bc Desk Calculator Language

This tutorial introduces **bc**, the calculator language for COHERENT. If you have not used **bc** before, this tutorial will introduce you to its features and functions. If you are familiar with **bc**, you can use it as a reference.

bc is a language that can calculate to high precision. It automatically adjusts the number of digits in a number to represent it correctly. It is like having a powerful calculator at your fingertips.

Entry and Exit

The **bc** calculator for COHERENT is easy to use. Whenever you wish to invoke **bc** all you do is type its name (**bc**), followed by a stroke of the carriage return key. When you are finished using the calculator and wish to exit, just type the word 'quit' or **<ctrl-D>**. **bc** exits and returns control to COHERENT.

Example of Simple Use

bc performs calculations on formulas that you type into it. The formulas are laid out as you would naturally write them. For example, to invoke **bc**, have it add 2+2, and then exit, type:

```
bc
2 + 2
```

bc replies:

```
4
```

Then, leave **bc** by typing:

```
quit
```

bc is an arbitrary precision calculator: the number of digits carried by **bc** depends upon the requirements of the calculation, and is automatically expanded by **bc**. Thus, **bc** will never overflow. The number of digits it carries is limited only by the amount of available computer memory. For example, invoke **bc** and then try this calculation:

```
2^500
```

The circumflex '^' character signifies a superscript; thus, we are asking **bc** to raise 2 to the 500th power. After a moment, **bc** will reply:

```
327339060789614187001318969682759915221664\
204604306478948329136809613379640467455488\
327009232590415715088668412756007100921725\
6545885393053328527589376
```

You have probably already noticed one nice thing about this calculator: you don't have to include a print statement as part of your command, because **bc** automatically prints the results onto your terminal screen. When **bc** sees any expression, like "2+2" or "37-7", it prints the result.

bc provides the common arithmetic operators for add, subtract, multiply, and divide, as illustrated by the following commands:

```
7 + 5
7 - 5
7 * 5
7 / 5
```

bc also provides the remainder operator '**%**'. To get a sense of how it works, type:

```
7 % 5
5 % 7
```

Here, **bc** prints the *remainder* of the first number divided by the second; in the case of the first example, **bc** prints 2, and in the second prints 5. As you saw above, **bc** also includes the exponentiation operator '**^**'.

With **bc**, you can also enter numbers with fractional parts. Type the following to illustrate:

```
        9.999 * 9.999
```

bc replies:

```
        99.980
```

You can save temporary calculations or repeated constants in *variables*. The following example shows you first how to define variables, and second how to use them:

```
        a = 1.1
        b = 2.2
        a
        b
        a * b
```

Variable names can be longer than one letter.

The basic calculations in the above examples show only part of what **bc** can do. The following section describes simple statements — the assignment of variables and abbreviations — that allow you to perform complex calculations easily.

Simple Statements

Although you can use **bc** as a simple calculator for manipulating numbers, you can take advantage of its greater power by using *variables*. Variables, as noted above, store parts of calculations or constants that you will use repeatedly in calculations. Variable names are simply "words" that you make up. Here are some examples of possible variable names:

```
        a
        b
        totaltaxesdue
        ratio
```

To use variables, simply give them a value, use them in a calculation in place of a number, or print them out.

To see how a variable can save you repetitive typing, and protect you from possible errors, invoke **bc** and type the following:

```
        x = 9.999
        x
        x * x
        x = x * x
        x
```

The following gives the example with **bc**'s replies *in italics*:

```
        x = 9.999
        x
        9.999
        x * x
        99.980
        x = x * x
        x
        99.980
```

bc did not reply to the assignment statements **x=9.999** and **x=x*x**. However, it did print the value of **x** when requested, and the results of arithmetic using **x**.

Calculations executed with hand-held calculators, with programming languages like C, or with **bc** often use the following formula:

```
        x = x + 1
```

To decrease the likelihood of error, **bc** offers you a shorthand expression for this common phrase:

```
        x += 1
```

What it means is, "add one to **x**". Type the following example into **bc** to see how this expression works:

```
x = 1
x * x
x += 1
x * x
x += 1
```

Likewise, **bc** provides an abbreviation for:

```
x = x - 2
```

The form should now be familiar:

```
x –= 2
```

The number to the right of the **-=** or **+=** operator can be replaced with a variable or even another calculation. When you type:

```
i = 4
x = 48
x –= i
x
```

bc replies:

```
44
```

Alternatively, if you type:

```
i = 4
x = 48
x –= i * i
x
```

then **bc** replies:

```
32
```

Similar abbreviations are provided for multiplication, division, remainder, and exponentiation. Here is a summary of this class of operation.

a **+=** 2	Replace *a* with a plus 2
b **+=** *a*	Replace *b* with *b* plus *a*
b **–=** *a*	Replace *b* with *b* minus *a*
c ***=** *b*	Replace *c* with *c* multiplied by *b*
c **/=** *a*	Replace *c* with *c* divided by *a*
c **%=** *b*	Replace *c* with remainder of *c* divided by *b*
d **^=** *3*	Replace *d* with *d* raised to the third power

bc also has an operator that increases a variable by one: '**++**'. When you type:

```
a = 1
++a
```

then **bc** replies:

```
2
```

To use this operator in an expression, combine it with a variable anywhere that a variable would normally be used. For example, entering

```
b = 1
a = 3
b = ++a
a
b
```

yields:

```
4
4
```

The '**++**' operator can also be put after a name. The resulting value in the expression is the value of the name *before* it is incremented. However, after the expression is evaluated, the name will have an incremented value. The following example shows the use of '**++**' both before and after a name:

```
a = 1
b = 1
a++
++b
a
b
```

bc replies:

```
1
2
2
2
```

Operators are used in this manner:

```
a = 1
b = 2
c = a++ + ++b
```

Similar to '**++**' is '—'. It behaves the same way, except that rather than adding one, it subtracts one.

Numbers with Fractions

Most of the examples presented earlier use whole numbers (integers). However, **bc** can use numbers with fractional parts. This section discusses the use of fractional numbers in **bc** and their precision under different operations.

The Scale of Numbers

The number of digits to the left of the decimal point carried by **bc** depends upon the requirements of the calculation. If you calculate a large number, as in:

```
2^500
```

the result will contain as many digits as needed to express the product.

The number of digits to the right of a decimal point is called the *scale* of the number. Scale depends upon the operation that produces the number of digits, and a variable called **scale** that will be described shortly.

To illustrate simple uses of numbers with fractions, invoke **bc** and then type:

```
a = .01
b = 0.99
a + b
```

bc replies:

```
1.00
```

Addition and Subtraction

bc will dynamically adjust the number of digits in the calculation. It deals similarly with fractional numbers. To the following example

```
a = 0.01
b = 0.001
a + b
```

bc reply:

```
.011
```

In addition and subtraction, the scale of the result is the *larger* of the scales of the two numbers involved. Results are not truncated in addition or subtraction operations.

Scale During Multiplication

Other arithmetic operations act differently with numbers that contain fractions. In the multiplication of two numbers, the scale of the product will at least equal the larger of the scales of the two numbers. For example, the input:

```
1.1 * 1.11
```

results in:

```
1.22
```

Setting the Scale of Results

To increase the number of fractional digits for higher accuracy, **bc** provides the built-in variable **scale**. The following example illustrates the **scale** variable:

```
scale = 3
1.1 * 1.11
```

The result from this example is:

```
1.221
```

Note, however, the scale of the product of a multiplication procedure never exceeds the sum of the scales of the two numbers being multiplied. For example,

```
scale = 10
1.1 * 1.11
```

yields the result:

```
1.221
```

If the variable **scale** is less than the sum of the scales of the numbers being multiplied, then the product will have a scale equal to that of the variable **scale**. For example,

```
scale = 4
1.11 * 2.222
```

yields:

```
2.4664
```

The scales of the operands are 2 and 3. The larger scale is 3, so the result of a multiplication will have a scale of at least 3, no matter what **scale** is set to. Also, the sum of the scales is 5, so the result will never have more than 5 digits to the right of the decimal point. In this example, **scale** has been set to a scale of 4. Therefore, the result has four digits to the right of the decimal point.

Scale for Divisions

For division and remainder, the scale of the result is determined only by the value of the variable **scale**. For example,

```
scale = 13
14 / 13
14 % 13
```

yields:

```
1.0769230769230
.0000000000010
```

For non-whole numbers, as well as for integers, the definition of remainder is chosen so that the relationship

```
dividend = (divisor * quotient) + remainder
```

is true.

Scale From Exponentiation

bc sets the **scale** of a result of exponentiation as if repeated multiplications had been performed. Thus, for

```
        5.992 ^ 5
```

the scale is chosen as if you typed:

```
        n = 5.992
        n * n * n * n * n
```

That is, the default is the scale of the largest (or, in this case, the only) number being multiplied; and scale cannot exceed the sum of the scales of the numbers being multiplied. Thus, the scale of the product in this example has a default setting of 3, and can be reset up to 15.

What Is the Current Scale?

The variable **scale** is just like other variables: you can assign values to it, as above. Because it is like regular variables, you can also use it in operations, as in this example:

```
        scale += 1
```

You can also print its value:

```
        scale
```

The value of the **scale** variable is zero until you explicitly change it.

The if Statement

The statements shown so far have been either assignment statements, giving a new value to a variable; or an expression, which prints the resulting value. Several other kinds of statements are available. These give you power to write programs that make decisions and perform iterative computations.

Using the if Statement

To see the **if** statement in action, type the following example into **bc**:

```
        x = 3
        if (x < 5) x
        if (x > 5) -x
```

bc replies:

```
        3
```

If the input is:

```
        x = 6
        if (x < 5) x
        if (x > 5) -x
        <return>
```

bc replies:

```
        -6
```

The part of the **if** statement in parentheses, such as **(x > 5)**, determines whether **bc** executes the statement that follows it, such as **-x**. If the expression is false, the following statement is not executed. If the expression is true, the following statement is executed.

Comparisons

The decision expression in an **if** statement is enclosed in parentheses. The decision can be based upon a comparison of two operands, or numbers. The kinds of comparisons that can be done are:

==	First operand equal to second
!=	First operand not equal to second
<=	First operand less than or equal to second
<	First operand less than second
>=	First operand greater than or equal to second
>	First operand greater than second

The **if** statement can include the sorts of the simple statements already shown. You can also include an **if** statement, as well as the **while**, **do**, and **for** statements, which will be discussed below. The following example illustrates the use of an **if** statement within an **if** statement:

```
a = 2
b = 6
if (a >= 2) if (b > a) a + b
<return>
```

bc replies, simply:

```
8
```

Because both of the **if** conditions were true, **bc** proceeded to add **a** and **b**. Note that nested **if** statements must appear on the same line. Therefore,

```
if (a == 3) if (b > a) a + b
```

does not print the result of **a + b** because not both conditions were true. However

```
if (a == 3)
if (b > a) a + b
```

prints the result of **a + b** because **bc** treats **if** statements one by one, and the second **if** statement's condition is true.

Grouped Statements

You can place more than one statement after the expression part of the **if** statement by using grouping braces '{' and '}'. This can be useful if you want to perform several calculations based on the result of an **if** statement comparison. The following example prints the value of **a** and **b** if the value of **b** is less than the value of **a**:

```
a = 1
b = .99
if (a > b) {
    a
    b
}
```

bc replies:

```
1
.99
```

Any statement may be enclosed within the group braces, as the following example shows:

```
a = 1
b = .99
if (a > b) {
    a
    b
    if ((a + b) >= 2) a + b
}
```

Many Statements Per Line

To this point, all of our examples typed each statement on its own line. This includes the group braces '{' and '}', the latter of which must appear on a line by itself. You can, however, place several statements on one line if you separate them with semicolons. If you do this, remember that the semicolon rather than the carriage return separates the statements. For example, if you type:

```
a = 1;b = 2;c = 3
a;b;c
```

bc replies:

```
1
2
3
```

You can use this in combination with the group braces:

```
a = 1;b = 2;c = 3
if ((a + b) >= c) {
    a; b; c; a + b; }
```

The reply from **bc** is:

```
1
2
3
3
```

This example can be compressed even further by putting all of the **if** statement on one line:

```
a = 1;b = 2;c = 3
if ((a + b) >= c) { a; b; c; a + b; }
```

You do not need to follow the '}' with a semicolon.

The while Statement

The **while** statement repeats calculations. This is useful in successive approximation calculations. The following example of the **while** loop prints the numbers one through ten:

```
i = 1
while (i <= 10) {
        i
        i = i + 1
}
```

bc replies:

```
1
2
3
4
5
6
7
8
9
10
```

The statement

```
i = i + 1
```

adds 1 to the variable **i**. The expression

```
(i <= 10)
```

compares **i** with ten. While **i** is less than or equal to ten, the **while** loop executes. When **i** is increased to greater than ten, the loop stops executing.

bc checks the comparison expression for the **while** loop before the loop is entered for the first time. If the comparison fails, the loop is not executed at all; otherwise the processing repeats as long as the comparison is true. For example, the following statements do not print anything:

```
i = 0
while (i > 1) i
quit
```

Abbreviations in the while Statement

If we recall the assignment statements from the previous section, we can shorten the **while** counting-to-ten example to:

```
i = 1
while (i <= 10) {
        i
        i += 1
}
```

The result remains the same — a list of numbers from one to ten.

Another abbreviation of the example uses the '**++**' operator. The variable **i** is incremented, then tested in the **while** expression, which simplifies the entire example to:

```
i = 0
while (++i <= 10) i
```

Before the **while** is executed, **i** is set to zero. Then, the **while** expression increments the value of **i** before it is used or compared, Thus, the first value compared, then printed, is one.

Finally, the example calculation can be shortened to one line. If a variable in **bc** is used before it is initialized, it will have the value of zero. For example:

```
zip
```

prints:

```
0
```

Using this in our counting-to-ten example yields:

```
while (++n <= 10) n
```

The for Statement

for is a statement that controls the execution of other **bc** statements. You should use **for** to write a formula to control the number of times a value is computed.

The previous section demonstrated how to print the numbers one to ten using a **while** statement. The following does the same task with a **for** statement:

```
for (i=1; i <= 10; ++i) i
```

Three Parts of the for Statement

The **for** statement is more complex than the **while** statement; its controlling expressions have three parts.

The first part, shown here in italics

`for (`*i=1*`; i <= 10; ++i) i`

sets up the initial condition. The second part

`for (i=1; `*i <= 10*`; ++i) i`

tests whether more iterations should be performed. **bc** performs this test *before* it executes the statements that are subordinate to the **for** statement. If the test fails, no more iterations are performed.

The third part

`for (i=1; i <= 10; `*++i*`) i`

is performed at the end of each iteration. In practically every instance, this part of the **for** statement modifies the value of the variable that the second part tests.

Taken together, these statements (1) set **i** to zero; (2) check whether **i** is less than or equal to ten; (3) if **i** proves to be so, prints **i**, and then increases it by one.

The following example of the **for** statement adds the squares of the numbers one through ten, prints each square, and then prints the sum of the squares at the end.

```
sum = 0
for (n=1; n <= 10; ++n) {
    sq = n * n
    sq
    sum += sq
}
sum
```

The result is:

```
1
4
9
16
25
36
49
64
81
100
385
```

Similarities Between the for and while Statements

To illustrate the similarity between the **for** statement and the simpler **while** statement, the following rewrites the above example, substituting the **while** for the **for**:

```
sum = 0
n = 0
while (++n <= 10) {
       sq = n * n
       sq
       sum += sq
}
sum
```

Functions in bc

bc allows you to name routines that you use repeatedly. You can then call them by name without having to retype them; obviously, this can be a great time-saver. These named routines are called *functions*. This section shows you how to define and use functions for your **bc** calculations.

Example of Function Use

The following example defines a function that calculates the area of a circle from its radius.

```
scale = 5
pi = 3.14159
define area (radius) {
       r2 = radius * radius
       return (pi * r2);
}
area (1.00)
area (2.00)
area (56)
```

The results will be:

```
3.14159
12.56636
9852.02624
```

The **define** keyword tells **bc** that you are defining a function. The name of the function follows. Then, in parentheses, come the *parameters* of the function. In this example, the only parameter, or *argument*, of the function is **radius**. Most functions have arguments, but they are not mandatory.

The **return** statement defines the value of the function. In the **area** example, the expression

```
area (1.00)
```

references the function **area**. **bc** then performs the calculation described by your definition of the function **area**. The number

```
1.00
```

is substituted wherever the parameter **radius** is shown.

The statement

```
r2 = radius * radius
```

is then executed, yielding this result:

```
1.00
```

Then, the statement

```
return (pi * r2)
```

calculates the area and returns its value. The statement

```
area (1.00)
```

then has the value calculated in the return statement.

Functions Using Other Functions

Functions in **bc** perform calculations using the same expressions as the rest of the **bc** program. This includes the use of functions. The **area** program can be written using another function, **sq**, to calculate the square of a number:

```
scale = 5
pi = 3.14159
define sq (number) {
       return (number * number)
       }
define area (radius) {
       return (sq (radius) * pi)
       }
area (1.00)
area (2.00)
area (56)
```

Again, the results will be identical:

```
3.14159
12.56636
9852.02624
```

Functions That Call Themselves

Not only can functions call other functions and perform regular calculations; a function can use itself in calculations. An example of this is the Fibonacci calculation:

```
define fib (f) {
       if (f == 0) return (0)
       if (f == 1) return (1)
       if (f > 1) return (fib (f - 1) + fib (f - 2))
}
fib (5)
fib (20)
```

Fibonacci numbers are defined in the following way: Fibonacci number zero is zero; similarly, Fibonacci number one is one. Any other Fibonacci number is defined as the sum of the two previous Fibonacci numbers. Fibonacci numbers are defined only for non-negative integers.

The defined function **fib** follows this definition by returning zero if the number requested is zero and one if the argument is one. If the number is neither of these, then the function calls itself to calculate the previous two numbers of the series and adds them together.

The auto Statement

Many functions that call other functions, including themselves, may require variables that are not changeable by the rest of the program. This is signalled to **bc** by the **auto** statement:

```
auto var1, var2
```

This declares **var1** and **var2** as local to the function that contains them.

To illustrate the use of **auto**, the following **bc** program calculates the factorial of a number:

```
define factorial (number) {
        auto value, i
        value = 1
        for (i = 1; i <= number; ++i) value *= i
        return (value)
}
value = 3
factorial (value)
i = 99
factorial (20)
value
i
```

The result is:

```
6
2432902008176640000
3
99
```

The first number, 6, results from:

```
factorial (value)
```

The second number is from:

```
factorial (20)
```

The last two numbers are from **value** and **i**, and are included to demonstrate that the variables in the function **factorial** appearing in this statement:

```
 auto value, i
```

are separate from the variables of the same name in the rest of the program.

If the function calls itself, as the **fib** example does above, any variable names noted in the **auto** statement are handled separately for each call of the function.

Programs in a File

Because its programs can be quite complex, **bc** lets you keep them in files. This lets you build a library of **bc** programs and functions that can be called up easily.

Using a Program From a File

To illustrate the use of programs stored in a file, type the following example into file **fib.bc** using the editor of your choice. The program defines the function **fib**:

```
define fib (f) {
        if (f == 0) return (0)
        if (f == 1) return (1)
        if (f > 1) return (fib (f - 1) + fib (f - 2))
}
```

To use a **bc** program that has been stored in a file, enter the file name on the **bc** command line, like this:

```
bc fib.bc
```

The function definition will be read in by **bc** and ready for your use. To use the function, simply type the function name with parameters.

So, if you type:

```
bc fib.bc
fib (6)
```

bc will reply:

```
8
```

Using Libraries

You can enter several useful programs in their own files and call them into **bc** at the same time. The following example creates another function that calculates the sum of the squares of integers up to a given number. Use an editor to type the following into a file named **sumsq.bc**:

```
define sumsq (number) {
        auto i, sum
        sum = 0
        for (i = number; i > 0; --i) sum += i ^ 2
        return (sum)
}
```

Now, you can use the **sumsq** function to print the sum of the squares for each number from one to ten:

```
bc sumsq.bc
for (i = 1; i <= 10; ++i) sumsq (i)
```

The result is:

```
1
5
14
30
55
91
140
204
285
385
quit
```

You can use the two functions stored in a file to print the difference between the sum of the squares of numbers, and the Fibonacci number:

```
bc fib.bc sumsq.bc
for (i = 1; i <= 10; ++i) sumsq (i) - fib (i)
quit
```

The result of this questionable computation is:

```
0
4
12
27
50
83
127
183
251
330
```

The bc Library

COHERENT provides an extended library to go with **bc**. It includes the following functions:

atan(*z*)	arctangent of *z*
cos(*z*)	cosine of *z*
exp(*z*)	exponential function of *z*
j(*n*,*z*)	*n*th order Bessel function of *z*
ln(*z*)	natural logarithm of *z*
pi	the value of pi to 100 digits
sin(*z*)	sine of *z*

The library is stored in file **/usr/lib/lib.b**. To use the library, invoke the **bc** command with the **-l** option.

To show how the library can be used in your work the following example computes the sine of an angle of one-third radian with scale set to 20:

```
bc -l
scale = 20
sin (1/3)
quit
```

The result is:

```
.32719469679615224418
```

Summary

The Lexicon entry for **bc** summarizes its commands, features, and libraries. It will also refer you to related commands and functions.