The Basics of Bash scripting

by Andy Pepperdine

This paper is an attempt to describe the syntactic elements of a bash script, to enable the reader to understand any scripts they see, and to write simple scripts themselves.

The descriptions below are neither complete nor rigorous, but hopefully will give a sufficient understanding that subsequent refinement will not be too strenuous.

Overview

A bash script is a text file that consists of a series of lines. Each line is read and interpreted, if necessary.

Any line that starts with a # symbol is treated as a comment and no further interpretation is performed on it. In fact, any line whose first non-blank character is a # symbol is a comment.

Any completely empty or completely blank line is also ignored.

All other lines will contain a command which will be interpreted according to the rules of the bash language.

If a line ends with a \ (back slash) and no other characters (including blanks), then the end of line is ignored and the next line is appended to the present one, the back slash being removed.

Variables

Variables can be defined. Their names begin with a letter (either upper of lower case) or an underscore, and continue with a string of either letters, underscore or digits.

To give a variable a value, the variable name is followed by an equal sign and then the value to assign to it. No other text should appear on the line or it will be interpreted differently.

To refer to the value contained in a variable, use the symbol $ immediately before the variable name. An alternative, which is sometimes necessary, is to enclose the name is {} (curly braces) before appending the $ symbol. A use of this will appear later.

There are extra features to how a variable can be referred to, but these are not elaborated here.

Commands

A command line consists of the name of the *command*, optionally followed by one or more *options*, and then optionally followed by one or more *arguments*. Each *command* name, *option* or option value and *argument* is separated from the following one by one or more blanks. Note that sometimes a *argument* is referred to as an *parameter*.

Options start with a – sign (minus or hyphen on the keyboard) and have a single character; or start with two – signs (--) and a longer string of characters. Some options make take values, which are then put after the appropriate option separated from it by a blank.

After all the options, the parameters are given in the order to be passed to the command.

Command names can be either the full path name of the command to execute, or just the command name. If the full path is not specified, then the name is looked up in a set of directories supplied in a special variable $PATH, each directory name is separated from the others by a colon.

When you log in, the variable $PATH is set automatically by the system. If a file called .profile exists in your home directory, this is then invoked and all the commands found there are executed. In this way the $PATH variable can be further modified to your own requirements.

A line may contain more than one command, provided they are separated by semicolons.

Command line substitution

As the interpreter reads each line, it first does various textual substitutions on it before breaking it up into the command name, options and parameters.

Each variable name beginning with a $ sign is replaced by the value of the variable.

A string beginning and ending with a ' (single quote) symbol is treated as a literal string consisting of the characters between the quotes, and is treated as a single value. Hence any blanks within the string are kept as part of the argument or value.

A string beginning and ending with a " (double quote) will also be treated as a single value, but other substitutions will be done first. So any variable name preceded by a $ will be replaced by its value.

Any name of a file or directory is known as a pathname. The shell will look for special characters in pathnames and expand them. These are known as wildcards. The most common and almost the only one you might use in the beginning is the * (asterisk) character. When an asterisk is encountered in a pathname, the pathname is treated as a pattern and is replaced a list of all names that match the other characters in place with the * replaced by zero or more characters. For example the string /bin/*sh might expand to

```
    /bin/bash /bin/dash /bin/rbash /bin/sh /bin/static-sh
```

But enclosing in either single or double quotes will not expand the pattern.

Other wildcards are ? to represent exactly one character; or the string [*set*] which will match any character found in *set*. Also, the pattern [!*set*] matches any character *not* in *set*.

There is a special character ~ which, when it occurs as the first character in a pathname, converts to the name of the home directory.

Special variables

Some strings starting with a $ have special significance, and are known as built-in shell variables but most of them are useful only when used inside scripts.

The string $1 refers to the first argument passed to the script when it is being executed. Similarly $2 is the second, $3 the third, etc. But after 9, you have to use ${10}, ${11} etc., in order to prevent the interpreter taking only a single digit.

$0 gives you the name of the command being executed. This will either be just the name, or the full pathname to the command depending on how it was invoked.

To find the number of arguments, $# will give it to you.

$@ will expand to contain all the arguments in the order provided to the command. If any arguments contain blanks, you may need instead "$@" which will ensure that the arguments are kept as individual arguments.

$* also expands to contain the arguments, but "$*" provides them all together in a single string.

Redirection

When a command executes it is provided with certain input and output files. It also returns an exit code. By convention, a return code of 0 is normal, and any other exit code is an error or abnormal. The special variable $? contains the value of the exit code from the last command executed.

There is a single standard input stream to each command. If you are executing it from a terminal, then by default, the stream is from the terminal and the command will read any further input you type until it terminates. If you want the command to read its standard input from a file, then the filename is preceded by a < character. For example:

```
command options arguments < filename
```

The blank between < and the filename is optional.

There are two output files, a standard output where normal output is directed, and an error output where error messages and other information can be written. By default the standard output is to the terminal if you are executing from the command line of a terminal. To write to a file instead, then use, for example:

```
command options arguments > filename
```

By default, the eror output is also sent to the terminal, but to write it instead to a file, then this example shows how:

```
command options arguments 2> filename
```

There is a special file named */dev/null* which always exists, but never contains anything and will swallow all output like a black hole. This can be very useful to delete unwanted output from a command, or to provide no input.

One of the most characteristic features in Unix systems is the idea of chaining of commands, pushing the output from one into the input to the next. The character | is used to accept output from the command on its left, and route it to the input of the command on its right. For example, if you wish to sort a file and record the number of lines that are equal, then this line will do so:

```
sort < file | uniq -c
```

Sometimes it very useful to capture the output of a command in a shell variable, and we do this with the syntax $(*command*). For example, to get what shell scripts exist on the system, we can do the following:

```
ls /bin/*sh
```

But to see what sorts of files they are:

```
file $(ls /bin/*sh)
```

The bash shell will execute the *ls* command and then place the result as a list to the *file* command. However, if any of the filenames retrieved by *ls* contain a blank, then the strings produced will not be what you want in this case. To keep each filename as a separate argument to the *file* command, you have to enclose the invocation in double quotes:

```
file "$(ls /bin/*sh)"
```

Or to put the list in a variable:

```
shells="$(ls /bin/*sh)"
```

If statements

We have seen that $? give the value of the exit code from a command, and so we can use that to determine what to do next by testing it. A normal return of 0 is treated as true, and an abnormal return of nonzero is false. The way we can do the test is:

```
if condition
  then
    commands
  else
    commands
fi
```

where *condition* is a command whose exit code will be tested, and *commands* are any number of other commands that are executed according to the condition detected. The *else* clause is optional.

There is a command called *test* that can give you many different pieces of information about files and directories. For example, to test whether a file with a certain name exists:

```
if test -e pathname
```

will execute the *then* clause if *pathname* exists, but it may be a directory, link, or other pseudo-file. To determine whether it is regular simple file, you can test it with:

```
if test -f pathname
```

To make it easier to read, the *test* command has an unusual alias, the symbol [. But in this case, a closing] must also appear. So the above test could also be written:

```
if [ -f pathname ]
```

Many sorts of test are possible concerning the type or attribute of pathnames; strings can be tested for equality, and numbers can be compared numerically. A list can be found in the reference.

Arithmetic

Bash contains some simple arithmetic operations. Values are all integers, not real. The operations include plus (+), minus (-), times (*) and divide (/); the divide operations truncates towards zero and give an integral answer.

To assign an integer value is exactly as expected:

```
P=2; echo $P
```

will display the value 2.

To perform arithmetic, the expression is enclosed by $((…)), so to change the value of P:

```
P=$((P – 5)); echo $P
```

will print -3.

Loops

The simplest loop is one which assigns to the loop variable a value from a list in turn:

```
for X in a e i o u
do
  grep $X textfile
done
```

will first print all the lines in *textfile* containing the letter *a*, then those which contain *e*, and so on, as the value of $X takes the value of each vowel in turn.

To loop for a specific number of times, say given by the value in variable N, then the following form will work:

```
for ((K = 1; K <= $N; K = K+1))
then
  commands
done
```

where K will take on the values 1, 2, 3, … upto the value of N in turn.

Another type of loop is one where the loop keeps repeating until a test fails to be satisfied. This is most useful when looping through a stream of data, but I refer you to the book in the reference for the details.

References

A good book on the bash shell language is Classic Shell Scripting by Arnold Robbins and Nelson Beebe, published by O'Reilly in 2005.