ICSI 311 Assignment 2 – The Lexer Part 2

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

You must submit .java files. Any other file type will be ignored. <u>Especially</u> ".class" files.

You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

You must submit every file for every assignment.

<u>You must submit buildable .java files for credit.</u>

Introduction

In this assignment, we will complete our lexer (yes, already!).

As you know, in programming languages (like Java), there are known words (like "for" and "while") and words that we can't forsee (like "myVariableName" and "lineNumber").

One concept that we will use extensively in this assignment is HashMap. Remember that these are O(1) lookups that have we can use as a lookup table. We will use this to look for keywords and convert them to Token Types.

Details

Make a HashMap of <String, TokenType> in your Lex class. Below is a list of the keywords that you need. Make token types and populate the hash map in your constructor (I would make a helper method that the constructor calls).

while, if, do, for, break, continue, else, return, BEGIN, END, print, printf, next, in, delete, getline, exit, nextfile, function

Modify "ProcessWord" so that it checks the hash map for known words and makes a token specific to the word with no value if the word is in the hash map, but WORD otherwise.

For example,

Input: for while hello do

Output: FOR WHILE WORD(hello) DO

In your loop in Lex, we need to deal with comments. Comments in AWK start with # and go to the end of the line (like // comments in Java). When you encounter a #, loop to the end of the line. No need to update line number or line index, because we aren't going to output any tokens for comments.

You are familiar with string literals in Java (String foo = "hello world";) AWK has them as well. Make a token type for string literals. In Lex, when you encounter a ", call a new method (I called it HandleStringLiteral()) that reads up through the matching " and creates a string literal token (STRINGLITERAL(hello world)). Be careful of two things: make sure that an empty string literal ("") works and make sure to deal with escaped " (String quote = "She said, \"hello there\" and then she left.";)

AWK builds in regular expressions as a literal. "Real" AWK uses slashes for their patterns: (example: /.*/). That makes the parser much harder since we use / for division. Instead, we will use the backtick (` - next to the "1" on your keyboard). The logic for this is very similar to StringLiteral (it just uses ` instead of "). Make a new token type, a new method (HandlePattern) and call it from Lex when you encounter a backtick.

The last thing that we need to deal with in our lexer is symbols. Most of these will be familiar from Java, but a few I will detail a bit more. We will be using two different hash maps – one for two-character symbols (like ==, &&, ++) and one for one character symbols (like +, -, $). Why? Well, some two-character symbols start with characters that are also symbols (for example, + and +=). We need to prioritize the += and only match + if it is not a +=.

Two-character symbols:

`>=   ++   --   <=   ==   !=   ^=   %=   *=   /=   +=   -=   !~   &&   >>   ||`

^ is the symbol in AWK for exponents (5^3 == 125).
~ is the symbol in AWK for match, so !~ is "does not match"
>> is the symbol in AWK (and BASH) for append.
Create token types and a hash map for these symbols.

Next create the token types and the hash maps for the single character symbols (I used String, not char):
`{ } [ ] ( ) $ ~ = < > !  + ^ - ?  : * / % ; \n | ,`

One note – AWK doesn't require semi-colons, but it allows them. For this reason, I mapped both \n and ; to SEPERATOR tokens.

Create a method called "ProcessSymbol" – it should use PeekString to get 2 characters and look them up in the two-character hash map. If it exists, make the appropriate token and return it. Otherwise, use PeekString to get a 1 character string. Look that up in the one-character hash map. If it exists, create the appropriate token and return it. Don't forget to update the position in the line. If no symbol is found, return null.

Finally, call ProcessSymbol in your lex() method. If it returns a value, add the token to the token list.

Unit test expectation go up for this assignment. You should be testing with whole AWK programs online. Lex them and **programmatically** ensure that they are lexing correctly.

Here is one of my unit tests to give you the idea:

```
public void TestFullLine1()
        {
            var lexer = new Lexer("$0 = tolower($0)");
            var tokens = lexer.Lex();
            Assert.AreEqual(8, tokens.Count);
            Assert.AreEqual(Token.TokenType.DOLLAR, tokens[0].Type);
            Assert.AreEqual(Token.TokenType.NUMBER, tokens[1].Type);
            Assert.AreEqual(Token.TokenType.ASSIGN, tokens[2].Type);
            Assert.AreEqual(Token.TokenType.WORD, tokens[3].Type);
            Assert.AreEqual("tolower", tokens[3].Value);
            Assert.AreEqual(Token.TokenType.OPENPAREN, tokens[4].Type);
            Assert.AreEqual(Token.TokenType.DOLLAR, tokens[5].Type);
            Assert.AreEqual(Token.TokenType.NUMBER, tokens[6].Type);
            Assert.AreEqual(Token.TokenType.CLOSEPAREN, tokens[7].Type);
        }
```

Yours may not look exactly like mine (I am not working in Java).

HINTS

Do not wait until the assignment is nearly due to begin. **Start early** so that you can ask questions.
Similarly, don't underestimate how long the unit tests will take.
You may find it useful to make some private helper methods in your lexer. **This is encouraged.**
Read the rubric carefully.
Be careful about good OOP – very little in this assignment should be "static". Members should always be private.
Your lex() method will probably be a big block of if/else if /else if… statements. That's normal.

Rubric	Poor	OK	Good	Great
Code Style	Few comments, bad names (0)	Some good naming, some necessary comments (3)	Mostly good naming, most necessary comments (6)	Good naming, non-trivial methods well commented, static only when necessary, private members (10)
Unit Tests	Don’t exist (0)	At least one (6)	Missing tests (12)	All functionality tested (20)
Keywords	Unmodified (0)	Processed ad hoc (3)	Hashmap created and populated (6)	Keywords properly recognized and proper tokens created (15)
Comments	Not skipped (0)			Rest of line ignored (5)
StringLiteral	Not processed (0)	Token type exists (3)	Token type exists and Method exists and is called (6)	String literals properly processed into their own tokens; line count and column updated (10)
Patterns	Not processed (0)	Token type exists (3)	Token type exists and Method exists and is called (6)	Patterns properly processed into their own tokens; line count and column updated (10)
Symbol – One Character	No hashmap (0)	Hash map exists (3)	Most symbols exist (6)	All symbols exist with reasonable token types (10)
Symbol – Two Character	No hashmap (0)	Hash map exists (3)	Most symbols exist (6)	All symbols exist with reasonable token types (10)
ProcessSymbol	None (0)	Exists, looks in one map (3)	Exists, looks in both maps (6)	Exists, looks in twoChar, then oneChar, updates StringManager and position correctly (10)