COMP 2718: Shell Scripts: Part 1

By: Dr. Andrew Vardy

Outline

- Shell Scripts: Part 1
- Hello World
- Shebang!
- Example Project
- Introducing Variables
- Variable Names
- Variable Facts
- Arguments
- Exit Status
- Branching: `if`
- `test` - A Condition For `if`

Shell Scripts: Part 1

We'll now start to consider **shell scripts** which are described in the textbook starting with chapter 24.

A shell script is a series of commands placed in a file. The shell executes the commands in order, just as if they had been entered on the command line.

Shell scripts are used to automate various tasks and are heavily used in system adminstration.

Lets start with our first script. . .

Hello World

```
#!/bin/bash
# This is our first script.
echo 'Hello World!'
```

If you try and execute this script you will first get a "command not found" error:

```
$ hello_world.sh
-bash: hello_world.sh: command not found
```

Why? Because the script's location is probably not in the PATH. One solution is to specify the path:

```
$ ./hello_world.sh
-bash: ./hello_world.sh: Permission denied
```

But we know how to fix this problem, right? The user needs execute permission:

```
$ chmod u+x hello_world.sh
```

Now it works!

```
$ ./hello_world.sh
Hello World!
```

You can execute it without the `./` by adding `.` to the end of your path. But many suggest this is unsafe: `http://unix.stackexchange.com/questions/65700/is-it-safe-to-add-to-my-path-how-come`.

Shebang!

Lets see our script again:

```
#!/bin/bash
# This is our first script.
echo 'Hello World!'
```

Only the first line should be unfamiliar. The `#!` character sequence is known as a **shebang**. The rest of the line specifies the name of the interpreter to use to execute the script—`bash` in this case.

The fact that the shebang starts with # is important as this is the comment character. So `bash` itself will ignore the line.

This works also for other interpreted languages such as python:

```
#!/usr/bin/python
print 'Hello World!'
```

Now this script can be executed as follows:

```
$ ./hello_world.py
Hello World!
```

Without the shebang, the python interpreter would have to be called:

```
$ python hello_world.py
Hello World!
```

Example Project

We will follow along with an example that begins in chapter 25 of the textbook. The task is to write a script that produces a report in HTML format. We will start by creating a script that produces the following minimal HTML page:

```
<HTML>
  <HEAD>
      <TITLE>Page Title</TITLE>
  </HEAD>
  <BODY>
      Page body.
  </BODY>
</HTML>
```

It would be easy enough to place this in a file directly, but we will use the following script to achieve this instead. We will then add features to the script to create the page dynamically...

```
#!/bin/bash
# Program to output a system information page
echo "<HTML>"
echo "  <HEAD>"
echo "      <TITLE>Page Title</TITLE>"
echo "  </HEAD>"
echo "  <BODY>"
echo "      Page body."
echo "  </BODY>"
echo "</HTML>"
```

We have to make this script executable. When we execute it all of the text is echoed to standard output. To create an html file, use redirection:

```
$ report1.sh > /tmp/report1.html
```

Open by directing your browser to this URL: `file:///tmp/report1.html`.

We always want to avoid repetition in code. We notice that all of the `echo` commands in the previous script could be combined into one:

```
#!/bin/bash
# Program to output a system information page
echo "<HTML>
  <HEAD>
      <TITLE>Page Title</TITLE>
  </HEAD>
  <BODY>
      Page body.
  </BODY>
</HTML>"
```

The shell will keep reading a quoted string until it encounters the closing quotation mark. So the newline characters are preserved in the output.

Introducing Variables

Now we want to give the report a better title. We will use the title in a couple of places, so it makes sense to use a variable for the title. Why? In case we need to change the title, we only need to change one line.

```
-----------------------------------------------
#!/bin/bash
# Program to output report with custom title
TITLE="System Report"
echo "<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
    </BODY>
</HTML>"
-----------------------------------------------
```

All of the shell expansions we have seen can be used when defining or using variables. e.g.

```
a=z
b="a string"
c="a string and $b"
d=$(ls -l foo.txt)
e=$((5 * 7))
f="\t\ta string\n"
```

In the following script we add some further data to the report:

```
#!/bin/bash
# Program to output report with custom title w/ date
TITLE="System Report for $(date +%F)"
echo "<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
        <P>Compiled by $USER</P>
    </BODY>
</HTML>"
```

Variable Names

Shell variables must being with a letter or underscore. The remaining characters can be letters, digits, or underscores. So the following are valid names:

- `_var1`
- `A2`
- `Tomato_33`

The following are **not** valid:

- `1var`
- `BIG CHEESE`
- `x-coordinate`

Variable Facts

Constants

It is a convention, that names in ALL-CAPS are treated as constants, but this is not enforced.
If you want to force a variable to be treated as read-only, use the `declare` builtin with `-r`:

```
declare -r TITLE="Page TItle"
```

Type

By default, all variables are strings. These strings can be converted into integers for processing (covered later).

Undefined Variables

An undefined variable expands to the empty string. No warnings are issued for using an undefined variable! This can cause problems:

```
$ foo=foo.txt
$ foo1=foo1.txt
$ cp $foo $fool  # An 'l' was typed instead of '1'
cp: missing destination file operand after `foo.txt'
```

Seperating Variables From Other Text

Often we want to concatenate strings. But if you add text to the end of a variable, then you will be referring to a new (probably undefined) variable. For example, lets say you want to rename a file with a '1' suffix.

```
$ file="foo"
$ mv $file $file1
mv: missing destination file operand after ‘foo’
```

Instead, use curly braces to delimit the variable name:

```
$ mv $file ${file}1
```

Arguments

A shell script can access command-line arguments. They are accessed through the following special variables:

`$1`	First argument
`$2`	Second argument
...	...
`$9`	Ninth argument

Subsequent arguments are accessed as follows: `${10}`, `${11}`, ...

Consider the following example scripts that require arguments.

```
#!/bin/bash
echo $1 $2 $3 $4
```

The next one just sums the arguments passed in.

```
#!/bin/bash
echo $(($1 + $2))
```

String arguments not convertible to integers are simply treated as zeros. Floating-points generate an error.

Exit Status

All commands issue a value when they terminate called their **exit status**. This applies to scripts, but also to individual commands like `ls`. The exit status is a value from 0 to 255 which indicates the following:

0	Success
1-255	Failure. The value may indicate a particular problem. See "Exit Status" in command's `man` page.

We can access the exit status of the **last command executed** via the special variable `$?`.

e.g.

```
$ ls -d /usr/bin
/usr/bin
$ echo $?
0

$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
$ echo $?
2
```

The shell provides two simple builtin commands called `true` and `false` that just terminate with a 0 (`true`) or 1 (`false`).
Note that in digital logic 0 is true, while 1 is false. Not so for `bash`.

```
$ true
$ echo $?
0

$ false
$ echo $?
1
```

We can now really get started to talk about branching in shell scripts...

Branching: `if`

The ability to test a condition and execute different **branches** of a program is fundamental to all programming languages. Branching is achieved through the `if` statement which has the following form:

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

Remember that the parts in [square brackets] are optional and that `...` means that the whole section can be repeated (i.e. there can be multiple `elif` blocks).

Lets look at an example:

```
x=5
if [ $x -eq 5 ]; then
     echo "yes"
else
     echo "no"
fi
```

We can put this in a script and execute it (add the shebang) or put it in the command-line:

```
$ x=5
$ if [ $x -eq 5 ]; then echo "yes"; else echo "no"; fi
```

Note that each keyword (`if`, `then`, `else`, `fi`) must begin a line or be preceded by `;`.

Here again is the form of `if`

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

The `if` statement depends on a condition which is really just a sequence of commands. The exit status of the last of those commands is what determines what branch is taken.

Recall that the exit status of `true` and `false` were 0 (success) and 1 (failure), respectively.

```
$ if true; then echo "It's true."; fi
It's true.

$ if false; then echo "It's true."; fi
$
```

`test` - A Condition For `if`

The `if` statement tests the exit status of a set of commands, which could be anything. However, the most common strategy is to use the `test` command. On its own, `test` has two equivalent forms:

```
test expression
```

and

```
[ expression ]
```

expression evalutes to either true or false. The `test` command returns an exit status of zero for a true expression, and a status of one otherwise.

There are a wide variety of possible expressions based on files, strings, and integers, with conditions that can be combined through logical operators.

File Operators

The following are possible expressions based on **file operators**:

Expression	Is True If:
`file1 -ef file2`	*file1* and *file2* have the same inode numbers (the two filenames refer to the same file by hard linking).
`file1 -nt file2`	*file1* is newer than *file2*.
`file1 -ot file2`	*file1* is older than *file2*.
`-b file`	*file* exists and is a block-special (device) file.
`-c file`	*file* exists and is a character-special (device) file.
`-d file`	*file* exists and is a directory.
`-e file`	*file* exists.
`-f file`	*file* exists and is a regular file.
`-g file`	*file* exists and is set-group-ID.
`-G file`	*file* exists and is owned by the effective group ID.
`-k file`	*file* exists and has its “sticky bit” set.
`-L file`	*file* exists and is a symbolic link.
`-O file`	*file* exists and is owned by the effective user ID.
`-p file`	*file* exists and is a named pipe.
`-r file`	*file* exists and is readable (has readable permission for the effective user).

On the next slide is a script that provides an example of some file operators. It contains tests such as the following:

```
[ -f "$FILE" ]
```

This one checks for the existence of a file. Notice that the variable `FILE` is quoted. This is to prevent spaces or special characters within this variable from invalidating the expression. If this is not a concern, the variable can be left unquoted.

You will also see a new statement in this example:

```
exit 1
```

This terminates the script with an exit status of 1, indicating a type of failure (remember that 0 is success).

```
FILE=~/.profile
if [ -e "$FILE" ]; then
    if [ -f "$FILE" ]; then
        echo "$FILE is a regular file."
    fi
    if [ -d "$FILE" ]; then
        echo "$FILE is a directory."
    fi
    if [ -r "$FILE" ]; then
        echo "$FILE is readable."
    fi
    if [ -w "$FILE" ]; then
        echo "$FILE is writable."
    fi
else
    echo "$FILE does not exist"
    exit 1
fi
```

String Operators

Operators on strings are summarized below:

Expression	Is True If...
`string`	*string* is not null.
`-n string`	The length of *string* is greater than zero.
`-z string`	The length of *string* is zero.
`string1 = string2` `string1 == string2`	*string1* and *string2* are equal. Single or double equal signs may be used, but the use of double equal signs is greatly preferred.
`string1 != string2`	*string1* and *string2* are not equal.
`string1 > string2`	*string1* sorts after *string2*.
`string1 < string2`	*string1* sorts before *string2*.

Warning: the `>` and `<` expression operators must be quoted (or escaped with a backslash) when used with `test`. If they are not, they will be interpreted by the shell as redirection operators, with potentially destructive results. Also note that while the `bash` documentation states that the sorting order conforms to the collation order of the current locale, it does not. ASCII (POSIX) order is used in versions of `bash` up to and including 4.0.

The following script provides some examples of string operators:

```
ANSWER=$1
if [ -z "$ANSWER" ]; then
    echo "There is no answer." 1>&2   # An error msg.
    exit 1                           # so redirect
fi                                   # to stderr

if [ "$ANSWER" == "yes" ]; then
    echo "The answer is YES."
elif [ "$ANSWER" == "no" ]; then
    echo "The answer is NO."
elif [ "$ANSWER" == "maybe" ]; then
    echo "The answer is MAYBE."
else
    echo "The answer is UNKNOWN."
fi
```

Integer Operators

Integer operators are summarized below:

Expression	Is True If...
integer1 `-eq` *integer2*	*integer1* is equal to *integer2*.
integer1 `-ne` *integer2*	*integer1* is not equal to *integer2*.
integer1 `-le` *integer2*	*integer1* is less than or equal to *integer2*.
integer1 `-lt` *integer2*	*integer1* is less than *integer2*.
integer1 `-ge` *integer2*	*integer1* is greater than or equal to *integer2*.
integer1 `-gt` *integer2*	*integer1* is greater than *integer2*.

The following script provides some examples of integer operators. . .

```
INT=$1
if [ -z "$INT" ]; then
    echo "INT is empty." >&2
    exit 1
fi
if [ $INT -eq 0 ]; then
    echo "INT is zero."
else
    if [ $INT -lt 0 ]; then
        echo "INT is negative."
    else
        echo "INT is positive."
    fi
    if [ $((INT % 2)) -eq 0 ]; then
        echo "INT is even."
    else
       echo "INT is odd."
    fi
fi
```

Logical Operators

All of the types of expressions described above can be combined using logical operators:

Operation	test
AND	`-a`
OR	`-o`
NOT	!

For example, to test whether `FILE` is both a regular file and is readable, do the following test:

```
[ -f $FILE -a -r $FILE ]
```

To test if a file is not readable, do the following:

```
[ ! -r $FILE ]
```

To test for the following logical condition:
$\neg(regular \wedge readable)$

Do the following:

```
[ ! \( -f $FILE -a -r $FILE \) ]
```

Note that the parentheses have to be escaped. Without parentheses the logical not would apply only to the first condition. Meanwhile, the above condition could be written without parentheses if we utilize De Morgan's Law:
$\neg(P \wedge Q) \iff (\neg P) \vee (\neg Q)$

This applies a logical not to each term individually and changes the operator from AND to OR:

```
[ ! -f $FILE -o ! -r $FILE ]
```