This lab introduces applications of regular expressions to pattern matching. It is due in class Monday, May 4 but earlier handins are welcome and encouraged.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself. You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.

This lab is adapted from a homework and lab originally written by David Eck.

Preliminaries

You must use Linux for Part 2, either in Demarest 002 or by connecting to the Linux VDI. To simplify keeping track of your files, it is recommended that you do all three parts in Linux and that you use Eclipse. Eclipse should be configured as described in the previous labs.

- Create a new Eclipse project named `regex`. Make sure the "Create module-info.java file" box is *not* checked before clicking Finish in the dialog box.
- Import the file `/classes/cs229/regex/GetPage.java` into your project. Make sure it goes into the default package under `src` rather than into a package or the top level of the project directory. You will use this program in Part 3.
- Create a text file in your project to hold your answers for Parts 1 and 2: Start with File→New→Untitled Text File, add your name at the beginning of the file, and save it with File→Save. Name the file `regex-exercises.txt` and be sure to save it in the top level of your project folder — choose the project folder as the parent folder in the Save As dialog.

Handin

- Make sure you name is at the beginning of your `regex-exercises.txt` file and that you have included a comment with your name at the beginning of each Java program.
- Make sure you have autoformatted your Java files.
- Hand in your work by copying the entire project directory `~/cs229/workspace/regex` to your handin directory `/classes/cs229/handin/`*`username`* (where *username* is replaced by your username).

1 Regex Patterns — Search and Substitution

This part makes use of a file `data.txt` containing an (anonymized) collection of course grade data. It contains lines of the form

```
1202,MATH 100,B-,26,7/2/2019
1196,MATH 130,A,28,7/2/2019
1196,MATH 130,NC,27,8/13/2019
1196,MATH 130,C+,19,8/27/2019
```

You can find a link for the whole file on the schedule page (along with the link for this lab).

Exercises

Use the notation discussed in class for writing regex patterns. You are encouraged to use the RegEx101 tool (`https://regex101.com`) to develop and test your answers — use the Java flavor for compatible syntax. The whole data file is quite large, but you can copy-and-paste parts of it for testing.

Put your answers in the `regex-exercise.txt` file.

1. Dates in the file are given in the format `M/D/YYYY`, where `M` and `D` are the month (1-12) and day (1-31) and `YYYY` is the four-digit year. Give a search pattern and a replace pattern to convert `M/D/YYYY` dates to the format `YYYY-M-D`.

2. The first item on each line of the file is the four-digit code for the academic term: the first digit is always 1, the second and third digits are the last two digits of the year, and the fourth digit indicates the term (2 for Spring, 4 for Summer, 6 for Fall, 8 for Winter). Give *four sets* of search and replace patterns that could be used to transform these codes to the more user-friendly format Spring 2024, Fall 2023, etc.

   (If you look carefully at the file, you may notice that the year in the code doesn't match the year in the date — that's an artifact of the manipulations done to anonymize the data. It doesn't affect this task, so no need to worry about it.)

3. Each line of the file contains five fields, separated by commas. Only the first three fields are meaningful for your application, so you would like to discard those as well as rearrange the remaining fields and add spaces after the commas. For example, the line

   `1202,MATH 100,B,24,1/15/2020`

   should become

   `MATH 100, B, 1202`

Give a search pattern and a replace pattern to accomplish this. Be careful to match the whole line and pull out the parts that you need. Hint: `.*` can be useful here.

4. The third field on each line contains the grade received. This includes grades such as CR, NC, VW, and so forth. Give a search pattern to match only lines with letter grades (A, B, C, D, or F, optionally followed by a `+` or `-` sign). Be careful — some of the letters can occur in other places in the file (e.g. `MATH 130` contains `A`). Hint: make use of the commas on each line!

---

2 Command-Line Tools

UNIX Utilities and the Bash Shell

Linux (like MacOS since Mac OS X) is a "UNIX-like operating system." What that means for us here is that it comes with a number of standard command line utilities — small programs that can be run on the command line. The program that implements the command line itself is called a "shell" or "command shell." On MacOS and most versions of Linux, the command shell program is *bash*. You run *bash* when you open a terminal window or when you log on remotely using a program such as *ssh*. Some of the commands that you might be used to, such as `cd`, are built into bash, but many are actually small programs.

Bash supports a basic programming language for scripting. This so-called "Bash shell scripting language" has variables, assignment statements, if statements, loops, and functions. But here, we are interested in some of the more basic syntax.

First of all, note that many command-line programs are designed for processing text. These programs read from *standard input* and write to *standard output*. Typically, when a command is used to operate on a file, standard input comes from the file and standard output goes to the command line. For example, the command

```
head MyProgram.java
```

will read the first 10 lines from the file *MyProgram.java* as input, and it will output those lines to the terminal window where the command was given. However, the shell can use *input/output redirection*. You can redirect the output from a command to a file by adding `> filename` to the end of the command. For example, the command

```
head MyProgram.java > out.txt
```

copies the first 10 lines of *MyProgram.java* to a file named *out.txt*. (Warning: An existing file will be overwritten without warning!) For redirecting input, use `<` in place of `>`.

But for this lab, we need the fact that you can *pipe* the output from one program into another program. This means that the standard output from the first program is sent to the standard input of the second program. The symbol for piping is the vertical bar `|`. For example,

```
ls | head
```

runs the `ls` command, which lists the contents of a directory, and sends the output from that command into the `head` program. The `head` program then displays just the first ten lines of the output from `ls`.

Another feature of UNIX commands is that they often have many features that can be enabled by adding an "option" to the command. Options have names that begin with `-` (single dash) or `--` (two dashes). For example, the `ls` command options include `-l` for showing extra information for each directory item, `-R` for listing the contents of directories recursively, and `-h`, used along with `-l` for showing file sizes in more "human-readable" form. Single-letter options that begin with `-` can be combined; for example: `ls -lh`.

Here, then are a few of the command-line utilities that can be used in a UNIX-like operating system, along with some of their options. For most of these commands, standard input can be taken from a file or can be piped from a previous program. If neither a file nor a pipe is provided, they might expect the user to type in the input.

- `cat` <*`files`*> — copies the contents of all the named files, one after the other, to standard output. <*`files`*> refers to a space-separated list of filenames; you do not literally type the word `files` or the < or > symbols. (The word "cat" here is short for "concatenate", but in practice, `cat` is used mostly for one file at a time.)
- `head` and `tail` — Display the first or last ten lines of their input. You can specify a different number of lines as an option. For example, `head -100` *`file`* shows the first 100 lines of the file, and `tail -3` shows just the last three lines of whatever input it is given.
- `wc` — outputs the numbers of lines, words, and characters in the input. To get just the number of lines, use `wc -l`
- `sort` — sorts lines of input alphabetically and sends the result to output. The option `-u` causes duplicate lines to be omitted from the output; the "u" stands for "unique". The option `-i` causes the sort to be case-insensitive. The option `-n` causes sort to do a numeric instead of an alphabetic sort.
- `cut` — selects just parts of the input line for output, where the parts of the line are divided by a specified delimiter. The default delimiter is tab (`\t`). To use a space as the delimiter, add the option `-d " "`. To use a colon as the delimiter, add the option `-d ":"`. To specify which fields you want, use the `-f` option,

which takes a field number or a range of field numbers or a list of field numbers, separated by commas. For example, use `-f 1` to print the first field from each line (that is, everything on the line up to the first occurrence of the delimiter). For example, `cut -d ":" -f 3` outputs everything between the second and third colon on each line.

- `perl -pe 's/`*`regex/replacement`*`/'` applies a "search-and-replace" command to each line of the input, and prints the result. This is discussed more below.
- `grep` and `egrep` — do regular expression matching. These are the main commands for this lab. More on them below.

Most commands can take multiple file names on a line. Note that file names use "globbing", which means that `*` and `?` are treated as wildcard characters, with `?` matching exactly one character and `*` matching any number of characters. For example, `javac *.java` will compile all .java files in the current directory. Note that `*.java` is not a regular expression here; the `*` does not mean repetition.

Substitutions With `perl -pe`

Perl is a powerful programming language. One small thing you can do with it is described here. A command of the form

```
perl -pe 'statement'
```

will execute the Perl *statement* for each line of input and will print the result. You can pipe the input from a previous command, or you can add an input file name to the command. Here, we are interested in a regular-expression substitution statement, which takes the form

```
s/search-expression/replacement-text/
```

The *search-expression* is a regular expression. This statement will look for the first matching substring (if any) in the input line, and will replace that substring with the replacement text. All or parts of the matched substring can be included in the replacement text by using backreferences `\0`, `\1`, `\2`, etc. The command given above only replaces the first matching substring in each line; to apply it to all matching substrings, append a "g" to the command. As a simple example, use

```
perl -pe 's/\t/   /g'
```

to transform every tab character in the input into a sequence of three spaces.

`grep` and `egrep`

The `grep` command takes a regular expression as its argument, and it prints every line from its input that contains a substring that matches the regular expression. `grep` uses

a somewhat more limited syntax for regular expressions than we have studied. For the full set of features, use `egrep` instead of `grep`. **For this lab, you can use `egrep` to avoid confusion about what features are and are not supported.** When using `egrep`, enclose the regular expression in single quotes (but the quotes are only really necessary if the expression contains characters that are special in the bash shell). For example,

```
egrep '".*"' MyProgram.java
```

will print every line from *MyProgram.java* that contains a string literal, using the regular expression `".*"` to match the strings. `grep` would also work here. (The single quotes are needed because both `"` and `*` are special characters for the bash shell.) And, using the pipe syntax discussed above,

```
egrep '".*"' MyProgram.java | wc -l
```

will just output the number of such lines that were found. The `grep` and `egrep` commands have several useful options, including

- `-i` — does a case-insensitive match.
- `-v` — prints out lines that do NOT match the regular expression.
- `-o` — prints out just the part of the line that matches the expression. If there are several matching parts in one line, then they are all output, each one on a separate line.
- `-r` — does recursive searching when applied to a directory; that is, all the files in the directory and in its sub-directories are searched.

Exercises

For exercises that involve multiple commands piped together, build up your command line one part at a time — seeing the output from the previous part can help you understand how to write the next part correctly.

You can use Regex101 (`https://regex101.com/`) to help test and debug your regular expressions. Set the flavor to "Python" in order to use the perl-syntax `\1`, `\2`, etc substitutions instead of `$1`, `$2`, etc used in class.

Put your answers in the `regex-exercise.txt` file.

Investigate user accounts.

On Linux systems, the file `/etc/passwd` contains information about local user accounts. You can view it with

```
cat /etc/passwd
```

If you want to know how many accounts there are, try the command

```
cat /etc/passwd | wc -l
```

Each line of the password file contains seven fields, separated by colons. You may be able to guess what some of the fields are for; more about the file format is addressed in the exercises below.

Networked user accounts are not stored in `/etc/passwd`, so you won't see your account listed there, but when the information is retrieved by the system, the same format is used. The file `/classes/cs229/regex/passwd` contains a dump of the department's user accounts from several years ago, when Linux accounts were separate from HWS network accounts.

5. The first field is the username. On our systems, a student user name is one that matches the regular expression `[a-z]{2}[0-9]{4}`.

   Write a command that will output the number of student accounts. You will need three individual commands, separated by two pipes. (Be sure to use `egrep` since `grep` doesn't support the $\{n\}$ syntax.) How many student accounts are there?

6. The third field is the user account number. Write a single command that will print the smallest student account number in the file, with no other output. (Hint: Use `sort -n` as one of your commands. And you will need some of the other utilities discussed above.) What's the smallest student account number?

7. The fifth field is the user name. For student accounts, the name is in the form `Alice Smith` (first name followed by last name, separated by a space). Write a command that will print out all student names in the form `"Smith, Alice"` including the quotation marks. Note that the order of the names is reversed — the result should have the last name first, followed by a comma, and then the first name. You can use `perl -pe` to rearrange the data into the required format.

   Also: in cases where there are more than two parts to the name, assume that the first space separates the first name and last name. For example, `John von Neumann` should become `"von Neumann, John"`. For this it is important to know that regex matching is *greedy* — quantifiers like `*` will match as much as possible. That means that `(.*) (.*)` applied to `John von Neumann` will match `John von` as the first group and `Neumann` as the second group — `.` matches any characters, including spaces, so it will match as much of the name as possible, leaving the last space in the name to match the space in the regular expression.

8. Now, write a command that will print out the student user name, followed by a space, followed by the name in the same format as the previous exercise. For example: `zz9999 "Smith, John"`.

9. Finally, write a command that will output the same information in the same

format as the previous example, but with the names in alphabetical order by last name. This is harder because you will have to put the information in one format for sorting, then rearrange the information into its final format.

Extra credit: `/classes/cs229/regex/passwd` contains two inconsistencies in the names — in some cases the name ends with a carriage return character (sometimes visible as `^M`) and in some cases the name is already in the form `Smith, John`. Give a solution for #9 which strips out the carriage return characters (`\r` is the character to match a carriage return in a regular expression) and formats every name as `"Smith, John"` (meaning that for names already in the form `Smith, John` you only need to add the quotes).

Investigate a web access log.

The 86-megabyte file `/classes/cs229/regex/access.log` contains the access log for the web server on math.hws.edu for a seven-day period. It has one line for each time someone on the Internet sent a request to our web server during that period. In this part of the lab, you will work with that file. **Do not make a copy of this file.** You can instead reference the file with its full pathname e.g.

```
cat /classes/cs229/regex/access.log
```

As you develop commands to operate on this file, you will sometimes need to see what the output from a command looks like, but you don't want to see 86 megabytes of output. Piping the output into `head` is a way to see just the first ten lines of output.

10. The first thing on each line in the file is an *IP address* that identifies the computer that sent the request to our server. The IP address is followed by a space. Write a command that determines the number of **different** IP addresses that sent requests from the file. The first step can be to use the `cut` to extract the IP address from the line. The IP address is everything from the start of the line up to the first space. But you will need to pipe the output from `cut` into another command to get a list that does not contain duplicate entries, and another command to count the lines of output. How many different IP addresses are there?

11. Lines that represent requests for specific files will contain a string of the form `"GET ` followed by a space (a double quote, followed by GET, followed by a space). This is followed by the path name of the file. The path name cannot contain a space. Write a command that determines how many requests were made for files beginning with `/javanotes`. Such a request will start with the string: `"GET /javanotes`. How many requests did you find? Now, write a command that will find out how many *different* IP addresses sent requests for such files. How many were there?

12. One of the fields on each line is the "referrer." If the request was generated

by a user clicking on a link, the referrer is the web address of the page that contained that link. If the referrer field starts with `"http://www.google.` or `"https://www.google.` (including the double quote mark at the start and the period at the end!), then the request was the result of a Google search. You can assume that any line that contains such a string represents a request generated from a Google search. Write a command to determine the number of such requests. How many were there?

13. The web site name in the referrer field is terminated by the first slash (`/`). For example: `"http://www.google.com/`, `"https://www.google.co.in/`, `"https://www.google.com.ng/`. The last two are Google's India and Singapore sites. Write a command to determine how many different Google sites occur in the referrer field. How many were there? (Note: The `-o` option for `egrep` will be useful.) For your own interest, you might be curious about some of the other countries where the Google sites were located. If so, a list of country codes can be found at `https://en.wikipedia.org/wiki/Country_code_top-level_domain#Lists"`.

14. Finally, write separate commands to determine how many google searches led to a page whose name started with `/javanotes` and how many different google sites led to a page whose name started with `/javanotes`. How many were there in each case?

---

3 Regex in Java

Java has support for regular expressions, provided by the classes `java.util.regex.Pattern` and `java.util.regex.Matcher`. Full details can be found in the API documentation for those classes — google `java 21 pattern` and `java 21 matcher`.

Java regular expressions are specified by strings, however there is one unfortunate complication when specifying a regular expression as a `String` literal in Java: `String` literals themselves have special characters that have to be escaped. For example, suppose you want to write the regular expression

```
\([^"]*\)
```

in Java. This expression matches a string that starts with a left parenthesis, ends with a right parenthesis, and contains no double quotation marks. To write this as a Java string literal, you have to escape the special characters `\` and `"` with backslashes and enclose the whole thing in quotation marks:

```
"\\([^\"]\\)"
```

This can get very ugly, but it is unavoidable. Remember in particular that you have to use *two* backslashes to "double escape" any character that is supposed to be escaped

in the regular expression.

When Java does regular expression search and replace, the syntax for backreferences in the replacement text uses `$` rather than `\`: `$0` represents the entire string that was matched; `$1` represents the string that matched the first parenthesized sub-expression, and so on. If you want to include a literal `$` or `\` in the replacement text, you have to escape them with a backslash: `\$` and `\\`.

Regular expressions come up in several places in the Java API. For example, the delimiter used by a `Scanner` is specified as a regular expression. The `String` class includes several instance methods that make it easy to use regular expressions for several purposes. The following methods are defined for an object of type `String`:

- `public boolean matches(String regex)` — test whether or not the entire string matches the regular expression *regex*
- `public String replaceAll(String regex, String replacement)` — replace all substrings that match the regular expression *regex* with the *replacement* text (which can include backreferences `$0`, `$1`, etc.). The return value is the string after the replacements have been made.
- `public String replace(String regex, String replacement)` — replace just the first matching substring. The return value is the string after the replacements have been made.
- `public String[] split(String regex)` — breaks the string into "tokens" separated by substrings that match the regular expression *regex*. The return value is an array containing all the tokens, but *not* the substrings that match the delimiting regular expression.

To do fancier stuff with regular expressions, you have to use the `Pattern` and `Matcher` classes. A `Pattern` represents a "compiled" regular expression. A pattern object is created using a static function in the `Pattern` class:

```
Pattern regexPattern = Pattern.compile(regexString);
```

where *regexString* specifies the expression as a string. A second parameter can be added to specify one or more flags such as `Pattern.CASE_INSENSITIVE` and `Pattern.MULTILINE`. Multiple flags can be combined with the bitwise or operator, `|`. However, the only case you are really likely to use is:

```
Pattern regexPattern = Pattern.compile(regexString,
                                       Pattern.CASE_INSENSITIVE);
```

In order to match the `Pattern` against a string, you have to create a `Matcher`:

```
Matcher regexMatcher = regexPattern.matcher(String stringToBeMatched);
```

There are three reasons to use a `Matcher` instead of simply using `stringToBeMatched.matches(regexString)`:

First, it is possible to search for a matching substring, instead of just trying to match the entire string. Do this with `regexMatcher.find()`, which returns a boolean to indicate whether or not a matching substring was found. By default, the next call to `find()` after a successful search will start looking for a match at the end of the string that was found by the previous match.

Second, it is possible to set a "region" within the string that is being matched. The region is the substring that is considered for the match. A successful `find()` operation sets the beginning of the region to be the position at the end of the substring that was found, but the region can also be set explicitly by calling `regexMatcher.reset(startIndex,endIndex)`. By default, the regular expression "anchor" characters `^` and `$` match the beginning and end of the region.

And third, it is possible to discover the entire string that was matched and the substrings that were matched by parenthesized sub-expressions. This is done by calling `regexMatcher.group(n)` after a successful match. This returns the entire matched string when n is 0 or the string that matched the n-th parenthesized sub-expression when $n > 0$. You can also find the starting and ending positions for group number n, by calling `regexMatcher.start(n)` and `regexMatcher.end(n)`.

Exercises

For this part of the lab, you will write two short Java programs that use regular expressions. `Pattern` and `Matcher` are in the package `java.util.regex`, so you will need the following at the beginning of each program:

```
import java.util.regex.*;
```

Using regular expressions in Java.

The first task is mostly to make sure that you can use `Pattern` and `Matcher` — read through the section above, and look at the example at the top of the `Pattern` API documentation.

15. Create a program named `PatternMatcher` which works as described below.

    Your program should ask the user to type in a regular expression, read a line of input from the user, and pass it to the `Pattern.compile` method to create an object of type `Pattern`.

    Then ask the user to type in additional lines of text to be matched against the pattern. For each line of text that is input, create a `Matcher` for the text, using the `matcher()` method in the `Pattern` object.

Call the matcher's `matcher.find()` method to test whether the string contains a substring that matches the regular expression. This returns a boolean value to tell you whether a matching substring was found. Tell the user the result. If the regular expression included parentheses, then `matcher.groupCount()` is the number of left parentheses, and `matcher.group(n)` is the substring that matched group `n` for `n` between 1 and `matcher.groupCount()`. Also, `matcher.group(0)` is in any case the entire matching substring. You should print out the substring for each group.

You can end the program when the user inputs an empty string. Here is an example of a session with a sample program:

```
input a regular expression: ([a-zA-Z]*), ([a-zA-Z]*)

input a string: Doe, John
that string matches
      Group 0 matched: Doe, John
      Group 1 matched: Doe
      Group 2 matched: John

input a string: Doe,John
that string does not match

input a string: Bond, 007
that string matches
      Group 0 matched: Bond,
      Group 1 matched: Bond
      Group 2 matched:

input a string: Fortunately, the reactor did not explode
that string matches
      Group 0 matched: Fortunately, the
      Group 1 matched: Fortunately
      Group 2 matched: the

input a string:
```

Your program's output does not need to match the example exactly, but it should contain the elements described above — tell the user whether or not a matching substring was found and print out the matching substring for each group.

Extract information from a web page.

The basic operation for this exercise — extract a list of web sites that a web page links to — is an important one, used, for example, by Google for building its index of the

Web.

16. The provided program *GetPage.java* reads lines from a web page and prints them out. Save a copy of this program as *ExtractURLs.java*, then modify *ExtractURLs.java* so that instead of printing out the lines from the file, it prints out the web addresses from any links that appear on the page. You will use regular expressions to find the links and extract the addresses.

    A web page is usually an HTML file, that is, a text file that contains the content of the page as well as special "markup" code. In the file, a link can look, for example, like one of these strings:

    ```
    <a href="http://google.com">
    <A target="mainframe" href='data321.html'>
    <a id='link23' HREF = "file23.html" target="_TOP">
    <a href="images/myhouse.png">
    ```

    The link must start with `<a` and must contain `href=`. (These are case-insensitive.) There can be spaces around the `=`, but not after the `<`. The web site is in single or double quotes after `href=`. For the examples above, the web addresses that your program would extract are

    ```
    http://google.com
    data321.html
    file23.html
    images/myhouse.png
    ```

    Your program should use a `Pattern` that will match such links and will extract the web site name. Use a `Matcher` for each line of input, to test whether it contains a link to a web site and, if so, to extract the web site. Print out all the web sites that are found, one to a line. Be sure to use a case-insensitive pattern.