CS 246 Fall 2017 - Tutorial 1

September 20th, 2017

1 Summary

- Shell Commands Review
- I/O Redirection
- Pipelining
- Embedded Commands
- Types of Quotes
- Regular Expressions and `egrep`
- Program Exit Codes

2 Shell Commands Review

- Commands you should definitely know:
- `cd` - change the current directory
  - With no directory or `~` returns you to your home directory
  - With - will return you to your previous current directory
- `ls` - view files in the current/specified directory
  - With `-l` returns a long form listing of the files
  - With `-a` returns all (including hidden) files
  - With `-h` returns human readable format for various fields (e.g. file sizes: 100M, 1G)
  - Can combine multiple options, e.g. `ls -al`
- `pwd` - prints the absolute path to the current directory
  - Same as `$PWD`
- `uniq` - removes consecutive duplicates (removes all duplicates if sorted)
  - The `-c` option will print counts of consecutive duplicates.

- `sort` - sort lines of a file/standard input
  - The `-n` option will sort strings of digits in numeric order
- `tail` - print last 10 lines of file/standard input

3 Output Redirection

- Suppose we have a program (`printer` - prints even numbers to stdout, odd to stderr) that prints to standard output and standard error. Give the redirection to redirect stdout to `print.out` and stderr to `print.err`:
  - `./printer > print.out 2> print.err`
- Suppose we want to redirect the output from standard output to standard error.
  - `echo "ERROR" 1>&2`
- To redirect standard output and standard error to the same file we need to tie them together.
  - For example, `./printer &> out`, which prints both stdout and stderr to `out`
  - Or `./printer > out 2>&1`
  - Or `./printer 2> out 1>&2`
- The order of redirection matters!
- What would be the purpose of redirecting output to `/dev/null`?
  - When we do not care about the actual output of the program but want it to perform some operation (e.g. checking if files are the same, executed correctly).

4 Pipelining

Suppose we want to determine the 10 most commonly occurring words in a collection of words (see `wordCollection` file) and output it to the file `top10`. How might we accomplish this?

Idea: Use some combination of `sort`/`uniq`/`tail`. But how? Probably need the `-c` option with `uniq` and the `-n` option with `sort`.

Okay. `uniq -c wordCollection | sort -n`

But what's the problem? `wordCollection` isn't sorted!

So now we have: `sort wordCollection | uniq -c | sort -n`

So this gives us counts from least to most. How do we get the top 10 and output it to the file `top10`?

Let's try `tail` now: `sort wordCollection | uniq -c | sort -n | tail > top10`

What if we wanted the word counts of the first 10 words alphabetically?

```
sort wordCollection | uniq -c | sort -k2,2r | tail > top10
```

What if we wanted the top 10 words but wanted to break ties based upon reverse alphabetical order?

```
sort wordCollection | uniq -c | sort -k1,1n -k2,2 | tail > top10
```

For fun (not covered material): `wordCollection` was created using the script `wordCollectionGenerator`; you can take a look through the script.

5 Embedded Commands

- We can use a subshell to embed commands as command line arguments to scripts.
- `egrep $(cat file) myfile.txt` could allow us to run `egrep` with the contents of a file being the regular expression.
- Note the difference between `x="echo cat"` and `x=$(echo cat)`. In the first expression, when `x` is evaluated, the output is the word "cat". In the second expression, `echo` outputs `cat` which is then run by the shell when `x` is evaluated.
- Back Ticks (`) works the same way; however, it does not support nesting.

  For example, `$(cat $(echo hello.txt))` is the same as `$(cat hello.txt)` (why?) but `` `cat `echo hello.txt`` `` does not do what you intend to do.

6 Types of Quotes

6.1 Double Quotes

- Suppresses globbing, but not others
- Allows variable substitutions and embedded commands:
  - `echo *   # returns names of all files in the current directory`
  - `echo "*" # returns *`
  - `echo "$(cat word.txt)" # will print contents of word.txt`
  - `echo "${HOME}" # will print the absolute path to the user's home directory`

6.2 Single Quotes

- No substitution or expansion will take place with anything inside of single quotes.
- Suppresses variable substitution and embedded commands.

- `echo '$(cat word.txt)'# Will print $(cat word.txt)`

Both single and double quotes can be used to pass multiple words as one argument. This is required for opening files and directories with spaces in their names.

7 `egrep` and Regular Expressions

- Recall that `egrep` allows us to find lines that match patterns in files/standard input.
- Some useful regular expression operators are:
  - `^` - matches the beginning of the line
  - `$` - matches the end of the line
  - `.` - matches any single character
  - `?` - the preceding item can be matched 0 or 1 times
  - `*` - the preceding item can be matched 0 or more times
  - `+` - the preceding item can be matched 1 or more times
  - `[...]` - match any **one** of the characters in the set
  - `[^..]` - match any one character not in the set
  - `\` - the character after this will be regarded as a character not an operator.
    i.e. `\[` matches the `[` character.
  - `expr1|expr2` - match expr1 or expr2
- Recall that concatenation is implicit
- Parentheses can be used to group expressions
- `egrep` can be especially useful for finding occurrences of variable/type names in source files
  - The option `-n` will print line numbers
- The following are some examples:
  - Find all lines containing "count" in all files ending in `.c`: `egrep "count" *.c`
  - Give a regular expression to find lines starting with 'a' or lines ending with 'z'.
    - `egrep "^a|z$" /usr/share/dict/words`
  - Give a regular expression to find lines with more than one occurrence of the characters a,e,i,o,u
    - We may try `egrep "[aeiou].*[aeiou]+" /usr/share/dict/words`
    - But `egrep "[aeiou].*[aeiou]" /usr/share/dict/words` would also suffice. Why?
  - We want all lines in all `.c` files that modify count by assigning either 0 or 1 aside from initialization.
    - Let's try the obvious thing first: `egrep "^ *count *= *0|1 *; *$" *.c`
    - This doesn't work. Why?
    - What about if we use parentheses? `egrep "^ *count *= *(0|1) *; *$" *.c`
- Remember: regular expressions **are not the same as** globbing patterns.

8 Bash Variables

- In bash, a variable is declared as follows: `var=42`.
  Note: There cannot be spaces on either side of the equals symbol.
- All variables are stored as strings.
- Unlike C variables, bash variables persist outside of the scope of if statements, loops, and scripts.
- Accessing the value in a variable: `$var`
- `${var%<end>}` removes the suffix `<end>` from the string stored in var. If `<end>` is not at the end of var, the string is unchanged.
- We can store a command in a variable and call it later.

9 Program Exit Codes

- When a program completes, it always returns a status code to signify if the program was a success.
- This is true of any C program you have written before now. The exit code is the value returned from main, hence the contract `int main();`. In C and C++, if you do not return from main, the exit code is 0.
- In bash, if a program is successful, the exit code is 0. Otherwise, the exit code is non-zero. The exit code is stored in the variable `?`.
- The exit code cannot be larger than 255. In bash if you return some return code larger than 255, you will get the code modulo 256.

10 Tip of the Week: Vi Shortcuts

You'll quickly notice that vi has a few basic modes. The one you are likely familiar with are the command and insert modes.

10.1 Insert Mode

- This is the mode where you can write text.
- Pressing escape (`Esc` key) when you are in insert mode switches to normal mode

10.2 Normal Mode

- During normal mode, many keys are hotkeys for various actions. The following list is a few useful hotkeys.
  - `i` Enter insert mode at the current position of the cursor.

- `I` Enter insert mode at the beginning of the line which the cursor is currently on.
- `a` Enter insert mode at the position after the current location of the cursor.
- `A` Enter insert mode at the end of the line which the cursor is currently on.
- `o` Enter insert mode on a new line after the line the cursor is currently on.
- `O` Enter insert mode on a new line before the line the cursor is currently on.
- `x` Delete the character the cursor is currently on.
- `0` Move the cursor to the beginning of the current line.
- `$` Move the cursor to the end of the current line.
- `:q` Close vi if no changes have been made to the file.
- `:q!` Close vi without saving change which have been made to the file (since the last save).
- `:w` Save changes to the current file without quitting.
- `:wq` Save changes to the current file and close vi.
- `:n` Move to line n of the file.
- `:$` Move to the last line of the file.
- `/<word>` Searches for <word> in the file and moves the cursor to the next occurrence of <word>. The <word> searched for can be a regular expression. Move to the next match by entering n. Move to the previous match by entering N.
- `?<word>` same as /<word> with n and N's roles reversed.