Deliverable 1 Scanner - Java

Darius Tiglao, Rosny Colas

CS 4308 SECTION 2

Deepa Muralidhar

Preface

This program acts as a scanner to read lexemes from a source text document. While the syntax that the scanner reads is based on Julia, the scanner should also be able to read other languages. The scanner should be able to read a source document’s lexemes by each source line and recognize each of those lexemes as a token. The token can take the form of an identifier, a constant, or a keyword. From there this scanner will be able to place these tokens into a global variable that is accessible by the program. Along with the token and its identifier, the scanner will be able to store the lexeme itself and the line it was derived from.

Code Implementation

The code for this scanner was implemented in Java. The choice of Java was simply because of the familiarity of the people in the group with the language and the readability and writability of it. While Python could have done this quite simply, the for each function was much easier to write for than the one in Python. The scanner was implemented by one method mostly, the “tokenCheck()” method was used to do the heavy lifting of the scanning, while the main program used a list of those TokenRecord’s to iterate through a scanned file and create tokens.

TokenRecord had three strings and one int respectively: lexeme, tokenCode, tokenName, srcLine. The lexeme string stored the literal string that was given by tokenCheck, a function that will be later explained. TokenCode was created to give the token, once identified, a code associated with what it was identified to be. TokenName would be the actual concept of what the lexeme was, may it be an identifier, constant, etc. The srcLine variable was the storing of what line the lexeme in question was found on. Along with the variables, a TokenRecord constructor was created to allow the parameters of the class to be determined.

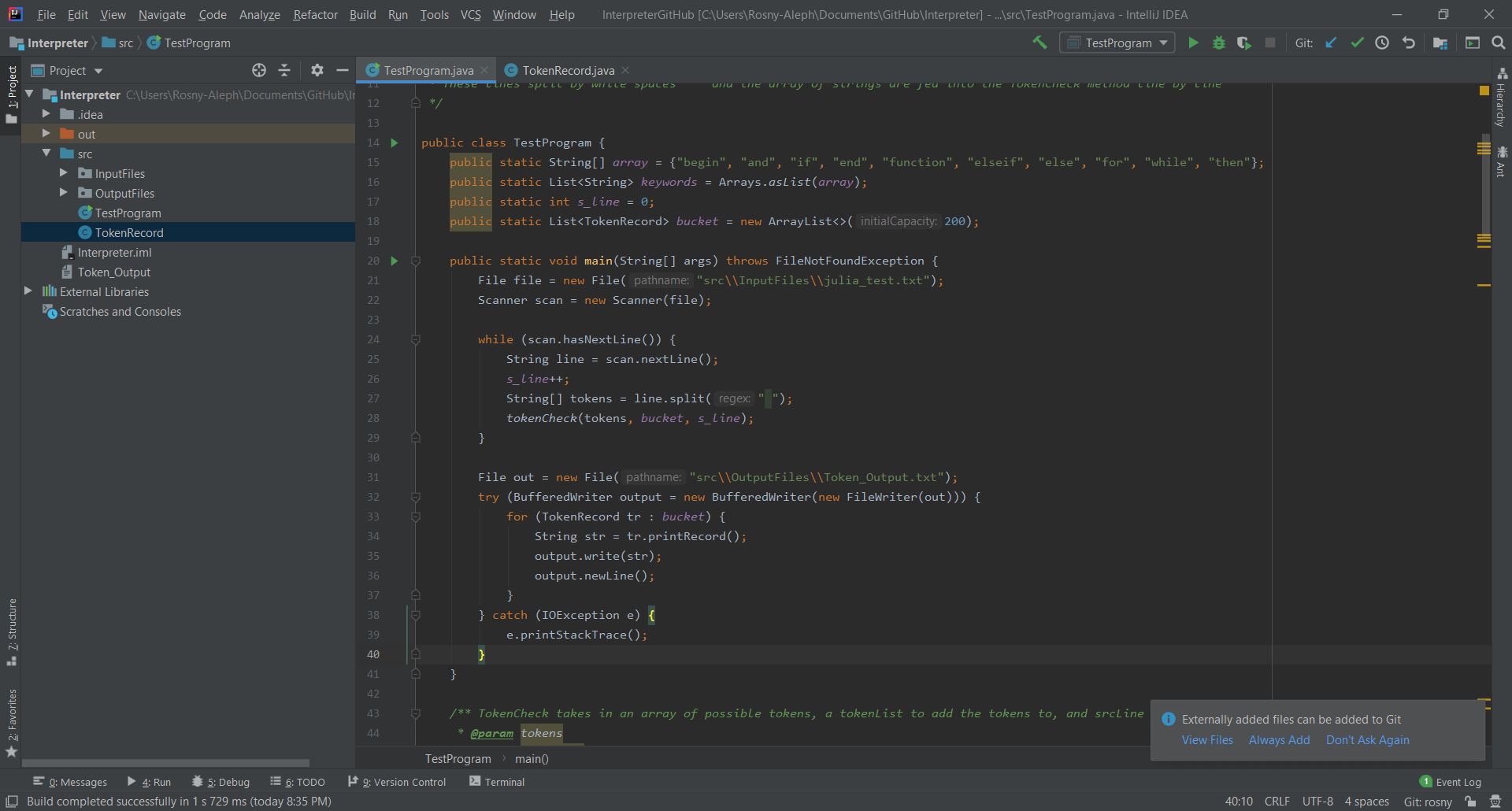
In TokenRecord is just one method, a print method. However, the tokenCheck() method in the testProgram class does most of the heavy lifting. TokenCheck(), was the most coded and heavily resourced. TokenCheck was given three variables: tokens, tokenList, and srcLine. Tokens was an array of strings that stored the tokens found in a file to be later turned into TokenRecord objects. TokenList was the current list of those TokenRecord objects. SrcLine was created to store the line number the token was found on. In the method, a for-each looped through the file and iterated until a semi-colon was found and took each segmented string via whitespace navigation. Along with this, there is also a check for tabs, which at the time gave a bug where the program found itself stuck inside the tab and creating stack overflow errors.

After the error-checking and cleaning the file, the TokenCheck method could sort through the strings given to create tokens out of them. The method by which the strings were checked was by a switch statement that took the string and compared it to various cases. These cases were broken down by operations, comparisons, and notations. If the character matched these then it was deemed to be the respective operation. If the character did not match any of the previous cases, it was deemed an identifier.

Along with the comparisons directly, each token is assigned a 4-digit code to determine the identity of the token. 0xxx for operations, 1xxx for keywords, 2xxx for literals, 3xxx for identfiers, and 4xxx for functions. This object creates a tokenRecord using these opcodes for the list in the TestProgram. Where the comparisons are made to the tokens provided by the string scanner.

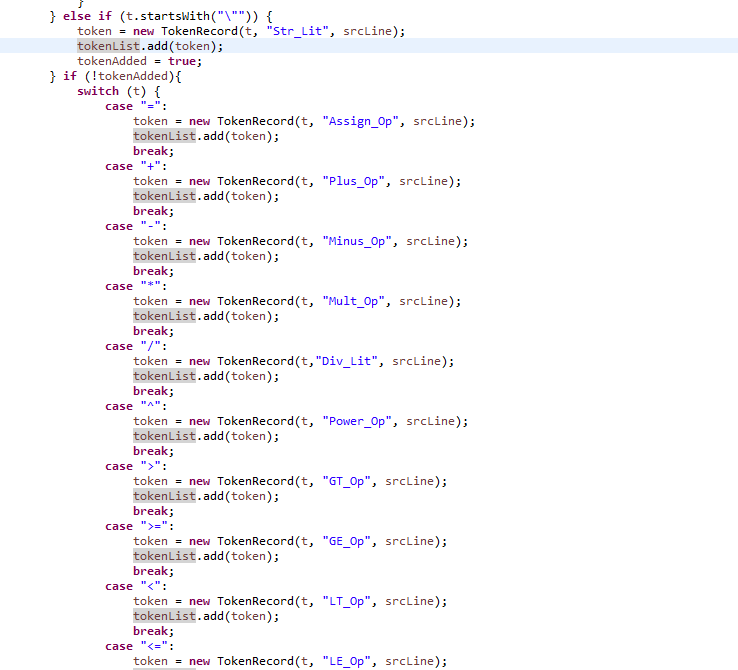
There were many issues with the implementation of the code. We had to determine how we wanted to segment the strings and read the file. There were guides that tried using a character-by-character flagging system that used a two-dimensional array that placed an identifier at the “x” axis and the actual lexeme at its “y” axis. But we opted to implement the string-fed method we had. This did create issues for identifying methods, since we could not determine the difference between parameters or operation-based parenthesis. We could use rhs-recursive iteration that could have allowed us to determine based on implicit clues, but we opted for what we had implemented.

Example/Code

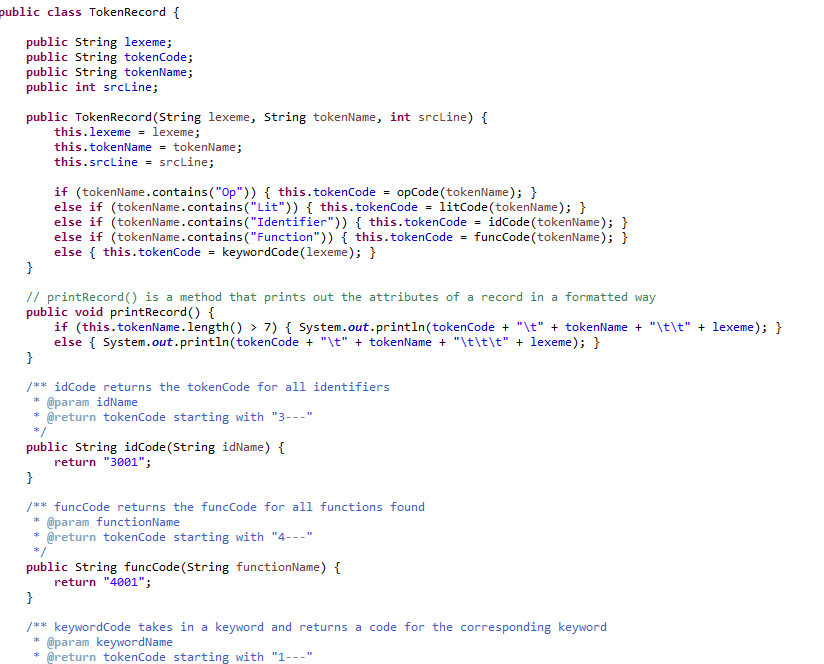


TestProgram.main()





tokenCheck – the switch statement extends to all operations for Julia; apply same logic for the assignment operator for all other operators



tokenRecord class

1001 Keyword begin

3001 Identifier x

0100 Assign\_Op =

2001 Int\_Lit 1

3001 Identifier y

0100 Assign\_Op =

2002 Float\_Lit 2.0

3001 Identifier k

0100 Assign\_Op =

3001 Identifier x

0107 Rem\_Op %

3001 Identifier y

4001 Function print("String")

2003 Str\_Lit "String"

3001 Identifier x

0100 Assign\_Op =

0116 LP\_Op (

3001 Identifier x

0101 Plus\_Op +

2001 Int\_Lit 3

0117 RP\_Op )

4001 Function print(4+7)

2001 Int\_Lit 4

0101 Plus\_Op +

2001 Int\_Lit 7

3001 Identifier y

0100 Assign\_Op =

2001 Int\_Lit 4

0101 Plus\_Op +

2001 Int\_Lit 5

3001 Identifier v

0100 Assign\_Op =

3001 Identifier x

0106 Power\_Op ^

3001 Identifier y

3001 Identifier v

0100 Assign\_Op =

3001 Identifier x

0106 Power\_Op ^

3001 Identifier y

1002 Keyword end

1003 Keyword function

4001 Function e()

4001 Function print("String")

2003 Str\_Lit "String"

1002 Keyword end

Example output using sample\_julia.txt – provided in d2l