Basic 4
Bash and Regex

EECS 201 Fall 2020

Submission Instructions

This assignment will be submitted as a repository on the UMich GitLab server. Create a Project on it with the name/path `eecs201-basic4` and add `brng` as a Reporter. The repository should have the following directory structure, starting from the repository's root:

```
/
|-- report.txt
|-- grep/
|   |-- cap-vow.sh
|   |-- ing.sh
|   |-- n-letter.sh
|   |-- same-lower-vowel.sh
|
|-- sed/
    |-- c89ify.sh
```

Preface

I highly suggest that you do this homework in a Linux environment, be it WSL on Windows (on the Linux filesystem, not the Windows filesystem) or your Ubuntu virtual machine. The reasoning for this is that some tools that deal with regular expressions (namely for this assignment: `sed` and `grep`) may differ in behavior depending on *nix system. Linux systems use GNU `sed` and `grep` while macOS (and FreeBSD) use BSD `sed` and `grep` which have some subtle differences in behavior.

In this assignment you'll be provided yet another zipped archive containing some starter empty files and scripts.

```
$ wget https://www.eecs.umich.edu/courses/eecs201/files/assignments/basic4.tar.gz
```

Initialize a Git repository in the extracted `basic4` directory, adding all of the script files and committing them. Create a **private** project with the name/URL `eecs201-basic4` on the UMich GitLab (`gitlab.umich.edu`) and add the instructor `brng` as a **Reporter**. Set this UMich GitLab project as your remote: you'll be pushing to it in order to submit.

1 Regex fun

As mentioned in lecture, `grep` is a utility that finds patterns in files. These patterns are by default POSIX basic regular expressions; `egrep` or `grep` with the `-E` flag will interpret patterns as POSIX extended regular expressions. For this question we'll be looking at an American English dictionary.
This file is located under `/usr/share/dict/american-english`. If you don't have it, on Ubuntu you can get it via the `wamerican` package and on Arch you can get it via the `words` package. (`/usr/share/dict/words` is a standard file that contains a list of dictionary words; it's probably symlinked to the appropriate dictionary file. You'll learn about symlinks in the next homework question).

1. `cd` into the `grep` directory.
2. Run `$ grep "world" /usr/share/dict/american-english`. This finds and prints out words in the dictionary that contain "world" anywhere in the word.
3. Run `$ grep "^[A-Z]" /usr/share/dict/american-english`. This finds and prints out words in the dictionary that start with a capital letter.

Now onto the question proper:

1. Implement the functionality described in each of the Bash scripts. You may only use one `grep` command in each script. Each script takes in the path to some dictionary file. This can be the American English dictionary, or your own test dictionary. Feel free to use ERE via `-E` or `egrep`.

2. Stage and commit your changes.

2 Searching and replacing text with `sed`

`sed` is a utility that is able to perform pattern searches and replacements. By default `sed` will use POSIX BRE unless a parameter is specified to use ERE.

1. `cd` into the `sed` directory.

2. Run `$ echo "hello user hello" | sed -e "s/hello/world/"`. What `sed` did is replace the first instance of "hello" with "world" of each line of input. `s` is the command to "substitute" texts, with the following charactering serving as a delimiter (`sed -e "s@hello@world@"` would also work, with it using '@' as a delimiter instead of '/'). The `s` command is of the format `s/pattern/replacement/flags`.

3. Run `$ echo "hello user hello" | sed -e "s/hello/world/g"`. Note that both "hello"s were replaced. The `g` at the end of the `sed` `s` command is a flag that says to replace all matches, not only the first.

4. Run `$ echo "hello user hello" | sed -e "s/\(hello\) \(.*\) \1/world \2 world/g"` Note the use of backreferences in both the pattern and replacement fields.

Now onto the question proper. Take a look at `saxpy.c`. Note how it uses `//` for comments.

1. Run `$ gcc -std=c99 saxpy.c`. This will compile the C program following the C99 standard. It should compile successfully. Feel free to run the `a.out` binary that is produced.

2. Now run `$ gcc -std=c89 saxpy.c`. This will compile the C program following the C89 ("ANSI C") standard. It should fail to compile. That is because C++ actually introduced the use of `//` for single single line comments, support for which the C99 standard added to the C language. (If your `gcc` is aliased to `clang` (e.g. on macOS), add the `-pedantic` flag; `clang` is less anal about this particular part of the language).

3. In the provided `c89ify.sh` file, implement the script as described by its usage printout. What this script is supposed to do is take in a list of C files as arguments and then for each C file, replace the `// comment` comments with `/* comment */` style comments and put a "fixed" version of the file into a new file that tacks on `.c89` before the `.c` extension. For example, `./c89ify.sh saxpy.c daxpy.c` will produce fixed versions in `saxpy.c89.c` and `daxpy.c89.c`.

4. To test if the replacement worked, try compiling the files with the `-std=c89` flag. (By the way, deleting the comments aren't a solution; we can check to see if the actual comment messages are still there :p

5. Stage and commit the finished `c89ify.sh` script.

3 Conclusion

1. Add and commit any changes you intend to submit.

2. Create a file called `report.txt`.

3. On the first line provide an integer time in minutes of how long it took for you to complete this assignment.

4. On the second line and beyond, write down what you learned while doing this assignment. If you already knew how to do all of this, put down "N/A".

5. Add and commit this `report.txt` file.

Symbolic links

This section is ungraded and is more for your personal edification.
A "symbolic link" is a special type of file that "points" to another file. (This is similar to the idea of a shortcut to a file). In this question we'll be exploring the use of symbolic links ("symlinks").

1. Use `mkdir` to create a directory called `symlink` and `cd` into it. Use `touch` to create an empty file named `hello`.
2. Using `mkdir`, make a directory called `links`.
3. Let's make a symbolic link to `hello` called `world`:
   run `$ ln -s hello world`
4. Open the `world` file that gets created in a text editor, writing "`Hello world!`" in it and saving it.
5. Open the `hello` file. Note how the changes to `world` appear here.
6. What `ln -s` did is create a special file called `world` that links to a file called `hello` in the *current directory*.
7. We can see what a symlink points to by running `$ readlink world`. Alternatively `$ ls -l` will also show where symlinks point. Note how `world` points to `hello` by itself.
8. Using `mv`, move the `world` file into `links` directory.
9. Now run `$ cat links/world`. Note how it can't resolve the target file as the link refers to a file named `hello` in the same directory as it. You can run `$ readlink -f links/world` to see the exact ("canonical") path that the symlink resolves to.
10. Now run `$ ln -s hello links/foo` and `$ ln -sr hello links/bar`
    Now run `$ ls -l links`. Note the difference in where `links/foo` and `links/bar` point. The `-r` flag makes the pointer to the target file **r**elative to the link's location. You can also provide absolute addresses (starting from root `/`) as targets to link (with all of the issues that they come with: what would happen if I cloned this repo and tried to access that symlink?).