NAME

bc - arbitrary-precision decimal arithmetic language and calculator

SYNOPSIS

bc [**-cCghilPqRsvVw**] [**--digit-clamp**] [**--no-digit-clamp**] [**--global-stacks**] [**--help**] [**--interactive**] [**--mathlib**] [**--no-prompt**] [**--no-read-prompt**] [**--quiet**] [**--standard**] [**--warn**] [**--version**] [**-e** *expr*] [**--expression**=*expr*...] [**-f** *file*...] [**--file**=*file*...] [*file*...] [**-I** *ibase*] [**--ibase**=*ibase*] [**-O** *obase*] [**--obase**=*obase*] [**-S** *scale*] [**--scale**=*scale*] [**-E** *seed*] [**--seed**=*seed*]

DESCRIPTION

bc(1) is an interactive processor for a language first standardized in 1991 by POSIX. (See the **STANDARDS** section.) The language provides unlimited precision decimal arithmetic and is somewhat C-like, but there are differences. Such differences will be noted in this document.

After parsing and handling options, this bc(1) reads any files given on the command line and executes them before reading from **stdin**.

This bc(1) is a drop-in replacement for *any* bc(1), including (and especially) the GNU bc(1). It also has many extensions and extra features beyond other implementations.

Note: If running this bc(1) on *any* script meant for another bc(1) gives a parse error, it is probably because a word this bc(1) reserves as a keyword is used as the name of a function, variable, or array. To fix that, use the command-line option **-r** *keyword*, where *keyword* is the keyword that is used as a name in the script. For more information, see the **OPTIONS** section.

If parsing scripts meant for other bc(1) implementations still does not work, that is a bug and should be reported. See the **BUGS** section.

OPTIONS

The following are the options that bc(1) accepts.

-C, **--no-digit-clamp**

Disables clamping of digits greater than or equal to the current **ibase** when parsing numbers.

This means that the value added to a number from a digit is always that digit's value multiplied by the value of ibase raised to the power of the digit's position, which starts from 0 at the least significant digit.

If this and/or the **-c** or **--digit-clamp** options are given multiple times, the last one given is used.

This option overrides the **BC_DIGIT_CLAMP** environment variable (see the **ENVIRONMENT VARIABLES** section) and the default, which can be queried with the **-h** or **--help** options.

This is a **non-portable extension**.

-c, **--digit-clamp**

Enables clamping of digits greater than or equal to the current **ibase** when parsing numbers.

This means that digits that the value added to a number from a digit that is greater than or equal to the ibase is the value of ibase minus 1 all multiplied by the value of ibase raised to the power of the digit's position, which starts from 0 at the least significant digit.

If this and/or the **-C** or **--no-digit-clamp** options are given multiple times, the last one given is used.

This option overrides the **BC_DIGIT_CLAMP** environment variable (see the **ENVIRONMENT VARIABLES** section) and the default, which can be queried with the **-h** or **--help** options.

This is a **non-portable extension**.

-E *seed*, **--seed**=*seed*

Sets the builtin variable **seed** to the value *seed* assuming that *seed* is in base 10. It is a fatal error if *seed* is not a valid number.

If multiple instances of this option are given, the last is used.

This is a **non-portable extension**.

-e *expr*, **--expression**=*expr*

Evaluates *expr*. If multiple expressions are given, they are evaluated in order. If files are given as well (see the **-f** and **--file** options), the expressions and files are evaluated in the order given. This means that if a file is given before an expression, the file is read in and evaluated first.

If this option is given on the command-line (i.e., not in **BC_ENV_ARGS**, see the **ENVIRONMENT VARIABLES** section), then after processing all expressions and files, bc(1) will exit, unless **-** (**stdin**) was given as an argument at least once to **-f** or **--file**, whether on the command-line or in **BC_ENV_ARGS**. However, if any other **-e**, **--expression**, **-f**, or **--file** arguments are given after **-f-** or equivalent is given, bc(1) will give a fatal error and exit.

This is a **non-portable extension**.

-f *file*, **--file**=*file*

Reads in *file* and evaluates it, line by line, as though it were read through **stdin**. If expressions are also given (see the **-e** and **--expression** options), the expressions are evaluated in the order given.

If this option is given on the command-line (i.e., not in **BC_ENV_ARGS**, see the **ENVIRONMENT VARIABLES** section), then after processing all expressions and files, bc(1) will exit, unless **-** (**stdin**) was given as an argument at least once to **-f** or **--file**. However, if any other **-e**, **--expression**, **-f**, or **--file** arguments are given after **-f-** or equivalent is given, bc(1) will give a fatal error and exit.

This is a **non-portable extension**.

-g, **--global-stacks**

Turns the globals **ibase**, **obase**, **scale**, and **seed** into stacks.

This has the effect that a copy of the current value of all four are pushed onto a stack for every function call, as well as popped when every function returns. This means that functions can assign to any and all of those globals without worrying that the change will affect other functions. Thus, a hypothetical function named **output(x,b)** that simply printed **x** in base **b** could be written like this:

```
define void output(x, b) {
  obase=b
  x
}
```

instead of like this:

```
define void output(x, b) {
  auto c
  c=obase
  obase=b
  x
  obase=c
}
```

This makes writing functions much easier.

(**Note**: the function **output(x,b)** exists in the extended math library. See the **LIBRARY** section.)

However, since using this flag means that functions cannot set **ibase**, **obase**, **scale**, or **seed** globally, functions that are made to do so cannot work anymore. There are two possible use cases for that, and each has a solution.

First, if a function is called on startup to turn bc(1) into a number converter, it is possible to replace that capability with various shell aliases. Examples:

```
alias d2o="bc -e ibase=A -e obase=8"
alias h2b="bc -e ibase=G -e obase=2"
```

Second, if the purpose of a function is to set **ibase**, **obase**, **scale**, or **seed** globally for any other purpose, it could be split into one to four functions (based on how many globals it sets) and each of those functions could return the desired value for a global.

For functions that set **seed**, the value assigned to **seed** is not propagated to parent functions. This means that the sequence of pseudo-random numbers that they see will not be the same sequence of pseudo-random numbers that any parent sees. This is only the case once **seed** has been set.

If a function desires to not affect the sequence of pseudo-random numbers of its parents, but wants to use the same **seed**, it can use the following line:

```
seed = seed
```

If the behavior of this option is desired for every run of bc(1), then users could make sure to define **BC_ENV_ARGS** and include this option (see the **ENVIRONMENT VARIABLES** section for more details).

If **-s**, **-w**, or any equivalents are used, this option is ignored.

This is a **non-portable extension**.

-h, **--help**

Prints a usage message and exits.

-I *ibase*, **--ibase**=*ibase*

Sets the builtin variable **ibase** to the value *ibase* assuming that *ibase* is in base 10. It is a fatal error if *ibase* is not a valid number.

If multiple instances of this option are given, the last is used.

This is a **non-portable extension**.

-i, **--interactive**

Forces interactive mode. (See the **INTERACTIVE MODE** section.)

This is a **non-portable extension**.

-L, **--no-line-length**

Disables line length checking and prints numbers without backslashes and newlines. In other words, this option sets **BC_LINE_LENGTH** to **0** (see the **ENVIRONMENT VARIABLES** section).

This is a **non-portable extension**.

-l, **--mathlib**

Sets **scale** (see the **SYNTAX** section) to **20** and loads the included math library and the extended math library before running any code, including any expressions or files specified on the command line.

To learn what is in the libraries, see the **LIBRARY** section.

-O *obase*, **--obase**=*obase*

Sets the builtin variable **obase** to the value *obase* assuming that *obase* is in base 10. It is a fatal error if *obase* is not a valid number.

If multiple instances of this option are given, the last is used.

This is a **non-portable extension**.

-P, **--no-prompt**

Disables the prompt in TTY mode. (The prompt is only enabled in TTY mode. See the **TTY MODE** section.) This is mostly for those users that do not want a prompt or are not used to having them in bc(1). Most of those users would want to put this option in **BC_ENV_ARGS** (see the **ENVIRONMENT VARIABLES** section).

These options override the **BC_PROMPT** and **BC_TTY_MODE** environment variables (see the **ENVIRONMENT VARIABLES** section).

This is a **non-portable extension**.

-q, **--quiet**

This option is for compatibility with the GNU bc(1) (https://www.gnu.org/software/bc/); it is a no-op. Without this option, GNU bc(1) prints a copyright header. This bc(1) only prints the copyright header if one or more of the **-v**, **-V**, or **--version** options are given unless the **BC_BANNER** environment variable is set and contains a non-zero integer or if this bc(1) was built with the header displayed by default. If *any* of that is the case, then this option *does* prevent bc(1) from printing the header.

This is a **non-portable extension**.

-R, **--no-read-prompt**

Disables the read prompt in TTY mode. (The read prompt is only enabled in TTY mode. See the **TTY MODE** section.) This is mostly for those users that do not want a read prompt or are not used to having them in bc(1). Most of those users would want to put this option in **BC_ENV_ARGS** (see the **ENVIRONMENT VARIABLES** section). This option is also useful in hash bang lines of bc(1) scripts that prompt for user input.

This option does not disable the regular prompt because the read prompt is only used when the **read()** built-in function is called.

These options *do* override the **BC_PROMPT** and **BC_TTY_MODE** environment variables (see the **ENVIRONMENT VARIABLES** section), but only for the read prompt.

This is a **non-portable extension**.

-r *keyword*, **--redefine**=*keyword*

Redefines *keyword* in order to allow it to be used as a function, variable, or array name. This is useful when this bc(1) gives parse errors when parsing scripts meant for other bc(1) implementations.

The keywords this bc(1) allows to be redefined are:

- **abs**

- **asciify**

- **continue**
- **divmod**
- **else**
- **halt**
- **irand**
- **last**
- **limits**
- **maxibase**
- **maxobase**
- **maxrand**
- **maxscale**
- **modexp**
- **print**
- **rand**
- **read**
- **seed**
- **stream**

If any of those keywords are used as a function, variable, or array name in a script, use this option with the keyword as the argument. If multiple are used, use this option for all of them; it can be used multiple times.

Keywords are *not* redefined when parsing the builtin math library (see the **LIBRARY** section).

It is a fatal error to redefine keywords mandated by the POSIX standard (see the **STANDARDS** section). It is a fatal error to attempt to redefine words that this bc(1) does not reserve as keywords.

-S *scale*, **--scale**=*scale*
Sets the builtin variable **scale** to the value *scale* assuming that *scale* is in base 10. It is a fatal error if *scale* is not a valid number.

If multiple instances of this option are given, the last is used.

This is a **non-portable extension**.

-s, **--standard**
Process exactly the language defined by the standard (see the **STANDARDS** section) and error if any extensions are used.

This is a **non-portable extension**.

-v, **-V**, **--version**
Print the version information (copyright header) and exits.

This is a **non-portable extension**.

-w, **--warn**
Like **-s** and **--standard**, except that warnings (and not errors) are printed for non-standard extensions and execution continues normally.

This is a **non-portable extension**.

-z, **--leading-zeroes**
Makes bc(1) print all numbers greater than **-1** and less than **1**, and not equal to **0**, with a leading zero.

This can be set for individual numbers with the **plz(x)**, **plznl(x)**, **pnlz(x)**, and **pnlznl(x)** functions in the extended math library (see the **LIBRARY** section).

This is a **non-portable extension**.

All long options are **non-portable extensions**.

STDIN

If no files or expressions are given by the **-f**, **--file**, **-e**, or **--expression** options, then bc(1) reads from **stdin**.

However, there are a few caveats to this.

First, **stdin** is evaluated a line at a time. The only exception to this is if the parse cannot complete. That means that starting a string without ending it or starting a function, **if** statement, or loop without ending it will also cause bc(1) to not execute.

Second, after an **if** statement, bc(1) doesn't know if an **else** statement will follow, so it will not execute until it knows there will not be an **else** statement.

STDOUT

Any non-error output is written to **stdout**. In addition, if history (see the **HISTORY** section) and the prompt (see the **TTY MODE** section) are enabled, both are output to **stdout**.

Note: Unlike other bc(1) implementations, this bc(1) will issue a fatal error (see the **EXIT STATUS** section) if it cannot write to **stdout**, so if **stdout** is closed, as in **bc >&-**, it will quit with an error. This is done so that bc(1) can report problems when **stdout** is redirected to a file.

If there are scripts that depend on the behavior of other bc(1) implementations, it is recommended that those scripts be changed to redirect **stdout** to **/dev/null**.

STDERR

Any error output is written to **stderr**.

Note: Unlike other bc(1) implementations, this bc(1) will issue a fatal error (see the **EXIT STATUS** section) if it cannot write to **stderr**, so if **stderr** is closed, as in **bc 2>&-**, it will quit with an error. This is done so that bc(1) can exit with an error code when **stderr** is redirected to a file.

If there are scripts that depend on the behavior of other bc(1) implementations, it is recommended that those scripts be changed to redirect **stderr** to **/dev/null**.

SYNTAX

The syntax for bc(1) programs is mostly C-like, with some differences. This bc(1) follows the POSIX standard (see the **STANDARDS** section), which is a much more thorough resource for the language this bc(1) accepts. This section is meant to be a summary and a listing of all the extensions to the standard.

In the sections below, **E** means expression, **S** means statement, and **I** means identifier.

Identifiers (**I**) start with a lowercase letter and can be followed by any number (up to **BC_NAME_MAX-1**) of lowercase letters (**a-z**), digits (**0-9**), and underscores (_). The regex is **[a-z][a-z0-9_]***. Identifiers with more than one character (letter) are a **non-portable extension**.

ibase is a global variable determining how to interpret constant numbers. It is the "input" base, or the number base used for interpreting input numbers. **ibase** is initially **10**. If the **-s** (**--standard**) and **-w** (**--warn**) flags were not given on the command line, the max allowable value for **ibase** is **36**. Otherwise, it is **16**. The min allowable value for **ibase** is **2**. The max allowable value for **ibase** can be queried in bc(1) programs with the **maxibase()** built-in function.

obase is a global variable determining how to output results. It is the "output" base, or the number base used for outputting numbers. **obase** is initially **10**. The max allowable value for **obase** is **BC_BASE_MAX** and can be queried in bc(1) programs with the **maxobase()** built-in function. The min allowable value for **obase** is **0**. If **obase** is **0**, values are output in scientific notation, and if **obase** is **1**, values are output in engineering notation. Otherwise, values are output in the specified base.

Outputting in scientific and engineering notations are **non-portable extensions**.

The *scale* of an expression is the number of digits in the result of the expression right of the decimal point, and **scale** is a global variable that sets the precision of any operations, with exceptions. **scale** is initially **0**. **scale** cannot be negative. The max allowable value for **scale** is **BC_SCALE_MAX** and can be queried in bc(1) programs with the **maxscale()** built-in function.

bc(1) has both *global* variables and *local* variables. All *local* variables are local to the function; they are parameters or are introduced in the **auto** list of a function (see the **FUNCTIONS** section). If a variable is accessed which is not a parameter or in the **auto** list, it is assumed to be *global*. If a parent function has a *local* variable version of a variable that a child function considers *global*, the value of that *global* variable in the child function is the value of the variable in the parent function, not the value of the actual *global* variable.

All of the above applies to arrays as well.

The value of a statement that is an expression (i.e., any of the named expressions or operands) is printed unless the lowest precedence operator is an assignment operator *and* the expression is notsurrounded by parentheses.

The value that is printed is also assigned to the special variable **last**. A single dot (**.**) may also be used as a synonym for **last**. These are **non-portable extensions**.

Either semicolons or newlines may separate statements.

Comments

There are two kinds of comments:

1. Block comments are enclosed in **/*** and ***/**.

2. Line comments go from **#** until, and not including, the next newline. This is a **non-portable extension**.

Named Expressions

The following are named expressions in bc(1):

1. Variables: **I**

2. Array Elements: **I[E]**

3. **ibase**

4. **obase**

5. **scale**

6. **seed**

7. **last** or a single dot (**.**)

Numbers 6 and 7 are **non-portable extensions**.

The meaning of **seed** is dependent on the current pseudo-random number generator but is guaranteed to not change except for new major versions.

The *scale* and sign of the value may be significant.

If a previously used **seed** value is assigned to **seed** and used again, the pseudo-random number generator is guaranteed to produce the same sequence of pseudo-random numbers as it did when the **seed** value was previously used.

The exact value assigned to **seed** is not guaranteed to be returned if **seed** is queried again immediately. However, if **seed** *does* return a different value, both values, when assigned to **seed**, are guaranteed to

produce the same sequence of pseudo-random numbers. This means that certain values assigned to **seed** will *not* produce unique sequences of pseudo-random numbers. The value of **seed** will change after any use of the **rand()** and **irand(E)** operands (see the *Operands* subsection below), except if the parameter passed to **irand(E)** is **0**, **1**, or negative.

There is no limit to the length (number of significant decimal digits) or *scale* of the value that can be assigned to **seed**.

Variables and arrays do not interfere; users can have arrays named the same as variables. This also applies to functions (see the **FUNCTIONS** section), so a user can have a variable, array, and function that all have the same name, and they will not shadow each other, whether inside of functions or not.

Named expressions are required as the operand of **increment**/**decrement** operators and as the left side of **assignment** operators (see the *Operators* subsection).

Operands

The following are valid operands in bc(1):

1. Numbers (see the *Numbers* subsection below).
2. Array indices (**I[E]**).
3. **(E)**: The value of **E** (used to change precedence).
4. **sqrt(E)**: The square root of **E**. **E** must be non-negative.
5. **length(E)**: The number of significant decimal digits in **E**. Returns **1** for **0** with no decimal places. If given a string, the length of the string is returned. Passing a string to **length(E)** is a **non-portable extension**.
6. **length(I[])**: The number of elements in the array **I**. This is a **non-portable extension**.
7. **scale(E)**: The *scale* of **E**.
8. **abs(E)**: The absolute value of **E**. This is a **non-portable extension**.
9. **is_number(E)**: **1** if the given argument is a number, **0** if it is a string. This is a **non-portable extension**.
10. **is_string(E)**: **1** if the given argument is a string, **0** if it is a number. This is a **non-portable**

extension.

11. **modexp(E, E, E)**: Modular exponentiation, where the first expression is the base, the second is the exponent, and the third is the modulus. All three values must be integers. The second argument must be non-negative. The third argument must be non-zero. This is a **non-portable extension**.

12. **divmod(E, E, I[])**: Division and modulus in one operation. This is for optimization. The first expression is the dividend, and the second is the divisor, which must be non-zero. The return value is the quotient, and the modulus is stored in index **0** of the provided array (the last argument). This is a **non-portable extension**.

13. **asciify(E)**: If **E** is a string, returns a string that is the first letter of its argument. If it is a number, calculates the number mod **256** and returns that number as a one-character string. This is a **non-portable extension**.

14. **asciify(I[])**: A string that is made up of the characters that would result from running **asciify(E)** on each element of the array identified by the argument. This allows creating multi-character strings and storing them. This is a **non-portable extension**.

15. **I()**, **I(E)**, **I(E, E)**, and so on, where **I** is an identifier for a non-**void** function (see the *Void Functions* subsection of the **FUNCTIONS** section). The **E** argument(s) may also be arrays of the form **I[]**, which will automatically be turned into array references (see the *Array References* subsection of the **FUNCTIONS** section) if the corresponding parameter in the function definition is an array reference.

16. **read()**: Reads a line from **stdin** and uses that as an expression. The result of that expression is the result of the **read()** operand. This is a **non-portable extension**.

17. **maxibase()**: The max allowable **ibase**. This is a **non-portable extension**.

18. **maxobase()**: The max allowable **obase**. This is a **non-portable extension**.

19. **maxscale()**: The max allowable **scale**. This is a **non-portable extension**.

20. **line_length()**: The line length set with **BC_LINE_LENGTH** (see the **ENVIRONMENT VARIABLES** section). This is a **non-portable extension**.

21. **global_stacks()**: **0** if global stacks are not enabled with the **-g** or **--global-stacks** options, non-zero otherwise. See the **OPTIONS** section. This is a **non-portable extension**.

22. **leading_zero()**: **0** if leading zeroes are not enabled with the **-z** or **-leading-zeroes** options, non-zero otherwise. See the **OPTIONS** section. This is a **non-portable extension**.

23. **rand()**: A pseudo-random integer between **0** (inclusive) and **BC_RAND_MAX** (inclusive). Using this operand will change the value of **seed**. This is a **non-portable extension**.

24. **irand(E)**: A pseudo-random integer between **0** (inclusive) and the value of **E** (exclusive). If **E** is negative or is a non-integer (**E**'s *scale* is not **0**), an error is raised, and bc(1) resets (see the **RESET** section) while **seed** remains unchanged. If **E** is larger than **BC_RAND_MAX**, the higher bound is honored by generating several pseudo-random integers, multiplying them by appropriate powers of **BC_RAND_MAX+1**, and adding them together. Thus, the size of integer that can be generated with this operand is unbounded. Using this operand will change the value of **seed**, unless the value of **E** is **0** or **1**. In that case, **0** is returned, and **seed** is *not* changed. This is a **non-portable extension**.

25. **maxrand()**: The max integer returned by **rand()**. This is a **non-portable extension**.

The integers generated by **rand()** and **irand(E)** are guaranteed to be as unbiased as possible, subject to the limitations of the pseudo-random number generator.

Note: The values returned by the pseudo-random number generator with **rand()** and **irand(E)** are guaranteed to *NOT* be cryptographically secure. This is a consequence of using a seeded pseudo-random number generator. However, they *are* guaranteed to be reproducible with identical **seed** values. This means that the pseudo-random values from bc(1) should only be used where a reproducible stream of pseudo-random numbers is *ESSENTIAL*. In any other case, use a non-seeded pseudo-random number generator.

Numbers

Numbers are strings made up of digits, uppercase letters, and at most **1** period for a radix. Numbers can have up to **BC_NUM_MAX** digits. Uppercase letters are equal to **9** plus their position in the alphabet, starting from **1** (i.e., **A** equals **10**, or **9+1**).

If a digit or letter makes no sense with the current value of **ibase** (i.e., they are greater than or equal to the current value of **ibase**), then the behavior depends on the existence of the **-c**/**--digit-clamp** or **-C**/**--no-digit-clamp** options (see the **OPTIONS** section), the existence and setting of the **BC_DIGIT_CLAMP** environment variable (see the **ENVIRONMENT VARIABLES** section), or the default, which can be queried with the **-h**/**--help** option.

If clamping is off, then digits or letters that are greater than or equal to the current value of **ibase** are not changed. Instead, their given value is multiplied by the appropriate power of **ibase** and added into

the number. This means that, with an **ibase** of **3**, the number **AB** is equal to **3^1*A+3^0*B**, which is **3** times **10** plus **11**, or **41**.

If clamping is on, then digits or letters that are greater than or equal to the current value of **ibase** are set to the value of the highest valid digit in **ibase** before being multiplied by the appropriate power of **ibase** and added into the number. This means that, with an **ibase** of **3**, the number **AB** is equal to **3^1*2+3^0*2**, which is **3** times **2** plus **2**, or **8**.

There is one exception to clamping: single-character numbers (i.e., **A** alone). Such numbers are never clamped and always take the value they would have in the highest possible **ibase**. This means that **A** alone always equals decimal **10** and **Z** alone always equals decimal **35**. This behavior is mandated by the standard (see the STANDARDS section) and is meant to provide an easy way to set the current **ibase** (with the **i** command) regardless of the current value of **ibase**.

If clamping is on, and the clamped value of a character is needed, use a leading zero, i.e., for **A**, use **0A**.

In addition, bc(1) accepts numbers in scientific notation. These have the form **<number>e<integer>**. The exponent (the portion after the **e**) must be an integer. An example is **1.89237e9**, which is equal to **1892370000**. Negative exponents are also allowed, so **4.2890e-3** is equal to **0.0042890**.

Using scientific notation is an error or warning if the **-s** or **-w**, respectively, command-line options (or equivalents) are given.

WARNING: Both the number and the exponent in scientific notation are interpreted according to the current **ibase**, but the number is still multiplied by **10^exponent** regardless of the current **ibase**. For example, if **ibase** is **16** and bc(1) is given the number string **FFeA**, the resulting decimal number will be **2550000000000**, and if bc(1) is given the number string **10e-4**, the resulting decimal number will be **0.0016**.

Accepting input as scientific notation is a **non-portable extension**.

Operators

The following arithmetic and logical operators can be used. They are listed in order of decreasing precedence. Operators in the same group have the same precedence.

++ --

Type: Prefix and Postfix

Associativity: None

Description: **increment**, **decrement**

- ! Type: Prefix

Associativity: None

Description: **negation**, **boolean not**

$ Type: Postfix

Associativity: None

Description: **truncation**

@ Type: Binary

Associativity: Right

Description: **set precision**

^ Type: Binary

Associativity: Right

Description: **power**

*** / %**

Type: Binary

Associativity: Left

Description: **multiply**, **divide**, **modulus**

+ - Type: Binary

Associativity: Left

Description: **add**, **subtract**

<< >>

Type: Binary

Associativity: Left

Description: **shift left**, **shift right**

= <<= >>= += -= *= /= %= ^= @=
Type: Binary

Associativity: Right

Description: **assignment**

== <= >= != < >
Type: Binary

Associativity: Left

Description: **relational**

&&
Type: Binary

Associativity: Left

Description: **boolean and**

|| Type: Binary

Associativity: Left

Description: **boolean or**

The operators will be described in more detail below.

++ --
The prefix and postfix **increment** and **decrement** operators behave exactly like they would in C. They require a named expression (see the *Named Expressions* subsection) as an operand.

The prefix versions of these operators are more efficient; use them where possible.

- The **negation** operator returns **0** if a user attempts to negate any expression with the value **0**. Otherwise, a copy of the expression with its sign flipped is returned.

! The **boolean not** operator returns **1** if the expression is **0**, or **0** otherwise.

This is a **non-portable extension**.

$ The **truncation** operator returns a copy of the given expression with all of its *scale* removed.

This is a **non-portable extension**.

@ The **set precision** operator takes two expressions and returns a copy of the first with its *scale* equal to the value of the second expression. That could either mean that the number is returned without change (if the *scale* of the first expression matches the value of the second expression), extended (if it is less), or truncated (if it is more).

The second expression must be an integer (no *scale*) and non-negative.

This is a **non-portable extension**.

^ The **power** operator (not the **exclusive or** operator, as it would be in C) takes two expressions and raises the first to the power of the value of the second. The *scale* of the result is equal to **scale**.

The second expression must be an integer (no *scale*), and if it is negative, the first value must be non-zero.

***** The **multiply** operator takes two expressions, multiplies them, and returns the product. If **a** is the *scale* of the first expression and **b** is the *scale* of the second expression, the *scale* of the result is equal to **min(a+b,max(scale,a,b))** where **min()** and **max()** return the obvious values.

/ The **divide** operator takes two expressions, divides them, and returns the quotient. The *scale* of the result shall be the value of **scale**.

The second expression must be non-zero.

% The **modulus** operator takes two expressions, **a** and **b**, and evaluates them by 1) Computing **a/b** to current **scale** and 2) Using the result of step 1 to calculate **a-(a/b)*b** to *scale* **max(scale+scale(b),scale(a))**.

The second expression must be non-zero.

+ The **add** operator takes two expressions, **a** and **b**, and returns the sum, with a *scale* equal to the max of the *scale*s of **a** and **b**.

- The **subtract** operator takes two expressions, **a** and **b**, and returns the difference, with a *scale* equal to the max of the *scale*s of **a** and **b**.

<< The **left shift** operator takes two expressions, **a** and **b**, and returns a copy of the value of **a** with its decimal point moved **b** places to the right.

The second expression must be an integer (no *scale*) and non-negative.

This is a **non-portable extension**.

>> The **right shift** operator takes two expressions, **a** and **b**, and returns a copy of the value of **a** with its decimal point moved **b** places to the left.

The second expression must be an integer (no *scale*) and non-negative.

This is a **non-portable extension**.

= <<= >>= += -= *= /= %= ^= @=

The **assignment** operators take two expressions, **a** and **b** where **a** is a named expression (see the *Named Expressions* subsection).

For **=**, **b** is copied and the result is assigned to **a**. For all others, **a** and **b** are applied as operands to the corresponding arithmetic operator and the result is assigned to **a**.

The **assignment** operators that correspond to operators that are extensions are themselves **non-portable extensions**.

== <= >= != < >

The **relational** operators compare two expressions, **a** and **b**, and if the relation holds, according to C language semantics, the result is **1**. Otherwise, it is **0**.

Note that unlike in C, these operators have a lower precedence than the **assignment** operators, which means that **a=b>c** is interpreted as **(a=b)>c**.

Also, unlike the standard (see the **STANDARDS** section) requires, these operators can appear anywhere any other expressions can be used. This allowance is a **non-portable extension**.

&&

The **boolean and** operator takes two expressions and returns **1** if both expressions are non-zero, **0** otherwise.

This is *not* a short-circuit operator.

This is a **non-portable extension**.

|| The **boolean or** operator takes two expressions and returns **1** if one of the expressions is non-zero, **0** otherwise.

This is *not* a short-circuit operator.

This is a **non-portable extension**.

Statements

The following items are statements:

1. **E**
2. **{ S ; ... ; S }**
3. **if (E) S**
4. **if (E) S else S**
5. **while (E) S**
6. **for (E ; E ; E) S**
7. An empty statement
8. **break**
9. **continue**
10. **quit**
11. **halt**

12. **limits**

13. A string of characters, enclosed in double quotes

14. **print E , ... , E**

15. **stream E , ... , E**

16. **I()**, **I(E)**, **I(E, E)**, and so on, where **I** is an identifier for a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section). The **E** argument(s) may also be arrays of the form **I[]**, which will automatically be turned into array references (see the *Array References* subsection of the **FUNCTIONS** section) if the corresponding parameter in the function definition is an array reference.

Numbers 4, 9, 11, 12, 14, 15, and 16 are **non-portable extensions**.

Also, as a **non-portable extension**, any or all of the expressions in the header of a for loop may be omitted. If the condition (second expression) is omitted, it is assumed to be a constant **1**.

The **break** statement causes a loop to stop iterating and resume execution immediately following a loop. This is only allowed in loops.

The **continue** statement causes a loop iteration to stop early and returns to the start of the loop, including testing the loop condition. This is only allowed in loops.

The **if else** statement does the same thing as in C.

The **quit** statement causes bc(1) to quit, even if it is on a branch that will not be executed (it is a compile-time command).

Warning: The behavior of this bc(1) on **quit** is slightly different from other bc(1) implementations. Other bc(1) implementations will exit as soon as they finish parsing the line that a **quit** command is on. This bc(1) will execute any completed and executable statements that occur before the **quit** statement before exiting.

In other words, for the bc(1) code below:

```
for (i = 0; i < 3; ++i) i; quit
```

Other bc(1) implementations will print nothing, and this bc(1) will print **0**, **1**, and **2** on successive lines before exiting.

The **halt** statement causes bc(1) to quit, if it is executed. (Unlike **quit** if it is on a branch of an **if** statement that is not executed, bc(1) does not quit.)

The **limits** statement prints the limits that this bc(1) is subject to. This is like the **quit** statement in that it is a compile-time command.

An expression by itself is evaluated and printed, followed by a newline.

Both scientific notation and engineering notation are available for printing the results of expressions. Scientific notation is activated by assigning **0** to **obase**, and engineering notation is activated by assigning **1** to **obase**. To deactivate them, just assign a different value to **obase**.

Scientific notation and engineering notation are disabled if bc(1) is run with either the **-s** or **-w** command-line options (or equivalents).

Printing numbers in scientific notation and/or engineering notation is a **non-portable extension**.

Strings

If strings appear as a statement by themselves, they are printed without a trailing newline.

In addition to appearing as a lone statement by themselves, strings can be assigned to variables and array elements. They can also be passed to functions in variable parameters.

If any statement that expects a string is given a variable that had a string assigned to it, the statement acts as though it had received a string.

If any math operation is attempted on a string or a variable or array element that has been assigned a string, an error is raised, and bc(1) resets (see the **RESET** section).

Assigning strings to variables and array elements and passing them to functions are **non-portable extensions**.

Print Statement

The "expressions" in a **print** statement may also be strings. If they are, there are backslash escape sequences that are interpreted specially. What those sequences are, and what they cause to be printed, are shown below:

\a: **\a**

\b: **\b**

\\: \

\e: \

\f: **\f**

\n: **\n**

\q: **"**

\r: **\r**

\t: **\t**

Any other character following a backslash causes the backslash and character to be printed as-is.

Any non-string expression in a print statement shall be assigned to **last**, like any other expression that is printed.

Stream Statement

The "expressions in a **stream** statement may also be strings.

If a **stream** statement is given a string, it prints the string as though the string had appeared as its own statement. In other words, the **stream** statement prints strings normally, without a newline.

If a **stream** statement is given a number, a copy of it is truncated and its absolute value is calculated. The result is then printed as though **obase** is **256** and each digit is interpreted as an 8-bit ASCII character, making it a byte stream.

Order of Evaluation

All expressions in a statment are evaluated left to right, except as necessary to maintain order of operations. This means, for example, assuming that **i** is equal to **0**, in the expression

```
a[i++] = i++
```

the first (or 0th) element of **a** is set to **1**, and **i** is equal to **2** at the end of the expression.

This includes function arguments. Thus, assuming **i** is equal to **0**, this means that in the expression

```
x(i++, i++)
```

the first argument passed to **x()** is **0**, and the second argument is **1**, while **i** is equal to **2** before the function starts executing.

FUNCTIONS

Function definitions are as follows:

```
define I(I,...,I){
   auto I,...,I
   S;...;S
   return(E)
}
```

Any **I** in the parameter list or **auto** list may be replaced with **I[]** to make a parameter or **auto** var an array, and any **I** in the parameter list may be replaced with ***I[]** to make a parameter an array reference. Callers of functions that take array references should not put an asterisk in the call; they must be called with just **I[]** like normal array parameters and will be automatically converted into references.

As a **non-portable extension**, the opening brace of a **define** statement may appear on the next line.

As a **non-portable extension**, the return statement may also be in one of the following forms:

1. **return**
2. **return ( )**
3. **return E**

The first two, or not specifying a **return** statement, is equivalent to **return (0)**, unless the function is a **void** function (see the *Void Functions* subsection below).

Void Functions

Functions can also be **void** functions, defined as follows:

```
define void I(I,...,I){
   auto I,...,I
   S;...;S
   return
}
```

They can only be used as standalone expressions, where such an expression would be printed alone, except in a print statement.

Void functions can only use the first two **return** statements listed above. They can also omit the return statement entirely.

The word "void" is not treated as a keyword; it is still possible to have variables, arrays, and functions named **void**. The word "void" is only treated specially right after the **define** keyword.

This is a **non-portable extension**.

Array References

For any array in the parameter list, if the array is declared in the form

```
*I[]
```

it is a **reference**. Any changes to the array in the function are reflected, when the function returns, to the array that was passed in.

Other than this, all function arguments are passed by value.

This is a **non-portable extension**.

LIBRARY

All of the functions below, including the functions in the extended math library (see the *Extended Library* subsection below), are available when the **-l** or **--mathlib** command-line flags are given, except that the extended math library is not available when the **-s** option, the **-w** option, or equivalents are

given.

Standard Library

The standard (see the **STANDARDS** section) defines the following functions for the math library:

s(x)

Returns the sine of **x**, which is assumed to be in radians.

This is a transcendental function (see the *Transcendental Functions* subsection below).

c(x)

Returns the cosine of **x**, which is assumed to be in radians.

This is a transcendental function (see the *Transcendental Functions* subsection below).

a(x)

Returns the arctangent of **x**, in radians.

This is a transcendental function (see the *Transcendental Functions* subsection below).

l(x) Returns the natural logarithm of **x**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

e(x)

Returns the mathematical constant **e** raised to the power of **x**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

j(x, n)

Returns the bessel integer order **n** (truncated) of **x**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

Extended Library

The extended library is *not* loaded when the **-s**/**--standard** or **-w**/**--warn** options are given since they are not part of the library defined by the standard (see the **STANDARDS** section).

The extended library is a **non-portable extension**.

p(x, y)

Calculates **x** to the power of **y**, even if **y** is not an integer, and returns the result to the current **scale**.

It is an error if **y** is negative and **x** is **0**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

r(x, p)

Returns **x** rounded to **p** decimal places according to the rounding mode round half away from **0** (https://en.wikipedia.org/wiki/Rounding#Round_half_away_from_zero).

ceil(x, p)

Returns **x** rounded to **p** decimal places according to the rounding mode round away from **0** (https://en.wikipedia.org/wiki/Rounding#Rounding_away_from_zero).

f(x) Returns the factorial of the truncated absolute value of **x**.

perm(n, k)

Returns the permutation of the truncated absolute value of **n** of the truncated absolute value of **k**, if **k <= n**. If not, it returns **0**.

comb(n, k)

Returns the combination of the truncated absolute value of **n** of the truncated absolute value of **k**, if **k <= n**. If not, it returns **0**.

l2(x)

Returns the logarithm base **2** of **x**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

l10(x)

Returns the logarithm base **10** of **x**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

log(x, b)

Returns the logarithm base **b** of **x**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

cbrt(x)

Returns the cube root of **x**.

root(x, n)

Calculates the truncated value of **n**, **r**, and returns the **r**th root of **x** to the current **scale**.

If **r** is **0** or negative, this raises an error and causes bc(1) to reset (see the **RESET** section). It also raises an error and causes bc(1) to reset if **r** is even and **x** is negative.

gcd(a, b)

Returns the greatest common divisor (factor) of the truncated absolute value of **a** and the truncated absolute value of **b**.

lcm(a, b)

Returns the least common multiple of the truncated absolute value of **a** and the truncated absolute value of **b**.

pi(p)

Returns **pi** to **p** decimal places.

This is a transcendental function (see the *Transcendental Functions* subsection below).

t(x) Returns the tangent of **x**, which is assumed to be in radians.

This is a transcendental function (see the *Transcendental Functions* subsection below).

a2(y, x)

Returns the arctangent of **y/x**, in radians. If both **y** and **x** are equal to **0**, it raises an error and causes bc(1) to reset (see the **RESET** section). Otherwise, if **x** is greater than **0**, it returns **a(y/x)**. If **x** is less than **0**, and **y** is greater than or equal to **0**, it returns **a(y/x)+pi**. If **x** is less than **0**, and **y** is less than **0**, it returns **a(y/x)-pi**. If **x** is equal to **0**, and **y** is greater than **0**, it returns **pi/2**. If **x** is equal to **0**, and **y** is less than **0**, it returns **-pi/2**.

This function is the same as the **atan2()** function in many programming languages.

This is a transcendental function (see the *Transcendental Functions* subsection below).

sin(x)

Returns the sine of **x**, which is assumed to be in radians.

This is an alias of **s(x)**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

cos(x)

Returns the cosine of **x**, which is assumed to be in radians.

This is an alias of **c(x)**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

tan(x)

Returns the tangent of **x**, which is assumed to be in radians.

If **x** is equal to **1** or **-1**, this raises an error and causes bc(1) to reset (see the **RESET** section).

This is an alias of **t(x)**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

atan(x)

Returns the arctangent of **x**, in radians.

This is an alias of **a(x)**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

atan2(y, x)

Returns the arctangent of **y/x**, in radians. If both **y** and **x** are equal to **0**, it raises an error and causes bc(1) to reset (see the **RESET** section). Otherwise, if **x** is greater than **0**, it returns **a(y/x)**. If **x** is less than **0**, and **y** is greater than or equal to **0**, it returns **a(y/x)+pi**. If **x** is less than **0**, and **y** is less than **0**, it returns **a(y/x)-pi**. If **x** is equal to **0**, and **y** is greater than **0**, it returns **pi/2**. If **x** is equal to **0**, and **y** is less than **0**, it returns **-pi/2**.

This function is the same as the **atan2()** function in many programming languages.

This is an alias of **a2(y, x)**.

This is a transcendental function (see the *Transcendental Functions* subsection below).

r2d(x)

Converts **x** from radians to degrees and returns the result.

This is a transcendental function (see the *Transcendental Functions* subsection below).

d2r(x)

Converts **x** from degrees to radians and returns the result.

This is a transcendental function (see the *Transcendental Functions* subsection below).

frand(p)

Generates a pseudo-random number between **0** (inclusive) and **1** (exclusive) with the number of decimal digits after the decimal point equal to the truncated absolute value of **p**. If **p** is not **0**, then calling this function will change the value of **seed**. If **p** is **0**, then **0** is returned, and **seed** is *not* changed.

ifrand(i, p)

Generates a pseudo-random number that is between **0** (inclusive) and the truncated absolute value of **i** (exclusive) with the number of decimal digits after the decimal point equal to the truncated absolute value of **p**. If the absolute value of **i** is greater than or equal to **2**, and **p** is not **0**, then calling this function will change the value of **seed**; otherwise, **0** is returned and **seed** is not changed.

srand(x)

Returns **x** with its sign flipped with probability **0.5**. In other words, it randomizes the sign of **x**.

brand()

Returns a random boolean value (either **0** or **1**).

band(a, b)

Takes the truncated absolute value of both **a** and **b** and calculates and returns the result of the bitwise **and** operation between them.

If you want to use signed two's complement arguments, use **s2u(x)** to convert.

bor(a, b)

Takes the truncated absolute value of both **a** and **b** and calculates and returns the result of the bitwise **or** operation between them.

If you want to use signed two's complement arguments, use **s2u(x)** to convert.

bxor(a, b)

Takes the truncated absolute value of both **a** and **b** and calculates and returns the result of the bitwise **xor** operation between them.

If you want to use signed two's complement arguments, use **s2u(x)** to convert.

bshl(a, b)

Takes the truncated absolute value of both **a** and **b** and calculates and returns the result of **a** bit-shifted left by **b** places.

If you want to use signed two's complement arguments, use **s2u(x)** to convert.

bshr(a, b)

Takes the truncated absolute value of both **a** and **b** and calculates and returns the truncated result of **a** bit-shifted right by **b** places.

If you want to use signed two's complement arguments, use **s2u(x)** to convert.

bnotn(x, n)

Takes the truncated absolute value of **x** and does a bitwise not as though it has the same number of bytes as the truncated absolute value of **n**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bnot8(x)

Does a bitwise not of the truncated absolute value of **x** as though it has **8** binary digits (1 unsigned byte).

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bnot16(x)

Does a bitwise not of the truncated absolute value of **x** as though it has **16** binary digits (2 unsigned bytes).

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bnot32(x)

Does a bitwise not of the truncated absolute value of **x** as though it has **32** binary digits (4 unsigned bytes).

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bnot64(x)

Does a bitwise not of the truncated absolute value of **x** as though it has **64** binary digits (8 unsigned bytes).

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bnot(x)

Does a bitwise not of the truncated absolute value of **x** as though it has the minimum number of power of two unsigned bytes.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brevn(x, n)

Runs a bit reversal on the truncated absolute value of **x** as though it has the same number of 8-bit bytes as the truncated absolute value of **n**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brev8(x)

Runs a bit reversal on the truncated absolute value of **x** as though it has 8 binary digits (1 unsigned byte).

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brev16(x)

Runs a bit reversal on the truncated absolute value of **x** as though it has 16 binary digits (2 unsigned bytes).

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brev32(x)

Runs a bit reversal on the truncated absolute value of **x** as though it has 32 binary digits (4 unsigned bytes).

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brev64(x)

Runs a bit reversal on the truncated absolute value of **x** as though it has 64 binary digits (8

unsigned bytes).

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brev(x)

Runs a bit reversal on the truncated absolute value of **x** as though it has the minimum number of power of two unsigned bytes.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

broln(x, p, n)

Does a left bitwise rotatation of the truncated absolute value of **x**, as though it has the same number of unsigned 8-bit bytes as the truncated absolute value of **n**, by the number of places equal to the truncated absolute value of **p** modded by the **2** to the power of the number of binary digits in **n** 8-bit bytes.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brol8(x, p)

Does a left bitwise rotatation of the truncated absolute value of **x**, as though it has **8** binary digits (**1** unsigned byte), by the number of places equal to the truncated absolute value of **p** modded by **2** to the power of **8**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brol16(x, p)

Does a left bitwise rotatation of the truncated absolute value of **x**, as though it has **16** binary digits (**2** unsigned bytes), by the number of places equal to the truncated absolute value of **p** modded by **2** to the power of **16**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brol32(x, p)

Does a left bitwise rotatation of the truncated absolute value of **x**, as though it has **32** binary digits (**2** unsigned bytes), by the number of places equal to the truncated absolute value of **p** modded by **2** to the power of **32**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brol64(x, p)

Does a left bitwise rotatation of the truncated absolute value of **x**, as though it has **64** binary digits (**2** unsigned bytes), by the number of places equal to the truncated absolute value of **p** modded by **2** to the power of **64**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brol(x, p)

Does a left bitwise rotatation of the truncated absolute value of **x**, as though it has the minimum number of power of two unsigned 8-bit bytes, by the number of places equal to the truncated absolute value of **p** modded by 2 to the power of the number of binary digits in the minimum number of 8-bit bytes.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

brorn(x, p, n)

Does a right bitwise rotatation of the truncated absolute value of **x**, as though it has the same number of unsigned 8-bit bytes as the truncated absolute value of **n**, by the number of places equal to the truncated absolute value of **p** modded by the **2** to the power of the number of binary digits in **n** 8-bit bytes.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bror8(x, p)

Does a right bitwise rotatation of the truncated absolute value of **x**, as though it has **8** binary digits (**1** unsigned byte), by the number of places equal to the truncated absolute value of **p** modded by **2** to the power of **8**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bror16(x, p)

Does a right bitwise rotatation of the truncated absolute value of **x**, as though it has **16** binary digits (**2** unsigned bytes), by the number of places equal to the truncated absolute value of **p** modded by **2** to the power of **16**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bror32(x, p)

Does a right bitwise rotatation of the truncated absolute value of **x**, as though it has **32** binary digits (**2** unsigned bytes), by the number of places equal to the truncated absolute value of **p** modded by **2** to the power of **32**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bror64(x, p)

Does a right bitwise rotatation of the truncated absolute value of **x**, as though it has **64** binary digits (**2** unsigned bytes), by the number of places equal to the truncated absolute value of **p** modded by **2** to the power of **64**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bror(x, p)

Does a right bitwise rotatation of the truncated absolute value of **x**, as though it has the minimum number of power of two unsigned 8-bit bytes, by the number of places equal to the truncated absolute value of **p** modded by 2 to the power of the number of binary digits in the minimum number of 8-bit bytes.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bmodn(x, n)

Returns the modulus of the truncated absolute value of **x** by **2** to the power of the multiplication of the truncated absolute value of **n** and **8**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bmod8(x, n)

Returns the modulus of the truncated absolute value of **x** by **2** to the power of **8**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bmod16(x, n)

Returns the modulus of the truncated absolute value of **x** by **2** to the power of **16**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bmod32(x, n)

Returns the modulus of the truncated absolute value of **x** by **2** to the power of **32**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bmod64(x, n)

Returns the modulus of the truncated absolute value of **x** by **2** to the power of **64**.

If you want to a use signed two's complement argument, use **s2u(x)** to convert.

bunrev(t)

Assumes **t** is a bitwise-reversed number with an extra set bit one place more significant than the real most significant bit (which was the least significant bit in the original number). This number is reversed and returned without the extra set bit.

This function is used to implement other bitwise functions; it is not meant to be used by users, but it can be.

plz(x)

If **x** is not equal to **0** and greater that **-1** and less than **1**, it is printed with a leading zero, regardless of the use of the **-z** option (see the **OPTIONS** section) and without a trailing newline.

Otherwise, **x** is printed normally, without a trailing newline.

plznl(x)

If **x** is not equal to **0** and greater that **-1** and less than **1**, it is printed with a leading zero, regardless of the use of the **-z** option (see the **OPTIONS** section) and with a trailing newline.

Otherwise, **x** is printed normally, with a trailing newline.

pnlz(x)

If **x** is not equal to **0** and greater that **-1** and less than **1**, it is printed without a leading zero, regardless of the use of the **-z** option (see the **OPTIONS** section) and without a trailing newline.

Otherwise, **x** is printed normally, without a trailing newline.

pnlznl(x)

If **x** is not equal to **0** and greater that **-1** and less than **1**, it is printed without a leading zero, regardless of the use of the **-z** option (see the **OPTIONS** section) and with a trailing newline.

Otherwise, **x** is printed normally, with a trailing newline.

ubytes(x)

Returns the numbers of unsigned integer bytes required to hold the truncated absolute value of **x**.

sbytes(x)

Returns the numbers of signed, two's-complement integer bytes required to hold the truncated value of **x**.

s2u(x)

Returns **x** if it is non-negative. If it *is* negative, then it calculates what **x** would be as a 2's-complement signed integer and returns the non-negative integer that would have the same representation in binary.

s2un(x,n)

Returns **x** if it is non-negative. If it *is* negative, then it calculates what **x** would be as a 2's-complement signed integer with **n** bytes and returns the non-negative integer that would have the same representation in binary. If **x** cannot fit into **n** 2's-complement signed bytes, it is truncated to fit.

hex(x)

Outputs the hexadecimal (base **16**) representation of **x**.

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

binary(x)

Outputs the binary (base **2**) representation of **x**.

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

output(x, b)

Outputs the base **b** representation of **x**.

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

uint(x)

Outputs the representation, in binary and hexadecimal, of **x** as an unsigned integer in as few power of two bytes as possible. Both outputs are split into bytes separated by spaces.

If **x** is not an integer or is negative, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

int(x)

Outputs the representation, in binary and hexadecimal, of **x** as a signed, two's-complement integer in as few power of two bytes as possible. Both outputs are split into bytes separated by spaces.

If **x** is not an integer, an error message is printed instead, but bc(1) is not reset (see the **RESET**

section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

uintn(x, n)

Outputs the representation, in binary and hexadecimal, of **x** as an unsigned integer in **n** bytes. Both outputs are split into bytes separated by spaces.

If **x** is not an integer, is negative, or cannot fit into **n** bytes, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

intn(x, n)

Outputs the representation, in binary and hexadecimal, of **x** as a signed, two's-complement integer in **n** bytes. Both outputs are split into bytes separated by spaces.

If **x** is not an integer or cannot fit into **n** bytes, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

uint8(x)

Outputs the representation, in binary and hexadecimal, of **x** as an unsigned integer in **1** byte. Both outputs are split into bytes separated by spaces.

If **x** is not an integer, is negative, or cannot fit into **1** byte, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

int8(x)

Outputs the representation, in binary and hexadecimal, of **x** as a signed, two's-complement integer in **1** byte. Both outputs are split into bytes separated by spaces.

If **x** is not an integer or cannot fit into **1** byte, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

uint16(x)

Outputs the representation, in binary and hexadecimal, of **x** as an unsigned integer in **2** bytes. Both outputs are split into bytes separated by spaces.

If **x** is not an integer, is negative, or cannot fit into **2** bytes, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

int16(x)

Outputs the representation, in binary and hexadecimal, of **x** as a signed, two's-complement integer in **2** bytes. Both outputs are split into bytes separated by spaces.

If **x** is not an integer or cannot fit into **2** bytes, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

uint32(x)

Outputs the representation, in binary and hexadecimal, of **x** as an unsigned integer in **4** bytes. Both outputs are split into bytes separated by spaces.

If **x** is not an integer, is negative, or cannot fit into **4** bytes, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

int32(x)

Outputs the representation, in binary and hexadecimal, of **x** as a signed, two's-complement integer in **4** bytes. Both outputs are split into bytes separated by spaces.

If **x** is not an integer or cannot fit into **4** bytes, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

uint64(x)

Outputs the representation, in binary and hexadecimal, of **x** as an unsigned integer in **8** bytes. Both outputs are split into bytes separated by spaces.

If **x** is not an integer, is negative, or cannot fit into **8** bytes, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

int64(x)

Outputs the representation, in binary and hexadecimal, of **x** as a signed, two's-complement integer in **8** bytes. Both outputs are split into bytes separated by spaces.

If **x** is not an integer or cannot fit into **8** bytes, an error message is printed instead, but bc(1) is not reset (see the **RESET** section).

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

hex_uint(x, n)

Outputs the representation of the truncated absolute value of **x** as an unsigned integer in hexadecimal using **n** bytes. Not all of the value will be output if **n** is too small.

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

binary_uint(x, n)

Outputs the representation of the truncated absolute value of **x** as an unsigned integer in binary using **n** bytes. Not all of the value will be output if **n** is too small.

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

output_uint(x, n)

Outputs the representation of the truncated absolute value of **x** as an unsigned integer in the current **obase** (see the **SYNTAX** section) using **n** bytes. Not all of the value will be output if **n** is too small.

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

output_byte(x, i)

Outputs byte **i** of the truncated absolute value of **x**, where **0** is the least significant byte and **number_of_bytes - 1** is the most significant byte.

This is a **void** function (see the *Void Functions* subsection of the **FUNCTIONS** section).

Transcendental Functions

All transcendental functions can return slightly inaccurate results, up to 1 ULP (https://en.wikipedia.org/wiki/Unit_in_the_last_place). This is unavoidable, and the article at https://people.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT explains why it is impossible and unnecessary to calculate exact results for the transcendental functions.

Because of the possible inaccuracy, I recommend that users call those functions with the precision (**scale**) set to at least 1 higher than is necessary. If exact results are *absolutely* required, users can double the precision (**scale**) and then truncate.

The transcendental functions in the standard math library are:

- **s(x)**
- **c(x)**
- **a(x)**
- **l(x)**
- **e(x)**
- **j(x, n)**

The transcendental functions in the extended math library are:

- **l2(x)**
- **l10(x)**
- **log(x, b)**
- **pi(p)**
- **t(x)**
- **a2(y, x)**
- **sin(x)**
- **cos(x)**

- **tan(x)**
- **atan(x)**
- **atan2(y, x)**
- **r2d(x)**
- **d2r(x)**

RESET

When bc(1) encounters an error or a signal that it has a non-default handler for, it resets. This means that several things happen.

First, any functions that are executing are stopped and popped off the stack. The behavior is not unlike that of exceptions in programming languages. Then the execution point is set so that any code waiting to execute (after all functions returned) is skipped.

Thus, when bc(1) resets, it skips any remaining code waiting to be executed. Then, if it is interactive mode, and the error was not a fatal error (see the **EXIT STATUS** section), it asks for more input; otherwise, it exits with the appropriate return code.

Note that this reset behavior is different from the GNU bc(1), which attempts to start executing the statement right after the one that caused an error.

PERFORMANCE

Most bc(1) implementations use **char** types to calculate the value of **1** decimal digit at a time, but that can be slow. This bc(1) does something different.

It uses large integers to calculate more than **1** decimal digit at a time. If built in a environment where **BC_LONG_BIT** (see the **LIMITS** section) is **64**, then each integer has **9** decimal digits. If built in an environment where **BC_LONG_BIT** is **32** then each integer has **4** decimal digits. This value (the number of decimal digits per large integer) is called **BC_BASE_DIGS**.

The actual values of **BC_LONG_BIT** and **BC_BASE_DIGS** can be queried with the **limits** statement.

In addition, this bc(1) uses an even larger integer for overflow checking. This integer type depends on the value of **BC_LONG_BIT**, but is always at least twice as large as the integer type used to store digits.

LIMITS

The following are the limits on bc(1):

BC_LONG_BIT

The number of bits in the **long** type in the environment where bc(1) was built. This determines how many decimal digits can be stored in a single large integer (see the **PERFORMANCE** section).

BC_BASE_DIGS

The number of decimal digits per large integer (see the **PERFORMANCE** section). Depends on **BC_LONG_BIT**.

BC_BASE_POW

The max decimal number that each large integer can store (see **BC_BASE_DIGS**) plus **1**. Depends on **BC_BASE_DIGS**.

BC_OVERFLOW_MAX

The max number that the overflow type (see the **PERFORMANCE** section) can hold. Depends on **BC_LONG_BIT**.

BC_BASE_MAX

The maximum output base. Set at **BC_BASE_POW**.

BC_DIM_MAX

The maximum size of arrays. Set at **SIZE_MAX-1**.

BC_SCALE_MAX

The maximum **scale**. Set at **BC_OVERFLOW_MAX-1**.

BC_STRING_MAX

The maximum length of strings. Set at **BC_OVERFLOW_MAX-1**.

BC_NAME_MAX

The maximum length of identifiers. Set at **BC_OVERFLOW_MAX-1**.

BC_NUM_MAX

The maximum length of a number (in decimal digits), which includes digits after the decimal point. Set at **BC_OVERFLOW_MAX-1**.

BC_RAND_MAX

The maximum integer (inclusive) returned by the **rand()** operand. Set at **2^BC_LONG_BIT-1**.

Exponent
: The maximum allowable exponent (positive or negative). Set at **BC_OVERFLOW_MAX**.

Number of vars
: The maximum number of vars/arrays. Set at **SIZE_MAX-1**.

The actual values can be queried with the **limits** statement.

These limits are meant to be effectively non-existent; the limits are so large (at least on 64-bit machines) that there should not be any point at which they become a problem. In fact, memory should be exhausted before these limits should be hit.

ENVIRONMENT VARIABLES

As **non-portable extensions**, bc(1) recognizes the following environment variables:

POSIXLY_CORRECT

If this variable exists (no matter the contents), bc(1) behaves as if the **-s** option was given.

BC_ENV_ARGS

This is another way to give command-line arguments to bc(1). They should be in the same format as all other command-line arguments. These are always processed first, so any files given in **BC_ENV_ARGS** will be processed before arguments and files given on the command-line. This gives the user the ability to set up "standard" options and files to be used at every invocation. The most useful thing for such files to contain would be useful functions that the user might want every time bc(1) runs.

The code that parses **BC_ENV_ARGS** will correctly handle quoted arguments, but it does not understand escape sequences. For example, the string **"/home/gavin/some bc file.bc"** will be correctly parsed, but the string **"/home/gavin/some "bc" file.bc"** will include the backslashes.

The quote parsing will handle either kind of quotes, **'** or **"**. Thus, if you have a file with any number of single quotes in the name, you can use double quotes as the outside quotes, as in **"some 'bc' file.bc"**, and vice versa if you have a file with double quotes. However, handling a file with both kinds of quotes in **BC_ENV_ARGS** is not supported due to the complexity of the parsing, though such files are still supported on the command-line where the parsing is done by the shell.

BC_LINE_LENGTH

If this environment variable exists and contains an integer that is greater than **1** and is less than

UINT16_MAX (**2^16-1**), bc(1) will output lines to that length, including the backslash (\). The default line length is **70**.

The special value of **0** will disable line length checking and print numbers without regard to line length and without backslashes and newlines.

BC_BANNER

If this environment variable exists and contains an integer, then a non-zero value activates the copyright banner when bc(1) is in interactive mode, while zero deactivates it.

If bc(1) is not in interactive mode (see the **INTERACTIVE MODE** section), then this environment variable has no effect because bc(1) does not print the banner when not in interactive mode.

This environment variable overrides the default, which can be queried with the **-h** or **--help** options.

BC_SIGINT_RESET

If bc(1) is not in interactive mode (see the **INTERACTIVE MODE** section), then this environment variable has no effect because bc(1) exits on **SIGINT** when not in interactive mode.

However, when bc(1) is in interactive mode, then if this environment variable exists and contains an integer, a non-zero value makes bc(1) reset on **SIGINT**, rather than exit, and zero makes bc(1) exit. If this environment variable exists and is *not* an integer, then bc(1) will exit on **SIGINT**.

This environment variable overrides the default, which can be queried with the **-h** or **--help** options.

BC_TTY_MODE

If TTY mode is *not* available (see the **TTY MODE** section), then this environment variable has no effect.

However, when TTY mode is available, then if this environment variable exists and contains an integer, then a non-zero value makes bc(1) use TTY mode, and zero makes bc(1) not use TTY mode.

This environment variable overrides the default, which can be queried with the **-h** or **--help** options.

BC_PROMPT

If TTY mode is *not* available (see the **TTY MODE** section), then this environment variable has no

effect.

However, when TTY mode is available, then if this environment variable exists and contains an integer, a non-zero value makes bc(1) use a prompt, and zero or a non-integer makes bc(1) not use a prompt. If this environment variable does not exist and **BC_TTY_MODE** does, then the value of the **BC_TTY_MODE** environment variable is used.

This environment variable and the **BC_TTY_MODE** environment variable override the default, which can be queried with the **-h** or **--help** options.

BC_EXPR_EXIT

If any expressions or expression files are given on the command-line with **-e**, **--expression**, **-f**, or **--file**, then if this environment variable exists and contains an integer, a non-zero value makes bc(1) exit after executing the expressions and expression files, and a zero value makes bc(1) not exit.

This environment variable overrides the default, which can be queried with the **-h** or **--help** options.

BC_DIGIT_CLAMP

When parsing numbers and if this environment variable exists and contains an integer, a non-zero value makes bc(1) clamp digits that are greater than or equal to the current **ibase** so that all such digits are considered equal to the **ibase** minus 1, and a zero value disables such clamping so that those digits are always equal to their value, which is multiplied by the power of the **ibase**.

This never applies to single-digit numbers, as per the standard (see the **STANDARDS** section).

This environment variable overrides the default, which can be queried with the **-h** or **--help** options.

EXIT STATUS

bc(1) returns the following exit statuses:

0 No error.

1 A math error occurred. This follows standard practice of using **1** for expected errors, since math errors will happen in the process of normal execution.

Math errors include divide by **0**, taking the square root of a negative number, using a negative number as a bound for the pseudo-random number generator, attempting to convert a negative

number to a hardware integer, overflow when converting a number to a hardware integer, overflow when calculating the size of a number, and attempting to use a non-integer where an integer is required.

Converting to a hardware integer happens for the second operand of the power (^), places (**@**), left shift (<<), and right shift (>>) operators and their corresponding assignment operators.

2 A parse error occurred.

Parse errors include unexpected **EOF**, using an invalid character, failing to find the end of a string or comment, using a token where it is invalid, giving an invalid expression, giving an invalid print statement, giving an invalid function definition, attempting to assign to an expression that is not a named expression (see the *Named Expressions* subsection of the **SYNTAX** section), giving an invalid **auto** list, having a duplicate **auto**/function parameter, failing to find the end of a code block, attempting to return a value from a **void** function, attempting to use a variable as a reference, and using any extensions when the option **-s** or any equivalents were given.

3 A runtime error occurred.

Runtime errors include assigning an invalid number to any global (**ibase**, **obase**, or **scale**), giving a bad expression to a **read()** call, calling **read()** inside of a **read()** call, type errors, passing the wrong number of arguments to functions, attempting to call an undefined function, and attempting to use a **void** function call as a value in an expression.

4 A fatal error occurred.

Fatal errors include memory allocation errors, I/O errors, failing to open files, attempting to use files that do not have only ASCII characters (bc(1) only accepts ASCII characters), attempting to open a directory as a file, and giving invalid command-line options.

The exit status **4** is special; when a fatal error occurs, bc(1) always exits and returns **4**, no matter what mode bc(1) is in.

The other statuses will only be returned when bc(1) is not in interactive mode (see the **INTERACTIVE MODE** section), since bc(1) resets its state (see the **RESET** section) and accepts more input when one of those errors occurs in interactive mode. This is also the case when interactive mode is forced by the **-i** flag or **--interactive** option.

These exit statuses allow bc(1) to be used in shell scripting with error checking, and its normal behavior can be forced by using the **-i** flag or **--interactive** option.

INTERACTIVE MODE

Per the standard (see the **STANDARDS** section), bc(1) has an interactive mode and a non-interactive mode. Interactive mode is turned on automatically when both **stdin** and **stdout** are hooked to a terminal, but the **-i** flag and **--interactive** option can turn it on in other situations.

In interactive mode, bc(1) attempts to recover from errors (see the **RESET** section), and in normal execution, flushes **stdout** as soon as execution is done for the current input. bc(1) may also reset on **SIGINT** instead of exit, depending on the contents of, or default for, the **BC_SIGINT_RESET** environment variable (see the **ENVIRONMENT VARIABLES** section).

TTY MODE

If **stdin**, **stdout**, and **stderr** are all connected to a TTY, then "TTY mode" is considered to be available, and thus, bc(1) can turn on TTY mode, subject to some settings.

If there is the environment variable **BC_TTY_MODE** in the environment (see the **ENVIRONMENT VARIABLES** section), then if that environment variable contains a non-zero integer, bc(1) will turn on TTY mode when **stdin**, **stdout**, and **stderr** are all connected to a TTY. If the **BC_TTY_MODE** environment variable exists but is *not* a non-zero integer, then bc(1) will not turn TTY mode on.

If the environment variable **BC_TTY_MODE** does *not* exist, the default setting is used. The default setting can be queried with the **-h** or **--help** options.

TTY mode is different from interactive mode because interactive mode is required in the bc(1) standard (see the **STANDARDS** section), and interactive mode requires only **stdin** and **stdout** to be connected to a terminal.

Command-Line History

Command-line history is only enabled if TTY mode is, i.e., that **stdin**, **stdout**, and **stderr** are connected to a TTY and the **BC_TTY_MODE** environment variable (see the **ENVIRONMENT VARIABLES** section) and its default do not disable TTY mode. See the **COMMAND LINE HISTORY** section for more information.

Prompt

If TTY mode is available, then a prompt can be enabled. Like TTY mode itself, it can be turned on or off with an environment variable: **BC_PROMPT** (see the **ENVIRONMENT VARIABLES** section).

If the environment variable **BC_PROMPT** exists and is a non-zero integer, then the prompt is turned on when **stdin**, **stdout**, and **stderr** are connected to a TTY and the **-P** and **--no-prompt** options were not used. The read prompt will be turned on under the same conditions, except that the **-R** and **--no-read-prompt** options must also not be used.

However, if **BC_PROMPT** does not exist, the prompt can be enabled or disabled with the **BC_TTY_MODE** environment variable, the **-P** and **--no-prompt** options, and the **-R** and **--no-read-prompt** options. See the **ENVIRONMENT VARIABLES** and **OPTIONS** sections for more details.

SIGNAL HANDLING

Sending a **SIGINT** will cause bc(1) to do one of two things.

If bc(1) is not in interactive mode (see the **INTERACTIVE MODE** section), or the **BC_SIGINT_RESET** environment variable (see the **ENVIRONMENT VARIABLES** section), or its default, is either not an integer or it is zero, bc(1) will exit.

However, if bc(1) is in interactive mode, and the **BC_SIGINT_RESET** or its default is an integer and non-zero, then bc(1) will stop executing the current input and reset (see the **RESET** section) upon receiving a **SIGINT**.

Note that "current input" can mean one of two things. If bc(1) is processing input from **stdin** in interactive mode, it will ask for more input. If bc(1) is processing input from a file in interactive mode, it will stop processing the file and start processing the next file, if one exists, or ask for input from **stdin** if no other file exists.

This means that if a **SIGINT** is sent to bc(1) as it is executing a file, it can seem as though bc(1) did not respond to the signal since it will immediately start executing the next file. This is by design; most files that users execute when interacting with bc(1) have function definitions, which are quick to parse. If a file takes a long time to execute, there may be a bug in that file. The rest of the files could still be executed without problem, allowing the user to continue.

SIGTERM and **SIGQUIT** cause bc(1) to clean up and exit, and it uses the default handler for all other signals. The one exception is **SIGHUP**; in that case, and only when bc(1) is in TTY mode (see the **TTY MODE** section), a **SIGHUP** will cause bc(1) to clean up and exit.

COMMAND LINE HISTORY

bc(1) supports interactive command-line editing.

If bc(1) can be in TTY mode (see the **TTY MODE** section), history can be enabled. This means that command-line history can only be enabled when **stdin**, **stdout**, and **stderr** are all connected to a TTY.

Like TTY mode itself, it can be turned on or off with the environment variable **BC_TTY_MODE** (see the **ENVIRONMENT VARIABLES** section).

If history is enabled, previous lines can be recalled and edited with the arrow keys.

Note: tabs are converted to 8 spaces.

LOCALES

This bc(1) ships with support for adding error messages for different locales and thus, supports **LC_MESSAGES**.

SEE ALSO

dc(1)

STANDARDS

bc(1) is compliant with the IEEE Std 1003.1-2017 ("POSIX.1-2017") specification at https://pubs.opengroup.org/onlinepubs/9699919799/utilities/bc.html . The flags **-efghiqsvVw**, all long options, and the extensions noted above are extensions to that specification.

In addition, the behavior of the **quit** implements an interpretation of that specification that is different from all known implementations. For more information see the **Statements** subsection of the **SYNTAX** section.

Note that the specification explicitly says that bc(1) only accepts numbers that use a period (**.**) as a radix point, regardless of the value of **LC_NUMERIC**.

This bc(1) supports error messages for different locales, and thus, it supports **LC_MESSAGES**.

BUGS

Before version **6.1.0**, this bc(1) had incorrect behavior for the **quit** statement.

No other bugs are known. Report bugs at https://git.gavinhoward.com/gavin/bc .

AUTHORS

Gavin D. Howard <gavin@gavinhoward.com> and contributors.