Bash Shell Scripting Tutorial

1 Introduction

The shell is the program we interact with when we type at a Unix command line prompt. There are actually several different Unix shell programs; the most commonly used is `bash`. `bash` and other shells include facilities for writing programs, called "shell scripts". (Different shells have overlapping but distinct capabilities and syntax; in this tutorial, we assume the shell is `bash`.)

Shell scripts can be used to automate tasks involving Unix, as well as command-line programs that run under Unix. The shell scripting language reflects an accretion of good ideas and convenient tricks, in contrast to coherently designed, full-featured programming languages such as Python, which can also be used to write scripts for automating tasks. However, shell scripting is convenient because of its direct relationship to the well-conceived set of command-line tools that Unix comprises. All Unix commands can be used directly in shell scripts; indeed, many useful shell scripts consist of little more than a sequence of commands that could be typed at the command line.

Shell scripting benefits immensely from the Unix concept of small, well-defined command line utilities that can be combined to perform larger tasks by use of pipes and redirection of standard input and output. Familiarity with this approach is a prerequisite for writing good scripts. Likewise, powerful and versatile command line tools such as `grep, sed,` and `awk` greatly extend the range of what simple scripts can do.

The disadvantages of shell scripting for tasks beyond a certain complexity arise as well from its simple design. First among these limitations is the lack of convenient facilities for mathematical operations on real numbers, or data structures beyond integers, strings, and simple arrays. As a result, shell scripts are preferred for automation tasks, but not for "number crunching".

Of course, other shell scripting tutorials exist; many are rather brief and thus of limited use. One extensive tutorial is `http://tldp.org/LDP/abs/html/`, which is a useful reference but formidable to digest. This tutorial presents those features of shell scripting that commonly appear in scripts written for my research group.

The elements of the language are introduced here with brief but functional sample code. By themselves, these examples are too limited to give a clear indication of how real scripts are constructed.

However, similar tasks and consequently similar techniques occur again and again in our scripts. Therefore, in the final part of this tutorial we present several complete scripts from real applications, annotated to highlight techniques we have found useful.

2 Language elements

2.1 Variables

The traditional first program in any language prints out "Hello world". Here is a shell script that illustrates the use of variables:

```
#!/bin/bash

# to use a variable, just assign it
sayIt="Hello"
sayit=$sayIt" world"
echo $
```

In bash scripts, `#` denotes a comment, and blank lines are allowed. The first line above is a comment to Unix, that the script should be executed with `bash`.

In the script above, `sayIt` is first assigned the value "Hello". Then, `sayIt` is assigned its old value, concatenated with " world". The value of a variable is accessed with `$`.

Shell variables need not be "declared" or "defined" first; just start using them. Shell variables do not have "types" (integer, real, string, etc.). More precisely, all shell variables are strings, but can be interpreted and used as integers (discussed below).

2.2 Arguments

Values can be passed to shell scripts by command-line arguments. Here is an example (`script1.sh`):

```
#!/bin/bash
echo Hello $1
```

`script1.sh Scott` prints out "Hello Scott". Up to 9 command-line arguments are accessed with `$1` and so forth. `$#` is the number of arguments passed.

2.3 Strings

Shell variables hold strings, which are assigned with `=` (no space on either side):

```
myString=Hello
```

To assign a string containing spaces, enclose it in double quotes:

```
myString="Hello world"
```

To assign a string containing special characters that should not be "processed", enclose it in single quotes:

```
myString='echo $1'
```

Strings are concatenated by placing them together:

```
string1="Hello "
string2="world!"
bothStrings=$string1$string2
```

2.4 Catching output from commands

Shell scripts can contain any Unix command. Often, we want to save the output of the command in a variable. This is done with backquotes (`` `...` ``), as follows:

```
myFiles=`ls`
```

myFiles now contains a string with all the filenames separated by spaces.

An equivalent form is `$(...)`:

```
myFiles=$(ls)
```

As usual in Unix, the command inside `` `...` `` or `$(...)` can be a sequence of commands separated by semicolons, or a compound command involving pipes.

2.5 Integers

Shell variables are fundamentally strings; however, shell scripting includes syntax that allows us to treat variables as integers.

One way is to use `let`, which allows us to do math on variables that contain integers, using operations `+, -, *, /, **` (exponentiation), and parentheses:

```
z=4
let z=($z/2 + 4**2)*$z
```

which replaces `z` with 72. Note that division (`/`) here is *integer*, i.e., 5/2=2. Spaces are not permitted around the `=`, but can appear in the expression.

A more convenient way to do math on integer-valued variables is with `((...))`. Expressions inside double parentheses act as they would in C/C++, with no need to use `$` to obtain the value of variables. For example:

```
((e=4*e+3**2))
```

The right side evaluates without typing `$e`, and the assignment is made. Operators like `++` (increment) and `%` (mod) also work inside `((...))`.

2.6 Loops

Automation scripts typically involve looping. We may loop over values of some parameter, or a list of files to be processed. Or, we may continue a process while or until some condition is met.

Shell scripting includes three constructs for looping: `for, while,` and `until`. All have a similar syntax, with the body of the loop enclosed by `do` and `done`. `while` and `until` loops are controlled by a logical test (described below):

```
while <logical test>
do
<command1>
<command2>
...
done
```

`for` loops in shell scripts are different from those in C/C++; they iterate over a list of choices:

```
for fruit in apple pear orange
do
     echo $fruit
done
```

The list of choices can be contained in a variable, or the output of a command:

```
choiceList="apple pear orange"
for fruit in $choiceList
...
```

```
for file in ‘ls‘
...
```

To get a C-like `for` loop, use the shell command `seq` to generate a sequence to iterate over:

```
for i in ‘seq 1 10‘
...
```

The general form of `seq` is `seq <start> <step> <end>`, where `<start>` and so forth are integers (or expressions that evaluate to integers), and `<step>` is optional (with default 1).

2.7 Logical tests

To compare two variables, we use a logical test (note the spaces between the brackets, variables, and `==`):

```
[ $string1 == $string2 ]
```

To test for inequality, use `!=`.

If the variables contain integers (see below), comparison operators `-eq, -ne, -le, -ge, -lt, -gt` can be used.

A more convenient form of logical test for integer-valued variables is to use `((...))`. In the same way as arithmetic operations on integers (see above), expressions inside `((...))` using the usual comparison operators `<, >, <=, >=, != ==, &&` (and), and `||` (or) will evaluate to “true” or “false”.

Thus we can write logical tests in loops like

```
while ((x**2+y**2 < 100))
...
```

and in if statements (see below) like

```
if ((x < 2*y));
then
...
```

2.8 Control statements

In addition to looping, shell scripting provides control statements to direct the execution. The simplest is the if-statement:

```
if [ $string1 == $string2 ];
then
    echo "equal"
else
    echo "not equal"
fi
```

The logical test `[...]` takes the form described in the previous section, and must be followed by a semicolon. The `then` and `else` sections can contain multiple commands. The `else` clause may be omitted.

An if-statement with an "else-if" secondary test can be written as

```
if [ $string1 == $string2 ];
then
    echo "equal"
elif [ $string1 == "default" ];
then
    echo "default"
fi
```

In the above example, for simplicity the `else` cases have been omitted from the primary and secondary tests.

2.9 Random choices

The shell provides a facility for generating 16-bit random integers (ranging from 0 to 2^{15}-1 = 32767). Each time the shell variable `$RANDOM` is evaluated, a different pseudorandom number is obtained.

These can be used to make random choices among a relatively small number of alternatives, in conjunction with the mod (`%`) operator and integer arithmetic.

The mod operator computes the remainder of an integer divided by the modulus: `echo $((5 % 2))` gives 1, while `echo $((-5 % 2))` gives -1.

For example, to choose between two possibilities with equal probability,

```
if (($RANDOM % 2 == 0));
...
```

2.10 Case statements

Multiway branching can be performed with a `case` statement. The comparison is different from `if`; here, the argument is matched to a succession of "patterns", each of which is terminated with a closing parenthesis `)`:

```
case $percent in
[1-4]*)
     echo "less than 50 percent" ;;
[5-8]*)
     echo "less than 90 percent" ;;
*)
    echo "90 percent or greater" ;;
esac
```

Note the double semicolon terminating each command (which as usual can be multiple commands on multiple lines).

The patterns to be matched use the same syntax as in other Unix operations (commonly used in `vi, grep, ls, rm`, and the like). For completeness, patterns are described briefly in the next section.

2.11 Patterns

Patterns (or "regular expressions") are used by several Unix command-line programs and utilities. File management programs like `ls`, `cp`, `mv`, `rm`, and `tar` use patterns to select files to operate on.

The Unix utility `grep` selects lines from its input (stdin) that match a pattern, and prints those lines to stdout. The Unix visual editor `vi`, and the stream editor `sed` use patterns in its searching and substitution.

Patterns are strings, possibly containing special characters, which match part or all of other strings. A simple example is `ls *.sh`, which lists all files with names ending in `.sh` (i.e., all shell scripts, if we adhere to that naming convention). Because effective use of patterns extends the capabilities of Unix tools like `grep` and `sed` that frequently appear in scripts, we include here a table of the special characters used in patterns and their meanings.

Table 1: Special characters used in regular expressions and their meanings.

pattern	matches
`^`	beginning of string
`$`	end of string
`.`	any single character
`*`	zero or more repeats of previous character
`[...]`	any one of the enclosed characters
`[a-k]`	any one of the range of characters
`[2-5]`	any one of the range of digits
`[^...]`	any character not in ...
`\<...\>`	pattern ... only matches a word
`\(...\)`	saves pattern matched by ...
`\1, \2, ...`	the first, second, ... pattern saved

Pattern matching is useful in many contexts, but can be tricky — it is often a good idea to test patterns in a safe situation before relying on them where errors can do damage (as with `rm`).

2.12 Functions

In longer scripts it is often convenient to define functions, to perform a well-define subtask that is called upon multiple times.

Command-line arguments can be passed to functions in the same way as for scripts generally, with variables `$1`, `$2` and so forth. Variables declared with `local` are protected from meddling by the main program. Shell script functions do not return values, so functions must communicate with the calling program through global variables.

The syntax for defining a function is simple:

```
function newFcn {
...
}
```

in which ... are one or more commands.

2.13 Return status

Functions can return a value that indicates the status of the function call; by convention, 0 indicates success and nonzero values indicate some error. For a user-defined function, the return value is set with `return <value>`.

All built-in shell functions return such values. The return value of the most recently called function can be accessed with `$?`.

Return values of functions can be used as logical tests. For example:

```
while read nextLine
do
   echo $nextLine
done
```

reads successive lines and echoes them until the file is empty, at which point `read` returns a nonzero value, and the `while` loop ends.

Another way to use return status of functions and commands, is together with the operators `&&` (logical and) and `||` (logical or). The syntax

```
<cmd1> && <cmd2>
```

executes the second command only if the first one *succeeds* (i.e., returns status 0). This construct can be used to run a "check" (for the existence of a file) before executing a command.

Similarly,

```
<cmd1> || <cmd2>
```

executes the second command only if the first one *fails* (i.e., returns status 1). This construct can be used to run a "fallback" command if the first one fails.

2.14 Writing to files

Output to files can be written using Unix command line tools, including `echo`, `cat`, and `printf`. `echo` writes strings; `cat` concatenates one or more files; and `printf` prints one or more variables in a specified format, similar to print statements in C/C++.

For example,

```
echo "var1 equals" $var1
```

writes the concatenated string consisting of the text strings and variable.

`printf` writes one or more variables using a C-style format string, which provides a convenient way of controlling the output format:

```
printf "%5s %10.5f%12.4g  %5i\n" $s1 $f1 $f2 $i1
```

writes string `$s1` in 5 spaces, followed by floating-point variable `$f1` in 10 spaces with 5 decimal places, `$f2` in scientific notation in 12 spaces with 4 decimal places, and integer `$i1` in five spaces.

Finally, `cat` concatenates a list of files into a single file directed to stdout, as in:

```
cat file1 file2 file3
```

2.15 Output redirection

All three utilities `echo`, `cat`, and `print` write to stdout (standard output). However, this output can be redirected to write or append to a file with `>` or `>>`. If a script is invoked with redirection, as in

```
myScript.sh > myFile.out
```

all commands in the script that write to stdout are redirected to `myFile.out`. This is convenient if the script only produces one output file, since redirection can be used to save the file with any desired name.

Sometimes, a script writes more than one output file. Then, it is convenient to redirect output of commands within the script. This is done with the same `>` and `>>` syntax. For example,

```
cat file1 file2  > file3
```

writes `file3` (overwriting if it exists); whereas,

```
cat file1 file2 >> file3
```

appends to `file3` if it exists (and creates it otherwise).

2.16 "here" documents

Sometimes, it is convenient to store a small file within a script, for use with a utility like `cat` that requires a file as input. This can be accomplished with a "here" document:

```
cat << EOF
first line
second line
third line
EOF
```

In this example, `cat` will write the file (consisting of three lines) to stdout. Here documents can be combined with output redirection, by replacing the first line with

```
cat > myFile.out << EOF
```

Here documents can be used with any command that requires a file as input, as an alternative to maintaining a separate "database file" that is redirected to input, like this:

```
cat < database.in > myFile.out
```

2.17 Reading input

The Unix utility `read` is used to get input from files. By default, `read` takes input from stdin. To read a single line of text:

```
read newLine
```

To read multiple arguments (separated by spaces) from a single line of text:

```
read arg1 arg2 arg3
```

Often, we want to read all the lines in a file, processing each line in some way until the file is empty. This can be done with a `while` loop:

```
while read newLine
do
   echo $newLine
done
```

The `read` command returns status 0 ("true") if a line is read, and nonzero ("false") if the end of file is reached. This serves as a logical test to control the `while` loop. If input is read from stdin (the keyboard), ctrl-D acts as an end-of-file to terminate input.

2.18 Reading from a string

Sometimes it is convenient to read from a string; for example, we can use read to parse a string into arguments. This may be regarded as a form of input redirection, or a variation on a "here" document.

To read from a string:

```
read arg1 arg2 arg3 <<< $mystring
```

2.19 Input redirection

If a script is invoked with input redirection, as in

```
myScript.sh < myFile.in
```

all `read` commands in the script that have not been redirected then take input from `myFile.in`. This is convenient if the script only reads from one file, which we specify when we invoke the script.

Sometimes, a script reads from more than one file. Then, we can also invoke input redirection within the script. For example:

```
while read newLine
do
echo $newLine
done < myFile.txt
```

Note how the redirection follows the entire while loop construct; all reads inside the `while` loop come from `myFile.txt`.

However, if we want to read two successive lines from a file, the obvious code

```
read line1 < myFile.txt
read line2 < myFile.txt
```

does not work as intended; both reads take the *first* line from `myFile.txt`. To achieve the desired result requires a different approach, described in the next section.

2.20 Open a file

In languages like C/C++ or Python, facilities exist for "opening" a file, which provides a file "pointer" that advances with each successive read. When reading is done, the file must be "closed".

The following syntax accomplishes this in a shell script:

```
# open the file
exec 3< inFile
# read from the file (first line)
read 0<&3 arg1 arg2
# read again (second line)
read 0<&3 arg3 arg4
# close the file
exec 3<&-
```

Here 3 is a file "channel", where 0 is standard input (stdin), 1 is standard output (stdout), and 2 is standard error (stderr). If more than one file must be opened at once, channels 4, 5, and so forth can be used.

2.21 Arrays

Full-featured programming languages offer data structures more complex than a single variable. The simplest of these are arrays. The shell provides array variables, which are lists of simple variables that can be accessed by index and iterated over.

An array can be created by reading a list of arguments separated by spaces from a single line (the flag `-a` directs read to create an array):

```
read -a myArray
```

Arrays can also be assigned by listing the elements within parentheses:

```
myArray=("apple" "pear" "orange" $otherFruit)
```

Elements of the array can be accessed by index, as

```
firstElement=${myArray[0]}
```

(The array index starts at 0.) The length of the array is accessible with the same syntax as the length of a string:

```
${#myArray[@]}
```

The entire array can be accessed as a single string (with elements separated by spaces):

```
${myArray[@]}
```

Arrays can be iterated over with a `for` loop:

```
for i in ${myArray[@]}
do
...
done
```

2.22 Substrings

Often we need to take part of a string. Substrings can be accessed as follows:

```
myString="abcdefg"
newString=${myString:2:3}
```

newString then equals `bcd`, i.e., a substring starting at character 2, of length 3.

If the `2` above is omitted, the substring starts at 1; thus `${myString::3}` equals `abc`.

Likewise, if the `:3` is omitted, the substring goes to the end; thus `${myString:2}` equals `bcdefg`.

Characters can be removed from the end of a string with

```
${myString%<pattern>}
```

where `<pattern>` is a string possibly including "wildcard" characters. For example, `?` matches any single character, so that

```
${myString%???}
```

drops the last three characters from `myString`.

Finally, the length of a string is given by

```
${#myString}
```

2.23 Floating point math

Floating point math operations in shell scripting are limited and awkward, which constrains its use in programs that crunch numbers. However, many shell scripts require only a few floating-point calculations. For this purpose, the Unix utility `bc` (basic calculator) can be used.

`bc` implements a rather modest calculator, with arithmetic operations `+, -, *, /`. When invoked with option `-l`, a very few scientific functions are available (`s(x)` = sine, `c(x)` = cosine, `a(x)` = arctan, `l(x)` = natural log, `e(x)` = exponential, `sqrt(x)` = square root).

`bc` is normally an interactive program; launched from the command line, it responds to typed input. To use `bc` in a script, input can be supplied from `echo` via a pipe:

```
echo "c(3.14159/2)" | bc -l
```

The output can be saved in a variable using backquotes `‘...‘`:

```
myResult=‘echo "c(3.14159/2)" | bc -l‘
```

Variables can be included in the input expression:

```
echo "$var1*$var2" | bc -l
```

The number of output digits can be controlled with the command `scale=<n>`. This only works if the expression includes division, which can be worked around by dividing by 1 somewhere.

```
echo "scale=3;3.14*7/2+1" | bc -l
```

2.24 Automating interactive programs

Using `echo` to supply what would have been typed input to `bc` is an example of a general approach to make interactive programs "scriptable".

Another way to do this is with input redirection from a string, which we encountered with `read` in constructions like `read arg1 arg2 <<< $inputLine`.

The same syntax can be used with `bc`, and any other program or utility that expects typed input from stdin:

```
myResult=‘bc -l <<< "c(3.14159/2)"‘
```

2.25 Auxiliary programs

Unix provides a number of very powerful utility programs, chief among them `grep`, `sed`, and `awk`, which can be used to great advantage in shell scripts.

`grep` (Globally search for Regular Expression and Print) searches for patterns ("regular expressions") on each line of its input, and writes the lines matching the pattern to stdout. The basic format is

```
grep <pattern> <file>
```

in which `<pattern>` is any string, possibly containing one or more regular expressions (see "Patterns" above). If `<file>` is not specified, `grep` takes its input from stdin.

`awk` (named for its co-creators Aho, Weinberger, Kernihan) acts on each line of its input, splitting it into fields, doing simple calculations, and printing output.

Finally, `sed` (Stream EDitor) acts with editing commands on each line of its input, producing a stream of output lines. An important use of `sed` is to produce a sequence of files from a common "template file", in which only a few parameters are replaced by some iterated value. A typical use of `sed` looks like this:

```
sed "s/ANG1/$myAng1/; s/ANG2/$myAng2/" < infile > outfile
```

Here all occurrences in `infile` of the "template parameter" `ANG1` are replaced by the value of the variable `$myAng1`, and likewise for `ANG2` and `$myAng2`. The commands within quotes are standard `vi` substitution commands.