Laith Hussin
921659619
Feb 27-2022

Assignment 2 Documentation

Github Repository

https://github.com/sfsu-csc-413-fall-2022-roberts/assignment-2---lexer-laith57th.git

Project Introduction and Overview

Using a code skeleton provided for the lexer class and other corresponding classes, I was required to improve the output of lexical analysis for readability, add a few new token types and reserved keywords, and implement an algorithm that scans and analyzes two new token type literals.

Scope of Work

Task		Completed
Add new tokens for lexical analysis		**X**
	Add new token types to the tokens file.	**X**
	Compile and run the TokenSetup class	**X**
	Check to make sure new tokens were added to the Tokens and TokenType classes.	**X**
Improve the lexical analysis output		**X**
	Create a toString() for the lexer class.	**X**
	Use String.format to better organize the output of lexical analysis.	**X**
	Change the Token class's toString to return: String.valueOf(symbol). (To get the String value of the token (i.e. INTeger, BOOLean, etc))	**X**
	Create an integer inside the toString that tracks the lineNumber and use that value to output the raw code of the x file upon successful completion of lexical analysis.	**X**
Implement the algorithm for Utf16StringLit token		**X**
	public void illegalch(String tok)	**X**
	Final int utfLength = 12;	**X**
	Add a condition inside the newToken method that checks if the first character in a token is a "\", if it is it triggers a while loop that checks	**X**

	the first part of a utf16 string until another “\” is scanned. Once the second “\” is scanned it triggers a nested while loop that checks the validity of the second half of the utf16 String. If the length of the scanned token = utfLength at the end of the string, then return the new utf16String.	
Implement the algorithm for TimeStampLit token		**X**
	public void illegalch(String tok)	**X**
	public void timeCalc(String tok, int index, int from, int to, char sym)	**X**
	Insert a condition inside the number scan that checks if the character after a 4-digit token = “~”. If this condition is met, create a switch statement that takes in the length of the token as reference and scan the token for any illegal TimeStamp type characters.	**X**
Change simple.x file to include TimeStampLit and utf16StringLit token and run the lexer class with all possible cases for each token. Make sure correct tokens are recorded and illegal tokens are thrown away with an error message.		**X**
	\uD83D\uDC7D Result: **legal**	**X**
	\ud83D\uDc7D Result: **legal**	**X**
	\uR123\uZ123 Result: **illegal**	**X**
	\uDf12\dF Result: **illegal**	**X**
	2022~02~15~12:15:22 Result: **legal**	**X**
	0001~12~15~12:30:59 Result: **legal**	**X**
	2022~14~15~39:15:22 Result: **illegal**	**X**
	2000~ Result: **illegal**	**X**
Allow x files to be entered from the command line		**X**
	Lexer lex = new Lexer(arg);	**X**

Execution and Development Environment

To complete this project, I used Visual Studio Code on my Macbook M1 pro.

Compilation Result

Using the terminal, I used the following commands on the project files:

```
> javac lexer/setup/TokenSetup.java
> java lexer.setup.TokenSetup
> javac lexer/Lexer.java
> java lever.Lexer filename.x
```

The program ran as expected and no error messages were displayed

Assumptions

I assumed that the file entered from the command line exists and is an x file. I also assumed that all timestamp elements will have trailing zeros behind numbers less than 4 or 2 digits (i.e. 0001 for year and 04 for month). Any other illegal or invalid characters and tokens inside the x file will be thrown away by the error handling.

Implementation

Output design

After studying the lexer class and its associated classes and files, I began to slowly understand the gist of how everything is operating. To improve the output, I knew I needed to move everything away from the main method and into a lexer toString. However, before doing that I tried to place the previous implementation of the output inside a for loop that takes an argument in the form of a String, or in this case, a file path.

After implementing a way to dynamically type and scan a file directly from the command line, I began to organize the output to the desired format inside the toString. I used String.format to create a consistent design for scanned tokens that shows all required information. To track the line number and use it in the output information, I created an instance variable and a getter method inside the lexer class.

Finally, to print the raw code upon successful completion of lexical analysis, I added the read lines during the analysis to a string and printed the string at the end of the output specifying each line.

New Token types and reserved keywords

This was one of the easier tasks in the implementation of this project. All I needed to do was add the token identifiers and their corresponding names to the tokens file, run the TokenSetup class and the tokens were automatically added to the Tokens and TokenType classes.

New Utf16StringLit token

For the implementation of this token literal, I knew I needed to check each half of the string separately with the “\” as the halfpoint. Therefore, Inside the nextToken() method in the lexer

class I created a new try->catch block that runs after a “\” is encountered. My implementation is simply a while loop that runs until a “\” is encountered, otherwise, it keeps scanning the next token. Within that while loop, I created a nested while loop for when the second “\” is encountered that checks the second half of the utf16String. I equipped each while loop with their own error checking conditions such as length and character value.

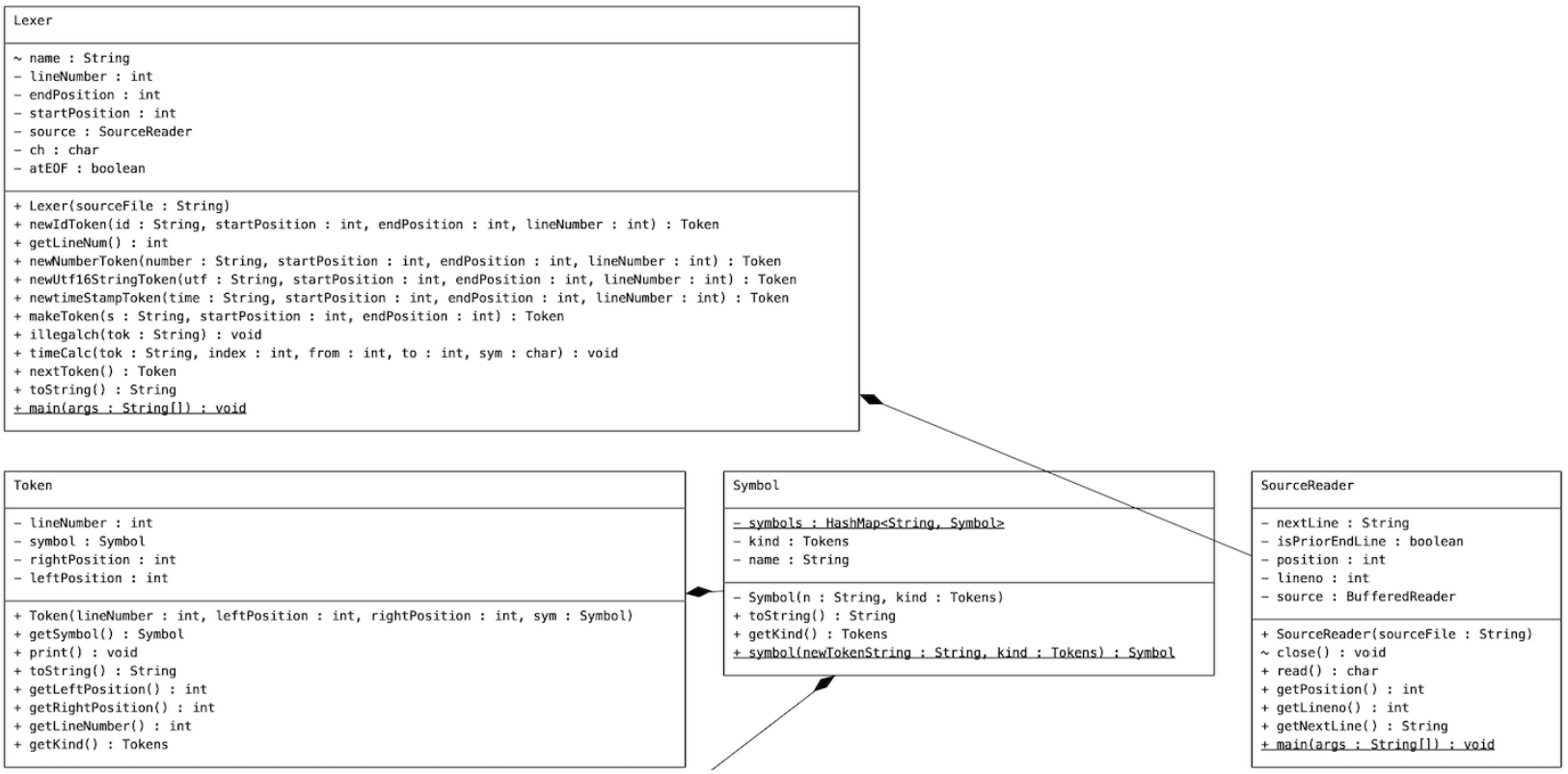
New TimeStampLit token

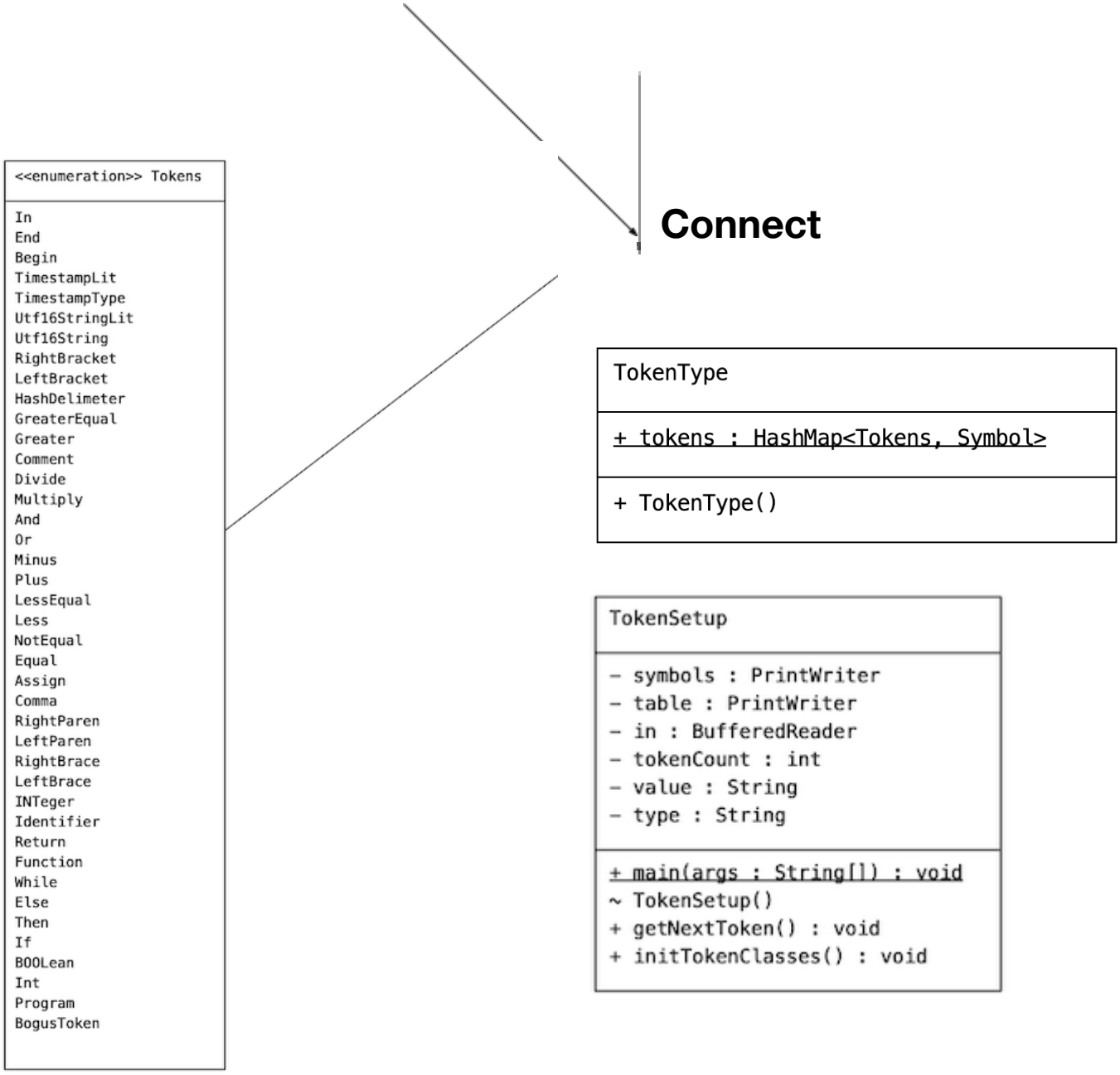
For the implementation of the TimeStampLit token, I created a helper method named, “timeCalc” that takes a String token, an integer index (String index at which a ‘~’ or ‘:’ are in the correct position in the token String), two integers “from” and “to” that hold the range of each element in the TimeStampLit token, and a char symbol that holds the symbol that is required to be at the index parameter. Next, I simply created a condition inside the digit checking algorithm within the nextToken() method that only executes if the character after a 4-digit number is a ‘~’. If this condition is met, then a switch statement that takes the length of the token and checks each important interval inside the token using the timeCalc() method.

Code Organization

To prevent repetition and code clutter I created the illegalch() and timeCalc() methods inside the lexer class. This also helped me debug more efficiently.

Class Diagram




<<enumeration>> Tokens
In
End
Begin
TimestampLit
TimestampType
Utf16StringLit
Utf16String
RightBracket
LeftBracket
HashDelimeter
GreaterEqual
Greater
Comment
Divide
Multiply
And
Or
Minus
Plus
LessEqual
Less
NotEqual
Equal
Assign
Comma
RightParen
LeftParen
RightBrace
LeftBrace
INTeger
Identifier
Return
Function
While
Else
Then
If
BOOLean
Int
Program
BogusToken
Connect
TokenType
+ tokens : HashMap<Tokens, Symbol>
+ TokenType()
TokenSetup
- symbols : PrintWriter
- table : PrintWriter
- in : BufferedReader
- tokenCount : int
- value : String
- type : String
+ main(args : String[]) : void
~ TokenSetup()
+ getNextToken() : void
+ initTokenClasses() : void

Results and Conclusion

This project taught me many new things about Strings and string manipulation in java, as well as, how to read and analyze pre-written code at a big scale. I successfully improved the output of lexical analysis to the required result and implemented a working algorithm that identifies and records the new Utf16String and timeStampLit tokens. While this project was a bit complex, my experiences and mistakes taught me lessons for the future and gave me confidence in my ability.

Challenges

Beginning this project was very tough since I'd never analyzed or edited a prewritten code project before and I felt extremely overwhelmed. Over some time, I examined every file in the lexer package and began to slowly grasp all the moving parts. Most of the challenges arose during the implementation of the Utf16String since setting up the double while loop made sense to me in theory; however, coding a successful algorithm that tests for every case required lots of trial and error. Implementing the timeStampLit algorithm inside the nextToken also gave me a few challenges with index positions and debugging for desired outcome.

Future Work

We can use the lexical class as part of a bigger parser algorithm to further give meaning to the stream of tokens inside an x file.