CIS*4650 (Winter 2020) Warm-up Assignment

Due on Thursday, Jan. 30, 2020 by 11:59 pm

Please read the following description carefully so that you know what needs to be done and what should be handed in. You are required to follow the *Coding Style Guidelines* at the end of this description to format your code. This assignment should be done individually in order for you to build a useful experience for text pre-processing.

Scanner for Tagged Documents:

A scanner is not only useful for the analysis of programming languages, but also helpful for the preprocessing of textual documents since the original text often needs to be broken down into words and transformed to a required format. In this assignment, you are asked to write a scanner that will filter out the irrelevant information in textual documents and convert the remaining relevant information into a simple format so that the preprocessed documents may be used for further processing (e.g., searching for relevant documents as in Google and classifying the review documents into positive and negative categories as in Sentiment Analysis).

You should use the Java programming language along with the scanner generation tool JFlex to implement your scanner. To help you get started, a sample scanner package for a tiny programming language (named as "java_sample.tgz") is also provided along with this description in our CourseLink account. You are allowed to use it as a starting point and further extend it for the implementation of this assignment.

The required scanner for this assignment takes as input a set of documents marked by SGML (Standard Generalized Markup Language), somewhat similar to HTML except that all tags come in pairs: one for opening and the other for closing of the embedded text. Along with the sample scanner package, please find a file named "newsdata.txt" for one-day worth of articles from a newspaper, which can be used to develop and test your scanner. Please view the file in a text editor so that you can see all the tags explicitly.

Token Types:

In SGML, every open tag has a corresponding close tag. For example, <Doc> is an open tag and </Doc> is a close tag for marking the two ends of a document. Furthermore, embedded tags should be fully nested. For example, it is acceptable to have the tag sequence: <Text> <P></P></Text>, but not the overlapped sequence: <Text><P></Text></P>. Although it is generally difficult to write regular expressions for such nested structures, for this assignment, however, we can get around the problem by explicitly maintaining a global stack. More

specifically, you will need to write regular expressions to recognize two types of tags from the input:

- OPEN-TAG: all open tags such as <Doc>. Note that open tags can potentially contain attributes such as <Text align="left"> although they may not be used in the file of newsdata.txt. To make your scanner more general, all the details within an open tag (including attribute-value pairs) need to be recognized from the input, but for matching with the related close tag, only the tag names such as "Text" are used, typically ignoring the cases so that "text", "Text" and "TEXT" are all matched.

- CLOSE-TAG: all close tags such as </Doc>.

To make sure that all tags are matched and nested properly, we should maintain a global stack WITHIN the scanner specification file for JFlex in your implementation. Every time we recognize an open tag, we will push its tag name onto the global stack, and every time we recognize a close tag, we will check its tag name against the name on the top of the stack by ignoring the cases. If they do not match, we will report an error and continue the scanning process by ignoring the current close tag. If they do match, we will pop up the top tag name from the stack. At the end of the input, we will also report an error if there are any unmatched tag names in the stack and display them all to the user.

Certain tags contain relevant information and need to be kept along with the embedded text for output. For example, <DocNo> and </DocNo> contain the unique ID for each article. The relevant tags for this assignment should include: <Doc>, <Text>, <Date>, <DocNo>, <Headline>, and <Length> for the given documents. Although you should write general regular expressions to recognize all kinds of OPEN-TAG and CLOSE-TAG, for the output of the relevant tags, however, we will generate more specific names for them. For example, <Doc> and </Text> should be recognized as OPEN-TAG and CLOSE-TAG, but for the output, their token types will be renamed as OPEN-DOC and CLOSE-TEXT, respectively.

Other tags are assumed to contain irrelevant information and should be filtered out for output along with the text embedded between them. In other words, they are very much like comments or whitespaces in a programming language: we still need to recognize them but we do not generate output tokens for them. The tags in this category include: <Byline>, <Correction>, <Correction-Date>, <Dateline>, <DocID>, <Graphic>, <Section>, <Subject>, and <Type>. Any other unknown tags should be treated as irrelevant and thus need to be filtered out as well.

Due to the distinction of relevant and irrelevant tags, we should check the open tags in the global stack to decide when the embedded text should be filtered out or not for output. If all tags in the stack are relevant, we will send the tokens of the embedded text to the output along with the associated open and close tags. On the other hand, if there are any irrelevant tags in the stack, we will filter out the tokens for the embedded text along with the associated tags. Note that <P> and </P> tags indicate a paragraph and can appear within both relevant and irrelevant tags. As a

result, they should be kept for output if all tags in the stack are relevant, but filtered out if there are irrelevant tags in the stack.

Within the relevant tags, we will further break the text into tokens of the following types and you need to write regular expressions for them:

- WORD: strings of letters and digits separated by spaces, tabs, newlines, and most of the punctuation marks. For example, “John”, “compiler”, “mp3”, “123abc” are all treated as WORD tokens.

- NUMBER: integers and real numbers, with possible positive and negative signs.

- APOSTROPHIZED: words such as “John’s”, “O’Reily”, “O’Reily’s”, and “You’re” should be treated as single tokens. However, “world’cup” and “this’is’just’a’test” are likely to be typos and perhaps should be split further. Due to the longest possible match in JFlex, however, it’s hard to separate these two cases for the time being. As a result, you can treat them all as APSOTROPHIZED and write a general pattern for them. This implies that strings like “world’cup” and “this’is’just’a’test” will all be treated as single apostrophized tokens for now. In practice, we could do post-processing to address this problem, but it’s beyond the scope of this assignment.

- HYPHENATED: words such as “data-base” and “father-in-law” should be treated as single tokens. However, “---“ should be treated as a sequence of punctuation marks. Note that when a hyphenated token is ended with an apostrophized suffix, it will be classified as an apostrophized token such as “father-in-law’s”. Again, for the time being, you can write a general pattern for hyphenated words that contain multiple parts separated by hyphens as single hyphenated tokens. For example, “this-is-just-a-test” can be treated as a hyphenated token for now.

- PUNCTUATION: Any symbol that does not contribute to the tokens above should be treated as a punctuation mark.

Note that when specifying a set of token types, you should make sure that the set of regular expressions for them are both mutually exclusive and exhaustive. This implies that any input can be tokenized, and no token can be classified into more than one type (for example, an APOSTROPHIZED cannot also be a WORD).

Implementation Guidelines:

- Your regular expressions should be as general and descriptive as possible, defining the largest set of possible matches (i.e., a number should include any reasonable representation of an integer or a float, either positive or negative). Similarly, you should write general expressions for OPEN-TAG and CLOSE-TAG. Simply listing all possible

tags in the given data file is not helpful for future revisions when tag names are changed or new tags are introduced to an existing document set.

- The global stack is used to match the related open and close tags so that it can be used to validate the properly nested tag structures of a document. It should be implemented within the specification file where you define all the regular expressions so that we can hide the implementation details within the scanner and also avoid generating tokens of the irrelevant information to the output.

- Your scanner should consume input from stdin until EOF is read and send output to stdout; do not prompt for input. The sample output for the beginning part of the first document in the given data set (named "sample.out") is provided along with the sample scanner package for your references.

- You are responsible for writing a Makefile to compile your scanner. Typing "make clean" will remove all the compiled and generated files, and typing "make" or "make all" should result in the compilation of all required components to produce your "scanner". You should ensure that all required dependencies are specified correctly.

- For programming assignments, it is always desirable to follow an incremental process. We recommend the following steps for this assignment: (1) get familiar with the given sample code; (2) write the required regular expressions to generate all possible tokens; (3) implement a global stack to check matched tags and nested structures; and (4) filter out the irrelevant information so that only the relevant tokens are sent to the output.

Guidelines for Assignment Submission:

The following should be handed in for your assignment submission:

- Source Code: Turn in all source code you have written or used along with any other files necessary to compile and run your program (such as Makefile and datafiles). Source code should be appropriately commented and follow reasonable style guidelines (see below).
- Documentation: a file called README should always be present where you can describe the general problem you are trying to solve; what are the assumptions and limitations of your solution; how can a user build and test your program; how is the program tested for correctness (i.e., the test plan should be part of the README file); and what possible improvements could be done if you were to do it again or have extra time available.
- Testing Environment: For the purpose of marking, we will run your programs on a Linux server at linux.socs.uoguelph.ca where both Java and JFlex are installed, and you are strongly encouraged to test your code on this server before uploading your submissions. For the instructions on how to gain remote access to this server, please check the document on "Server Login Instructions" prepared by a former TA (David Wickland) along with the sample scanner package.

Steps for submitting your assignment:

1. Organize all of your files into a directory named by the pattern "<userid>_a1" (e.g., my email userid is "fsong" and I will name the directory "fsong_a1").

2. Run tar and gzip to the above directory to create a single compressed file using the following commands on Linux:
   "tar -cvf fsong_a1.tar fsong_a1" and "gzip fsong_a1.tar" to create "fsong_a1.tar.gz",
   or simply "tar –czvf fsong_a1.tgz fsong_a1" to create "fsong_a1.tgz".

3. Upload the compressed file into the corresponding dropbox on CourseLink by the specified due time.

4. Verify that your submission is indeed uploaded successfully by downloading your compressed file from CourseLink to your local machine and examine the details. You can expand your compressed file using the following Linux commands:
   "gzip –d fsong_a1.tar.gz" and "tar –xvf fsong_a1.tar" to recreate "fsong_a1" directory, or "tar –xzvf fsong_a1.tgz" to do the same.

Coding Style Guidelines:

- All class files should have a comment header, describing the purpose of the related class; and similarly, all methods should have a comment header, describing the purpose of the related method.
- All identifiers for classes, variables, and methods should have descriptive names (with the possible exception of index variables for loops).
- Literal constants (or so-called "magic numbers") should be avoided; instead use named constants for counting limits, array sizes, and so forth.
- Indentation and use of whitespaces should be consistent throughout the code.
- A method should ideally be a small block of code that performs a single task. If a method is getting too long or is doing too many things, you should probably divide it into several methods.
- Other commenting should be appropriate (e.g., you would not comment an increment statement as it is too obvious) and helpful for someone to understand your code (usually highlighting the major steps).