bc Calculator

Language Tutorial

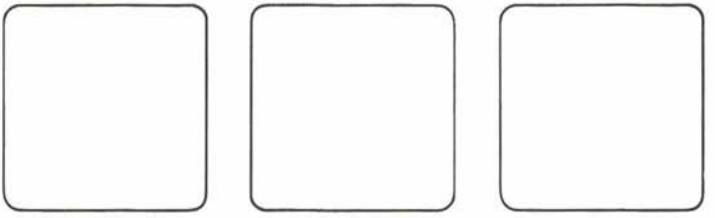

Table of Contents

1. Introduction

This is a tutorial for **bc**, the COHERENT™ System calculator language. If you have not used **bc** before, you can start here. If you are familiar with **bc**, you may wish to use this manual for reference.

The **bc** command provides a desk calculator language with high-precision calculation capability. The number of digits in numbers is automatically adjusted to contain enough digits to correctly represent the number. It is like having a powerful calculator at your fingertips.

2. Entry and exit

The COHERENT calculator couldn't be easier to use. Whenever you want to invoke **bc**, all you do is type out its name (bc), remembering to conclude with a stroke of the carriage return key. And when you are finished using the calculator and wish to exit from **bc**, all you do is type the word 'quit'. (Alternatively, you can type <**Ctrl-D**>; **bc** will exit and return control to the COHERENT shell.)

Example of simple use

Calculations in **bc** are performed by typing in formulas as you would naturally write them. The example below shows you how to call up **bc**, how to add 2 + 2, and then how to exit:

```
bc
2 + 2
quit
```

To which **bc** will reply:

```
.4
```

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

```
2^500
```

This prints the value of 2 raised to the 500th power. (The '^' character signifies the power operation.)

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 out the results onto your terminal screen. When **bc** sees any expression, like "2+2" or "3777", it outputs the result.

bc provides the common operators for add, subtract, multiply and divide. These are illustrated by the following commands:

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

Also provided by **bc** is the remainder operator, '%'. To get a sense of how it works, type:

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

The response to the first will be 2, and to the second, 5. (The power operator '^' has already been illustrated.)

You can also enter numbers with fractional parts. Type the following to illustrate:

```
9.999 * 9.999
```

And the reply will be:

```
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 will allow you to streamline increasingly complex calculations.

Summary

To call up **bc**, type 'bc'; to exit after finishing your calculations, type 'quit' (or <**Ctrl-D**). Remember that calculations are entered literally, just as you would write them out on paper. **bc** provides

the common operators for the basic arithmetic functions (+ - * /), as well as the remainder operator, '%', and the power operator, '^'. Variables were also introduced.

3. Simple statements

While you can use **bc** as a simple calculator for calculating numbers, you can take advantage of its greater power by using *variables*. Variables, as we noted in the previous section, 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. Type in the following example:

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

Note that an assignment does not print a value, unlike normal expressions. For example, this example first assigns the value 9.999 to the variable **x**, and prints it out. Then, the value of **x** multiplied by **x** is printed out. Next, **x** is given a new value and printed out. The result printed at your terminal will be:

```
9.999
99.980
99.980
```

When performing calculations, either with hand-held calculators, with programming languages like C, or with **bc**, the following calculation is frequently used:

```
x = x + 1
```

bc has a shorthand for this that reduces the amount of typing necessary to write this common phrase, thus decreasing the likelihood of error. The above expression can be written in shorthand as:

```
x += 1
```

What it means is: "add one to x". Type in the following example to see how it works.

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

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

```
x = x - 2
```

The form is familiar:

```
x -= 2
```

The number following the $-=$ or $+=$ can be replaced by a variable or even another calculation. When you type:

```
i = 4
x = 48
x -= i
x
```

then **bc** will respond:

```
44
```

Alternatively, if you type:

```
i = 4
x = 48
x -= i * i
x
```

then **bc** will respond:

```
32
```

Similar abbreviations are provided for multiplication, division, remaindering, 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 3rd power         */
```

There is an operator which, when used in an expression, increments a variable by one: '+ +'. When you type:

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

then **bc** will reply:

```
2
```

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

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

the reply will be:

```
4
4
```

The '+ +' operator can also be put before a name. The resulting value in the expression will be 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 the '+ +' operator both before and after a name:

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

The result printed by **bc** is:

```
1
2
2
2
```

Operators are used in this manner:

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

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

Summary

This section went into variables with more depth; it also showed how to use abbreviations for useful operations involving arithmetic functions, remaindering, and exponentiation. You will find the use of variables and operators convenient in equations where basic formulas are used repeatedly. Simple statements can also be combined with the more complex statements, described in Sections 5 to 8, governing operations specifically required to make decisions for you.

4. Numbers with fractions

The examples presented in earlier sections use whole numbers or integers. However, **bc** is capable of using 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 necessary.

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

To illustrate simple uses of numbers with fractions, type:

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

and **bc** will reply:

```
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 will reply:

```
.011
```

In addition and subtraction, the scale of the result will be 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 containing fractions. In the multiplication of two numbers, the scale of the result will be at least equal to the higher of the scales of the two numbers. For example, the input:

```
1.1 * 1.11
```

will result in:

```
1.22
```

Setting the scale of results

To retain more fractional digits for better accuracy, **bc** provides the built-in variable, **scale**. The following example illustrates this:

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

The result from this example is:

```
1.221
```

If the value of **scale** is greater than the sum of the scales of the two numbers involved, the result will have a scale equal to this sum. For this example:

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

the result will still be:

```
1.221
```

If the variable **scale** is less than the larger of the scales of the two numbers, then the result will have a scale equal to the larger of the scales of the operands. The following example illustrates this point:

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

The result from this is:

```
2.4664
```

Thus, the scales of the numbers are 2 and 3. The largest 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 number between 3 and 5, namely 4. Therefore, the result has a scale of 4.

Scale for divisions

For division and remaindering, the scale of the result is determined only by the value of the variable **scale**. This is illustrated in the example below:

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

To which **bc** will reply:

```
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.

The **scale** of a result from exponentiation is done as if repeated multiplications were performed. That is, for this example:

```
5.992 ^ 5
```

the scale is chosen as if you typed:

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

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 = scale + 1
```

You can also print its value:

```
scale
```

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

Summary

For addition and subtraction involving fractions, the scale is automatically set according to the number that has the greatest number of decimal points. For multiplication, you can retain more fractional digits for better accuracy, by setting a current value for the built-in variable, **scale**. And for division and remaindering, the scale of the result is determined solely by the current value of **scale**.

scale is like other variables in that you can assign values to it; you can also print out its value for reference. And its versatility is increased by its ability to take on any value you specify.

5. 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 out the resultant value. Several other kinds of statements are available. These statements give you power to write programs that make decisions and perform iterative computations.

Using the if statement

To see the **if** statement in operation, type in the following example:

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

The reply will be:

```
3
```

If the input is:

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

bc will reply:

```
-6
```

The part of the **if** statement in parentheses, such as **(x > 5)**, determines whether or not **bc** executes the statement following it, such as **−x**. If the expression is false, the following statement will not be executed. If the expression is true, the following statement will be 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 kinds of statements that can be in the **if** statement include 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 in later sections. 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
```

Grouped statements

More than one statement can be put after the expression part of the **if** statement by the use of the 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 will print 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
}
```

The result produced by **bc** is:

```
1
.99
```

Any statement may be included in the statements enclosed by the group braces. This is illustrated in the following example:

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

Many statements per line

Until now, each statement in **bc** has been on its own line; you have effectively been concluding each statement by hitting the carriage return key. This includes the group braces '{' and '}', which must be alone on a line.

You can 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 is separating the statements. The following example illustrates the use of the semicolon:

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

bc will reply:

```
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.

Summary

The **if** statement brings the decision-making function of **bc** into play. **bc** will make comparisons between, and respond to, hypothetical situations automatically when the **if** statement is used. Group braces allow you to carry out various calculations based on one **if** statement, and semicolons eliminate the need for successive carriage returns. These symbols, and the simple statements and operators described above, may also be used with the **while** and **for** statements discussed in the following sections.

6. The while statement

The **while** statement is used to repeat calculations. This is useful in successive approximation calculations.

Example of while usage

The following example of the **while** loop prints the numbers one through ten:

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

The reply from **bc** will be:

```
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** to 10. If **i** is less than or equal to 10, the **while** loop executes one more cycle. If **i** is more than 10, the loop is not executed again.

The comparison expression for the **while** loop is checked before the loop is entered for the first time. If the comparison fails, the loop is not executed at all. Otherwise the processing will repeat as long as the comparison is true. The following statements will not print any numbers:

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

Abbreviations in the while statement

If we recall the assignment statements from the previous section, the **while** counting-to-ten example can be shortened 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, simplifying the entire example to:

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

Notice that the **while** expression increments the value of **i** before it is used or compared, so before the **while** is executed, **i** is set to zero. 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
```

will print:

```
0
```

And using this in our counting-to-ten example yields:

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

Summary

The **while** statement, like the **if** statement, requires **bc** to selectively carry out operations based on a comparison expression set by the user. Again, abbreviations increase the efficiency of the **while** statement; assignment statements within the **while** statement can be shortened considerably by the inclusion of group braces and special operators.

7. The for statement

Like the **while** statement, the **for** statement controls the repetition of other **bc** statements. The **for** statement is useful if you can write a formula for the number of cycles of computation that you need.

Example of use

The previous section demonstrated how to print the numbers from one to ten using a **while** statement. The same task can be performed 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.

There are three parts to the **for** statement's controlling expressions.

The first part, which in the one-to-ten example is:

```
i = 1
```

is a statement that is performed once, and sets up the initial conditions required by the rest of the statements in the range of the **for**.

The second part,

```
i <= 10
```

is a test to determine whether or not more iterations should be performed. This test is performed *before* the iterations of the **for** statement. If the test fails, no more iterations are performed.

The third and final part:

```
++i
```

is performed at the end of each iteration. This part is sometimes called the **increment** part.

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

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

The result produced 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 same example is written using the **while** rather than the **for**:

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

You will notice one difference when you enter this example. In the **while** version of the example, the

```
++n
```

will print out the new value of **n**, whereas in the **for** example, the value will not be printed.

Summary

Although the **for** statement is more complicated than the **while** statement, the three parts of its controlling expressions—a statement that sets up the initial conditions required by the rest of the statements; a test determining if more iterations are required; and the **increment** part—are straightforward and allow you to define a specific range of actions for **bc** to undertake.

8. Functions in bc

Functions are a way to express repeatedly-used calculations in shorthand. 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 the value of it. 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 containing 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 printed on your terminal will be:

```
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.

Summary

Functions are a way to express repeatedly-used calculations in shorthand, for example calculating the area of a circle from its radius. First you **define** your function, then you list your parameters, in parentheses. Most functions have arguments, but they are not mandatory. To get the value of the function, you must use the **return** statement.

Functions in **bc** can use the same expressions as the rest of the **bc** program. Consequently, functions can use other functions. And a function can also call itself.

Defining functions for **bc** can be an involved process, especially when calculations call for the use of functions within functions. The COHERENT system will, however, store **bc** programs for you. Section 9 outlines methods for the storage and subsequent use of functions in files.

9. Programs in a file

Since **bc** programs can be quite complex, the COHERENT system provides for keeping them in files. This lets you build a library of **bc** programs and functions that can easily be called up.

Using a program from a file

To illustrate the use of programs stored in a file, the following example creates a file with **ed** containing the definition of the function **fib**:

```
ed
a
bc
define fib (f) {
    if (f==0) return (0)
    if (f==1) return (1)
    if (f > 1) return (fib (f-1) + fib (f-2))
}
.
w fib.bc
q
```

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)
quit
```

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:

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

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)
quit
```

The result will be:

```
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

An extended library is provided with **bc**. The functions provided include trigonometric functions, such as **sin**; the exponential functions **exp** and **ln**; and the variable **pi**, which is defined to 100 digits.

To use the library, invoke the **bc** command with the **-l** option. The following example computes the sine of an angle of one third radian with scale set to twenty:

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

The result is:

```
.32719469679615224418
```

Summary

The storage of programs within files and files within libraries allows powerful processes to operate through designated file names on a single command line. **bc** retains the definitions of the functions you create and reads them in according to a given file name. Obviously the efficiency and power of **bc** is increased in proportion to the

accretion of commands within your **bc** library. The extended library provided also carries a number of intricate functions that can be invoked with previously set file names.

Index

User Reaction Report

To keep this manual and COHERENT free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name: _____

Company: _____

Address: _____

_____

Phone: _____ Date: _____

Version and hardware used: _____

Did you find any errors in the manual? _____

_____

_____

Can you suggest any improvements to the manual? _____

_____

_____

Did you find any bugs in the software? _____

_____

_____

Can you suggest improvements or enhancements to the software?

_____

_____

_____

Additional comments: (Please use other side.)