Document Version: December 27, 2018

COM 3640 – Programing Languages
Final Project

Due Date: January 1, 2019, 11:59 PM.

Project Overview

Two command line tools that are commonly used on UNIX systems for searching text files are `grep` and `wc`. For purposes of this project, we are going to use the following modified definitions of these commands:

grep

input:
1) *search string:* a string containing the text to search for. The search string can be comprised of a single word or of multiple words surrounded by double quotes
2) *path:* an absolute path to the file in which grep must search for the search string

output: every line in the file in which the search string appears.

wc

input: a path to the file to search
output: a list of all unique words in the file, and the number of times each word appears. It ignores and removes punctuation, capitalization, and white space.

It is very common, when using UNIX systems, to pipe the output of one program into the input of another using the pipe command. The pipe command is a vertical line: "|". Assuming we have a text file named `silly.txt` whose content reads as follows:

```
this assignment is going to be really easy.
I think I am going to wait until the night before it's due to start.
Actually, on second thought, that is a really bad idea.
I think I'm going to start this week. Yes, I really really am.
```

If I wanted to get the word count of all unique words for all lines on which the word "this" appears, I would enter the following at the command line: `grep this silly.txt | wc`
and I would get the following output:

```
this  2
assignment 1
is    1
going 2
to    2
be    1
really 3
easy  1
i     2
think 1
im    1
start 1
week  1
```

```
yes   1
am    1
```

The word "this" only appeared in the first and last lines of the file, and thus those are the only lines output by `grep` and piped into `wc`. As you can see in the output, "this", "going", "to", and "I" appeared twice across those two lines, "really" appeared 3 times, and the rest of the words appeared one time.

In this project, you will implement `grep`, `wc`, and `grep | wc` (according to the description above, not the full UNIX commands) eight different times.

Details

You will implement `grep`, `wc`, and `grep | wc` eight different times, as follows:

- In Java, Python, and JavaScript, you will implement it twice each – once in Object Oriented style and once in Functional Programming style.
- In C, you will implement it twice – once in imperative style and once in Functional Programing style.

Regardless of the language and style, you must define a separate method/function/procedure for each of the following steps:

1. Filter-Lines: filter out lines that don't meet grep search criteria
2. Convert-Case: put everything in lowercase
3. Find-Words: split the text into individual words
4. Non-Alphabetic-Filter: strip out all non-alphabetic characters. Eliminate any "words" that are just white space.
5. Count-Words: produce a set of unique words and the number of times they each appear

Not all these steps are used in all 3 command combinations. Specifically:

- `grep` only uses step 1
- `wc` uses steps 2 through 5
- `grep | wc` uses all 5 steps

Object Oriented Implementation

Builder: You will use the builder pattern. Specifically, you will define a DocumentProcessorBuilder, which has a separate setter method for each of the five steps above. Each setter method takes as its parameter an object which will be used for that step. In your Java implementation, you must write 5 interfaces, with each interface specifies the required API for any class that implements one step above (e.g. LineFilterer, NonABCFilterer, WordFinder, etc.). You must then write an implementation of each interface to do the actual work.

No Explicit Command Combination: You do not explicitly tell the DocumentProcessor which of the three command combinations the user passed in at the command line – it figures that out based on which objects it was given by the build process.

No imperative steps in main(): you may not do any part of the logic in imperative code. All your main method is allowed to do is create an instance of some class and pass it the command line arguments.

Program flow: your main will create some class (I'll call it MyApp, but please do not use such a vague name) which comprises the primary logic of your program. MyApp will parse all the command line arguments and figure out what classes to instantiate and pass to the DocumentProcessorBuilder.

MyApp will create an instance of DocumentProcessorBuilder, call all the relevant setters, call build() to get a fully built instance of DocumentProcessor, and then call "DocumentProcessor.process" which takes one parameter - a String that represents the contents of the document that you read off disk. The "DocumentProcessor.process" method will return a map that maps a unique word to the number of times it appeared in the document. MyApp will then print out the results to the console, with one word-number combination per line.

Object Interactions: DocumentProcessor doesn't do any of the processing itself – it uses the objects passed in when it was being built to do all the work. Each of those objects must also have a "process" method. Filter-Lines and Convert-Case take a String as their input and return a new String as their output. Find-Words takes a string and returns a list of all words. Non-Alphabetic-Filter's input and output are both lists of words. Count-Words takes a list of all words and returns a map of word-count pairs, in which each word only appears as a key once.

Functional Implementation

- Each of the five steps must be implemented either by free-standing functions defined independently in the program text, or by lambdas or helper functions passed as parameters to calls to `map` , `filter`, `reduce`, `collect`, or the equivalent methods in whatever language you are working in. Lambdas are preferred, but helper functions can be used when using a lambda is impossible.
- The functions must provide a Fluent Interface, i.e. the return value of one step must be an object on which the next step can be called
- The "main method" logic of the program must be one long line of fluent/pipelined code, e.g. a().b().c().d()
- Be sure that your logic is really functional – no side effects, etc.
- You must use `map` , `filter`, `reduce`, `collect`, or the equivalent methods in whatever language you are working in, wherever possible to implement the logic within a single step of the 5 steps, as opposed to using imperative control structures, such as loops. One good introduction to thinking this way is here: https://codewords.recurse.com/issues/one/an-introduction-to-functional-programming
- Since C does not have any notion of objects, the composition approach to functions will have to be used as opposed to the Fluent Interface approach for the C functional implementation. For example, if step 1 is h(x), step 2 is g(x), and step 3 is f(x), the code in your main method will have the following structure: f(g(h(x))). (This is probably the hardest part of this project). Each C procedure may not access global variables or modify that which was passed to it as a parameter.

Imperative Implementation in C

Each of the five steps is implemented as its own procedure. It may access global variables and modify any of the parameters passed to it. The code should be done in accordance with clean, structured programming approaches in mind - no monster/huge procedures, good decomposition into blocks/procedures, etc.

Script to Run It

You must submit a UNIX shell script (.sh) OR windows batch file (.bat) that will take the command line arguments (commands, string to search for, path to file, as specified above), build and call all 8 implementations, and clearly print out the results from each.

Your project must work with JDK 11, nodejs 10.14.*, and Python 3.7.*

You can assume all 3 of these runtimes is installed on my machine and on the $PATH