* in radians
a(*x***)**	The arctangent, in radians, of *x*
l(*x***)**	The natural logarithm of *x*
e(*x***)**	The exponential of *x*
j(*n***,***x***)**	the Bessel function of integer order *n* of *x*.

The scale of the result of the invocation of any of these functions is the value of **scale** at the time the function is called. The behavior of the function is undefined if it is called with arguments that lie outside the domain of the mathematical function it represents. The logarithm of 0 or negative numbers, and the Bessel function of non-integer orders are not defined.

3. Rationale

(This subclause is not a part of P1003.2)

Examples, Usage

This description is based on *BC--An Arbitrary Precision Desk-Calculator Language* by Lorinda Cherry and Robert Morris, in the BSD User Manual {B28}.

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

In the shell, the following assigns an approximation of the first ten digits of π to the variable *x*

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

The following bc program prints the same approximation of π, 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
    }
}
```

In C-BC, a corresponding definition may look like this

```
scale = 20
define number e(number x) {
 number xn, nf, n, dy, y;
 for (xn = nf = 1, y = n = 0; (dy = xn/nf) != 0; xn *= x, nf *= ++n, y += dy) ;
 return y;
}
```

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

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

In C-BC, one may write out the formatted list

```
 for (i = 0; i < 10; ) <- i++, ": ", e(i), "\n"
```

History of Decisions Made

The bc utility is traditionally implemented as a front-end processor for dc; dc was not selected to be part of the standard 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 been left unspecified for several reasons

(1) The bc utility is used in both interactive and noninteractive situations. Different exit codes may be appropriate for the two uses.

(2) It is unclear when a nonzero exit should be given; divide-by-zero, undefined functions, and syntax errors are all possibilities.

(3) It is not clear what utility the exit status has.

(4) In the 4.3BSD, System V, and Ninth Edition implementations, bc works in conjunction with dc. dc 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 *file*$_1$ *file*$_2$ is used most often when at least *file*$_1$ contains data/function declarations/initializations. Having bc continue with prerequisite files missing is probably not useful. There is no implication in the Consequences of Errors subclause 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 members of the balloting group preferred to see either a pure subset of the C language or some changes to make the language more compatible 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 interpreter 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 existing practice, and it was not within the scope of POSIX.2 to define such a language and utility. If and when they are defined, it may be appropriate to include them in a future revision of this standard. This left the following alternatives:

(1) *Exclude any calculator language from the standard*. The consensus of the working group was that a simple programmatic calculator language is very useful. Also, an interactive version of such a calculator would be very important for the POSIX.2a revision. 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 address the needs of current application writers.

(2) *Standardize the existing dc, possibly with minor modifications*. The consensus of the working group was that dc is a fundamentally 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 existing 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 incompatibilities with C were resolved. Since most of the changes considered most desirable would break existing applications and require significant modification to existing implementations, almost no modifications were made. The one significant modification that was made was the replacement of the traditional bc's assignment operators =+ et al. with the more modern += et al. The older versions are considered to be fundamentally flawed because of the lexical ambiguity in uses like **a=-1**. In order to permit implementations to deal with backward compatibility as they see fit, the behavior of this one ambiguous construct 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 traditional implementations of bc, and has been maintained for the sake of existing applications despite its nonintuitive nature.

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 unambiguous, while the locale-specific character is used in input and output. Because there is no distinction between source and input in bc, this arrangement would not be possible. Using the locale-specific character in bc's input would introduce ambiguities into the language; consider 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 confusing in either pipeline usage or interactive usage, so period is also used in output.

Traditional 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 interpret input properly for values of ibase outside greater than 16. This is because numeric constants are recognized syntactically, rather than lexically, as described in the standard. They are built from lexical tokens of single hexadecimal digits and periods. Since <blank>s between tokens are not visible at the syntactic level, it is not possible to properly recognize the multidigit ``digits" used in the higher bases. The ability to recognize input in these bases was not considered useful enough to require modifying these implementations. Note that the recognition of numeric constants at the syntactic level is not a problem with conformance to the standard, as it does not impact the behavior of portable applications (and correct bc programs). Traditional 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 undefined, 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> character and/or up to four leading 0 characters may be output before the character zero. This behavior is considered a bug; it is unlikely that any currently portable 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. Traditional implementations use a faster approximation, and this is permitted. Note that the requirements apply only to values of obase that the standard requires implementations to support (in particular, not to 1, 0, or negative bases, if an implementation supports them as an extension).