-- --

IEEE P1003.2 Draft 11.2 – September 1991

## 4.3 `bc` — Arbitrary-precision arithmetic language

### 4.3.1 Synopsis

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

### 4.3.2 Description

The `bc` utility shall implement an arbitrary precision calculator. It shall take input from any files given, then read 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*, causing behavioral constraints described in the following subclauses.

### 4.3.3 Options

The `bc` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`-l` (The letter ell.) Define the math functions and initialize `scale` to 20, instead of the default zero. See 4.3.7.

### 4.3.4 Operands

The following operands shall be supported by the implementation:

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

### 4.3.5 External Influences

#### 4.3.5.1 Standard Input

See Input Files.

#### 4.3.5.2 Input Files

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

#### 4.3.5.3 Environment Variables

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.

#### 4.3.5.4 Asynchronous Events

Default.

### 4.3.6 External Effects

#### 4.3.6.1 Standard Output

The output of the `bc` utility shall be controlled by the program read, and shall
consist of zero or more lines containing the value of all executed expressions 2
without assignments. The radix and precision of the output shall be controlled by
the values of the `obase` and `scale` variables. See 4.3.7.

#### 4.3.6.2 Standard Error

Used only for diagnostic messages.

#### 4.3.6.3 Output Files

None.

### 4.3.7 Extended Description

#### 4.3.7.1 bc Grammar

The grammar in this subclause and the lexical conventions in the following subclause shall together describe the syntax for bc programs. The general conventions for this style of grammar are described in 2.1.2. A valid program can be represented as the nonterminal symbol program in the grammar. Any discrepancies found between this grammar and other descriptions in this subclause (4.3.7) shall be resolved in favor of this grammar.

```
%token  EOF NEWLINE STRING LETTER NUMBER

%token  MUL_OP
/*      '*', '/', '%'                           */

%token  ASSIGN_OP
/*      '=', '+=', '-=', '*=', '/=', '%=', '^=' */

%token  REL_OP
/*      '==', '<=', '>=', '!=', '<', '>'        */

%token  INCR_DECR
/*      '++', '--'                              */

%token   Define    Break    Quit    Length
/*      'define', 'break', 'quit', 'length'     */

%token   Return    For    If    While    Sqrt
/*      'return', 'for', 'if', 'while', 'sqrt'  */

%token   Scale    Ibase    Obase    Auto
/*      'scale', 'ibase', 'obase', 'auto'       */

%start  program

%%

program              : EOF
                     | input_item program
                     ;

input_item           : semicolon_list NEWLINE
                     | function
                     ;

semicolon_list       : /* empty */
                     | statement
                     | semicolon_list ';' statement
                     | semicolon_list ';'
                     ;

statement_list       : /* empty */
                     | statement
                     | statement_list NEWLINE
                     | statement_list NEWLINE statement
                     | statement_list ';'
                     | statement_list ';' statement
                     ;
```

```
statement              : expression
                       | STRING
                       | Break
                       | Quit
                       | Return
                       | Return ’(’ return_expression ’)’
                       | For ’(’ expression ’;’
                             relational_expression ’;’
                             expression ’)’ statement
                       | If ’(’ relational_expression ’)’ statement
                       | While ’(’ relational_expression ’)’ statement
                       | ’{’ statement_list ’}’
                       ;

function               : Define LETTER ’(’ opt_parameter_list ’)’
                             ’{’ NEWLINE opt_auto_define_list
                             statement_list ’}’
                       ;

opt_parameter_list     : /* empty */
                       | parameter_list
                       ;

parameter_list         : LETTER
                       | define_list ’,’ LETTER
                       ;

opt_auto_define_list   : /* empty */
                       | Auto define_list NEWLINE
                       | Auto define_list ’;’
                       ;

define_list            : LETTER
                       | LETTER ’[’ ’]’
                       | define_list ’,’ LETTER
                       | define_list ’,’ LETTER ’[’ ’]’
                       ;

opt_argument_list      : /* empty */
                       | argument_list
                       ;

argument_list          : expression
                       | argument_list ’,’ expression
                       ;

relational_expression  : expression
                       | expression REL_OP expression
                       ;

return_expression      : /* empty */
                       | expression
                       ;

expression             : named_expression
                       | NUMBER
                       | ’(’ expression ’)’
                       | LETTER ’(’ opt_argument_list ’)’
                       | ’-’ expression
                       | expression ’+’ expression                         1
```

```
                      | expression '-' expression                          1
                      | expression MUL_OP expression
                      | expression '^' expression
                      | INCR_DECR named_expression
                      | named_expression INCR_DECR
                      | named_expression ASSIGN_OP expression
                      | Length '(' expression ')'
                      | Sqrt '(' expression ')'
                      | Scale '(' expression ')'
                      ;

named_expression      : LETTER
                      | LETTER '[' expression ']'
                      | Scale
                      | Ibase
                      | Obase
                      ;
```

### 4.3.7.2 bc Lexical Conventions

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

(1) Except as noted, `bc` shall recognize the longest possible token or delimiter beginning at a given point.

(2) A comment shall consist of any characters beginning with the two adjacent characters /* and terminated by the next occurrence of the two adjacent characters */. Comments shall have no effect except to delimit lexical tokens.

(3) The character `<newline>` shall be recognized as the token `NEWLINE`.

(4) The token `STRING` shall represent a string constant; it shall consist of any characters beginning with the double-quote character (") and terminated by another occurrence of the double-quote character. The value of the string shall be the sequence of all characters between, but not including, the two double-quote characters. All characters shall be taken literally from the input, and there is no way to specify a string containing a double-quote character. The length of the value of each string shall be limited to {BC_STRING_MAX} bytes.

(5) A `<blank>` shall have no effect except as an ordinary character if it 1
appears within a `STRING` token, or to delimit a lexical token other than 1
`STRING`. 1

(6) The combination of a backslash character immediately followed by a 2
`<newline>` character shall delimit lexical tokens with the following 2
exceptions: 2

— It shall be interpreted as a literal `<newline>` in `STRING` tokens. 2

— It shall be ignored as part of a multiline `NUMBER` token. 2

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

```
NUMBER   : integer
         | '.' integer
         | integer '.'
         | integer '.' integer
         ;

integer  : digit
         | integer digit
         ;

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

(8) The value of a NUMBER token shall be interpreted as a numeral in the base specified by the value of the internal register `ibase` (described below). Each of the `digit` characters shall have the value from 0 to 15 in the order listed here, and the period character shall represent the radix point. The behavior is undefined if digits greater than or equal to the value of `ibase` appear in the token. (However, note the exception for single-digit values being assigned to `ibase` and `obase` themselves, in 4.3.7.3).

(9) The following keywords shall be recognized as tokens:

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

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

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

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

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

(12) If an = character, as the beginning of a token, is followed by a – character with no intervening delimiter, the behavior is undefined.

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

```
*    /    %
```

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

```
==    <=    >=    !=    <    >
```

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

```
++    --
```

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

`<newline> ( ) , + - ; [ ] ^ { }` 1

(17) The token `EOF` shall be returned when the end of input is reached.

#### 4.3.7.3 bc Operations

There are three kinds of identifiers: ordinary identifiers, array identifiers, and function identifiers. All three types consist of single lowercase letters. Array identifiers shall be followed by square brackets (`[ ]`). An array subscript is required except in an argument or auto list. Arrays are singly dimensioned and can contain up to {BC_DIM_MAX} elements. Indexing begins at zero so an array is indexed from 0 to {BC_DIM_MAX}−1. Subscripts shall be truncated to integers. Function identifiers shall be followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict.

Table 4-3 summarizes the rules for precedence and associativity of all operators. Operators on the same line shall have the same precedence; rows are in order of decreasing precedence.

**Table 4-3 – bc Operators**

| Operator | Associativity |
|---|---|
| ++, −− | not applicable |
| unary − | not applicable |
| ^ | right to left |
| *, /, % | left to right |
| +, binary − | left to right |
| =, +=, −=, *=, /=, %=, ^= | right to left |
| ==, <=, >=, !=, <, > | none |

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

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

The internal registers `scale`, `ibase`, and `obase` are all named expressions. The scale of an expression consisting of the name of one of these registers shall be zero; values assigned to any of these registers shall be truncated to integers. The `scale` register shall contain a global value used in computing the scale of expressions (as described below). The value of the register `scale` shall be limited to 0 ≤ `scale` ≤ {BC_SCALE_MAX} and shall have a default value of zero. The `ibase` and `obase` registers are the input and output number radix, respectively. The value of `ibase` shall be limited to

2 ≤ `ibase` ≤ 16

The value of `obase` shall be limited to

2 ≤ `obase` ≤ {BC_BASE_MAX}

When either `ibase` or `obase` is assigned a single `digit` value from the list in 4.3.7.2, the value shall be assumed in hexadecimal. (For example, `ibase=A` sets to base ten, regardless of the current `ibase` value.) Otherwise, the behavior is undefined when digits greater than or equal to the value of `ibase` appear in the input. Both `ibase` and `obase` shall have initial values of 10.

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

For all values of `obase` specified by this standard, numerical values shall be output as follows:

(1) If the value is less than zero, a hyphen (`-`) character shall be output.

(2) One of the following shall be output, depending on the numerical value:

— If the absolute value of the numerical value is greater than or equal to one, the integer portion of the value shall be output as a series of digits appropriate to `obase` (as described below). The most significant nonzero digit shall be output next, followed by each successively less significant digit.

— If the absolute value of the numerical value is less than one but greater than zero and the scale of the numerical value is greater than zero, it is unspecified whether the character 0 is output.

— If the numerical value is zero, the character 0 shall be output.

(3) If the scale of the value is greater than zero, a period character shall be output, followed by a series of digits appropriate to `obase` (as described below) representing the most significant portion of the fractional part of the value. If *s* represents the scale of the value being output, the number of digits output shall be *s* if `obase` is 10, less than or equal to *s* if `obase` is greater than 10, or greater than or equal to *s* if `obase` is less than 10. For `obase` values other than 10, this should be the number of digits needed to represent a precision of $10^s$.

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

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

which represent the values zero through fifteen, respectively.

For bases greater than 16, each ‘‘digit’’ shall be written as a separate multidigit decimal number. Each digit except the most significant fractional digit shall be preceded a single `<space>` character. For bases from 17 to 100, `bc` shall write two-digit decimal numbers; for bases from 101 to 999, three-digit decimal strings, and so on. For example, the decimal number 1024 in base 25 would be written as:

```
Δ01Δ15Δ24
```

in base 125, as:

```
Δ008Δ024
```

Very large numbers shall be split across lines with 70 characters per line in the POSIX Locale; other locales may split at different character boundaries. Lines that are continued shall end with a backslash (\).

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

The result of `length`(*expression*) shall be the total number of significant decimal digits in the expression. The scale of the result shall be zero.

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

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

The sequence `(` *expression* `)` shall be an expression with the same value and scale as *expression*. The parentheses can be used to alter the normal precedence.

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

–*expression*
: The result shall be the negative of the *expression*. The scale of the result shall be the scale of *expression*.

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

++*named-expression*
: The named expression shall be incremented by one. The result shall be the value of the named expression after incrementing.

– –*named-expression*
: The named expression shall be decremented by one. The result shall be the value of the named expression after decrementing.

*named-expression* ++

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

*named-expression* −−

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

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

*expression* ^ *expression*

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

```
if b >= 0 min(a * b, max(scale, a))      2
if b <  0 scale                          2
```

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

*expression* * *expression*

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

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

*expression* / *expression*

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

*expression* % *expression*

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

(1) Compute `a/b` to current scale.

(2) Use the result to compute

```
a - (a / b) * b
```

to scale

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

The scale of the result shall be

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

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

*expression* + *expression*

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

*expression* − *expression*

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

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

*named-expression* = *expression*
: This expression results in assigning the value of the expression on the right to the named expression on the left. The scale of both the named expression and the result shall be the scale of *expression* .

The compound assignments forms

*named-expression* *<operator>*= *expression*

shall be equivalent to:

*named-expression* = *named-expression* *<operator>* *expression*

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

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

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

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

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

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

*expression1* == *expression2*
: The relation shall be true if the values of *expression1* and *expression2* are equal.

*expression1* != *expression2*
: The relation shall be true if the values of *expression1* and *expression2* are unequal.

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

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

When a statement is an expression, unless the main operator is an assignment, execution of the statement shall write the value of the expression followed by a <newline> character.

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

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

```
input_item : semicolon_list NEWLINE
```

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

In an `if` statement [`if` (*relation*) *statement* ] the *statement* shall be executed if the relation is true.

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

A `for` statement [`for` (*expression* ; *relation*; *expression* ) *statement* ] shall be the same as:

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

All three expressions shall be present.

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

The `auto` statement [`auto` *identifier* [,*identifier* ] . . . ] shall cause the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers shall be specified by following the array name by empty square brackets. The `auto` statement shall be the first statement in a function definition.

A `define` statement:

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

defines a function named *LETTER*. If a function named *LETTER* was previously defined, the `define` statement shall replace the previous definition. The expression

*LETTER* ( *opt_argument_list* )

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

The `return` statements [`return` and `return`(*expression*)] shall cause termination of a function, popping of its auto variables, and specifies the result of the function. The first form shall be equivalent to `return(0)`. The value and scale of an invocation of the function shall be the value and scale of the expression in parentheses.

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

The following functions shall be defined when the `−l` option is specified:

| | |
|---|---|
| `s` ( *Expression* ) | Sine of argument in radians |
| `c` ( *Expression* ) | Cosine of argument in radians |
| `a` ( *Expression* ) | Arctangent of argument |
| `l` ( *Expression* ) | Natural logarithm of argument |
| `e` ( *Expression* ) | Exponential function of argument |
| `j` ( *Expression* , *Expression* ) | Bessel function of integer order |

The scale of an invocation of each of these functions shall be the value of the `scale` register when the function is invoked. The behavior is undefined if any of these functions is invoked with an argument outside the domain of the mathematical function.

### 4.3.8 Exit Status

The `bc` utility shall exit with one of the following values:

| | |
|---|---|
| 0 | All input files were processed successfully. |
| *unspecified* | An error occurred. |

### 4.3.9 Consequences of Errors

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

In an interactive invocation of `bc`, the utility should print an error message and recover following any error in the input. In a noninteractive invocation of `bc`, invalid input causes undefined behavior.

### 4.3.10 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
      }
}
```

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

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

**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 *file1 file2* is used most often when at least *file1* contains data/function declarations/initializations. Having bc continue with prerequisite files missing is probably not 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).

## 4.4 `cat` — Concatenate and print files

### 4.4.1 Synopsis

`cat [–u]` [*file* ...]

### 4.4.2 Description

The `cat` utility reads files in sequence and writes their contents to the standard output in the same sequence.

### 4.4.3 Options

The `cat` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`–u` Write bytes from the input file to the standard output without delay as each is read.

### 4.4.4 Operands

The following operand shall be supported by the implementation:

*file* A pathname of an input file. If no *file* operands are specified, the standard input is used. If a *file* is –, the `cat` utility shall read from the standard input at that point in the sequence. The `cat` utility shall not close and reopen standard input when it is referenced in this way, but shall accept multiple occurrences of – as a *file* operand.

### 4.4.5 External Influences

#### 4.4.5.1 Standard Input

The standard input is used only if no *file* operands are specified, or if a *file* operand is –. See Input Files.

#### 4.4.5.2 Input Files

The input files can be any file type.

#### 4.4.5.3 Environment Variables

The following environment variables shall affect the execution of `cat`:

**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).

**LC_MESSAGES** This variable shall determine the language in which messages should be written.

#### 4.4.5.4 Asynchronous Events

Default.

### 4.4.6 External Effects

#### 4.4.6.1 Standard Output

The standard output shall contain the sequence of bytes read from the input file(s). Nothing else shall be written to the standard output.

#### 4.4.6.2 Standard Error

Used only for diagnostic messages.

#### 4.4.6.3 Output Files

None.

### 4.4.7 Extended Description

None.

### 4.4.8 Exit Status

The `cat` utility shall exit with one of the following values:

- 0 All input files were output successfully.
- >0 An error occurred.

### 4.4.9 Consequences of Errors

Default.

### 4.4.10 Rationale. *(This subclause is not a part of P1003.2)*

**Examples, Usage**

Historical versions of the `cat` utility include the options `−e`, `−t`, and `−v`, which permit the ends of lines, `<tab>`s, and invisible characters, respectively, to be rendered visible in the output. The working group omitted these options because they provide too fine a degree of control over what is made visible, and similar output can be obtained using a command such as:

```
sed -n -e 's/$/$/' -e l pathname
```

The `−s` option was omitted because it corresponds to different functions in BSD and System V-based systems. The BSD `−s` option to squeeze blank lines will be handled by `more -s` in the UPE. The System V `−s` option to silence error messages can be accomplished by redirecting the standard error. An alternative to `cat −s` is the following shell script using `sed`: