ULI101 Assignment 3:

3.1: Regular Expressions Using grep

3.1.1: Introduction

Before we start working on regular expressions, you must bear in mind:

REGULAR EXPRESSIONS are NOT THE SAME as AMBIGUOUS FILENAMES!

Although both are used to match patterns, they cannot be swapped, and they use the same symbols to mean different things.

DO NOT GET THEM MIXED UP IN YOUR MIND!

Regular expressions are text search patterns. They are written using a specialized language, and there are several versions of that language.

There are several basic components used to build simple regular expressions (or "regexps"):

- Characters and Character Classes
- Wildcard
- Repetition
- Anchors

Our sample file "names" contains these lines:

```
Mr. Smith
Jenn Bewlite
Ms Carmichael
Dr Ivan James
Mrs Holly Alva Beswol
Mrs James Sheepwool
Mr. Hitchcock
Miss Jennifer Olgovie
Melissa (Missy) Smith
JoJo Smythe
Dr Smooth
Mr. John Waterson
Sara Oooosterinnk
Ms Ella Moloska
James Ngenda
Dr and Mr Wilson
```

3.1.2: Characters

Characters & Character Classes

Characters may be literal characters, such as letters or digits, which match exactly the same letter or digit.

Try this example:

```
$  grep "Smith" names
Mr. Smith
Melissa (Missy) Smith
```

Note that the quotes were not required in the last example, but it's easier to use them all the time than to think about when they are and are not required.

```
$  grep -i "smith" names
Mr. Smith
Melissa (Missy) Smith
```

If you want to search for a character other than a letter or digit, you will need to know if that character has a special meaning. If it does, then you can remove the special meaning by placing a backslash in front of the character.

```
$  grep "M[rs] " names
Ms Carmichael
Ms Ella Moloska
Dr and Mr Wilson
```

Notice that "M[rs] " didn't match "Mrs", because the regular expression specifies the letter M followed by EITHER an r or an s, followed by a space.

Remember that a character class matches only ONE character.

3.1.3: The Wildcard

A period "." character is called a wildcard, and will match any one character.

```
$  grep "Sm.th" names
Mr. Smith
Melissa (Missy) Smith
JoJo Smythe
```

To match two characters, you could just place two periods in a row.

```
$  grep "Sm..th" names
Dr Smooth
```

3.1.4: Repetition

The asterisk character, "*", means that the pattern should match the previous character zero or more times. That means that the previous character does not have to be present at all.

```
$  grep "wo*l" names
Jenn Bewlite
Mrs Holly Alva Beswol
Mrs James Sheepwool
```

```
$  grep "Dr.*James" names
Dr Ivan James
```

3.1.5: Anchors

Anchors

Anchors let us search for text only at the start or end of the line.
The symbols are:

```
   ^     anchor to the start of the line
   $     anchor to the end of the line
```

```
$  grep "^Miss" names
Miss Jennifer Olgovie
```

```
$  grep "James$" names
Dr Ivan James
```

3.1.6: Review

The data file for the review questions, named "inventory", looks like this:

```
Strawberry Jam,300,4
Raspberry Jam,1216,7
Blueberry Jam,96,195
Strawberry Compote,49,621
Raspberry Compote,1937,624
Blueberry compote,200,625
Frozen Strawberries,130,1941
Straw Hats,16,2047
```

The first field is the product name, the second field is the quantity on hand, and the third field is the product code. The fields are separated by commas.

In each of the following questions, write a command using "grep" and a regular expression. The data file in all cases is named "inventory".

Question 1

Display all of the lines in the file that contain the characters "Jam".

Remember that we're using "grep" and the data file is named "inventory".

```
$  grep "Jam" inventory
Strawberry Jam,300,4
Raspberry Jam,1216,7
Blueberry Jam,96,195
```

Question 2

Display all of the lines in the file that contain the word "Straw" right before a space.

```
$  grep "Straw " inventory
Straw Hats,16,2047
```

Question 3

Display all of the lines in the file that contain "Compote" or "compote".

(Use a character class, do not use the -i option).

```
$  grep "[Cc]ompote" inventory
Strawberry Compote,49,621
Raspberry Compote,1937,624
Blueberry compote,200,625
```

Review Question 4

Display all of the lines in the file that contain "Straw" at the beginning of the line.

```
$  grep "^Straw" inventory
Strawberry Jam,300,4
Strawberry Compote,49,621
Straw Hats,16,2047
```

Review Question 5

Display all of the lines in the file where the last field is one digit long. Search for ',' before the field, then use a character class to make sure it's one digit, and anchor it to the end of the line to make sure it's the last field.

```
$  grep ",[0-9]$" inventory
Strawberry Jam,300,4
Raspberry Jam,1216,7
```

Review Question 6

Display all of the lines that contain "Straw" followed later in the line by "Hat".

```
$  grep "Straw.*Hat" inventory
Straw Hats,16,2047
```

3.2: Interactive Shell Environment

3.2.1: Environment Variables

There are many pieces of information that can be passed to programs through command-line options and arguments. This can lead to a lot of typing.

Fortunately, Unix/Linux provides a way to reduce the amount of typing by enabling some information to be stored in 'Environment Variables', which are passed to all child processes.

Every process receives environment variables from its parent, and every process passes environment variables to processes that it starts (the child processes). Shells also permit you to view, create, remove, and alter environment variables.

The command to view all of your shell's current environment variables is 'env'.

```
$  env
```

That was too much information for your screen! Try using 'more' to view it one screen at a time.

Remember! - press 'q' to quit from 'more' when you are ready to continue.

```
$  env | more
```

There were many variables in there. Some of the key ones are:

```
  HOME -- contains your home directory
  MAIL -- specifies where your e-mail mailbox file is located
  PATH -- tells the system which directories to search for commands
  PS1 -- sets your primary shell prompt
  TERM -- tells programs what type of terminal you're using
```

To set a variable, just enter its name, then an equal sign, then the value. Do not use any spaces or tabs.

Let's assign the value '5' to the variable 'FOO':

```
$  FOO=5
```

Now use 'env' to see if 'FOO' is in the list (remember to use 'more'!).

Even though variable 'FOO' has been given a value, it is not yet an environment variable -- it is a shell variable accessible only by your shell. To make it into an environment variable, we must 'export' it so that child processes receive its value.

```
$  export FOO
```

You can also remove an environment variable with the 'unset' command.

Let's remove 'FOO'.

```
$  unset FOO
```

One of the most important environment variables is 'PATH'.

PATH contains a list of directories, separated by colons. When you enter a command, the shell will search those directories to find the command.

Your PATH is currently set to:

```
/home/uli101/bin:/usr/local/rvm/gems/ruby-2.4.1/bin:/usr/local/rvm/gems/ruby-
2.4.1@global/bin:/usr/local/rvm/rubies/ruby-
2.4.1/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/opt/oracle/insta
ntclient_12_2:/usr/local/rvm/bin:/home/ywang585/.local/bin:/home/ywang585/bin
```

Change your PATH to only include the root directory '/'.

```
$  PATH=/
```

```
$ ls
bash: ls: command not found
```

ls was not found because it is not found in any of the directories in the PATH.

(PATH has been automatically reset for you).

You can retrieve the value of any environment variable using a dollar sign and the variable name. You can put this in any command.

For example...

```
$  echo $PATH

/home/uli101/bin:/usr/local/rvm/gems/ruby-2.4.1/bin:/usr/local/rvm/gems/ruby-
2.4.1@global/bin:/usr/local/rvm/rubies/ruby-
2.4.1/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/opt/oracle/insta
ntclient_12_2:/usr/local/rvm/bin:/home/ywang585/.local/bin:/home/ywang585/bin
```

Notice that you only need to place a dollar sign in front of a variable name if you are retrieving the variable value. In any other situation we leave it off.

We can combine retrieving a variable with assigning a value to a variable.

Let's say you wanted to add the directory '/sbin' to your PATH.

```
$  PATH="$PATH:/sbin"
```

In the last example, we placed quotes around the value that we were assigning just in case there were spaces in the value. It is a good idea to always use quotes around the value when assigning a variable.

Exported environment variables are inherited by child processes, so every program that you start will receive their values if you set them up as soon as you login.

The file '~/.bash_profile' for the bash shell or '~/.profile' for ksh is run when you login, so you can add any environment variable set-up lines to that file. Note that since it is a regular text file, you can use any regular text editor to change it.

That concludes this section.

Remember:

To create a variable: variablename="value"
To export a variable: export variablename
To access a variable: $variablename (in a command)
To remove a variable: unset variablename
To see environment variables: env
To see ALL variables: set

And to have an environment variable created every time you login, place the appropriate commands in ~/.bash_profile (bash) or ~/.profile (ksh).

3.2.2: Variable Assignment & Quoting

To set a variable, just enter its name, then an equal sign, then the value. Do not use any space or tabs.

```
$  province=Ontario
```

To retrieve the value of a variable, we place a dollar sign in front of the variable name.

```
$  echo My province is $province
My province is Ontario
```

Let's see what happens when we try to use spaces around the '=' sign:

```
$ city = Brampton
bash: city: command not found
```

```
$ city=My city is Brampton
bash: city: command not found
```

```
$  city='My city is Brampton'
```

Double quotes are also called weak quotes, because they allow variables to be substituted by their values:

```
$  echo "My province is $province"
My province is Ontario
```

Single quotes are also called strong quotes, because they do not allow variable substitution:

```
$  echo 'My province is $province'
My province is $province
```

3.2.3: Command Substitution

Command substitution takes the output of a command and treats it as a character string.

The older form of command substitution uses backquotes:

```
$  echo "My user name is `whoami`"
My user name is ywang585
```

The newer form of command substitution is more powerful, since it allows nesting of command substitution. It uses '$' followed by parentheses surrounding the command:

```
$  echo "My user name is $(whoami)"
My user name is ywang585
```

The result of command substitution can be used anywhere a string can be used. For example, it may be placed into a variable:

```
$  username=$(whoami)
$  echo "My user name is $username"
My user name is ywang585
```

3.2.4: Using The alias Command

The 'alias' command can be used to give a command a different name.

```
$  alias display=echo
```

```
$ display "My user name is $(whoami)"
My user name is ywang585
```

An alias can be removed using the 'unalias' command:

```
$  unalias display
```

```
$ display "My user name is $(whoami)"
bash: display: command not found
```

The 'alias' command can also be used to add options or arguments to a command.

```
$  alias md='mkdir -p'
```

```
$ md a3_testdir/a/b/c/d
```

```
$  ls -ld a3_testdir/a/b/c/d
drwxr-xr-x 2 ywang585 users 6 Jul 25 23:19 a3_testdir/a/b/c/d
```

3.2.5: Variables

Besides the user-defined variables and environment variables seen in earlier sections, there are lots of pre-defined variables that can be used on the command line and within scripts.

Positional parameters are available as the variables $1, $2, $3, and so on. For example, if we write a script called 'script1', and call it as follows:

```
     script1 Toronto Brampton 'North York'
```

then the value of variable $1 is the string 'Toronto', $2 has the value 'Brampton', and $3 has the value 'North York'.

Similarly, these variables can be set on the command line by using the 'set' command. For example:

```
     set Toronto Brampton 'North York'
```

then the value of variable $1 is the string 'Toronto', $2 has the value 'Brampton', and $3 has the value 'North York'. Try this now:

```
$  set Toronto Brampton 'North York'
```

Now display the value of the variable $1:

```
$ echo $1
Toronto
```

The variable $* represents all of the positional parameters as a single string, with the parameters separated by a single space. Display the value of the variable $*:

```
$ echo $*
Toronto Brampton North York
```

The variable $@ represents all of the positional parameters as separate strings. The difference between $* and $@ will be easier to show when looping is discussed in a later section.

The variable $# represents the number of positional parameters. Display the value of the variable $#:

```
$ echo $#
3
```

The command 'shift' shifts all of the positional parameters by one position. The value of the first positional parameter disappears, so that $1 has the value of the second positional parameter, and so forth.

```
$  shift
```

```
$ echo "$1 and $2"
Brampton and North York
```

```
$ echo "$3 and $#"
 and 2
```

The variable $$ contains the process id number, or PID, of the current process. All processes running on the system, including this Assignment, have a unique process id. This is very useful when creating temporary files within a script, allowing temporary files to have a unique name regardless of how many times the script is being executed at the same time.

Again, this Assignment is an example of that, where multiple students can simultaneouly run the Assignment program without interfering with each other. Display the value of the variable $$:

```
$  echo $$
53587
```

The variable $? contains the 'exit status' of the last command that was executed. The 'exit status' is usually used as an indication of the success or failure of a command. Zero indicates success, non-zero indicates failure. This is very useful in scripting, where we can determine what the script will do next based on whether the previous command was successful or not.

3.2.6: Review Exercise

Question 1

The following command has been executed:

```
     set 10 20 30 40 50
```

Enter the value of the variable $3: 30

Question 2

The following command has been executed:

```
     set 10 20 30 40 50
```

Enter the value of the variable $#: 5

Question 3

The following commands have been executed:

```
set 10 20 30 40 50

shift

shift
```

Enter the value of the variable $3: 50

Question 4

The following commands have been executed:

```
set 10 20 30 40 50

shift

shift
```

Enter the value of the variable $#: 3

Question 5

The following commands have been executed:

```
set 10 20 30 40 50

shift

shift
```

Enter the value of the variable $*: 30 40 50

Question 6

Enter the command to display a line that looks like the following:

```
My process id is ####
```

where '####' is the process id of this Assignment (Hint: use a variable, not a command substitution):

```
$  echo "My process id is $$"
```

Question 7

What command would you use to add the directory '/usr/share/bin' to the end of your current PATH?

```
PATH=$PATH: "/usr/share/bin"
```

3.3: Introduction To Scripting (phone)

3.3.1: 'phone' Script

In the other shell window, display the file:

```
     ~uli101/2018b/phonebook
```

```
grep -i cheryl ~uli101/2018b/phonebook
```

Now create a file called 'phone', within the ~/scripts directory. It should start with a line indicating that it needs to run in a bash shell, followed by the same line you just ran:

```
     #!/bin/bash

     grep -i cheryl ~uli101/2018b/phonebook
```

Try using vi for this, just for the practice.

Now give yourself execute permission for the 'phone' script, for example:

```
     chmod u+x phone
```

Also, add your 'scripts' directory to your PATH:

```
     PATH=$PATH:~/scripts
```

Now try running 'phone' and make sure it works.

```
./script/phone
```

Now change the 'phone' script so that it will allow the name to be specified as a command line argument. For example, 'phone cheryl' should display the results for 'cheryl', and 'phone joel' should display the results for 'joel'. To do this, change the second line in the script so that instead of searching for 'cheryl', it searches for $1:

```
     grep -i $1 ~uli101/2018b/phonebook
```

Now try running 'phone cheryl' and 'phone joel' and make sure your script works.

```
./phone Cheryl
./phone joel
```

3.3.2: 'phone2' Script

Copy your script 'phone' to 'phone2'.

Now try running 'phone2 cheryl' and 'phone2 joel' (in another shell window) and make sure your script works.

```
./phone2 Cheryl
./phone2 joel
```

Change 'phone2' to prompt the user for a name to search for in the phonebook. In order to do that, add the following two lines before the grep:

```
     echo -n "Enter a name to search for: "

     read name
```

In order to use the name typed in by the user, the 'grep' needs to use the variable value $name instead of $1. Now the user can simply enter 'phone2', then enter a name to search for in response to the prompt.

Make the necessary changes, and try your 'phone2' script out, using both the names 'cheryl' and 'joel'.

```
./phone2
Enter a name to search for:cheryl
CREATORE CHERYL       LE      6251   HEALTH SCIENCES      250
```

3.3.3: 'phone3' Script

Copy your script 'phone2' to 'phone3'.

Now try running 'phone3' for both 'cheryl' and 'joel' (in another shell window) and make sure your script works.

```
cp phone2 phone3
./phone3
Enter a name to search for:Cheryl
CREATORE CHERYL       LE      6251   HEALTH SCIENCES      250
```

Change 'phone3' to prompt the user for a name to search for in the phonebook, but only if a name was not given as a positional parameter. In other words, 'phone3 cheryl' would search for 'cheryl', but 'phone3' would request a name to search for.

In order to do that, we'll assign the value of $1 to the variable 'name'. Then we'll use an 'if' statement to determine if $name has a value. If not, then we'll use a 'read' statement to get a value from the user.

Use the following five lines to replace the 'echo' and 'read' statements:

```
     name=$1

     if [ "$name" = "" ]

        then echo -n "Enter a name to search for: "

             read name

     fi
```

When a name was not given as a positional parameter:

```
./phone3 cheryl
./phone3: line 5: [cheryl: command not found
CREATORE CHERYL        LE      6251   HEALTH SCIENCES      250
```

So when a name was given as a positional parameter:

```
./phone3 $*
Enter a name to search for:cheryl
CREATORE CHERYL        LE      6251   HEALTH SCIENCES      250
```

3.3.4: 'phone4' Script

Try running 'phone4 xyz' and see what happens.

Modify your program so that if no matching name is found, an appropriate message is displayed: "Name 'xyz' not in directory".

You could use an 'if' statement to check the value of $? to see if the grep command was successful (remember, '0' indicates success). If the grep is NOT successful, then echo the message (which includes the value of $name). Give it a try.

Make sure this works with both command line arguments and with a name read in from the user, and make sure you use the message EXACTLY as shown.

```
#!/bin/bash

name=$1

if ["$name" = ""]

   then echo -n "Enter a name to search for:"

        read name

fi

grep -i $name ~uli101/2018b/phonebook

if [ $? = 1 ]; then

   echo "Name '$name' not in directory"

fi
```

3.4: More Scripting (add)

3.4.1: 'add' Script

If you haven't already done so, edit the file ".bashrc" in your home directory and add the following two lines at the end of the file:

```
PATH=$PATH:~/scripts
```

```
umask 077
```

The first line will ensure that you can execute your scripts regardless of your current directory, and the second line will ensure that other students can't copy your files.

If you don't have a file called ".bashrc" in your home directory, create it and add the above two lines.

```
vi ~/.bashrc
```

If you have changed the ".bashrc" file during this session, enter the "bash" command by itself in order to start an interactive subshell. This well allow ".bashrc" to execute. Pay close attention to make sure there are no errors.

You will write a bash shell script called 'add' that satisfies the following requirements:

Usage: add number-list

'add' will add the numbers (integers) in the list and display the total. For example:

add 4 -3 12 9

will produce the number "22" as output.

```
#!/bin/bash

A=$1
B=$2
C=$3
D=$4

echo `expr $A + $B + $C + $D`
```

Within 'add', use the "for" control structure to loop through all the positional parameters. Use a meaningful loop variable, for example "number" would make sense.

Within the loop, just echo the value of the "number" variable, to make sure you're looping the right way.

Save the program, give yourself execute permission, and test it out.

For example:

add 4 -3 12 9

should produce the output:

```
4
-3
```

```bash
#!/bin/bash

for num in {$1$2}
do
  echo $1
  echo $2
done
```

Make sure the current version of your script works correctly before going on.

Next, initialize a variable called "sum" to 0, before the "for" loop. You will use this variable to add each positional variable one-by-one, so starting it at zero should make sense.

Don't worry about the adding for now, but echo out the value of the variable "sum" after the loop.

Test your script again, make sure it displays the positional parameters and the zero at the end.

```bash
#!/bin/bash

sum=0

for num in {$1$2}
do
  echo $1
  echo $2
done

echo $sum
```

Finally, within the loop, instead of echoing out the value of the loop variable "number", add it to the variable "sum".

Use the x=$((x + y)) arithmetic format, as described in the lecture notes.

Again, check that it works. Make sure that

add 4 -3 12 9

will produce the number "22" as output. Try 'add' with other numbers as well.

```
#!/bin/bash

sum=0

for num in $*
do
  sum=$(($sum + $num))
done

echo $sum
```

```
./add 4 -3 12 9
22
```

3.4.2: 'add2' Script

Copy your script 'add' to 'add2'.

Try running:

```
add2 4 -3 twelve nine
```

You will change the script so that it will display the following message EXACTLY:

```
add2 4 -3 twelve nine
Sorry, 'twelve' is not a number
```

Notice that only the first problem was found, and then the script terminated. The best approach is to check each positional parameter within the loop, before the addition.

You will use an "echo" to display the value of the "number" variable, and pipe the output into a "grep". The "grep" should search for a single character which is NOT a digit or a plus or minus sign. A character class, something like "[^0-9+-]"
would be the easiest way to do this.

If the "grep" is successful, then the message should be displayed and the script terminated.

Step1:

Start this step by just adding the correct "echo" piped into the correct "grep", without redirecting the output, and make sure the script acts appropriately,
for example:

```
add2 4 -3 12 9
22

add2 4 -3 twelve nine
twelve
nine
1
```

```bash
#!/bin/bash

sum=0

for num in $*
do
echo $num | grep "[^0-9+-]"
sum=$(($sum + $num))
done

echo $sum
```

Step2:

Now add an "if" structure right after the "echo ... | grep ...", testing the exit status of the "grep". If the "grep" was successful, then the appropriate message should be displayed, and the script terminated. Make sure the script acts appropriately, for example:

```
add2 4 -3 12 9
22

add2 4 -3 twelve nine
twelve
Sorry, 'twelve' is not a number
```

```bash
#!/bin/bash

sum=0

for num in $*
do
   echo $num | grep '[^0-9+-]'
   if [ $? -eq 0 ]
      then echo "Sorry, '$num' is not a number"
      exit
   else
      sum=$(($sum + $num))
   fi
done

echo $sum
```

```
Can use
   if [ $? = 0 ]
or
   if [ $? != 1 ]
instead of
   if [ $? -eq 0 ]
```

Step3:

Finally, redirect the output of the grep to "/dev/null". Make sure the script acts correctly, for example:

```
add2 4 -3 12 9
22

add2 4 -3 twelve nine
Sorry, 'twelve' is not a number
```

```
#!/bin/bash

sum=0

for num in $*
do
   echo $num | grep '[^0-9+-]' > /dev/null
   if [ $? = 0 ]
      then echo "Sorry, '$num' is not a number"
      exit
   else
      sum=$(($sum + $num))
   fi
done

echo $sum
```

3.5: Yet More Scripting (oldfiles)

3.5.1: 'oldfiles' script

You will write a bash shell script called 'oldfiles' which takes one argument, the name of a directory, and adds the extension ".old" to all visible files in the directory that don't already have it. Treat subdirectories the same as ordinary files. For example:

```
$ ls
file1 file2.old file3old file4.old
$ oldfiles .
$ ls
file1.old file2.old file3old.old file4.old
```

Step1:

Within 'oldfiles', use a "for" control structure to loop through all the non-hidden filenames in the directory name in $1. Check the lecture notes for an example.
Use a meaningful loop variable, for example "filename" would make sense.
Also, use command substitution with "ls $1" instead of an ambiguous filename, or you'll descend into subdirectories.

Within the loop, just echo the value of the "filename" variable, to make sure you're looping the right way.

Save the program, give yourself execute permission, and test it out. To aid your testing, create a test directory and use touch to create some filenames with and without the ".old" extension. Then test your script and make sure that all the filenames in the specified directory are displayed.

```
#!/bin/bash

filename=

for filename in $(ls $1)
do
    echo $filename
done
```

Step2:

Make sure the current version of your script works correctly before going on.

Next, within the loop, continue to use an "echo" to display the value of the "filename" variable, and pipe the output into a "grep". The grep should search for the ".old" extension. A regular expression such as "\.old$" would be the easiest
way to do this.

Don't worry about changing the names yet, simply let "grep" display the filenames that already have the extension.

Test your script again, and make sure it displays the correct filenames.

```
#!/bin/bash

filename=

for filename in $(ls $1)
do
    echo $filename | grep "\.old$"
done
```

Step3:

Within the loop, check the exit status of the "grep" with an "if" control structure. If the "grep" is unsuccessful in finding ".old", then the file should be renamed.

The renaming can be done with a simple "mv" command, renaming "$1/$filename" to "$1/$filename.old".

Once this works, don't forget to redirect the "grep" output to "/dev/null".

Again, check that your script works correctly.

```
#!/bin/bash

filename=

for filename in $(ls $1)
do
    echo $filename | grep "\.old$" > /dev/null
    if [ $? -eq 1 ]
       then
           mv "$1/$filename" "$1/$filename.old"
    fi
done
```

3.5.2: 'oldfiles2' script

Copy your script 'oldfiles' to 'oldfiles2'.

The way the 'oldfiles' script was written is inefficient from a computer utilization point of view. We're looping through ALL the filenames, and then using an "if" inside the loop to determine if each file should be renamed.

It would be more efficient to just loop through the filenames that need to be changed.

Step1:

Modify the command substitution that's being used to create the loop values that will be placed into the "filename" variable.

Instead of just an "ls $1", pipe the output into a "grep". The "grep" will search for all filenames that DO NOT end in ".old". This can easily be done with the "grep -v" option.

With this approach, you can get rid of the "echo ... | grep ..." and the "if" control structure inside the loop, and simply do the rename.

Again, check that your script works correctly.

```
#!/bin/bash

filename=

for filename in $(ls $1 | grep -v "\.old$" )
do
      mv "$1/$filename" "$1/$filename.old"
done
```

Assignment 3.6: sed And awk

3.6.1: Using sed

sed is a 'stream editor'. This means it can be used to change (edit) a file non-destructively. In other words, it will send the changed file to standard output, but the original file is left unchanged.

The true power of sed can be seen when it's combined with other commands, either within a series of piped commands, or within a script.

In this section, we're going to create a series of piped commands to solve a problem, using the 'cars' file that you know and love:

```
plym    fury    77      73      2500
chevy   nova    79      60      3000
ford    mustang 65      45      17000
volvo   gl      78      102     9850
ford    ltd     83      15      10500
Chevy   nova    80      50      3500
fiat    600     65      115     450
honda   accord  81      30      6000
ford    thundbd 84      10      17000
toyota  tercel  82      180     750
chevy   impala  65      85      1550
ford    bronco  83      25      9525
```

We have a customer who wants to see a list of all the cars on the lot that cost less than $10,000, except she doesn't want a chevy. The list should be sorted from lowest cost to highest, and we want to produce the list so that the alphabetic characters are in uppercase.

Let's start by displaying all the records in 'cars' that are not chevy's.
Use a grep command to do this, and don't forget to ignore case:

```
$  grep -iv "chevy" cars
```

Use sed to delete which has 5 digital number at the end of the lines:

```
$  grep -iv chevy cars | sed '/[0-9][0-9][0-9][0-9][0-9]$/ d'
```

Now let's sort the cars by price. Pipe the output of the sed into a sort to do this, and don't forget to sort numerically on the 5th field:

```
$  grep -iv chevy cars | sed '/[0-9][0-9][0-9][0-9][0-9]$/ d' | sort -nk5
```

Finally, let's display the output in uppercase. Pipe the output of the sort into a tr command:

```
$  grep -iv chevy cars| sed '/[0-9][0-9][0-9][0-9][0-9]$/ d' | sort -nk 5 |
tr a-z A-Z
```

3.6.2: Using awk

awk can make substantial changes to a file, and like sed, it will send the changed file to standard output, but the original file is left unchanged.

We have another customer who wants to rebuild a classic car
built between 1975 and 1983. He's willing to pay up to $9,000, but he cares less about price than he does about low mileage, so the list should be sorted from lowest mileage to highest.

Let's start by displaying all the records in 'cars' that are newer than 1974.
Use an awk command to do this, selecting those records that have a third field greater than 74:

```
$  awk '$3>74' cars
```

Now let's pipe the output of the awk into a second awk, selecting those records that have a third field less than 84:

```
$  awk '$3 > 74' cars | awk '$3 < 84'
```

Now let's delete the cars that are more than $9,000. Pipe the output of the second awk into a third awk to do this, selecting those records that have a fifth field less than or equal to 9000:

```
$  awk '$3 > 74' cars | awk '$3 < 84' | awk '$5 <= 9000'
```

Finally, let's sort the cars by mileage. Pipe the output of the third awk into a sort to do this, and don't forget to sort numerically on the 4th field:

```
$  awk '$3 > 74' cars | awk '$3 < 84' | awk '$5 <= 9000' | sort -nk4
```

3.6.3: Review

The data file for the review questions, named "inventory", looks like this:

```
Strawberry Jam,300,4
Raspberry Jam,1216,7
Blueberry Jam,96,195
Strawberry Compote,49,621
Raspberry Compote,1937,624
Blueberry compote,200,625
Frozen Strawberries,130,1941
Straw Hats,16,2047
```

The first field is the product name, the second field is the quantity on hand, and the third field is the product code. The fields are separated by commas.

In each of the following questions, write a command using "sed" or "awk". The data file in all cases is named "inventory".

Question 1

Display only the fifth line in the file, using sed.

```
$  sed -n '5 p' inventory
```

Question 2

Display all of the lines in the file, changing the characters "Jam" to "Marmalade", using sed.

```
$  sed 's/Jam/Marmalade/' inventory
```

Question 3

Display all of the lines in the file showing only the quantity and product name, in that order and separated by a space, using awk. Product name is the first field, and quantity is the second field.

```
$  awk -F, '{print $2,$1}' inventory
```

Question 4

Display all of the lines in the file with less than 100 items in inventory, using awk. Quantity is the second field.

```
$  awk -F, '$2 < 100' inventory
```