COSC 3750

Shell Scripts

Kim Buckner

University of Wyoming

Jan. 27, 2022

Last time

- Discussed various utilities.
- Text utilities.
- Process control utilities.
- File utilities

Introduction

- Shell scripting can be simple or complex.
- Best to start with the simple and work up.
- Usually use either Bourne shell or the C shell.

Bourne shell scripts

- I think these are most effective and portable.
- Shell scripts are a text file that is (usually) executable.
- The lines are equivalent to typing at the command prompt.
- If you want the file to be **correctly** executed, it needs a special line.

(more . . .)

- Begin the file with a line like this

  `#!/bin/sh`

- Every modern shell understands this syntax and executes the listed program.
- Then passes the rest of the file to the program as input.
- This works with many things other than shells, like gawk, sed, and perl.

(more . . .)

- The *pound-bang* should always be
  - at the left margin and
  - the first line in the file.
- Always, always use the full path, never just "sh". Security issue.
- In general, comments begin with a # and continue to the end of the line.
- Good editors can do syntax highlighting.

Then . . .

- After that first line, just type in the commands, standard system utilities, programs you have written, or shell functions and shell syntax.
- **Make no assumptions**, shell programming does NOT rely on some programming language like C.
- Most of the syntax is old and slightly cumbersome, with quirks.

Variables

- Variables follow pretty standard naming rules.
- No data type, all are strings
- Declaration and assignment are done at the same time
- `var=value`
- Referenced with a dollar sign:
  `if [ $x == "bill" ]; then`

Bourne shell

- For the Bourne shell, arguments are readily available inside the script
- The first argument is $1, the second $2, etc. thru $9
- The entire command line is $*.
- The number of arguments is $#.
- $0 is the name used to invoke the script.

Shift

- This is a command that "shifts" the arguments.
- The easy version is just `shift`.
- That moves $2 to $1, $3 to $2 and so on. The original $1 is LOST.
- Can do `shift n` where n is an integer and n arguments are shifted, not just 1.

Quotation marks

- There are often problems with strings with spaces or special characters.
- Using quotation marks around a string makes the shell treat it as a single word.
- Double quotation (quote) marks allow variables to be expanded within the quotes.
- Single quotation (apostrophe or tick) marks remove all special meaning from characters like the dollar sign.

Back quotes (backtick)

- Not interchangeable with the other types of quotation marks.
- Everything within the back quotes is treated as a command.
- It is executed in a subshell and the standard output is substituted within the current script as a string.

Syntax

- Simple but the rules are a little clunky.
- Everything must be delimited with whitespace.
- For instance `if [-z "$x"]; then` is a syntax error.
- Must be `if [ -z "$x" ]; then`.

Flow control constructs

- There are several different ones, the most common are `if`, `for` and `case`.
- There are others and you might need them but I usually don't.
- These are `while` and `until`.

if

- `if` *list*; `then` *list*; `fi`
- Can add any number of `elif` *list*; `then` *list*
- And can have a single `else` *list*;
- Here, *list* is a list of commands.

(more . . .)

- Most commonly, the *list* immediately after the `if` or `elif` is a “test”.
- This is actually a command. By using the brackets (square), you get the shell’s version.
- All commands have **some** exit status, an integer value. This is NOT printed to standard output or standard error.
- In C and C++, this is the value returned by main().

Test

- The basic syntax of the test is
  `[ var op var ]`
- There are a large number of operators, some numeric, some string, and some file system.
- For instance `[ -x "$f" ]` would test the string in $f and if it was the name of an executable file then the test would be true.
- Of course that means that “test” returns **0**.

(more . . .)

- The IXes are odd to many because when a program executes **correctly**, the program signifies this by returning 0.
- Anything else is an error (incorrect operation).

(more . . .)

- The IXes are odd to many because when a program executes **correctly**, the program signifies this by returning 0.
- Anything else is an error (incorrect operation).
- Another example is `[ $# -lt 5 ]`.
- This is actually an arithmetic test but the “$<$” operator is reserved for **string** operations.

(more . . .)

- You can test multiple things at once, as in C
- `[ $# -gt 2 -a "$1" == "-q" ]`
- Make sure that you test such things thoroughly before deleting files.

for I

- **Bash** supports versions that I am not sure are portable. Stick with the basics.
- `for` *name [in list]*; `do` *list*; `done`
- *name* is an identifier that is used as the loop control.
- When the name first occurs, do NOT use dollar sign.

for II

- The loop repeats, each time assigning the next one of *list* to *name* for use in the loop.
- If the optional '`in` *list*' is omitted, the script's arguments are used.

Example

```
for i
do
  echo $i
done
```

Oddities

- The *list* of strings is assumed to be whitespace delimited.
- This can be a problem; all of a sudden, too many strings.
- If the `do` is not on a separate line, must be preceded by a semicolon.

case

```
case word in
(pattern)
    list ;;
⋮
esac
```

(more . . .)

- Each pattern is compared to the *word*.
- The patterns can be multiple patterns using | as "or".
- Each '`pattern` *list*' set MUST end with ;; or ;& or ;;&
- ;; is equivalent to break. ;& is the fall through and ;;& is continue testing patterns.
- The last two may not be portable. (*bash* peculiar)

Now what?

- There are two other script-like programs I find helpful.
    - *ed*, the line editor and
    - *sed*, the "stream editor".
- I am not the greatest with these but they are sometimes very helpful.

The line editor

- Sometimes it is needful to modify a text file from a script.
- The editor that can be used for this is *ed*.
- Part of what makes *ed* useful is that some of you may be familiar with parts of it from VI(M).
- But the problem is that *ed* is a line editor.
- Not a common thing anymore so ...

Invocation

- `ed filename`
- But this first prints out the number of bytes in the file.
- In a script we will use `ed -s file`, the ‘s’ means silent.
- Then we use basic editing commands you could be familiar with.

Commands

- (.)a – appends text *after* the addressed line.
- The address can be 0, which means that the lines will be added before any others in the file.
- (.)i – inserts text *before* the addressed line. Again, 0 is a valid address for this.
- (.,.)d – deletes the addressed lines.

(more . . .)

- (.,.)c – change. The addressed lines are deleted and the text is inserted in their place.
- (.,.)s/RE/REPLACE/ – The first match of RE on each of the range of lines is replaced with REPLACE.
- A "g" after the command makes it global.
- An integer after the command makes it the the N'th match.

Addresses

- A range like (.,.) can be 2,5
- It can be .,9 where the period means "the current line".
- It can be 1,$ where the $ means the last line.
- It can also be a single line number.
- The default is just the current line.

Other commands

- (.,.)l – list
- (.,.)p – print, similar to list
- (.,.)n – print with line numbers
- (1,$)w FILE – writes lines to FILE. If no FILE then uses this one.
- If no range then the entire file.
- q – quit. Warned about unwritten changes.
- u – undo last modification.

So what?

- How do we use *ed* in a shell script?
- Use the “here-document” redirection.
- This is input to a command, like it was typed from the keyboard.

Here document

```
<< word
  here-document
delimiter
```

(more . . .)

- The *word* and *delimiter* are the same for our purposes.
- Everything between is taken as lines of text that are input to the command
- The only real restriction on the *delimiter* is that it is unique within the text of the here-document.

Example

```
ed -s myfile << END
0a
This is a new first line.
And this is the second.
.
w
q
END
```

sed

- By default it operates on lines, and does not really care where they come from.
- There are several options, which I almost never use, but you should look at the man page.
- Basically, *sed* is used to modify the input as it passes by.
- Our usage is "`$>sed script`".

Regular expressions

- The script is the magic part. A regular expression and a command.
- The *man* page for sed says that the POSIX BREs are supported.
- One place these are described is https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html
- The regular expressions are really same ones you might be familiar with from VI (vim).

(more . . .)

- Characters represent themselves. The asterisk is a modifier that means 0 or more.
- As GNU extensions \+ and \? are available. The **plus** is 1 or more, the **question mark** is 0 or 1.
- The backslashes are **required**.

(more . . .)

- \(\) are used for grouping subexpressions.
- \{J\} is exactly J repetitions of the preceding expression.
- \{J,K\} is at least J but not more than K
- \{J,\} is J or more

(more . . .)

- [ ] enclose a character class.
- [^] reverses the sense of the character class
- There are two “anchors” in these regex (not in character classes)
  - ^ is the beginning of a line and
  - $ is the end of a line (not the newline)

(more . . .)

- The period '.' matches any character including a newline.
- To explicitly match a period you have to use \\.
- The \\| is alternation (or) as in REGEXP\\|REGEXP
- \\DIGIT matches the DIGIT'th subexpression.

(more . . .)

- \n matches the newline (might not be useful)
- But that and \\are the only portable character escapes.
- Specifically, do not depend on \t matching anything but **t**.

The “s” command

- This is substitute and is probably the most used command.
- s/REGEXP/REPLACEMENT/FLAGS
- If the REGEXP is matched the REPLACEMENT is substituted for the match.
- The FLAGS can change what happens, for instance “g” means the replacement is done to all matches in the pattern space.

Examples

```
sed 's/\.tzt/\.txt/'
sed 's/^\(.*\)\.txt$/PROG_\1_base.tex/'
```