Lexical Analyzer Assignment

Assignment objective:

Implement a lexical analyzer for the language called Teeny Tiny C, TTC.

Design:

1. Download the files **Assignment2.cpp**, **LexicalAnalyzer.cpp**, **LexicalAnalyzer.h**, **Token.cpp**, **Token.h**, and **TokenCodes.h** provided along with this assignment in Blackboard and include them in your project.

2. The file **Assignment2.cpp** is the driver code for the assignment. The main function receives, via the command line argument, the name of the text file that your lexical analyzer processes. The main function contains a loop to call the lexical analyzer's **getNextToken()** method to get the next token and print it out with each token printed on a separate line. You **cannot** modify any of the code in this file.

3. The files **Token.cpp** and **Token.h** contain the Token class used by the **getNextToken()** method of the lexical analyzer class to return a token which consists of two values: the token symbol, and the lexeme string of the token (use the empty string for the EOI token.) You **cannot** modify any of the code in these two files.

4. The file **TokenCodes.h** contains the enumerated type which includes the token symbols for all lexemes in TTC. You **cannot** modify any of the code in this file.

5. LexicalAnalyzer is the lexical analyzer's class. Complete the code to implement the lexical analyzer by adding your code to the files **LexicalAnalyzer.cpp** and **LexicalAnalyzer.h**. You **cannot** change any of the code that is already in those files but you can add code anywhere within those two files. You must declare private, all the data members you add to the lexical analyzer class by adding them under **private:** in the file **LexicalAnalyzer.h**. You must declare private, all the methods you add to the lexical analyzer class by adding their declaration under **private:** in the file **LexicalAnalyzer.h** and defining them in the file **LexicalAnalyzer.cpp**.

6. Copy your code from the body of the **isEOF()** function from Assignment 1 to the body of the **isEOI()** method in the file **LexicalAnalyzer.cpp**. Copy your code from the body of the **readNextLine()** function from Assignment 1 to the body of the **readNextLine()** method in the file **LexicalAnalyzer.cpp**. Copy your code from the body of the **getChar()** function from Assignment 1 to the body of the **getNextChar()** method in the file **LexicalAnalyzer.cpp**. You can modify the code within these methods to fix any errors as described in your grade report from Assignment 1.

7. Use the **Lexical Analyzer Pseudocode** (found in the course's presence in BlackBoard in the tab labeled **Notes & Code**) as the basis for coding the lexical analyzer for Teeny Tiny C. The code to recognize the next token begins in the method **getNextToken()** and only uses the method **getNextChar()** to get the next character from the file. Your token recognition code can also call the method **isEOI()** but it cannot use the file variable **sourceCodeFile** or call the method **readNextLine()**. The lexical analyzer examines one character at time looking for just the next token, stopping at the character right after the last character of that next token. Therefore, **getNextToken()** returns the next token in the file. You can create any additional methods as needed.

8. I execute your lexical analyzer in Linux compiled using g++ with this command:
   ```
   g++ -std=c++11 *.cpp
   ```
   The command I use to execute your program, assuming example.c contains a TTC program and the name of the executable code of your program is a.out, is:
   ```
   a.out example.c
   ```

9. Test data for the lexical analyzer does not need to be a TTC program. Just create a text file containing a good sample of TTC lexemes and other characters that are not found in any of the lexemes in TTC. The file **test_c.pdf** provided along with this assignment in Blackboard is a very simple TTC program. The file **test_txt.pdf** provided along with this assignment in Blackboard is the output generated by the lexical analyzer for the very simple TTC program in **test_c.pdf**.

10. Page 3 lists the table of the lexemes for TTC. All lexemes (except for *identifier*, *numeric literal*, *not a lexeme*, and *end of input*) are individual lexemes consisting of the character string listed in the **Lexeme** column and their corresponding token symbol listed in the **Token Symbol** column. For example, the lexeme **!=** corresponds to the token symbol NEQ. The description for each of the other four lexemes are given below.

11. An **identifier** lexeme starts with a letter and the remaining characters (if any) are either letters or digits. Assume there is not a maximum length for an identifier, and I will not test your code using identifiers that are really long. Identifier lexemes are case insensitive, which means, for example, Count and count are the same identifier name.

12. A **numeric literal** lexeme consists of either one or more digits, or one or more digits, followed by a period, followed by zero or more digits. Assume there is not a maximum length of a numerical literal, and I will not test your code using numeric literals that are really long.

13. The **not a lexeme** lexeme is a single character lexeme: a) for any character not found in any of the other lexemes of the TTC language, b) for any characters which are not the first character in any of the other lexemes in the TTC language, as well as c) a character which is the first character of a lexeme in the TTC language but by itself does not form a lexeme in the TTC language. The lexical analyzer returns the single character for the token's lexeme string, and the NAL token symbol for the **not a lexeme** lexeme. For example, the lexical analyzer returns the NAL token symbol for the ~ character.

14. The lexical analyzer returns the **end of input** lexeme when it reaches the end of the file.

15. **Tip:** Make your program as modular as possible, not placing all your code in a single method. You can create as many methods as you need in addition to the methods already in the lexical analyzer class. Methods being reasonably small follow the guidance that **A function does one thing and does it well**. You will lose a lot of points for code readability if you don't make your program as modular as possible. But, do not go overboard on creating methods. Your common sense guides your creation of methods.

16. Do **NOT** type any comments in your program. If you do a good job of programming by following the advice in number 15 above then it will be easy for me to determine the task of your code.

TTC Lexeme List

Lexeme	**Token Symbol**
`+`	PLUS
`-`	MINUS
`*`	TIMES
`/`	SLASH
`%`	MOD
`(`	LPAREN
`)`	RPAREN
`{`	LBRACE
`}`	RBRACE
`,`	COMMA
`;`	SEMICOLON
`||`	OR
`&&`	AND
`=`	ASSIGN
`==`	EQL
`!`	NOT
`!=`	NEQ
`<`	LSS
`<=`	LEQ
`>`	GTR
`>=`	GEQ
`bool`	BOOLSYM
`do`	DOSYM
`else`	ELSESYM
`false`	FALSESYM
`float`	FLOATSYM
`for`	FORSYM
`if`	IFSYM
`int`	INTSYM
`printf`	PRINTFSYM
`return`	RETURNSYM
`scanf`	SCANFSYM
`true`	TRUESYM
`void`	VOIDSYM
`while`	WHILESYM
`identifier`	IDENT
`numeric literal`	NUMLIT
`end of input`	EOI
`not a lexeme`	NAL

Grading Criteria:

The assignment is worth a total of 20 points, broken down as follows:

1. If your code does not implement the task described in this assignment, then the grade for the assignment is zero.
2. If your program does not compile successfully then the grade for the assignment is zero.
3. If your program produces runtime errors which prevents the grader from determining if your code works properly then the grade for the assignment is zero.

If the program compiles successfully and executes without significant runtime errors, then the grade computes as follows:
Followed proper submission instructions, 4 points:

1. Was the file submitted a zip file?
2. The zip file has the correct filename.
3. The contents of the zip file are in the correct format.

Code implementation and Program execution, 12 points:

- The code performs all the tasks as described in the assignment description.
- The code is free from logical errors.
- Program input, the program properly processes the input.
- Program output, the program produces the proper results for the assignment.

Code readability, 4 points:

- Good variable, method, and class names.
- Variables, classes, and methods that have a single small purpose.
- Consistent indentation and formatting style.
- Reduction of the nesting level in code.

Late submission penalty: assignments submitted after the due date are subjected to a 2-point deduction for each day late.

Late submission policy: you **CAN** submit your assignment early, before the due date. You are given plenty of time to complete the assignment well before the due date. Therefore, I do **NOT** accept any reason for not counting late points if you decide to wait until the due date (and the last possible moment) to submit your assignment and something happens to cause you to submit your assignment late. I only use the date submitted, ignoring the time as well as Blackboard's late submission label.

Submission Instructions:

Go to the folder containing your **LexicalAnalyzer.cpp** and **LexicalAnalyzer.h** files, select them, and place only them in a Zip file. The file can **NOT** be a **7z** or **rar** file! Follow the directions below for creating a zip file depending on the operating system running on the computer containing your assignment's lexical analyzer files.

Creating a Zip file in Microsoft Windows (any version):

1. Right-click any of the two files to display a pop-up menu.
2. Click on **Send to**.
3. Click on **Compressed (zipped) Folder**.
4. Rename your Zip file as described below.
5. Follow the directions below to submit your assignment.

Creating a Zip file in Mac OS X:

1. Click **File** on the menu bar.
2. Click on **Compress ? Items** where ? is 2, the number of files you selected.
3. Mac OS X creates the file **Archive.zip**.
4. Rename **Archive** as described below.
5. Follow the directions below to submit your assignment.

Save the Zip file with the filename having the following format:
your last name,
followed by an underscore _,
followed by your first name,
followed by an underscore _,
followed by the word **Assignment2**.
For example, if your name is John Doe then the filename would be: **Doe_John_Assignment2**

Once you submit your assignment you will not be able to resubmit it!
Make absolutely sure the assignment you want to submit is the assignment you want graded.
There will be **NO** exceptions to this rule!

You will submit your Zip file via your CUNY Blackboard account.
The only accepted submission method!

Follow these instructions:

Log onto your CUNY Blackboard account.
Click on the CSCI 316 course link in the list of courses you are taking this semester.
Click on the **Assignments** tab in the red area on the left side of the webpage.
You will see the **Lexical Analyzer Assignment**.
Click on the assignment.
Upload your Zip file and then click the submit button to submit your assignment.

Due Date: Submit this assignment on or before 11:59 p.m. Monday, April 3, 2023.