UNIX LAB

CSC 2500

Department of Computer Science
Tennessee Tech University

LAB 09: Getting Familiarized with `sed` and `awk`

General Instructions

This assignment corresponds with chapters 14 and 15 of your textbook. You should utilize online resources to answer these questions as well.

Objective

- Replacing text in line oriented files with `sed`
- Processing line oriented files with `awk`

Description

Part 1: `sed`
For this lab, you will practice using `sed`. The `sed` utility reads lines of text and processes each line of text according to `sed` commands supplied by the user. This utility is incredibly useful for processing text files, and you can accomplish in just a few commands what would require many lines of code if written in a general-purpose programming language such as C++, C#, or Java.

The `sed` utility has `man` commands, but for the purpose of this lab, you will focus on `sed`'s ability to transform lines of text using the s, or substitute, command. According to `sed`'s man page, the substitution function has the following format:

```
[2addr]s/regular expression/replacement/flags
```

In other words, the substitution command starts with an optional address range that is then followed by the letter 's', a regular expression, a replacement expression, and, finally, extra flags that control the substitution. Now that you know the general syntax, you need a data file to process. Create a file of names that looks like the following and call it names.txt (use your favorite text editor):

```
ed wright
tom tisdale
mary cobb
cindy parker
```

Create the file exactly as above, including the lack of capitalization.

Example 1: Now you will write a `sed` command that turns to all the names in names.txt to upper case. At your terminal, type in the following command and run it:

```
$ sed -E 's/(.*)/\U\1/' names.txt
```

The above command prints out all the names in the file, but with all the letters capitalized. The above command specifies no address range, so every line is processed. The next argument to the substitute command is `(.*)`, which is a regular expression that matches the whole line. It is surrounded by parenthesis so that it can be referenced in following arguments. The next argument is `\U\1`. This argument determines what the substitute command will substitute for the previously matched text. The `\U` is a special character extension that means "capitalize the rest of the substitution". The `\1` references all the text matched by the parenthetical expression in the previous argument. Note `sed` has other extensions to modify the substitution for a match. These are `\l  \U` `\u` and `\E`. Read the man page for more information on these extensions.

Example 2: For the next example, you will write a `sed` command that will turn only the first letter of each word to upper case. At your terminal, type in the following command and run it:

```
$ sed -E 's/(.*) (.*)/\u\1 \u\2/' names.txt
```

The regular expression matches any sequence of character followed by a space that is then followed by another sequence of characters. The parenthesis around the sequence of characters forms a numbered group. The first group is given the number 1, and the second group is given the number 2. Then, in the next argument to the substitution command (the \u\1 \u\2), the \u capitalizes the next letter, the \1 substitutes the first group. Then the \u\2 substitutes the next group with its first letter capitalized.

Example 3: For the last example command, you will transform a number such that the number will contain commas after every three digits. For example, 100000000 will become 100,000,000. At your terminal, type in the following command and run it:

```
echo "100000000" | sed -E 's/([0-9]{3})/,\1/2g'
```

The regular expression argument finds occurrences of a sequence of three digits and puts them in group 1. The substitution argument, ,\1 , substitutes a comma followed by the group 1. The magic that makes the substitution work is in the flags. The `g` flags substitutes for every match, and the 2 before the `g` skips the first match. So, in English, the command does the following: Find every sequence of three digits. For the first triplet, just print it out, but for each subsequent triplet, print out a comma followed by the triplet.

Part 2: `awk`

For part 2, you will experiment with the `awk` utility. The `awk` utility is similar to `sed` in that it reads lines on input and processes those lines to produce output. However, `awk` has its own fully

fledged programming language. Therefore, `awk` can be more powerful, but it can also take more code to transform the text.

The basic structure of an awk program is:

```
pattern1 {code}; pattern2 {code} ...
```

The `awk` utility parses lines of text that match a pattern into variables, where each variable hold the value that is delimited by the separator (the default separator is white space), then runs the corresponding code for that pattern. The `awk` utility has two special patterns. One is called `BEGIN`. The corresponding code for `BEGIN` is executed when `awk` starts. The other special pattern is `END`, and its code is executed when `awk` exits.

Example 4: At your terminal, type in the following command and run it:

```
echo "" | awk 'BEGIN {print("Start...")}; END {print("done")}'
```

The `echo` pipes the empty string to `awk` so that it has some input to process (the empty string). Once `awk` starts, the `BEGIN` pattern is automatically matched and its corresponding code is executed, which prints `"Start..."`. After `awk` processes all the input, the `END` pattern is automatically matched and its code prints the string `"done"`.

Example 5: Now you will get the opportunity to try `awk` with a pattern that matches lines of text. At your terminal, type in the following command and run it and try to understand what the script did:

```
awk '/^ed|^mary/ {print $0}' names.txt
```

Note that $0 is a special variable whose content is the whole line that was matched.

Example 6: Now, try the following script that turns the first letter of each word to upper case (just like you did with `sed`). However, you will add line numbers:

```
awk '
  BEGIN {
    count=0
  };
```

```
  {
    count++;
    print(count ". " toupper(substr($2,1,1)) substr($2,2) ","
toupper(substr($1,1,1)) substr($1,2))
  }
' names.txt
```

This script has no pattern. A script that has no pattern is applied to every line of the input. Also, note that `$1` is a special variable that holds the first word of the file (the text delimited by white space). The `$2` variable holds the next word, and so on. Also, note that this script creates a variable called `count`. You can create your own variables in `awk`. All you have to do is assign the variable a value and it is created automatically. This script, for every line of the input, updates the `count` by 1, prints the value of count, and then uses the substring function to get the first character of the first word, capitalize it, and then concatenate it with the rest of the word. Then the script continues by concatenating a comma, then the first letter of the second word, and then, finally, concatenating the rest of the second word. The concatenation operator in `awk` is the `space`.

Example 7: For a final example, you will change the FS variable. The FS variable holds the characters that should be used to parse the line into the `$1, $2, . . .` variables. The default is white space. However, for this last example, you will change the FS variable so you can parse the `/etc/passwd` file, which uses colons as separators. At your terminal, type in the following command and run it:

```
awk '
   BEGIN {
       FS=":";
       printf("%-20s %-5s    %-5s\n", "Login name", "User id", "Group")
   };
   {
      printf("%-20s %5d    %5d\n", $1, $3, $4)
   }
' /etc/passwd
```

The above script changes the field separator to a colon, prints a header, and then for each line in `/etc/passwd` prints the login name, user id, and group so that those values line up in columns. Note that `awk` has a `printf` that operates almost exactly like C's `printf`. The should look like the following:

```
Login name           User id    Group
root                     0        0
daemon                   1        1
bin                      2        2
sys                      3        3
sync                     4    65534
games                    5       60
...
```

Example 8 (Practice):

Write an `awk` command that converts numbers as you did in part1 with `sed`. In other words, turn 100000000 into 100,000,000

Submission

Submit a shell script (lab09.sh) that has all your code. Your shell script should echo a message for each example, and then run its `sed` or `awk` command. For example, your script should start like so:

```
#!/bin/bash


echo "Example 1:"
sed -E 's/(.*)/\U\1/' names.txt
echo ""
echo "Example 2:"
```

```
sed -E 's/(.*) (.*)/\u\1 \u\2/' names.txt
```

(and so on)

Submission Deadline:

Submission Site: iLearn (a Dropbox folder named “Lab 09”)