The Unix Shell's Humble If

Pat Brisbin – January 6, 2014 UPDATED ON March 28, 2019

UNIX, SHELL

The Unix shell is often overlooked by software developers more familiar with higher level languages. This is unfortunate because the shell can be one of the most important parts of a developer's toolkit. From the one-off `grep` search or `sed` replacement in your source tree to a more formal script for deploying or backing up an entire application complete with options and error handling, the shell can be a huge time saver.

To help shed light on the power and subtlety that is the Unix shell, I'd like to take a deep dive into just one of its many features: the humble `if` statement.

Syntax

The general syntax of an `if` statement in any POSIX shell is as follows:

```
if command ; then
  expressions

elif command ; then  # optionally
  expressions

else                 # optionally
  expressions

fi
```

The `if` statement executes *command* and determines if it exited successfully or not. If so, the "consequent" path is followed and the first set of *expressions* is executed. Otherwise, the "alternative" is followed. This may mean continuing similarly with an `elif` clause, executing the *expressions* under an `else` clause, or simply doing nothing.

```
if grep -Fq 'ERROR' development.log; then
  # there were problems at some point
elif grep -Fq 'WARN' development.log; then
  # there were minor problems at some point
else
```

```
  # all ok!
fi
```

The command can be a separate binary or shell script, a shell function or alias, or a variable referencing any of these. Success is determined by a zero exit-status or return value, anything else is failure. This makes sense: there may be many ways to fail but there should be exactly one way to succeed.

```
is_admin() {
  return 1
}

if is_admin; then
  # this will not run
fi
```

If your command is a pipeline, the exit status of the last command in the pipeline will be used:

```
# useless use of cat for educational purposes only!
if cat development.log | grep -Fq 'ERROR'; then
  # ...
fi
```

For the most part, this is intuitive and expected. In cases where it's not, some shells offer the `pipefail` option to change that behavior.

Negation, True, and False

The `!` operator, when preceding a command, inverts its exit status. If the status is successful (0), it changes it to unsuccessful (1). If the status is unsuccessful (anything but 0), it changes it to successful (0). Additionally, both `true` and `false` are normal commands on your system which do nothing but exit appropriately:

```
true; echo $?
# => 0

false; echo $?
# => 1

! true; echo $?
# => 1
```

The `!` operator allows us to easily form an "if-not" statement:

```
if ! grep -Fq 'ERROR' development.log; then
  # All OK
fi
```

The availability of `true` and `false` is what makes statements like the following work:

```
if true; then
  # ...
fi

var=false

if ! "$var"; then
  # ...
fi
```

However, you should avoid doing this. The idiomatic (and more efficient) way to represent booleans in shell scripts is with the values `1` (for true) and `0` (for false). This idiom is made more convenient if you have `((` available, which we'll discuss later.

The `test` Command

The `test` command performs a test according to the options given, then exits successfully or not depending on the result of said test. Since this is a command like any other, it can be used with `if`:

```
if test -z "$variable"; then
  # $variable has (z)ero size
fi

if test -f ~/foo.txt; then
  # ~/foo.txt is a regular (f)ile
fi
```

`test` accepts a few symbolic options as well, to make for more readable statements:

```
if test "$foo" = 'bar'; then
  # $foo equals 'bar', as a string
fi

if test "$foo" != 'bar'; then
  # $foo does not equal bar, as a string
fi
```

The `=` and `!=` options are only for string comparisons. To compare numerically, you must use `-eq` and `-ne`. See `man 1 test` for all available numeric comparisons.

Since commands can be chained together logically with `&&` and `||`, we can combine conditions intuitively:

```
if test "$foo" != 'bar' && test "$foo" != 'baz'; then
  # $foo is not bar or baz
fi
```

Be aware of precedence. If you need to enforce it, group your expressions with curly braces.

```
if test "$foo" != 'bar' && { test -z "$bar" || test "$foo" = "$bar"; }
  # $foo is not bar and ( $bar is empty or $foo is equal to it )
fi
```

(Note the final semi-colon before the closing brace)

If your expression is made up entirely of `test` commands, you can collapse them using `-a` or `-o`. This will be faster since it's only one program invocation:

```
if test "$foo" != 'bar' -a "$foo" != 'baz'; then
  # $foo is not bar or baz
fi
```

The `[` Command

Surprisingly, `[` is just another command. It's distributed alongside `test` and its usage is identical with one minor difference: a trailing `]` is required. This bit of cleverness leads to an intuitive and familiar form when the `[` command is paired with `if`:

```
if [ "string" != "other string" ]; then
  # same as if test "string" != "other string"; then
fi
```

Unfortunately, many users come across this usage first and assume the brackets are part of `if` itself. This can lead to some nonsensical statements.

Rule: Never use commands and brackets together

Case in point, this is incorrect:

```
if [ grep -q 'ERROR' log/development.log ]; then
  # ...
fi
```

And so is this:

```
if [ "$(grep -q 'ERROR' log/development.log)" ]; then
  # ...
fi
```

The former is passing a number of meaningless words as arguments to the `[` command; the latter is passing the string output by the (quieted) `grep` invocation to the `[` command.

There are cases where you might want to test the output of some command as a string. This would lead you to use a command and brackets together. However, there is almost always a better way.

```
# this does work
if [ -n "$(grep -F 'ERROR' log/development.log)" ]; then
  # there were errors
fi
```

```
# but this is better
if grep -Fq 'ERROR' development.log; then
  # there were errors
fi

# this also works
if [ -n "$(diff file1 file2)" ]; then
  # files differ
fi

# but this is better
if ! diff file1 file2 >/dev/null; then
  # files differ
fi
```

As with most things, quoting is extremely important. Take the following example:

```
var="" # an empty string

if [ -z $var ]; then
  # string is empty
fi
```

You'll find if you run this code, it doesn't work. The `[` command returns false even though we can clearly see that `$var` is in fact empty (a string of `z`ero size).

Since `[ OPTION` is valid usage for `[`, what's actually being executed by the shell is this:

```
if [ -z ]; then
  # is the string "]" empty? No.
```

```
fi
```

The fix is to quote correctly:

```
if [ -z "$var" ]; then
  # is the string "" empty? Yes.
fi
```

When are quotes needed? Well, to paraphrase Bryan Liles…

Rule: Quote All the Freaking Time

Examples: `"$var"`, `"$(command)"` `"$(nested "$(command "$var")")"`

In addition to properly quoting, other steps may be required to prevent `test` (or `[`) from incorrectly parsing one of your positional arguments as an option. Consider the following:

```
var='!'

if [ "$var" = "foo" ]; then
  # ...
fi
```

Some implementations of `test` will interpret `"$var"` as its `!` option rather than the literal string `"!"`:

```
if [ ! = "foo" ]; then
  # equivalent to: test ! = "foo"
  # => error: invalid usage
fi
```

Note that it’s very hard to trigger this behavior in modern shells; most will recognize the ambiguity and correctly interpret the expression. However, if you are deeply concerned with portability, one way to mitigate the risk is to use the following:

```
var='!'

if [ x"$var" = x"foo" ]; then
  # ...
fi
```

The prefix will prevent “x!” from being interpreted as an option. The character chosen doesn’t matter, but `x` and `z` are two common conventions.

Non-POSIX Concerns

In most modern shells like bash and zsh, two built-ins are available: `[[` and `((`. These perform faster, are more intuitive, and offer many additional features compared to the `test` command.

Best Practice: If you have no reason to target POSIX shell, use `[[`

Bracket-Bracket

`[[` comes with the following features over the normal `test` command:

- Use familiar `==`, `>=`, and `<=` operators
- Check a string against a regular expression with `=~`
- Check a string against a glob with `==`
- Less strict about quoting and escaping

You can read more details about the difference here.

While the operators are familiar, it’s important to remember that they are string (or file) comparisons only.

Rule: Never use `[[` for numeric comparisons.

For that, we’ll use `((` which I’ll explain shortly.

When dealing with globs and regular expressions, we immediately come to another rule:

Rule: Never quote a glob or regular expression

I know, I just said to quote everything, but the shell is an epic troll and these are the only cases where quotes can hurt you, so take note:

```
for x in "~/*"; do
  # This loop will run once with $x set to "~/*" rather than once
  # for every file and directory under $HOME, as was intended
done

for x in ~/*; do
  # Correct
done

case "$var" of
  'this|that')
    # This will only hit if $var is exactly "this|that"
    ;;
  '*')
    # This will only hit if $var is exactly "*"
    ;;
esac

# Correct
case "$var" of
  this|that) ;;
  *) ;;
esac

foo='foobarbaz'

if [[ "$foo" == '*bar*' ]]; then
  # True if $foo is exactly "*bar*"
fi

if [[ "$foo" == *bar* ]]; then
  # Correct
fi

if [[ "$foo" =~ '^foo' ]]; then
  # True if $foo is exactly "^foo", but leading or trailing
  # whitespace may be ignored such that this is also true if $foo is
  # (for example) "  ^foo  "
fi

if [[ "$foo" =~ ^foo ]]; then
  # Correct
fi
```

If the glob or regular expression becomes unwieldy, you can place it in a variable and use the (unquoted) variable in the expression:

```
pattern='^Home sweet'

if [[ 'Home sweet home' =~ $pattern ]]; then
  # ...
fi

myfiles='~/*'

for file in $myfiles; do
  # ...
done
```

After regular expression matches, you can usually find any capture groups in a magic global. In bash, it’s `BASH_REMATCH`.

```
if [[ 'foobarbaz' =~ ^foo(.*)baz$ ]]; then
  echo ${BASH_REMATCH[1]}
  # => "bar"
fi
```

And in zsh, it’s `match`.

```
if [[ 'foobarbaz' =~ ^foo(.*)baz$ ]]; then
  echo $match[1]
  # => "bar"
fi
```

(Note that in zsh, you don’t need curly braces for array element access)

Math and Numerical Comparisons

The built-in `((` or Arithmetic Expression is concerned with anything numeric. It’s an enhancement on the POSIX `$(( ))` expression which replaced the ancient `expr` program for doing integer math.

```
i=1

# old, don't use!
i=$(expr $i+1)

# better, POSIX
i=$((i+1))

# valid in shells like bash and ksh93
((i++))

# alternate syntax
let i++
```

The difference between `$((expression))` and `((expression))` or `let expression` is whether you want the result or not. Also notice that in either form, we don’t need to use the `$` when referencing variables. This is true in most but not all cases (`$#` is one where it’s still required).

When comparison operators are used within `((` , it will perform the comparison and exit accordingly (just like `test`). This makes it a great companion to `if` :

```
if ((x == 42)); then
  # ...
fi

if ((x < y)); then
  # ...
fi
```

Here’s a more extended example showing that it can be useful to perform arithmetic and comparisons in the same expression:

```
retry() {
  local i=1 max=5

  while ((i++ <= max)); do
    if try_something; then
      printf "Call succeeded.\n"
      return 0
    fi
  done

  printf "Maximum attempts reached!\n" >&2
  return 1
}
```

The `((` form can also check numbers for “truthiness”. Namely, the number `0` is false. This makes our boolean idiom a bit more convenient:

```
var=1

# POSIX
if [ "$var" -eq 1 ]; then
  # ...
fi

# bash, zsh, etc
if ((var)); then
  # ...
fi

# example use-case. $UID of the root user is 0.
if ((UID)); then
  error "You must be root"
fi
```

This will perform better than a fork-exec of `/bin/true` or `/bin/false`.

Conclusion

To recap, we’ve seen that `if` in the Unix shell is both simple and complex. It does nothing but execute a command and branch based on exit status. When combined with the `test` command, we can make powerful comparisons on strings, files, and numbers while upgrading to `[` gives the same comparisons a more familiar syntax. Additionally, using non-POSIX enhancements like `[[` and `((` gives us globs, regular expressions, and better numeric comparisons.

You’ve also seen a number of rules and best practices to ensure your shell scripts act as you intend in as many shells as you choose to run them.

If you enjoyed this post, you might also like:

Code Sleuthing with Git
Rebuilding Git in Ruby
2.5 Ways to Execute a Shell Command