Week 10 Lab

Scripting with `bash`

Task 1. Counting files

a) In lectures, you saw that it is possible to write a shell one-liner that uses `ls` with `wc` to count the number of files in a directory. Use this method to write a `bash` script called `filecount` that takes 0 or 1 parameters.

- If there are no parameters, `filecount` should count the number of files in the current directory.
- If there is a parameter, `filecount` should check that it is the name of a directory. If not, print a suitable error message; if so, print the number of files in that directory.
- If there are two or more parameters, `filecount` should print a usage message.

To get the test to work, you might need to look up `bash` conditional expressions. Here's a link to GNU's documentation: `http://www.gnu.org/software/bash/manual/html_node/Bash-Conditional-Expressions.html`

b) Write a `bash` script called `filetypes` that is called the same way as `filecount` but tells the user how many of each different type of file is in the directory: named pipes, block devices, character devices, directories, symbolic links, and regular files.

Task 2. To-do list

Write a `bash` script called `todo` that takes a directory as a parameter (or operates on the current directory if no parameter is given).

Your script should go through each `.c` and `.h` file in its target directory and look for lines that contain the strings `TODO:`, `FIXME:`, or both. You should output the filename and line number, but not the line itself. Your script should recurse into subdirectories.

At the end, you should output the string "`Your to-do list has` *`NN`* `items on it.`" where *`NN`* is the total number of `TODO`s and `FIXME`s found.

Task 3. CLI menu

a) Create a `bash` script that initially displays the following:

```
Welcome to RetroComputings menu-driven Easyshell!
Enter a command:

h : print this help message
e filename: edit filename with vi
t filename: touch this filename
q: quit to bash shell

Command:
```

and lets the user enter commands with the effects as described above. Use a loop construct, the `read` builtin command, and either a cascading `if` or `case` construct.

b) Add menu items `l : list C source files` that gives the full paths to all `.c` and `.h` files under the user's home directory, and `f pattern : find files whose names contain pattern`. Use `find` to implement both of these options.

Task 4. Word counter

Write a `bash` script that takes one parameter (a text file) and counts how many `unique` words it has – that is, a word such as *the* should only be counted once no matter how many times it appears in the file.

If you find that you need to use temporary files to enable this functionality, that is fine provided you put them in the `/tmp` directory, include the number of your process in the filename to ensure that no other process tries to access it (the special shell variable `$$` will help you here) and delete all temporary files once execution is complete.

Task 5. Anagram solver

You have seen how `grep` can be used to search files for various different patterns.

a) Using `grep` on the dictionary file `/usr/share/dict/words`, unscramble these letters: `yxusonlia` anxiously (by ingesting grep result) (how to find directly from grep?)

b) Write a `bash` script called `simple-anagram` that takes one parameter and looks for anagrams for it in `/usr/share/dict`. This program may assume that letters in its parameter are not repeated. Ignore case; your program should find `Paris` as an anagram of `rsipa` and `flora` as an anagram of `ARFLO`. Your script may output multiple answers if the search string has several anagrams in the file, e.g. `simple-anagram tac` is allowed to output both `act` and `cat`.

c) Write a `bash` script called `final-anagram` that takes one or two parameters. The first parameter is the sequence of letters to scramble; the second parameter, if provided, is a text file to scan, in the same format as `/usr/share/dict/words`.

   This script should *not* assume that there are no repeated letters in the search string. Instead, it should behave correctly, finding `steep` as an anagram of `eepst` but not finding `pests`.

   You might find that the `cut` program is useful here. Read its `man` page for more information.