The bc programming language

- Shell Commands: bc
- What Good Is bc?

Shell Commands: bc

- ❒ **This command invokes an interpreting calculator that can do calculations in any number base from 2 to 16 (although it is intended for bases 2, 8, 10, and 16 primarily)**

Syntax

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

Where

- ✗ **i** indicates interactive mode
  - ➢ Note: in non-interactive mode, if an error occurs, a message is issued and the **bc** interpreter is terminated; in interactive mode, if an error occurs, a message is issued and operation continues
  - ➢ Note: in interactive mode, there is a prompt (the colon character (**:**) to indicate bc is waiting for input)
- ✗ **l** (lower case alpha letter EL) indicates **bc** should load a library of standard functions before continuing
- ✗ *file* is a file containing **bc** instructions
  - ➢ If no file is specified, **bc** reads from stdin until it encounters the **quit** instruction
  - ➢ Notice you can have multiple files, and they are processed in the order specified on the bc statement

Shell Commands: bc, continued

- **At its simplest level,** bc **reads in a number and displays it back**
  - **Behind the scenes, you need to realize that bc maintains two variables that effect all operations:** ibase **and** obase
    - **ibase** is the number base you are entering numbers in
      - When you key in a number, **bc** assumes it is in the number base of ibase
      - You can change ibase by an assignment instruction; *e.g.*: ibase = 3
        - If an input number is not correct format for ibase (for example, entering 177 when ibase is set to 3; (you're not supposed to have digits greater than the ibase - 1), **bc** assumes each excessive digit is base 10 and each OK digit is base ibase)
        - In our example, 177 would be interpreted as 1*9 + 7*3 + 7*1 = 9 + 21 + 7 = 37
      - Internally, all numbers are stored as strings of numeric digits and converted to internal formats each time they are used
        - The interpretation of these strings depends on the current setting of ibase
    - **obase** is the number base to be used for output displays
      - **bc** always converts results to obase before displaying
      - You can change obase by an assignment instruction; *e.g.*: obase = 7
      - As a handy converter, then, key in a number and press <Enter> and the number converted to obase is displayed

Shell Commands: bc, continued

- **You can enter simple calculations**
  - **4+3**
    - **bc** returns this value on the next stdout line:
      7
    - **bc** always keeps the last value in the special variable **.** (dot)
- **Numbers in** bc **are composed of: optional sign (+ or -) followed by zero or more digits, optional decimal point (.) followed by zero or more digits; numbers can be arbitrarily long**
  - **There must be at least one digit either before or after the decimal point, if present**
  - **Note there cannot be any commas, although there may be spaces**
    - 123456 is valid
    - 123 456 is valid
    - 123,456 is invalid
  - **Uppercase A-F represent hex digits, with decimal values of 10-15, as usual**
    - Note: use "ibase = A" to reset ibase to base 10 if necessary

Shell Commands: bc, continued

❒ bc **allows the use of variables (also called** identifiers **in the literature)**

- **Variable names are case sensitive and composed of any number of letters, digits, and underscore (_) characters**

  - ✗ The first of which must be a lower-case letter

- **Variable names are global for the duration of the** bc **run**

❒ **You can create your own** bc functions **in the** bc **programming language (details in a bit)**

- **Function names have the same rule as variable names**

- **A reference to a function must always be followed by a set of parentheses containing zero or more arguments separated by commas (*e.g.*: sqrt(in_num))**

❒ **You can create a variable in** bc **that is an** array**, a list of elements**

- **Array names follow the same rules as variable names; only one dimensional arrays are supported**

- **A reference to an array must always be followed by brackets with a subscript (*e.g.*: scores[tot_num])**

- **You do not need to declare the size of an array;** bc **creates elements dynamically as needed (initialized to 0)**

Shell Commands: bc, continued

❒ **You can enter assignment instructions**

- **a=4** — **defines global variable a and gives it an initial value of 4**

- **a=a+b** — **undefined variables (b in this case) are given initial value of 0; you can perform basic numeric operations using variables or numeric literals**

  ✗ a = a - 5 — notice spaces are OK, but not required

  ✗ b = b * a

  ✗ c = b ^ a — exponentiation

  ✗ sum = sum / count

- **Now we must introduce an important element of** bc: <u>scale</u>

- **Unlike** let**, which only works with integers,** bc **works with numbers of any precision and scale**

Shell Commands: bc, continued

- ❒ **The** scale **is the number of digits to be retained to the right of the decimal point when doing calculations**

  - ♦ bc **maintains a built-in variable called** <u>scale</u> **to specify this**

  - ♦ **The default value for** <u>scale</u> **is 0**

  - ♦ **If you invoke** bc **with the -l option,** <u>scale</u> **is set to 20**

  - ♦ **You can explicitly change** <u>scale</u> **with an assignment instruction:** *e.g.*: **scale = 12**

  - ♦ **Each numeric value has an implicit scale when entered; the** <u>scale</u> **setting determines the scale of a calculation result**

  - ♦ **The maximum value for** <u>scale</u> **is maintained in the configuration variable BC_SCALE_MAX**

    - ✗ The IBM-supplied value for this is currently 32767

  - ♦ **Long numbers are output by** bc **with a maximum of 70 characters per line; if a number is longer than a line, a backslash (\) is appended to the display to indicate the value is continued on the next line (all lines are displayed at once)**

  - ♦ **Internal calculations are always done in decimal (base 10)**

    - ✗ So the number of places after the decimal point are dictated by <u>scale</u> when numbers are expressed in decimal form

Shell Commands: getconf

❒ **As a digression,** configuration variables **are variables set by IBM and possibly modified by your staff at installation time that specify the limits, defaults, and sources of information for your installation**

- ✗ You can view all your current configuration variables by issuing the shell command **getconf -a**

❒ **The relevant configuration variables here are:**

- **BC_BASE_MAX - maximum number supported for obase**
- **BC_DIM_MAX - maximum number of elements in an array**
- **BC_SCALE_MAX - maximum scale supported**
- **BC_STRING_MAX - maximum number of characters in a** bc **instruction**

Shell Commands: bc, continued

- **The implementation of** bc **on UNIX System Services includes a number of extensions to the standards, which will be noted as encountered**

  - **For example, in the standards, identifier names are only one character long**

Shell Commands: bc, continued

❒ **The** scale **used when doing calculations is determined this way:**

- **Addition and subtraction of two operands, A + B or A - B**

  max(scale(A), scale(B))

- **Multiply A * B**

  min(<u>scale(A) + scale(B)</u>, max(<u>scale</u>, <u>scale(A)</u>, <u>scale(B)</u>)

- **Divide A / B**

  scale

- **Remainder: A % B**

  First, calculate A / B using current scale; then calculate remainder as A - (A / B) * B using scale of max(scale+scale(B), scale(A))

- **Exponentiation: A * B (B must be an integer)**

  min(scale(A) * abs(B), max(scale, scale(A))

Shell Commands: bc, continued

❐ **When you enter a line into** bc**, if an assignment is involved, it changes the value of the target variable, of course**

- **The right hand component of assignment instructions can involve numbers, variables, calculations, and logical expressions**

- **But even lines that perform calculations without an assignment produce a** result

- **For example, entering** a * 2 **produces a result that is twice the value in variable** a**, but no variable is changed**

- **By way of contrast, the** increment **and** decrement **operations can be entered without being part of an assignment and a result is produced and a variable is changed**

++var	adds one to var; result is new value of var
var++	adds one to var; result is old value of var
--var	subtracts one from var, result is new value of var
var--	subtracts one from var, result is old value of var

Shell Commands: bc, continued

- bc **supports the traditional short hand assignment operators of C; in particular...**

  - ✗ var **^=** *value* is the same as var = var ^ value
  - ✗ var *= value is the same as var = var * value
  - ✗ var /= value is the same as var = var / value
  - ✗ var %= value is the same as var = var % value
  - ✗ var += value is the same as var = var + value
  - ✗ var -= value is the same as var = var - value

Shell Commands: bc, continued

- bc **also supports relational operators: these return 1 if true or 0 if false:**

  - ✗ value_1 **==** value_2 returns 1 if and only if the values are equal

  - ✗ value_1 <= value_2 returns 1 if and only if the first value is less than or equal to the second value

  - ✗ value_1 >= value_2 returns 1 if and only if the first value is greater than or equal to the second value

  - ✗ value_1 != value_2 returns 1 if and only if the values are not equal

  - ✗ value_1 < value_2 returns 1 if and only if the first value is less than the second value

  - ✗ value_1 > value_2 returns 1 if and only if the first value is greater than the second value

Shell Commands: bc, continued

- **Also, logical operators:**
  - ✗ A && B returns 1 if A is true (nonzero) and B is true; note that if A is not true, B is not even evaluated
  - ✗ A || B returns 1 if A is true or B is true; note that if A is true, B is not even evaluated
  - ✗ !A - returns 1 if A is false, 0 if A is true
  - ✗ -A is a unary minus (takes the negative of a number)
  - ✗ (A) - indicates that expression A should be evaluated before any other operations are performed in the instruction
    - ➢ Example: if you enter **a = b +5**, bc will not display anything; if you enter **a = (b+5)**, bc will calculate b+5, display the result, then place the result into a

- **Complex operations may be constructed in the usual fashion**
  - ♦ **Including using parentheses to explicitly indicate the order of precedence**
    - ✗ Note that bc's default order of precedence is not the same as C's in every case, so explicitly using parentheses in complex expressions is always a good idea

Shell Commands: bc, continued

❒ **As indicated earlier, not only can** bc **be interactive, but you may also construct a file (script) of** bc **instructions**

bc **instructions include**

♦ expressions **-** bc **calculates the value**

♦ assignments **-** bc **calculates a value and puts result into a variable**

♦ comments **- begin with /*, end with */ (can cross line boundaries)**

✗ Also, a pound sign (#) can be used to indicate the rest of the line is a comment (this is an extension to the standard)

♦ quit **- terminate** bc **(if no** quit **is encountered in a script, you remain in** bc **even after the end of the script is reached)**

♦ conditional instructions **(if - discussed shortly)**

♦ looping instructions **(for, while, break - discussed shortly)**

♦ void *expression* **- calculate the value of** *expression* **but do not display it; useful, for example, with increments and decrements: void ++able**

♦ sh *statement* **- send a [single line]** *statement* **to the shell for execution**

♦ miscellaneous **(braces, print, sequence) - discussed shortly**

Shell Commands: bc, continued

❒ Some quick points

- quoted strings - bc **simply displays the string, with no newline character following**

  - ✗ So, for example, the lines

```
b=5
a = 3.14 * (b^2)
"The result is "
a
```

  - ✗ will display

```
The result is 78.50
```

- **The** semi-colon **can be used to separate multiple instructions on a single line, so an equivalent script would be:**

```
b=5
a = 3.14 * (b^2)
"The result is " ; a
```

- **Or even:**

```
b=5 ; a = 3.14 * (b^2) ; "The result is " ; a
```

  - ✗ This represents the sequence construct

Shell Commands: bc, continued

❒ The print instruction

Syntax

print **[***expression***] [,** *expression***] [...]**

Where

- **If there are no** *expressions***, a blank line is printed**
- **Each** *expression* **may be a quoted string, a numeric literal, a numeric variable, or an arithmetic expression**
  - ✗ Expressions must be separated by commas
- **All** *expression***s on one** print **instruction are displayed on a single line**
- **A single space is displayed between adjacent numbers, but not between numbers and strings (so be sure to include spaces as necessary in quoted strings)**
- **If the last argument is null, subsequent output continues on the same line**
- **The** print **instruction of the** bc **command is an extension to the standards**

Shell Commands: bc, continued

❒ The if instruction in bc

Syntax

if (*relation_test***)** *instruction*$_1$ **[else** *instruction*$_2$ **]**

- **The parentheses are needed as shown**

- **If** *relation_test* **is true,** *instruction1* **is executed, otherwise** *instruction*$_2$ **is executed**

- *instruction*$_1$ **and** *instruction*$_2$ **can be simple instructions, for example:**

```
if (score[sub] == 0) "score not used"
```

```
if (score[sub] == 0) "score not used"
else total += score[sub]
```

Shell Commands: bc, continued

❒ **The** if **instruction in** bc**, continued**

- **If you want to perform multiple instructions on the** if **or the** else **portion, you must enclose the instructions in braces**

- **And, the opening brace must be on the same line as the** if **or** else **clause; for example**

```
if (score[sub] == 0) { "score not used"
} else { print "score[",sub,"] = ", score[sub] ; total += score[sub] }
```

- **or, perhaps better:**

```
if (score[sub] == 0) { "score not used"
} else {
      print "score[",sub,"] = ", score[sub]
      total += score[sub]
       }
```

- **Note, too, that if the** else **portion uses braces, the** if **must also, and the closing brace of the** if **must be on the same line as the** else

Shell Commands: bc, continued

❒ **The** for **instruction in** bc

♦ **This instruction allows for looping - for repeating a set of instructions as long as some condition / relation remains true**

Syntax

for (*init_expr***;** *relation***;** *end_expr***)** *instruction*

Where

♦ *init_expr* **is some expression that initializes a variable**

♦ *relation* **is any of the relation tests we've already seen**

✗ Typically checking the initialized variable for a limit or boundary

♦ *end_expr* **is an expression that indicates what to do after executing** *instruction* **and before testing** *relation*

✗ Typically updating the initialized and tested variable

♦ *instruction* **is a single instruction or a braces-bound series of instructions**

♦ **Although this is similar to the C construct, unlike C all three parts must be explicitly present for the** bc **version**

Shell Commands: bc, continued

❐ The for instruction in bc, continued

Examples

```
for (i = 0; i<=no_scores; ++i) total += scores[i]
print "total scores = ",total
```

```
for (i = 0; i <= no_scores; ++i)   {
      print "scores[",i,"] = ",scores[i]
      total += scores[i]
                                          }
print "total scores = ",total
```

```
for (i = 0; i <= no_scores; ++i)   {
      if (scores[i] == 0)  {
        print "score of zero not used. i = ",i
        } else   {
            print "scores[",i,"] = ",scores[i]
            total += scores[i]
                  }
                                          {
print "total scores = ",total
```

♦ **Notice that whenever you have braces-bounded instructions, the open brace has to be on the same line as the starting instruction (**if **and** else **as seen before,** for **as seen here,** while **as we shall see shortly)**

Shell Commands: bc, continued

❒ **The** while **instruction in** bc

- **This instruction also allows for looping**

  ✗ But it doesn't include the initialization and loop stepping logic explicitly - you have to add that

<u>Syntax</u>

while (*relation***)** *instruction*

- **We show the same code we used with for using while this time on the next page**

Shell Commands: bc, continued

❒ **The** while **instruction in** bc**, continued**

Examples

```
i = 0
while (i <= no_scores) {total += scores[i]; ++i}
print "total scores = ",total
```

```
i = 0
while (i <= no_scores)             {
      print "scores[",i,"] = ",scores[i]
      total += scores[i]
      ++i
                                   }
print "total scores = ",total
```

```
i = 0
while (i <= no_scores)             {
      if (scores[i] == 0)  {
        print "score of zero not used. i = ",i
        } else   {
            print "scores[",i,"] = ",scores[i]
            total += scores[i]
                  }
      ++i
                                   }
print "total scores = ",total
```

- **Now, these don't act exactly like the earlier code using for**
  - ✗ The explicit ++i incrementing displays the result value each time, where the embedded ++i in the for does not

Shell Commands: bc, continued

❒ **The** while **instruction in** bc**, continued**

To get the examples to produce the exact same output:

```
i = 0
while (i <= no_scores) {total += scores[i]; i = i + 1}
print "total scores = ",total
```

```
i = 0
while (i <= no_scores)             {
      print "scores[",i,"] = ",scores[i]
      total += scores[i]
      i = i + 1
                                   }
print "total scores = ",total
```

```
i = 0
while (i <= no_scores)             {
      if (scores[i] == 0)  {
        print "score of zero not used. i = ",i
        } else   {
            print "scores[",i,"] = ",scores[i]
            total += scores[i]
                  }
      i = i + 1
                                   }
print "total scores = ",total
```

✗ You have to either convert **++i** to **i = i +1** or code as **void ++i**

Shell Commands: bc, continued

❒ **The** break **instruction in** bc

- **Now, suppose you want to jump out of a** for **or** while **loop early - the** break **instruction is for you**

- **Using the most complex of our previous sets of examples, here's how you would stop your calculations when you found, say, a score of zero:**

```
for (i = 0; i <= no_scores; ++i)   {
      if (scores[i] == 0)  {
        print "score of zero not used. i = ",i
        break
        } else   {
            print "scores[",i,"] = ",scores[i]
            total += scores[i]
                  }
                                          {
print "total scores = ",total
```

```
i = 0
while (i <= no_scores)              {
      if (scores[i] == 0)  {
        print "score of zero not used. i = ",i
        break
        } else   {
            print "scores[",i,"] = ",scores[i]
            total += scores[i]
                  }
      i = i + 1
                                          }
print "total scores = ",total
```

Shell Commands: bc, continued

❒ **Functions in** bc

- **There are nine built-in functions supplied with** bc**:**

  - ✗ **length(***expression***)** - returns number of decimal digits (including before and after the decimal point) in *expression*

    - ➢ For example, length(2593.88768) is 9

  - ✗ **scale(***expression***)** - returns the scale of *expression*

    - ➢ For example, scale(2593.88768) is 5

    - ➢ Note: length(*expression*) - scale(*expression*) tells you how many decimal digits to the left of the decimal point in *expression*

  - ✗ **sqrt(***expression***)** - calculate the square root of *expression*; scale of result is max(scale, scale(*expression*))

Shell Commands: bc, continued

❐ **Functions in** bc, **continued**

- **There are nine built-in functions supplied with** bc, **continued:**

- **These functions are available only if you invoke** bc **with the l (lowercase letter el) flag**

  ✗ **arctan(***expression***)** or **a(***expression***)** or **atan(***expression***)** - return the arctangent of *expression*

  ✗ **bessel(***integer,expression***)** or **j(***integer,expression***)** or **jn(***integer,expression***)** - return the Bessel function of *expression* with order *integer*

  ✗ **cos(***expression***)** or **c(***expression***)** - return the cosine of *expression*

  ✗ **exp(***expression***)** or **e(***expression***)** - return the exponential of *expression* (that is, $\mathbf{e}^{\text{expression}}$)

  ✗ **ln(***expression***)** or **l(***expression***)** or **log(***expression***)** - return the natural logarithm of *expression*

  ✗ **sin(***expression***)** or **s(***expression***)** - return the sine of *expression*

Shell Commands: bc, continued

❒ **User functions in** bc

- **You can also create your own functions in** bc

- **A function definition ...**

  ✗ Begins with a define statement that names any parameters and specifies the body in braces

  ✗ The body contains any of the instructions we've discussed in this section plus possibly two other: **auto** and **return**

- **The resulting function is invoked through a function reference in one or more** bc **instructions**

- **Usually, function definitions are included in the body of a** bc **script file since there is no include or copy type capability:**

```
define func_x(..) {
.
.
.     }
define func_y(...) {
.
.
.     }

code, including function references
.
.
.
```

Shell Commands: bc, continued

❒ User functions in bc, continued

- **The** define **instruction specifies the name of the function and a list of** <u>parameters</u> **to be passed...**

```
define   calc_int(prin,rate)
```

- **Followed by the definition of the function in braces:**

```
define   calc_int(prin,rate)   {
      scale = 2
      int = prin * rate
      return(int)
                               }
```

<u>**Notes so far**</u>

- *prin* **and** *rate* **are parameters, names used only in the function definition itself**

- **The** return **statement passes back a single value that replaces the function reference at run time, for example:**

```
interest = calc_int(value1,value2)
print "Interest is", interest
```

- **or even just:**

```
print "Interest is", calc_int(value1,value2)
```

Shell Commands: bc, continued

- ❒ **User functions in** bc**, continued**

  - ♦ **Generally speaking, variables in** bc **code are global**

    - ✗ So variables declared in function definitions are known inside and outside the functions

    - ✗ And variables declared outside function definitions are also known inside and outside the functions

  - ♦ **You use parameters so you can call a function using different variables**

  - ♦ **In a function you can create a local variable by using the** auto **instruction:**

Syntax

auto *variable_name* **[,** *variable_name* **...]**

- ✗ All named variables are known only in the function body

- ✗ If there is a variable outside the function body of the same name, they are different variables

- ✗ If you have an auto instruction, it must come first in the body of the function

- ✗ Variables declared as auto are initialized to zero on every entry to the function

Shell Commands: bc, continued

❒ **User functions in** bc**, continued**

- **Both parameters and auto variables may be arrays**

  ✗ Simply specify with brackets and no subscript:

```
define   avg_score(scores[], no_scores) {
      auto i,   total
      for (i = 0; i <= no_scores; ++i) {
            total += scores[i] }
      return(total/no_scores)
                                                     }
```

and

```
define   name_reverse(in_names[], num_names) {
      auto   work_table[]
      .
      .
      .
            }
```

- **Note that if a function definition does not have a** return **instruction, the function returns on encountering the last instruction in the definition, with a function value of 0**

What Good Is bc?

❒ **The** bc **statement is handy for doing interactive calculations**

- **But suppose you would like to have a** bc **script invoked from a shell script**

- **It turns out this is not simple**

- **And yet you would like to be able to do something like this:**

  ✗ Query the person running the shell script for some values

  ✗ Call bc to calculate the values and report the result, perhaps using user-defined functions

❒ **The problem is**

- bc **instructions cannot reference shell variables, even if the variables are exported**

❒ **You can use** echo **and** read **inside a** bc **script (using the** sh **... instruction), but the values returned can still not be grabbed in the** bc **script!**

What Good Is bc?, continued

❒ **The way to take advantage of a** bc **script from a shell script is**

- **Use** echo **and** read **in the usual fashion to obtain input values in shell variables**

- **Use redirection to create a file containing these values assigned to** bc **variables**

- **Have a** bc **script written that processes the values and, perhaps, displays the results**

❒ **We show an example on the following pages**

What Good Is bc?, continued

❒ **First, here is a pre-written bc script, called <u>bcioc</u>, to process some variables:**

```
d = a / b
print a," divided by ",b," with scale ",scale," is ",d
quit
```

❒ **Second, here is a <u>shell script</u> (not a bc script) to gather values, put them into a bc script file, and then run the file followed by the bcioc file:**

```
# This is a shell script to request two numbers
# and pass those numbers to the bc interpreter
# for doing a calculation

read var1?"Enter numerator (top number) "
read var2?"Enter denominator (bottom number) "
read var3?"Enter scope (number of decimal places) "

cat <<eee > bcfileo
   a = $var1
   b = $var2
   scale = $var3
eee

bc bcfileo bcioc
```

- **Suppose this shell script is named <u>bciom</u>**

What Good Is bc?, continued

- **Now, when** bciom **is run, the user is prompted for three variables, and these are placed into var1, var2, and var3**

  - **Next a file is created called** bcfileo **that contains three assignments - note that the shell will do variable substitution of the $\$var_n$ symbols before the lines are written out**

  - **Finally,** bc **is invoked and it runs** bcfileo **followed by** bcioc

  - **Here's an example of running the bciom script:**

```
Enter numerator (top number) 23
Enter denominator (bottom number) 4
Enter scope (number of places to keep in decimal portion) 6
23 divided by 4 with scale 6 is 5.750000
```

- **bc is a rich programming language and can be explored in many ways**

This page intentionally left almost blank.