

Lecture 2

Shell Programs, Variables & Expansion

Shell Programs

The author uses the term *Command File* to mean an *executable shell program*. This is not a very widely accepted term. It is used only once in the **sh** man page and for a different meaning, and was never used in Mr. Bourne's original shell documentation. We will use the term *shell script* or *shell program*. You will learn the subtle differences of executing shell commands in files, and how to use them, later.

Why does your shell program need to have *read* access permissions to be executable? Should it not be sufficient to give the file *execute* permissions? In fact, *all* programs must have *read* access to run. The shell needs to open the file to read its contents, which of course is a list of shell commands. For binary executables, the kernel opens the program file, reads the contents, and loads it into memory to begin execution. Without *read* access, a program cannot be executed, because the program *is* the contents of the file.

On page 104 of our textbook, it says all you have to do to execute a shell program is to give execute permission to the shell program file - this is true. However, the book also says that once this is done, you can just execute the command by name - this is *not* true. Even after you have made the shell program executable, you will find that you get an error if you try to run it. Here's an example:

```
$ cat foo
#!/bin/sh

echo It Works!
$ chmod a+x foo
$ foo
foo: not found
$ ls -l foo
-rwx--x--x  1 cappella staff      26 Jan 14 11:50 foo
```

What happened? The file *foo* exists and is executable (and readable)! The problem occurs because of the way the shells look for commands. Specifically, your **PATH** environment variable (you will learn more about this later) is not set to look in the current directory to find commands. To run your shell program, or any shell program you create, you must specify it by pathname (relative or absolute). The easiest way to run the program is by the command

```
$ ./foo
It Works!
```

which is using a relative path to execute the *foo* command. The *./* in front of the file is just a path component that means in the current directory. You can execute the **ls** program using the full pathname **/usr/bin/ls** or if your current working directory is **/usr/bin**, you can execute **ls** with the relative pathname **./ls**. Likewise, you can execute the **foo** program with the relative **./foo** pathname.

If you know how to modify your `PATH` variable, you can add the path component `.` to the end of `PATH`. This will cause the shell to look elsewhere first for commands, and then finally in the local directory. This does create a security problem, however, in that you may actually run Trojan horse commands. This is more problematic if you are a system administrator running as the **root** user. This will be explained again in more detail later. For now, notice that by adding the current working directory to the end of the `PATH` environment variable that the command can be executed by name alone (without a relative or absolute pathname).

```
$ PATH=$PATH:.
$ foo
It Works!
```

Comments

Comments in shell scripts are useful for many reasons - the book explains several. In addition, comments can also be used to temporarily disable part of your script for debugging purposes. For example, if the shell script was not working correctly, you could comment-out part of it to isolate what was going wrong. The command line below should sort the output of an **echo** command, but it does not work as expected.

```
$ echo a c b | sort
a c b
```

Disabling the pipe to the **sort** command by placing a comment character before the pipe completely disables the remainder of that line. This helps you see that **sort** appears to not be doing anything at all.

```
$ echo a c b # | sort
a c b
```

You recall that **sort** only works on several *lines* of input, not *within* a single line. So, you place newline characters in your **echo** command to create multiple lines:

```
$ echo 'a\nc\nb' # | sort
a
c
b
```

Now, with the output looking correct for **sort**, remove the comment character to sort the output, and then add a **paste** command to join the lines together again:

```
$ echo 'a\nc\nb' | sort
a
b
c
$ echo 'a\nc\nb' | sort | paste -s -d" " -
a b c
```

While this is clearly a contrived example, commenting out part of your program is very useful when you have long pipelines inside of complex shell scripts. In fact, it is a good technique for disabling entire sections of your shell program. Inside **vi** the command `.,,$s/^/#` would place a comment character from the cursor position to the end of the file at the beginning of each line in your shell script, thus disabling those lines.

Variables

Many people have difficulty with the concept of a *variable*. In a nut shell, a variable is simply a name that holds a value. As I teach in my Introduction to UNIX course, think of variables as *boxes* that you name. The name of a *box* is the name of a variable. And what do you put inside the box? The value of the variable goes inside the box. When you refer to the variable, think about the box itself, and not its contents. When you refer to the value of the variable, you are referring to what is inside the named box.

Variables are extremely important in shell programming, as they provide you with a means to use one name consistently throughout your script. Consider a shell script that uses your login name to perform some action. You could write your login name everywhere it is required (this is called *hard coding*). However, what if you needed to change the user name to some other user's login name? You would have to change every line of your shell program that contains your login name to the new login name.

Instead, a more practical, useful way to code the directory name is to use a variable. For example, you could create a variable **Username** and assign your login name to that variable (i.e. places *cappella* in the box named **Username**):

```
$ Username=cappella
```

Then, throughout the remainder of your shell program, you would not *hard code* your login name, but instead you would use the variable itself. For example:

```
$ echo This script is written by user $Username
```

Recall from your text that a **\$** character in front of a variable name means the *value* of the variable (i.e. what is contained inside the box). This gives you a way to make only a single change to the variable assignment, and the rest of your shell program will just work correctly without modification. In addition, using variables, especially those located at beginning of your shell program can help make your code clearer and if named properly, self documenting. Consider this script:

```
#!/bin/sh

tmpFile=/tmp/my-script-tmp

echo Some output > $tmpFile
echo More output >> $ tmpFile
echo Even more output >> $tmpFile
```

It should be much clearer that the use of the variable **tmpFile** helps you to see that it refers to some sort of temporary output file used in the script. If you needed to change the name of this file, you could change it in a single location, instead of throughout the entire script. If a variable was not used, then you would have to examine the entire script to see what was going on. This can be very difficult, time consuming, and error prone for large, complex scripts. Using variables makes your script cleaner, more easily modified later, and more readable.

The Null Value

The concept of the **null** value is also difficult for some to understand, especially when used in certain shell control statements that you will learn about later. It is very important to understand that a variable that has

no value (its value is **null**) is literally replaced with *nothing*. Going back to the box example, the **null** value for a variable is the same as having literally nothing inside the box. The box itself exists - it is just empty.

A shell variable holds zero or more characters; when it has 1 or more characters, it is said to have a value. When it holds zero characters, it is said to be **null** or empty. Assume that variables **A**, **B**, and **C** are set as shown below (note that **B** is set to **null**)

```
$ A=ls
$ B=
$ C=-la
```

What would be the output of the command line below?

```
$ echo $A $B $C
```

The shell replaces **\$A** with **ls**, **\$B** is **null** so it is replaced with literally *nothing*, and **\$C** is replaced with **-la**, and the output would be:

```
ls -la
```

Command Line Expansion

The textbook begins to describe how the shell handles each command line. It may not be obvious from your experience, but the shell in effect looks through your command line several times, and replaces certain parts of the command line when it is executed. This modification is called **expansion** (or **substitution**). Thus far, you have seen a couple of ways that the shell modifies the command line. The first form of command line expansion is filename expansion (also called filename wildcards or globbing). The second form is variable substitution. The concept of expansion is really a simple one. Expansion means to replace some special characters in the command line with some other non-special characters.

Recall that the characters *****, **?**, and **[]** have a special meaning on the command line – they are *meta-characters* that the shell matches against filenames. These characters are literally replaced or substituted on the command line with the files that match to wildcard characters. And you have also learned about variable expansion - a **\$** in front of a word means that the word is a variable and **\$** in prepended to the word means the value of the variable

Now that you know about more than one expansion, you have to know in what order in which they are expanded. That is, which expansion does the shell perform first, second, etc.? Learning the types of expansion and order of expansion is critical to write more complex shell scripts, since you will get different results depending on the order of expansion. The book shows a good example using a ***** as the value of a variable. Review that example and make sure you understand it. And you will see soon enough that there are even more substitutions performed by the shell.

So what is the expansion order performed by the shell for filename matching and variable substitution? Does filename matching come first? Or does variable substitution come first? The answer is that variable substitution is performed *before* filename matching is done. Remember, the shell is looking several times through your line, once for each type of expansion. The first time it looks through the command line, it replaces any **\$**variables with their actual values. The second time through, it looks for any filename wildcard

meta-characters (*, ?, and []), and replaces any words that contain these characters with any filename matches. Here's an example:

```
$ A=/etc/pass??
$ B=/etc/w*
$ echo $A $B
/etc/passwd /etc/wall /etc/whodo /etc/wtmp /etc/wtmpx
```

The shell assigned the variables **A** and **B** with the characters strings show above. As mentioned in the book, the shell does not perform filename matching when assigning to variables (only when commands are executed is this done). The shell looks through the **echo** command line several times. The first time through, it does variable substitution, and replaces the **\$A** with **/etc/pass??** and the **\$B** with **/etc/w***. The new command line looks like **echo /etc/pass?? /etc/w***. Now the shell does filename matching. It expands the **/etc/pass??** to **/etc/passwd** and the **/etc/w*** to **/etc/wall /etc/whodo /etc/wtmp /etc/wtmpx**. Finally, it executes the internally re-written command as:

```
$ echo /etc/passwd /etc/wall /etc/whodo /etc/wtmp /etc/wtmpx
```

Since the shell works this way, it may occur to you that you can have the shell perform substitution or expansion on even the command name itself (word 1 of the command line). Consider the example below:

```
$ A=ls
$ B=' '
$ C=-la
$ D=/usr/bin/z
$ E=*
$ $A$B$C$B$D$E
```

What would the shell do with this? Well, let's go through the steps. First, perform variable expansion. This turns the command line internally into:

```
$ ls -la /usr/bin/z*
```

Next, perform filename matching. This turns the command line into:

```
$ ls -la /usr/bin/zcat /usr/bin/zipinfo
```

Finally, execute the command:

```
$ ls -la /usr/bin/zcat /usr/bin/zipinfo
-r-xr-xr-x  3 bin      bin          15168 Sep  1  1998 /usr/bin/zcat
-r-xr-xr-x  2 bin      bin          442368 Sep 10  1998 /usr/bin/zipinfo
```

Try it - see if you get the same answer. More importantly, it hopefully makes sense.