

Duccio Rocca  
ID = 922254031  
February 28th, 2022

# Assignment 2 Documentation

|  |           |
|--|-----------|
| <b>Github Repository .....</b>                     | <b>2</b>  |
| <b>Project Introduction and Overview .....</b>     | <b>2</b>  |
| <b>Execution and Development Environment .....</b> | <b>2</b>  |
| <b>Assumptions.....</b>                            | <b>2</b>  |
| <b>Scope of Work .....</b>                         | <b>3</b>  |
| <b>Compilation Result .....</b>                    | <b>4</b>  |
| <b>Implementation.....</b>                         | <b>5</b>  |
| <b>Lexer .....</b>                                 | <b>5</b>  |
| Filename provided as a command line argument.....  | 5         |
| Lexer Output.....                                  | 5         |
| Lexer Printout .....                               | 6         |
| NextToken method .....                             | 6         |
| <b>Tokens file .....</b>                           | <b>7</b>  |
| <b>Token class .....</b>                           | <b>7</b>  |
| <b>Code Organization .....</b>                     | <b>7</b>  |
| <b>Class Diagram .....</b>                         | <b>8</b>  |
| <b>Results and Conclusion .....</b>                | <b>9</b>  |
| <b>Challenges .....</b>                            | <b>9</b>  |
| <b>Future Work .....</b>                           | <b>9</b>  |
| <b>Summary of Technical Work.....</b>              | <b>10</b> |

## Github Repository

[Assignment two repository.](#)

Or

<https://github.com/sfsu-csc-413-fall-2022-roberts/assignment-2---lexer-RINO-GAELICO>

## Project Introduction and Overview

This project required me to modify a program that performs lexical analysis on a make-up language called X.

It required me to extend the Lexer class, in order to process additional tokens.

Moreover, it required me to improve the output to match a specific format provided with the instructions.

## Execution and Development Environment

As IDE I used Visual Studio Code on my MacBook Pro and I tested in both IDE and terminal through the command line.

JDK is Openjdk, version "17.0.2" 2022-01-18. The OpenJDK Runtime Environment is build 17.0.2+8-86, and OpenJDK 64-Bit Server VM is build 17.0.2+8-86 (mixed mode, sharing).

## Assumptions

I assumed that the lexical analysis tool was already able to recognize lexemes and create tokens based on the list in the tokens file. Therefore, my main task was to modify the program not to write it from scratch. Obviously, I had to understand the logic of the program in order to do that.

I also assumed that the TokenSetup class was correctly implemented to generate the two classes TokenType.java and Symbol.java.

I didn't assume that all tokens found by the lexer were valid, on the contrary, I accounted for the invalid token to be encountered and properly handled.

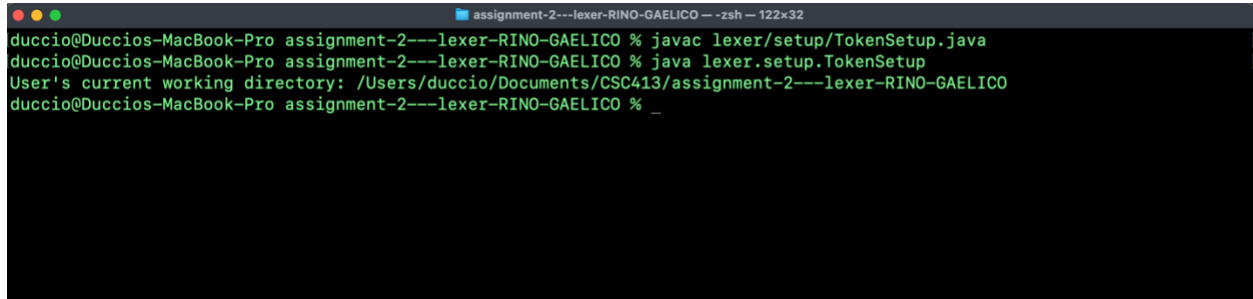
## Scope of Work

| Task   | Completed |
|--|-----------|
| Allow input via a filename provided as a command line argument                       | X         |
| Main method is currently commented out - you should uncomment and update this method | X         |
| In the event that no filename is supplied, a usage instruction should be displayed:  | X         |
| <pre>&gt; java lexer.Lexer usage: java lexer.Lexer filename.x</pre>                  |           |
| Tokens file must be updated  | X         |
| 1. Greater: >  | X         |
| 2. GreaterEqual: >=  | X         |
| 3. HashDelimiter: #  | X         |
| 4. LeftBracket: [  | X         |
| 5. RightBracket: ]   | X         |
| 6. Utf16String: utf16string  | X         |
| 7. Utf16StringLit: <utf16string>   | X         |
| 8. TimestampType: timestamp  | X         |
| 9. TimestampLit: <timestamp>   | X         |
| 10. Reserved words   | X         |
| 10.1 Begin: begin  | X         |
| 10.2 End: end  | X         |
| 10.3 In: in  | X         |
| TokenSetup run in order to re-generate the Tokens and TokenType classes.             | X         |
| The Token class must be updated to include the line number that a token was found    | X         |
| Lexer output must be updated for readability:  | X         |
| to include the line number from the Token  | X         |
| as well as the type of the token created   | X         |
| The format for each of the token lines is:   | X         |
| 1. 11 columns, left aligned, for the token description, then a space                 | X         |
| 2. left:, then a space   | X         |
| 3. 8 columns, left aligned, for the left position, then a space                      | X         |
| 4. right:, then a space  | X         |
| 5. 8 columns, left aligned, for the right position, then a space                     | X         |
| 6. line:, then a space   | X         |
| 7. 8 columns, left aligned, for the line number, then a space                        | X         |
| 8. The symbol <sup>[1]</sup> <sub>SEP</sub>  | X         |
| Lexer output must be updated to include a printout                                   | X         |
| with line number, of each of the lines read in from the source file                  | X         |
| Line numbers for here should be printed in 3 columns, right aligned                  | X         |

## Compilation Result

Using the instructions provided in the assignment 2 specification I typed in my terminal:

```
javac lexer/setup/TokenSetup.java
java lexer.setup.TokenSetup
```

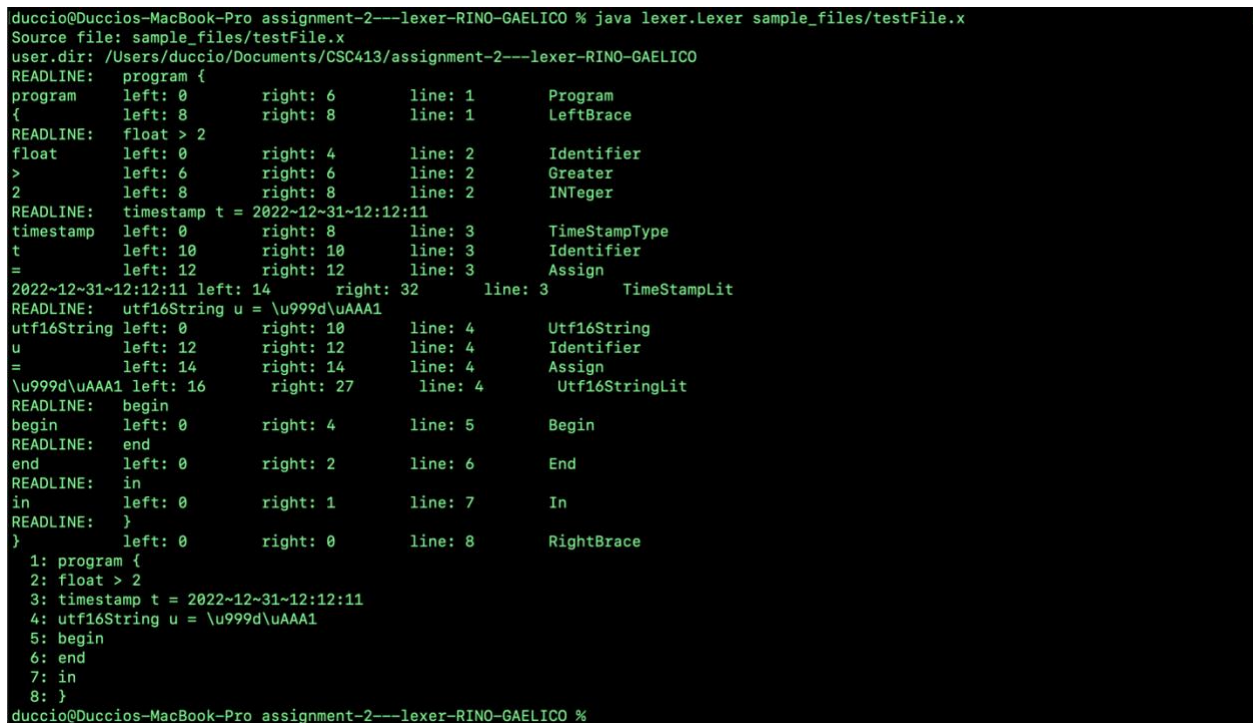


```
duccio@Duccios-MacBook-Pro assignment-2---lexer-RINO-GAELICO % javac lexer/setup/TokenSetup.java
duccio@Duccios-MacBook-Pro assignment-2---lexer-RINO-GAELICO % java lexer.setup.TokenSetup
User's current working directory: /Users/duccio/Documents/CSC413/assignment-2---lexer-RINO-GAELICO
duccio@Duccios-MacBook-Pro assignment-2---lexer-RINO-GAELICO % _
```

No error messages or warnings were displayed, and the application ran as expected.

Then to compile and run the Lexer.java file, which contained a main method, I typed:

```
javac lexer/Lexer.java
java lexer.Lexer sample_files/testFile.x
```



```
duccio@Duccios-MacBook-Pro assignment-2---lexer-RINO-GAELICO % java lexer.Lexer sample_files/testFile.x
Source file: sample_files/testFile.x
user.dir: /Users/duccio/Documents/CSC413/assignment-2---lexer-RINO-GAELICO
READLINE:  program {
program    left: 0      right: 6      line: 1      Program
{          left: 8      right: 8      line: 1      LeftBrace
READLINE:  float > 2
float      left: 0      right: 4      line: 2      Identifier
>          left: 6      right: 6      line: 2      Greater
2          left: 8      right: 8      line: 2      INteger
READLINE:  timestamp t = 2022-12-31-12:12:11
timestamp  left: 0      right: 8      line: 3      TimeStampType
t          left: 10     right: 10     line: 3      Identifier
=          left: 12     right: 12     line: 3      Assign
2022-12-31-12:12:11 left: 14      right: 32     line: 3      TimeStampLit
READLINE:  utf16String u = \u999d\uAAA1
utf16String left: 0      right: 10     line: 4      Utf16String
u          left: 12     right: 12     line: 4      Identifier
=          left: 14     right: 14     line: 4      Assign
\u999d\uAAA1 left: 16     right: 27     line: 4      Utf16StringLit
READLINE:  begin
begin      left: 0      right: 4      line: 5      Begin
READLINE:  end
end        left: 0      right: 2      line: 6      End
READLINE:  in
in         left: 0      right: 1      line: 7      In
READLINE:  }
}          left: 0      right: 0      line: 8      RightBrace

1: program {
2: float > 2
3: timestamp t = 2022-12-31-12:12:11
4: utf16String u = \u999d\uAAA1
5: begin
6: end
7: in
8: }
duccio@Duccios-MacBook-Pro assignment-2---lexer-RINO-GAELICO %
```

No error messages or warnings were displayed, and the application ran as expected.

## Implementation

### Lexer

Lexer was the class that required most of the modifications and adjustments. In fact, Lexer hosts the main function and it is also Lexer that returns the new token through its method `nextToken()`.

Main had to be modified in order to allow the user to input the filename through the command line. Main was also responsible for the output, which required some modifications to display the information in the format provided in Appendix A of the instructions. Lastly, it was Lexer that had the `SourceReader` object as its instance variable. This was responsible for opening the file and reading its content. Therefore, it was required to modify Lexer and the way it was interacting with its instance variable `source` to print each line correctly as shown in the instructions.

### Filename provided as a command-line argument

The implementation of this particular feature had to be carried out within the main function. The main method has a specific parameter that allows passing strings from the command line. **This** parameter is conventionally called `args` and it is an array of `Strings`. **To** allow the user to specify the filename through the command line **I had to make use of args**.

I decided to check if the length of the mentioned array was less than one, and, in case it was, a message was printed out in the console. The message was inviting the user to input a filename, showing the proper way to do it.

On the other hand, if the length of the arguments was more or equal to one, the first argument was used by the program as the filename.

I opted to check for greater or equal to one argument because I preferred the most conservative approach, which implied the use of the first argument regardless of other arguments entered. A different approach would have prevented the usage of the filename in case more than one argument was provided.

### Lexer Output

Another requirement consisted in implementing an output that would look exactly like the one provided in Appendix A. The specifications were provided in the instructions.

To do so, I used `System.out.format`. `Format` is a method from `PrintStream` class that takes two arguments. One is the string that specifies the formatting to be used and the other is a list of the variables to be printed using that formatting. In our case, the string providing the formatting had to follow the detailed specifications provided in the instructions. While the variables to be printed had to be obtained from the token object.

To obtain the correct information from the token object, I had to implement a new method called `getLineNumber` in the `Token` class. It consisted of a simple method returning the line number where the token was found within the file. The other methods used were `token.toString()`, `token.getLeftPosition()`, `token.getRightPosition()`, `token.getLineNumber()`, `token.getKind()`. These methods are getters and allow the public access to the instance variables of the token object.

## Lexer Printout

One more implementation consisted in introducing a printout at the end of each file that would represent the content of the file line by line. Each line of the printout had a particular format specified in the instructions. Moreover, in case an error was found the printout should have printed each line up to and included the line where the error was found.

To do so, I had to make sure that the content of each line was stored somewhere that could be easily accessible from the static main function in `Lexer`. To guarantee that the most recent line was always available I created a new instance variable in the `Lexer` class called `lastLine`. `LastLine` is a `String` variable that is updated every time a new line is reached, even though a new token is not created yet. This way, even though an error occurred before any token could be created, the content of the line was stored in a variable that could be accessed from other classes through a specific getter.

The other adjustment that I had to do was in the main function. I created an `ArrayList` (called `linesPrintOut`) that was updated every time a new line was encountered. To make sure the program knew if a new line was encountered I used a conditional that checks if the last token belongs to the same line of the previous token or a different line. This check is performed within the while loop in the main function. One more check is performed out of the loop, in case a new line was reached but no token was created because of invalid character or other errors.

The printout is finally outputted performing a loop that displays each element in the `ArrayList` `linesPrintOut`.

## NextToken method

The bulk of the work consisted in updating the `nextToken` method so that could recognize the newly added tokens, in particular, the two literals `TimeStamp` and `Uft16String`.

For `TimeStamp` extra care was needed since it could be easily confused with an integer. The way I decide to implement it was through a series of conditionals that step by step would check if the proper format was met for it to be a valid `TimeStamp`. Following the single responsibility principle, I implemented four helper methods that would check if the delimiter was correct (tilde or colon); one method would check if the two following characters were digits and another method would check if the limit (max and min) were respected. Finally, another helper method would advance the character and keep track of the whole string, incrementing it with the new character found.

I tried to make the code as much readable as possible, for example not having many deeply nested if statements. To do so I used inverted logic and called two other helper methods in case an invalid character was found or a number passed the limits (max or min).

For the `utf16String`, I implemented a similar mechanism, which involved the use of two helper methods. One would check the validity of the substring, from the backslash to the last hex digit. The other method would specifically verify if the 4 digits after the lower case 'u' were valid hex digits. This last method is called from inside the routine of the previous method. For this literal, the step is repeated twice, identically.

## **Tokens file**

This was the simplest step to implement. I simply added the token provided in the instructions to the list of tokens that were contained in the tokens file. Each Token had to be entered as a descriptor and as a type.

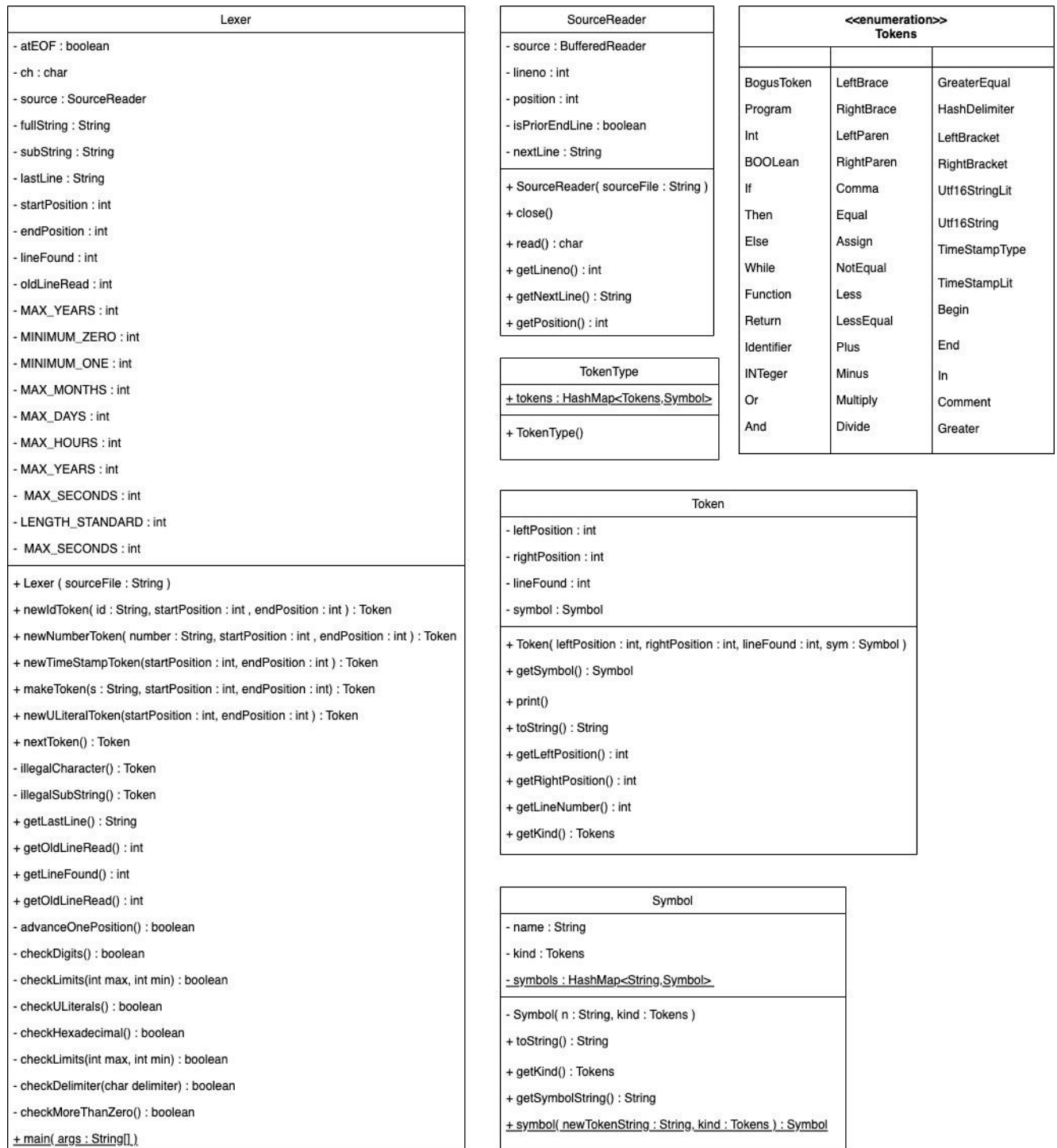
## **Token class**

I had to add an instance variable to be able to retrieve which line number of file the token was found.

This step didn't represent a major challenge, I simply added the integer attribute `lineFound`, and consequently, I added the getter method to get access to the instance variable.

## **Code Organization**

The project was already organized into packages, therefore I didn't change the organization of the folder. `Lexer` package contains the following classes: `Lexer.java`, `SourceReader.java`, `Symbol.java`, `Token.java`, `Tokens.java`, `TokenType.java`. Moreover, it contains two subfolders: `setup` and `sample_files`. The former is the package that contains `TokenSetup.java` and the file `tokens`, while the latter is the folder that contains the sample files written in language x.



## Class Diagram

The above class diagram shows the various classes included in the project (I did not include TokenSetup.java because I did not modify anything in that particular class).



## Results and Conclusion

I found this project extremely interesting since it gave me the opportunity to reflect and dive in on the very first phase of the work of a compiler. The challenges of breaking all those strings into small pieces and correctly identifying each lexeme. Even though the tokens we added were extremely complex, it took me some time to figure out the logic that had to be followed in order to make sure every token was treated correctly.

## Challenges

Debugging was profoundly helpful in the first part of the assignment, it helped me understand how the classes were interacting and the role of each class. I also used debugging when it came to making sure every corner case was accounted for and treated appropriately it helped me refine my code and improve the logic of the program.

The most challenging part was represented by the correct recognition and proper handling of the TimeStamp literal. I wasn't satisfied by my first solution because it didn't account for special cases like the one represented by 5 numeric digits followed by a tilde.

In that case, an Integer should have been formed and then the tilde should have been declared an invalid character. I then found a better way to handle that particular case and it involved a better order of the conditionals and when `source.read()` was called.

The most challenging aspect was probably to have the patience to analyze any possible combination characters that could be entered similar or resembling a valid TimeStamp literal. Once an ordered solution was found, it boiled down to refine the code to account for any possible corner case represented by a weird or unusual combination of characters.

## Future Work

I would like to be able to master the lexical analysis process and add new more complex tokens. I am also very interested in seeing this lexer program integrating with a parser that could simulate the actual work of a compiler.

Some tokens would require more attention and care, for example how to deal with decimals or other more ambiguous tokens.

I would like also to implement a tool that performs syntax analysis in coordination with the lexical analysis operated by the lexer. This is why I am looking forward to the next assignment.

## Summary of Technical Work

|                           |            |   |   |  |
|---------------------------|------------|---|---|--|
| <b>Documentation (27)</b> |            | A PDF is submitted that contains:   | ✓ |  |
|                           |            | Full name   | ✓ |  |
|                           | Overview   | A link to the github repository   | ✓ |  |
|                           |            | Project introduction  | ✓ |  |
|                           | Overview   | Execution and development environment described   | ✓ |  |
|                           | Overview   | Scope of work described (including what work was completed) - bullet points with checkboxes OK  | ✓ |  |
|                           |            | Instructions to compile and execute from command line - should be a list of instructions required, in order                                       | ✓ |  |
|                           |            | Assumptions made  | ✓ |  |
|                           | Discussion | Class diagram with hierarchy - only needs to include the objects that were touched for this assignment  | ✓ |  |
|                           | Discussion | Implementation decisions  | ✓ |  |
|                           | Discussion | Code organization   | ✓ |  |
|                           |            | Results/Conclusions   | ✓ |  |
|                           |            | Formatting  | ✓ |  |
| <b>Code Quality (15)</b>  |            | Code is clean, well formatted (appropriate white space and indentation)   | ✓ |  |
|                           |            | Classes, methods, and variables are meaningfully named (no unneeded comments exist to explain functionality - the identifiers serve that purpose) | ✓ |  |

|                      |                     |   |   |  |
|----------------------|---------------------|---|---|--|
|                      |                     | Methods are small and serve a single purpose            | ✓ |  |
|                      |                     | Code is well organized into a meaningful file structure | ✓ |  |
| <b>Requirement 1</b> | Lexer updated       |   | ✓ |  |
| <b>Requirement 2</b> | New tokens added    |   | ✓ |  |
| <b>Requirement 3</b> | Token Class Updated |   | ✓ |  |
| <b>Requirement 4</b> | Output updated      |   | ✓ |  |
| <b>Requirement 5</b> | Printout updated    |   | ✓ |  |
|                      |                     |   |   |  |