COMP421
Unix Environment for Programmers
Lecture 05: Shell Scripts

Jeff Wiegley, Ph.D.
Computer Science

jeffw@csun.edu

09/12/2005

"Go away or I will replace you with a very small shell script" –*ThinkGeek T-Shirt*

Motivation

- The shell is most often used for "single" command entry
- "single" may include multiple, piped commands.
  - `who | cut -c1-8 | sort -u | pr -l1 -8 -w78 -t`[a]
- The shell can also do more complicated programming chores:

```
for file in `ls -1`; do
  if grep -q quote $file; then
    echo "File $file contains what you seek"
  fi
done
```

- Such constructions can become very long and tedious to retype.[b]

[a] Prints a sorted list of all users currently logged in

[b] longer than even shell conveniences such as command history and completion can be useful for.

Shell Scripts:

It would be much easier to wrap up long, complicated sequences in single "script" that can be executed and edited on demand.

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

who | cut -c1-8 | sort -u | pr -l1 -8 -w78 -t

$ cat poormanssearch
#!/bin/sh

for file in `ls -1`; do
  if grep -q quote $file; then
    echo "File $file contains what you seek"
  fi
done
```

Such files are called "shell scripts". They are interpreted and are the basis for rapid prototyping and task automation in Unix.

They can be very powerful and are used as the foundation for controlling how operating system services are stopped and started.

The $PATH to enlightenment:

- Shell scripts must be interpreted by a shell.
- When a command is typed the interpreting shell searches the directories listed in the shell/environment variable "$PATH" for the program.
- Typing: "$ loggedin" probably will result in a "command not found" error because the current directory is not part of $PATH by default.
- There are three ways to overcome the problem:
  1. Add "." to the $PATH variable.

     ```
     PATH=$PATH:.
     export PATH
     ```

  2. Specify an absolute path instead

     ```
     $ ./loggedin
     ```

  3. Pass the filename as an argument to a shell.

     ```
     $ /bin/sh loggedin  [spawns a new shell process]
     $ . loggedin        [Current shell does interpretation]
     ```

Shell tricks:

There are few handy syntax tricks to know when dealing with the Shell.

- I/O redirection is possible.

  `{>,<,&>,#>}`

- The "command" `!!` means execute the last command line entered.
- The "command" `!foo` means execute the last command line entered that started with "foo".
- The ';' terminates a command and therefore allows multiple commands to be given on the same line.[a][b]

  ```
  echo hello; echo there
  ```

- Commands can be continued across lines by using the '\' character followed immediately by an end of line.

[a] The enter key or end of line also always indicates the end of a given command.

[b] bash will continue commands across lines if the context's syntax indicates the command is not complete.

Which shell?:

- More than one shell has been written and adopted into widespread use.
  - Bourne shell (sh)
  - C shell (csh)
  - Korn shell (ksh)
  - tcsh
  - Bourne-again shell (bash)
- Not all of the shells have 100% compatible syntax and features.
  - For instance, ksh and bash can do arithmetic functions with `x=$((x+1))` sh and csh cannot.[a]

[a] On linux /bin/sh is bash so it will appear to do math but such a /bin/sh script will fail to function on a Solaris machine where /bin/sh is truly the Bourne-shell.

Variables: a cursed blessing:

- Blessing: Variables in shell are simple; comprised only of "strings".
  - They do not have to be declared and are all global.[a]
- Curse: They are very limited. The shell relies on other utility programs such as `sed` to manipulate the data.
- Variables are set through assignment.
  - `$ name=value`
  - `$ mylist="jane mike jim kim"`
  - `$ set mylist="jane mike jim kim"`
- There cannot be any whitespace around the assignment character "`=`".
- Variables contents can be printed with the "echo" command.
  - `$ echo "$mylist"`

[a] right away you might guess some of the curses from the fact that variables are global.

Conditional branchings :

Conditional branching is handled a number of ways:

1. `if` statements

   `if` *conditional*
   `then`
   One or more commands...
   `else`
   One or more commands
   `fi`

2. boolean construction

   ```
   cp src dest || echo "failed to copy"
   ```

3. `case` statements, the syntax for which is covered later.

Conditionals :

The only "test" that the shell knows about is the exit value of a command.[a]

For example if the `grep` command finds any matches then it exits with a return value of `0`, otherwise it exits with a non-zero value.

This makes prototyping fast and efficient:

```
if grep -q file1 file2 filen
then
   echo "Something was found"
else
   echo "Give up!"
fi
```

This makes testing the contents of a variable rather challenging. how do you do `$mylist == "hello"`?

[a] Which is why all C or C++ programs **must** return an integer under Unix-like operating systems.

All hail `test`!, the almighty tester!

- `/usr/bin/test` is a program that does just testing.
- Uses command line arguments to specify what and how to test.
- returns a 0 if the test suceeeded, non-zero otherwise.[a]

```
if test "$mylist" = "hello"; then
  echo "They're the same!!"
fi
```

- The command run in this example is `test "$mylist" = "hello"`
- `test` is very powerful and includes all the possible comparisons one would want. Including boolean operators and arithmetic comparisons which convert their string arguments to integers.

[a]The astute reader will notice that the author packed up two command lines by use of the '`;`' character.

All hail []! the almighty... imposter?:

- It gets very boring and tedious to keep writing `test` for every conditional you want.
- `test` also doesn't look elegant or like anything similar to `( ... )` which is common syntax in other programming languages.
- So shells have a shortcut. You can use `[ ... ]` instead of test.

```
if [ "$mylist" = "hello" ]; then
  echo "They're the same!!"
fi
```

- `[ ... ]` functions identical to `test`.
- You **must** have whitespace before and after these delimiters.

Loops:

There are two looping constructs that test the exit status of command

1. until, which executes a command until its command returns 0:

```
until who | grep "^barb"; do
  sleep 60
done
```

2. while, which executes its command until its command returns non-zero. (Same syntax)

A third looping construct exists called a `for` loop that does not test an exit status but rather processes a list of things:

```
for varname in "one" "two" "three" "four"; do
  echo $varname
done
```

If the "in *list*" portion is omitted then the `for` loop iterates over the command line arguments.

Automatic variables:

Certain variables exist automatically.

`$@`: consists of the entire command line passed to the shell.

`$0`: the command name used to invoke the shell.

`$1`...`$9`: The first nine individual command line arguments.[a]

`$?` the exit value of the last command executed.

[a] All command line arguments can be "shifted" left by one position by using the `shift` command.