Bash test and comparison functions

By Ian Shields

Published February 20, 2007

The Bash shell is available on many Linux® and UNIX® systems today, and is a common default shell on Linux. Bash includes powerful programming capabilities, including extensive functions for testing file types and attributes, as well as the arithmetic and string comparisons available in most programming languages. Understanding the various tests and knowing that the shell can also interpret some operators as shell metacharacters is an important step to becoming a power shell user. This article, excerpted from the developerWorks tutorial LPI exam 102 prep: Shells, scripting, programming, and compiling, shows you how to understand and use the test and comparison operations of the Bash shell.
This tip explains the shell test and comparison functions and shows you how to add programming capability to the shell. You may have already seen simple shell logic using the && and || operators, which allow you to execute a command based on whether the previous command exits normally or with an error. In this tip, you will see how to extend these basic techniques to more complex shell programming.

Tests

In any programming language, after you learn how to assign values to variables and pass parameters, you need to test those values and parameters. In shells, the tests set the return status, which is the same thing that other commands do. In fact, `test` is a builtin *command*!

test and [

The `test` builtin command returns 0 (True) or 1 (False), depending on the evaluation of an expression, *expr*. You can also use square brackets: `test _expr` and [*expr*] are equivalent. You can examine the return value by displaying `$?`; you can use the return value with && and ||; or you can test it using the various conditional constructs that are covered later in this tip.

Listing 1. Some simple tests

```
[ian@pinguino ~]$ test 3 -gt 4 && echo True || echo false
false
[ian@pinguino ~]$ [ "abc" != "def" ];echo $?
0
[ian@pinguino ~]$ test -d "$HOME" ;echo $?
0
```

In the first example in Listing 1, the `-gt` operator performs an arithmetic comparison between two literal values. In the second example, the alternate `[ ]` form compares two strings for inequality. In the final example, the value of the HOME variable is tested to see if it is a directory using the `-d` unary operator.
You can compare arithmetic values using one of `-eq`, `-ne`, `-lt`, `-le`, `-gt`, or `-ge`, meaning equal, not equal, less than, less than or equal, greater than, and greater than or equal, respectively.
You can compare strings for equality, inequality, or whether the first string sorts before or after the second one using the operators =, `!=`, <, and >, respectively. The unary operator `-z` tests for a null string, while `-n` or no operator at all returns True if a string is not empty.
Note: the < and > operators are also used by the shell for redirection, so you must escape them using \< or \>. Listing 2 shows more examples of string tests. Check that they are as you expect.

Listing 2. Some string tests

```
[ian@pinguino ~]$ test "abc" = "def" ;echo $?
1
[ian@pinguino ~]$ [ "abc" != "def" ];echo $?
0
[ian@pinguino ~]$ [ "abc" \< "def" ];echo $?
0
[ian@pinguino ~]$ [ "abc" \> "def" ];echo $?
1
[ian@pinguino ~]$ [ "abc" \<"abc" ];echo $?
1
[ian@pinguino ~]$ [ "abc" \> "abc" ];echo $?
1
```

Some of the more common file tests are shown in Table 1. The result is True if the file tested is a file that exists and that has the specified characteristic.

Table 1. Some common file tests

Operator	Characteristic
-d	Directory
-e	Exists (also -a)
-f	Regular file
-h	Symbolic link (also -L)
-p	Named pipe

-r	Readable by you
-s	Not empty
-S	Socket
-w	Writable by you
-N	Has been modified since last being read

In addition to the unary tests above, you can compare two files with the binary operators shown in Table 2.

Table 2. Testing pairs of files

Operator	True if
-nt	Test if file1 is newer than file 2. The modification date is used for this and the next comparison.
-ot	Test if file1 is older than file 2.
-ef	Test if file1 is a hard link to file2.

Several other tests allow you to check things such as the permissions of the file. See the man pages for bash for more details or use `help test` to see brief information on the test builtin. You can use the `help` command for other builtins too.
The `-o` operator allows you to test various shell options that may be set using `set -o _option`, returning True (0) if the option is set and False (1) otherwise, as shown in Listing 3.

Listing 3. Testing shell options

```
[ian@pinguino ~]$ set +o nounset
[ian@pinguino ~]$ [ -o nounset ];echo $?
1
[ian@pinguino ~]$ set -u
[ian@pinguino ~]$ test  -o nounset; echo $?
0
```

Finally, the `-a` and `-o` options allow you to combine expressions with logical AND and OR, respectively, while the unary `!` operator inverts the sense of the test. You may use parentheses to group expressions and override the default precedence. Remember that the shell will normally run an expression between parentheses in a subshell, so you will need to escape the parentheses using (and) or enclosing these operators in single or double quotes. Listing 4 illustrates the application of de Morgan's laws to an expression.

Listing 4. Combining and grouping tests

```
[ian@pinguino ~]$ test "a" != "$HOME" -a 3 -ge 4 ; echo $?
1
[ian@pinguino ~]$ [ ! \( "a" = "$HOME" -o 3 -lt 4 \) ]; echo $?
1
[ian@pinguino ~]$ [ ! \( "a" = "$HOME" -o '(' 3 -lt 4 ')' ")" ]; echo $?
1
```

((and [[

The `test` command is very powerful, but somewhat unwieldy given its requirement for escaping and given the difference between string and arithmetic comparisons. Fortunately, bash has two other ways of testing that are somewhat more natural for people who are familiar with C, C++, or Java® syntax.
The `(( ))`compound command_ evaluates an arithmetic expression and sets the exit status to 1 if the expression evaluates to 0, or to 0 if the expression evaluates to a non-zero value. You do not need to escape operators between `(( ` and ` ))`. Arithmetic is done on integers. Division by 0 causes an error, but overflow does not. You may perform the usual C language arithmetic, logical, and bitwise operations. The `let` command can also execute one or more arithmetic expressions. It is usually used to assign values to arithmetic variables.

Listing 5. Assigning and testing arithmetic expressions

```
[ian@pinguino ~]$ let x=2 y=2**3 z=y*3;echo $? $x $y $z
0 2 8 24
[ian@pinguino ~]$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y $w
0 3 8 16
[ian@pinguino ~]$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y $w
0 4 8 13
```

As with `(( ))`, the `[[ ]]` compound command allows you to use more natural syntax for filename and string tests. You can combine tests that are allowed for the `test` command using parentheses and logical operators.

Listing 6. Using the `[[` compound

```
[ian@pinguino ~]$ [[ ( -d "$HOME" ) && ( -w "$HOME" ) ]] &&
>  echo "home is a writable directory"
home is a writable directory
```

The `[[` compound can also do pattern matching on strings when the = or `!=` operators are used. The match behaves as for wildcard globbing as illustrated in Listing 7.

Listing 7. Wildcard tests with `[[`

```
[ian@pinguino ~]$ [[ "abc def .d,x--" == a[abc]*\ ?d* ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def c" == a[abc]*\ ?d* ]]; echo $?
1
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* ]]; echo $?
1
```

You can even do arithmetic tests within `[[` compounds, but be careful. Unless within a `((` compound, the < and > operators will compare the operands as strings and test their order in the current collating sequence. Listing 8 illustrates this with some examples.

Listing 8. Including arithmetic tests with `[[`

```
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || (( 3 > 2 )) ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || 3 -gt 2 ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || 3 > 2 ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || a > 2 ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || a -gt 2 ]]; echo $?
-bash: a: unbound variable
```

Conditionals

While you could accomplish a huge amount of programming with the above tests and the `&&` and `||` control operators, bash includes the more familiar “if, then, else” and case constructs. After you learn about these, you will learn about looping constructs and your toolbox will really expand.

If, then, else statements

The bash `if` command is a compound command that tests the return value of a test or command (`$?`) and branches based on whether it is True (0) or False (not 0). Although the tests above returned only 0 or 1 values, commands may return other values. Learn more about these in the LPI exam 102 prep: Shells, scripting, programming, and compiling tutorial.
The `if` command in bash has a `then` clause containing a list of commands to be executed if the test or command returns 0, one or more optional `elif` clauses, each with an additional test and `then` clause with an associated list of commands, an optional final `else` clause and list of commands to be executed if neither the original test, nor any of the tests used in the `elif` clauses was true, and a terminal `fi` to mark the end of the construct.
Using what you have learned so far, you could now build a simple calculator to evaluate arithmetic expressions as shown in Listing 9.

Listing 9. Evaluating expressions with if, then, else

```
[ian@pinguino ~]$ function mycalc ()
> {
>   local x
>   if [ $# -lt 1 ]; then
>     echo "This function evaluates arithmetic for you if you give it some"
>   elif (( $* )); then
>     let x="$*"
>     echo "$* = $x"
>   else
>     echo "$* = 0 or is not an arithmetic expression"
>   fi
> }
[ian@pinguino ~]$ mycalc 3 + 4
3 + 4 = 7
[ian@pinguino ~]$ mycalc 3 + 4**3
3 + 4**3 = 67
[ian@pinguino ~]$ mycalc 3 + (4**3 /2)
-bash: syntax error near unexpected token `('
[ian@pinguino ~]$ mycalc 3 + "(4**3 /2)"
3 + (4**3 /2) = 35
[ian@pinguino ~]$ mycalc xyz
xyz = 0 or is not an arithmetic expression
[ian@pinguino ~]$ mycalc xyz + 3 + "(4**3 /2)" + abc
xyz + 3 + (4**3 /2) + abc = 35
```

The calculator makes use of the `local` statement to declare x as a local variable that is available only within the scope of the `mycalc` function. The `let` function has several possible options, as does the `declare` function to which it is closely related. Check the man pages for bash, or use `help let` for more information.
As you saw in Listing 9, you need to make sure that your expressions are properly escaped if they use shell metacharacters such as (,), *, >, and <. Nevertheless, you have quite a handy little calculator for evaluating arithmetic as the shell does it.
You may have noticed the `else` clause and the last two examples in Listing 9. As you can see, it is not an error to pass `xyz` to mycalc, but it evaluates to 0. This function is not smart enough to identify the character values in the final example of use and thus be able to warn the user. You could use a string pattern matching test such as

```
[[ ! ("$*" == *[a-zA-Z]* ]]
```

(or the appropriate form for your locale) to eliminate any expression containing alphabetic characters, but that would prevent using hexadecimal notation in your input, since you might use 0x0f to represent 15 using hexadecimal notation. In fact, the shell allows bases up to 64 (using `base_# _value` notation), so you could legitimately use any alphabetic character, plus _ and @ in your input. Octal and hexadecimal use the usual notation of a leading 0 for octal and leading 0x or 0X for hexadecimal. Listing 10 shows some examples.

Listing 10. Calculating with different bases

```
[ian@pinguino ~]$ mycalc 015
015 = 13
[ian@pinguino ~]$ mycalc 0xff
0xff = 255
[ian@pinguino ~]$ mycalc 29#37
29#37 = 94
[ian@pinguino ~]$ mycalc 64#1az
64#1az = 4771
[ian@pinguino ~]$ mycalc 64#1azA
64#1azA = 305380
[ian@pinguino ~]$ mycalc 64#1azA_@
64#1azA_@ = 1250840574
[ian@pinguino ~]$ mycalc 64#1az*64**3 + 64#A_@
64#1az*64**3 + 64#A_@ = 1250840574
```

Additional laundering of the input is beyond the scope of this tip, so use your calculator with care.

The `elif` statement is very convenient. It helps you in writing scripts by allowing you to simplify the indenting. You may be surprised to see the output of the `type` command for the `mycalc` function as shown in Listing 11.

Listing 11. Type mycalc

```
[ian@pinguino ~]$ type mycalc
mycalc is a function
mycalc ()
{
    local x;
    if [ $# -lt 1 ]; then
        echo "This function evaluates arithmetic for you if you give it some";
    else
        if (( $* )); then
            let x="$*";
            echo "$* = $x";
        else
            echo "$* = 0 or is not an arithmetic expression";
        fi;
    fi
}
```

Of course, you could just do shell arithmetic by using `$(( _expression_ ))` with the `echo` command as shown in Listing 12. You wouldn't have learned anything about functions or tests that way, but do note that the shell does not interpret metacharacters, such as *, in their normal role when inside `(( _expression_ ))` or `[[ _expression_ ]]`.

Listing 12. Direct calculation in the shell with echo and `$(( ))`

```
[ian@pinguino ~]$  echo $((3 + (4**3 /2)))
35
```