Unix Shell Scripting Tutorial

Ashley J.S Mills

<ashley@ashleymills.com>

Table of Contents

1. Introduction

Time is precious. It is non-sense-ical to waste time typing a frequently used sequence of commands at a command prompt, more especially if they are abnormally long or complex. Scripting is a way by which one can alleviate this necessity by automating these command sequences in order to make ones life at the shell easier and more productive. Scripting is all about making the computer, *the tool*, do the work. Hopefully by the end of this tutorial you should have a good idea of the kinds of scripting languages available for Unix and how to apply them to your problems.

2. Environment

In order to peruse this tutorial with ease it is probably necessary to know a little bit about how Unix works, if you are new to Unix you should read the documentation: *Configuring A Unix Working Environment* [../unixenvars/unixenvarshome.html], if you are not new to Unix you should browse it to check that you know everything it discusses.

Unix contains many wonderful and strange commands that can be very useful in the world of scripting, the more of the tools you know and the better you know them, the more use you will find for them in your scripting. Most of the Unix commands and many of the builtin commands have man pages, *man* pages contain the usage instructions and other stuff pertaining to the parent tool. They are not always very clear and may require reading several times. In order to access a man page in Unix the following command sequence is applied:

```
man command
```

If a man page exists for the command specified the internal viewer will be invoked and you will be able to read about the various options and usage instructions etc.

3. Shell Scripting

3.1. Shell Scripting Introduction

Unix uses shells to accept commands given by the user, there are quite a few different shells available. The most commonly used shells are SH(Bourne SHell) CSH(C SHell) and KSH(Korn SHell), most of the other shells you encounter will be variants of these shells and will share the same syntax, KSH is based on SH and so is BASH(Bourne again shell). TCSH(Extended C SHell) is based on CSH.

The various shells all have built in functions which allow for the creation of shell scripts, that is, the stringing together of shell commands and constructs to automate what can be automated in order to make life easier for the user.

With all these different shells available, what shell should we script in? That is debatable. For the purpose of this tutorial we will be using SH because it is practically guaranteed to be available on most Unix systems you will encounter or be supported by the SH based shells. Your default shell may not be SH. Fortunately we do not have to be using a specific shell in order to exploit its features because we can specify the shell we want to interpret our shell script within the script itself by including the following in the first line.

```
#!/path/to/shell
```

Usually anything following (#) is interpreted as a comment and ignored but if it occurs on the first line with a (!) following it is treated as being special and the filename following the (!) is considered to point to the location of the shell that should interpret the script.

When a script is "executed" it is being interpreted by an invocation of the shell that is running it. Hence the shell is said to be running non-interactively, when the shell is used "normally" it is said to be running interactively.

Note

There are many variations on the basic commands and extra information which is too specific to be mentioned in this short tutorial, you should read the man page for your shell to get a more comprehensive idea of the options available to you. This tutorial will concentrate on highlighting the most often used and useful commands and constructs.

3.2. Shell Scripting Basics

3.2.1. Command Redirection and Pipelines

By default a normal command accepts input from standard input, which we abbreviate to stdin, standard input is the command line in the form of arguments passed to the command. By default a normal command directs its output to standard output, which we abbreviate to stdout, standard output is usually the console display. For some commands this may be the desired action but other times we may wish to get our input for a command from somewhere other than stdin and direct our output to somewhere other than stdout. This is done by redirection:

- We use > to redirect stdout to a file, for instance, if we wanted to redirect a directory listing generated by the **ls** we could do the following:

  ```
  ls > file
  ```

- We use < to specify that we want the command immediately before the redirection symbol to get its input from the source specified immediately after the symbol, for instance, we could redirect the input to **grep**(which searches for strings within files) so that it comes from a file like this:

  ```
  grep searchterm < file
  ```

- We use >> to append stdout to a file, for instance, if we wanted to append the date to the end of a file we could redirect the output from **date** like so:

  ```
  date >> file
  ```

- One can redirect standard error (stderr) to a file by using **2>**, if we wanted to redirect the standard error from commandA to a file we would use:

  ```
  commmandA 2>
  ```

Pipelines are another form of redirection that are used to chain commands so that powerful composite commands can be constructed, the pipe symbol '|' takes the stdout from the command preceding it and redirects it to the command following it:

```
        ls -l | grep searchword | sort -r
```

The example above firsts requests a long (`-l` directory listing of the current directory using the **ls** command, the output from this is then piped to **grep** which filters out all the listings containing the searchword and then finally pipes this through to **sort** which then sorts the output in reverse (`-r`, **sort** then passes the output on normally to stdout.

3.2.2. Variables

3.2.2.1. Variables

When a script starts all environment variables are turned into shell variables. New variables can be instantiated like this:

```
name=value
```

You must do it exactly like that, with no spaces either side of the equals sign, the name must only be made up of alphabetic characters, numeric characters and underscores, it cannot begin with a numeric character. You should avoid using keywords like *for* or anything like that, the interpreter will let you use them but doing so can lead to obfuscated code ;)

Variables are referenced like this: `$name`, here is an example:

```
#!/bin/sh
msg1=Hello
msg2=There!
echo $msg1 $msg2
```

This would echo "Hello There!" to the console display, if you want to assign a string to a variable and the string contains spaces you should enclose the string in double quotes ("), the double quotes tell the shell to take the contents literally and ignore keywords, however, a few keywords are still processed. You can still use *$* within a (") quoted string to include variables:

```
#!/bin/sh
msg1="one"
msg2="$msg1 two"
msg3="$msg2 three"
echo $msg3
```

Would echo "one two three" to the screen. The escape character can also be used within a double quoted section to output special characters, the escape character is *"\"*, it outputs the character immediately following it literally so \\ would output \. A special case is when the escape character is followed by a newline, the shell ignores the newline character which allows the spreading of long commands that must be executed on a single line in reality over multiple lines within the script. The escape character can be used anywhere else too. Except within single quotes.

Surrounding anything within single quotes causes it to be treated as literal text that is it will be passed on exactly as intended, this can be useful for sending command sequences to other files in order to create new scripts because the text between the single quotes will remain untouched. For example:

```
#!/bin/sh
echo 'msg="Hello World!"' > hello
echo 'echo $msg' >> hello
chmod 700 hello
./hello
```

This would cause "msg="Hello World!" to be echoed and redirected to the file `hello`, "echo $msg" would then be echoed and redirected to the file `hello` but this time appended to the end. The *chmod* line changes the file permissions of `hello` so that we can execute it. The final line executes **hello** causing it output "Hello World". If we had not used literal quotes we never would have had to use escape characters to ensure that ($) and (") were echoed to the file, this makes the code a little clearer.

A variable may be referenced like so *${VARIABLENAME},* this allows one to place characters immediately preceding the variable like *${VARIABLENAME}*aaa without the shell interpreting *aaa* as being part of the variable name.

3.2.2.2. Command Line Arguments

Command line arguments are treated as special variables within the script, the reason I am calling them variables is because they can be changed with the **shift** command. The command line arguments are enumerated in the following manner *$0*, *$1*, *$2*, *$3*, *$4*, *$5*, *$6*, *$7*, *$8* and *$9*. *$0* is special in that it corresponds to the name of the script itself. *$1* is the first argument, *$2* is the second argument and so on. To reference after the ninth argument you must enclose the number in brackets like this *${nn}*. You can use the **shift** command to shift the arguments 1 variable to the left so that *$2* becomes *$1*, *$1* becomes *$0* and so on, *$0* gets scrapped because it has nowhere to go, this can be useful to process all the arguments using a loop, using one variable to reference the first argument and **shifting** until you have exhausted the arguments list.

As well as the commandline arguments there are some special builtin variables:

- *$#* represents the parameter count. Useful for controlling loop constructs that need to process each parameter.
- *$@* expands to all the parameters separated by spaces. Useful for passing all the parameters to some other function or program.
- *$-* expands to the flags(options) the shell was invoked with. Useful for controlling program flow based on the flags set.
- *$$* expands to the process id of the shell innovated to run the script. Useful for creating unique temporary filenames relative to this instantiation of the script.

Note

The commandline arguments will be referred to as parameters from now on, this is because SH also allows the definition of functions which can take parameters and when called the *$n* family will be redefined, hence these variables are always parameters, its just that in the case of the parent script the parameters are passed via the command line. One exception is *$0* which is always set to the name of the parent script regardless of whether it is inside a function or not.

3.2.2.3. Command Substitution

In the words of the SH manual “Command substitution allows the output of a command to be substituted in place of the command name itself”. There are two ways this can be done. The first is to enclose the command like this:

```
$(command)
```

The second is to enclose the command in back quotes like this:

```
`command`
```

The command will be executed in a sub-shell environment and the standard output of the shell will replace the command substitution when the command completes.

3.2.2.4. Arithmetic Expansion

Arithmetic expansion is also allowed and comes in the form:

```
$((expression))
```

The value of the expression will replace the substitution. Eg:

```
!#/bin/sh
echo $((1 + 3 + 4))
```

Will echo "8" to stdout

3.2.3. Control Constructs

The flow of control within SH scripts is done via four main constructs; *if...then...elif..else*, *do...while*, *for* and *case*.

3.2.3.1. If..Then..Elif..Else

This construct takes the following generic form, The parts enclosed within ([) and (]) are optional:

```
if list
then list
[elif list
then list] ...
[else list]
fi
```

When a Unix command exits it exits with what is known as an *exit status*, this indicates to anyone who wants to know the degree of success the command had in performing whatever task it was supposed to do, usually when a command executes without error it terminates with an exit status of zero. An exit status of some other value would indicate that some error had occurred, the details of which would be specific to the command. The commands' manual pages detail the exit status messages that they produce.

A list is defined in the SH as "a sequence of zero or more commands separated by newlines, semicolons, or ampersands, and optionally terminated by one of these three characters.", hence in the generic definition of the *if* above the list will determine which of the execution paths the script takes. For example, there is a command called **test** on Unix which evaluates an expression and if it evaluates to true will return zero and will return one otherwise, this is how we can test conditions in the *list* part(s) of the *if* construct because **test** is a command.

We do not actually have to type the **test** command directly into the *list* to use it, it can be implied by encasing the test case within ([) and (]) characters, as illustrated by the following (silly) example:

```
#!/bin/sh
if [ "$1" = "1" ]
then
   echo "The first choice is nice"
elif [ "$1" = "2" ]
then
   echo "The second choice is just as nice"
elif [ "$1" = "3" ]
then
   echo "The third choice is excellent"
else
   echo "I see you were wise enough not to choose"
   echo "You win"
fi
```

What this example does is compare the first parameter (command line argument in this case) with the strings "1", "2" and "3" using **test**s' (=) test which compares two strings for equality, if any of them match it prints out the corresponding message. If none of them match it prints out the final case. OK the example is silly and actually flawed (the user still wins even if they type in (4) or something) but it illustrates how the *if* statement works.

Notice that there are spaces between (if) and ([), ([) and the test and the test and (]), these spaces must be present otherwise the shell will complain. There must also be spaces between the operator and operands of the test otherwise it will not work properly. Notice how it starts with (if) and ends with (fi), also, notice how (then) is on a separate line to the test above it and that (else) does not require a (then) statement. You must construct this construct exactly like this for it to work properly.

It is also possible to integrate logical AND and OR into the testing, by using two tests separated by either "&&" or "||" respectively. For example we could replace the third test case in the example above with:

```
elif [ "$1" = "3"] || [ "$1" = "4" ]
then echo "The third choi...
```

The script would print out "The third choice is excellent" if the first parameter was either "3" *OR* "4". To illustrate the use of "&&

```
elif [ "$1" = "3"] || [ "$2" = "4" ]
then echo "The third choi...
```

The script would print out "The third choice is excellent" if and only if the first parameter was "3" *AND* the second parameter was "4".

"&&" and "||" are both lazily evaluating which means that in the case of "&&", if the first test fails it wont bother evaluating the second because the list will only be true if they *BOTH* pass and since one has already failed there is no point wasting time evaluating the second. In the case of "||" if the first test passes it wont bother evaluating the second test because we only need *ONE* of the tests to pass for the whole list to pass. See the **test** manual page for the list of tests possible (other than the string equality test mentioned here).

3.2.3.2. Do...While

The *Do...While* takes the following generic form:

```
while list
do list
done
```

In the words of the SH manual "The two lists are executed repeatedly while the exit status of the first list is zero." there is a variation on this that uses *until* in place of *while* which executes *until* the exit status of the first list is zero. Here is an example use of the *while* statement:

```
#!/bin/sh
count=$1                                  # Initialise count to first parameter
while [ $count -gt 0 ]                    # while count is greater than 10 do
do
   echo $count seconds till supper time!
   count=$(expr $count -1)                # decrement count by 1
   sleep 1                                # sleep for a second using the Unix sleep command
done
echo Supper time!!, YEAH!!                # were finished
```

If called from the commandline with an argument of 4 this script will output

```
4 seconds till supper time!
3 seconds till supper time!
2 seconds till supper time!
1 seconds till supper time!
Supper time!!, YEAH!!
```

You can see that this time we have used the `-gt` of the **test** command implicitly called via '[' and ']', which stands for greater than. Pay careful attention to the formatting and spacing.

3.2.3.3. For

The syntax of the for command is:

```
            for variable in word ...
            do list
            done
```

The SH manual states “The words are expanded, and then the list is executed repeatedly with the variable set to each word in turn.”. A word is essentially some other variable that contains a list of values of some sort, the *for* construct assigns each of the values in the *word* to *variable* and then *variable* can be used within the body of the construct, upon completion of the body *variable* will be assigned the next value in *word* until there are no more values in *word*. An example should make this clearer:

```
#!/bin/sh
fruitlist="Apple Pear Tomato Peach Grape"
for fruit in $fruitlist
do
   if [ "$fruit" = "Tomato" ] || [ "$fruit" = "Peach" ]
   then
      echo "I like ${fruit}es"
   else
```

```
      echo "I like ${fruit}s"
   fi
done
```

In this example, *fruitlist* is *word*, *fruit* is *variable* and the body of the statement outputs how much this person loves various fruits but includes an *if...then..else* statement to deal with the correct addition of letters to describe the plural version of the fruit, notice that the variable *fruit* was expressed like *${fruit}* because otherwise the shell would have interpreted the preceding letter(s) as being part of the variable and echoed nothing because we have not defined the variables *fruits* and *fruites* When executed this script will output:

```
I like Apples
I like Pears
I like Tomatoes
I like Peachs
I like Grapes
```

Within the *for* construct, *do* and *done* may be replaced by *'{'* and *'}'*. This is not allowed for *while*.

3.2.3.4. Case

The case construct has the following syntax:

```
case word in
pattern) list ;;
...

esac
```

An example of this should make things clearer:

```
!#/bin/sh
case $1
in
1) echo 'First Choice';;
2) echo 'Second Choice';;
*) echo 'Other Choice';;
esac
```

"1", "2" and "*" are patterns, *word* is compared to each pattern and if a match is found the body of the corresponding pattern is executed, we have used "*" to represent everything, since this is checked last we will still catch "1" and "2" because they are checked first. In our example *word* is "$1", the first parameter, hence if the script is ran with the argument "1" it will output "First Choice", "2" "Second Choice" and anything else "Other Choice". In this example we compared against numbers (essentially still a string comparison however) but the pattern can be more complex, see the SH man page for more information.

3.2.4. Functions

The syntax of an SH function is defined as follows:

```
name ( ) command
```

It is usually laid out like this:

```
name() {
commands
}
```

A function will return with a default exit status of zero, one can return different exit status' by using the notation *return exit status*. Variables can be defined locally within a function using *local name=value*. The example below shows the use of a user defined increment function:

Example 1. Increment Function Example

```
#!/bin/sh
inc() { ❶                              # The increment is defined first so we can use it
   echo $(($1 + $2))                # We echo the result of the first parameter plus the second paramete
}

                                    # We check to see that all the command line arguments are present
if [ "$1" "" ] || [ "$2" = "" ] || [ "$3" = "" ]
then
```

```
   echo USAGE:
   echo "   counter startvalue incrementvalue endvalue"
else
   count=$1                       # Rename are variables with clearer names
   value=$2
   end=$3
   while [ $count -lt $end ]      # Loop while count is less than end
   do
      echo $count
      count=$(inc $count $value)  ❷# Call increment with count and value as parameters
   done                           # so that count is incremented by value
fi
```

❶

```
inc() {
   echo $(($1 + $2))
}
```

The function is defined and opened with *inc() {*, the line *echo $(($1 + $2))* uses the notation for arithmetic expression substitution which is *$((*expression*))* to enclose the expression, *$1 + $2* which adds the first and second parameters passed to the function together, the echo bit at the start echoes them to standard output, we can catch this value by assigning the function call to a variable, as is illustrated by the function call.

❷

```
count=$(inc $count $value)
```

We use command substitution which substitutes the value of a command to substitute the value of the function call whereupon it is assigned to the *count* variable. The command within the command substitution block is *inc $count $value*, the last two values being its parameters. Which are then referenced from within the function using *$1* and *$2*. We could have used the other command substitution notation to call the function if we had wanted:

```
count=`inc $count $value`
```

We will show another quick example to illustrate the scope of variables:

Example 2. Variable Scope, Example

```
#!/bin/sh
inc() {
   local value=4  ❶
   echo "value is $value within the function\\n"
   echo "\\b\$1 is $1 within the function"
}
value=5
echo value is $value before the function
echo "\$1 is $1 before the function"
echo
echo -e $(inc $value)  ❷
echo
echo value is $value after the function
echo "\$1 is $1 after the function"
```

❶

```
inc() {
   local value=4
   echo "value is $value within the function\\n"
   echo "\\b\$1 is $1 within the function"
}
```

We assign a local value to the variable *value* of 4. The next three lines construct the the output we would like, remember that this is being echoed to some buffer and will be replace the function call with all the stuff that was passed to stdout within the function when the function exits. So the calling code will be replaced with whatever we direct to standard output within the function. The function is called like this:

```
echo -e $(inc $value)
```

We have passed the option `-e` to the **echo** command which causes it to process C-style backslash escape characters, so we can process any backslash escape characters which the string generated by the function call contains.

If we just echo the lines we want to be returned by the function it will not pass the newline character onto the buffer even if we explicitly include it with an escape character reference so what we do is actually include the sequence of characters that will produce a new line within the string so that when it is echoed by the calling code with the `-e` the escape characters will be processed and the newlines will be placed where we want them.

```
echo "value is $value within the function\\n"
```

Notice how the newline has been inserted with *\\n*, the first two backslashes indicate that we want to echo a backslash because within double quotes a backslash indicates to process the next character literally, we have to do this because we are only between double quotes and not the literal-text single quotes. If we had used single quotes we would had have to echo the bit with the newline in separately from the bit that contains *$value* otherwise *$value* would not be expanded.

```
echo "\\b\$1 is $1 within the function"
```

This is our second line, and is contained within double quotes so that the variable *$1* will be expanded, *\\b* is included so that *\b* will be placed in the echoed line and our calling code will process this as a backspace character. We have to do this because for some reason the shell prefixes a space to the second line if we do not, the backspace removes this space.

The output from this script called with 2 as the first argument is:

```
value is 5 before the function
$1 is 2 before the function

value is 4 within the function
$1 is 5 within the function

value is 5 after the function
$1 is 2 after the function
```

Tip

You can use ". DIRECTORY/common.sh" to import functions from a script called common.sh in DIRECTORY, a quick example is shown below, first is test.sh:

```
#!/bin/sh
. ./common.sh
if [ "$1" = "" ]; then
   echo USAGE:
   echo "sh test.sh type"
   exit
fi

if `validtype $1`; then
   echo Valid type
else
   echo Invalid type
fi
```

Here is common.sh:

```
#!/bin/sh
validtype() {
   if [ "$1" = "TYPEA" ] ||
      [ "$1" = "TYPEB" ] ||
      [ "$1" = "TYPEC" ] ||
      [ "$1" = "TYPED" ] ||
      [ "$1" = "TYPEE" ];
   then
      exit 0
   else
      exit 1
   fi
}
```

If you need to learn more, checkout http://www.japarker.btinternet.co.uk/sh/ for what looks like an excellent Bourne Shell Programming tutorial.