Adrien Tremblay
40108982

29th January 2023

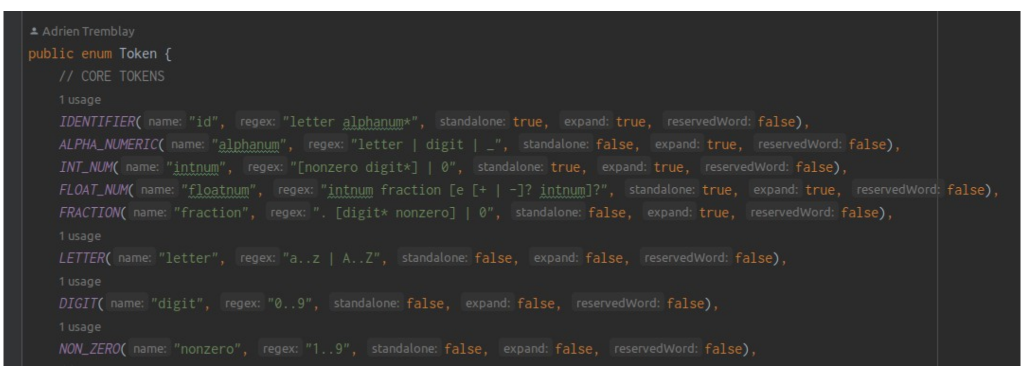
Assignment 1 Document

COMP442 Compiler Design

Section 1 – Lexical Specification

My solution takes the lexical specification regex as an input and converts it to an NDFA using *Thompson's Construction*. The NDFA is then converted to a DFA using *Rabin-Scott Powerset Construction*. The final DFA is represented as a State and Edge objects connecting them in the code. The DFA is then used as a guide during the lexing process. As such, I made a couple of minor adjustments for clarity sake to eliminate ambiguities that would otherwise be clear to human readers:

- Changed integer ::= nonzero digit* | 0 to be integer ::= [nonzero digit*] | 0
- Changed float ::= integer fraction [e [+|-] integer] to be float ::= integer fraction [e [+|-]? integer]?. The '?' character after a block '[]' specifies that the block is optional. So it can appear either 0 or 1 times.
- Changed fraction ::= .digit* nonzero | .0 to be fraction ::= [.digit nonzero] | 0



```
Adrien Tremblay
public enum Token {
    // CORE TOKENS
    1 usage
    IDENTIFIER( name: "id",  regex: "letter alphanum*",  standalone: true,  expand: true,  reservedWord: false),
    ALPHA_NUMERIC( name: "alphanum",  regex: "letter | digit | _",  standalone: false,  expand: true,  reservedWord: false),
    INT_NUM( name: "intnum",  regex: "[nonzero digit*] | 0",  standalone: true,  expand: true,  reservedWord: false),
    FLOAT_NUM( name: "floatnum",  regex: "intnum fraction [e [+ | -]? intnum]?",  standalone: true,  expand: true,  reservedWord: false),
    FRACTION( name: "fraction",  regex: ". [digit* nonzero] | 0",  standalone: false,  expand: true,  reservedWord: false),
    1 usage
    LETTER( name: "letter",  regex: "a..z | A..Z",  standalone: false,  expand: false,  reservedWord: false),
    1 usage
    DIGIT( name: "digit",  regex: "0..9",  standalone: false,  expand: false,  reservedWord: false),
    1 usage
    NON_ZERO( name: "nonzero",  regex: "1..9",  standalone: false,  expand: false,  reservedWord: false),
```

Figure 1.1 – The Lexical Specification in the code

Token		Regex
id	::=	letter alphanum*
alphanum	::=	Letter | digit | _
integer	::=	[nonzero digit*] | 0
float	::=	integer fraction [e [+|-]? integer]?
fraction	::=	.digit nonzero] | 0
letter	::=	a..z | A..Z
digit	::=	0..9
nonzero	::=	1..9

Figure 1.2 – Lexical Specification Regex Table

Section 2 – Finite State Automaton

The below NDFA and DFA are generated by my code from the Regex for the tokens. I use the *graphviz-java* (https://github.com/nidi3/graphviz-java) library to generate the image representations of each automata. They can be found in the */images* folder of the solution.

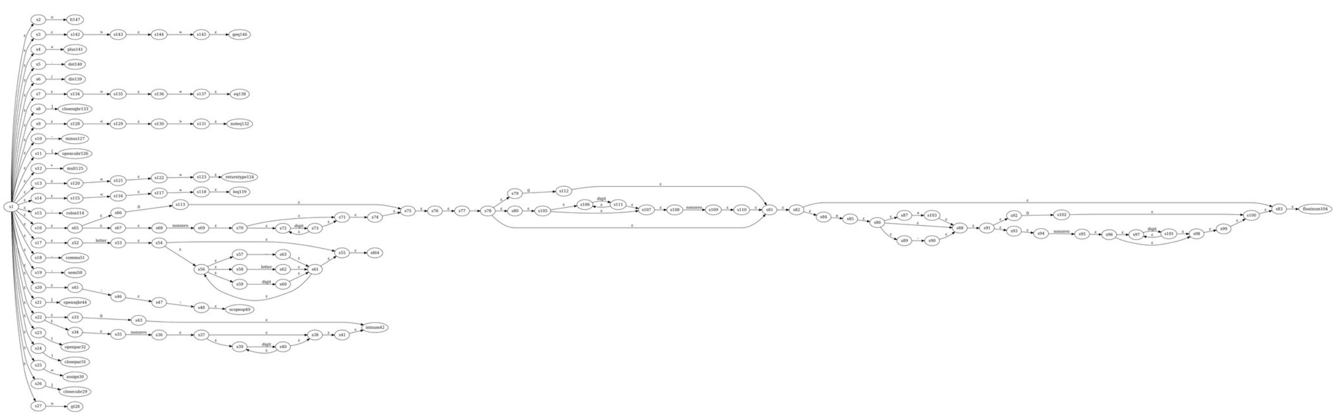


Figure 2.1 – The NDFA

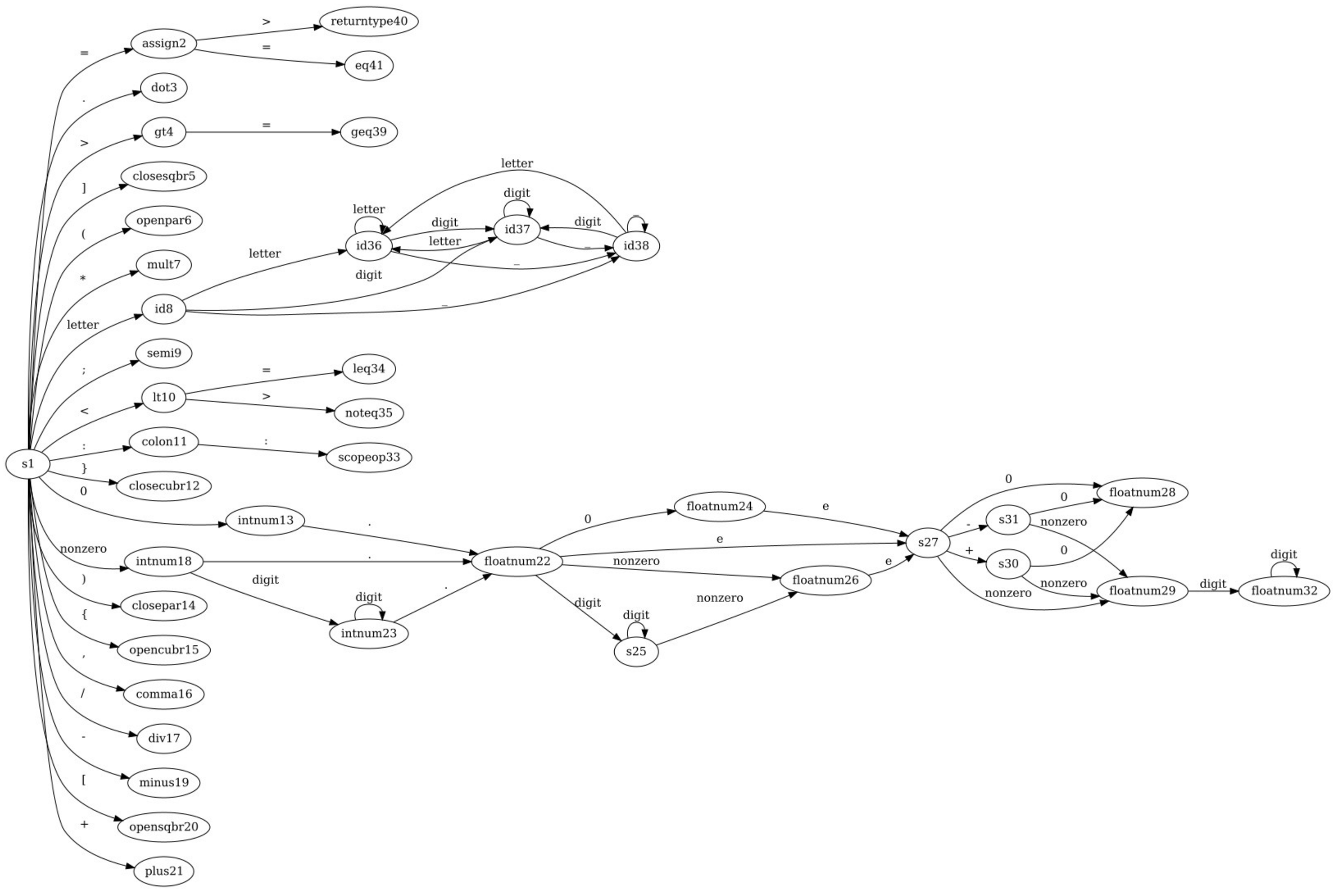


Figure 2.2 – The DFA

Each state for both graphs has a unique numerical id at the end of their name. Non-terminal states simply have the name s, while terminal states have the name of the token they are a terminal state for.

Adrien Tremblay
40108982

29th January 2023

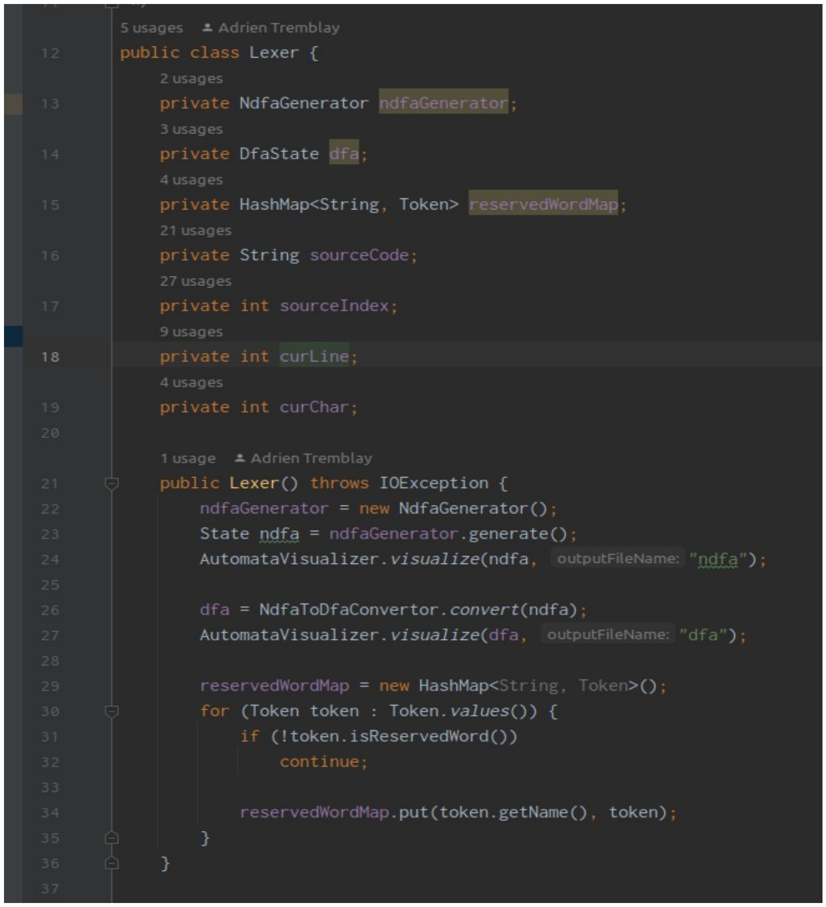
Section 3 – Design

This assignment was written in Java using Open JDK 18. The main method is location in src/Driver.class.

```java
import lexical_analysis.Lexer;
import lexical_analysis.TokenPrinter;

import java.io.IOException;

/**
 * I would have used JUnit to separate my test cases but the assignment asks for a Driver class
 */
public class Driver {
    public static void main(String args[]) throws IOException {
        Lexer lexer = new Lexer();
        TokenPrinter tokenPrinter = new TokenPrinter(lexer);

        String[] sourceFilePaths = {
                "test_source_files/my_test/my_test.src",
                "test_source_files/bubble_sort/bubble_sort.src",
                "test_source_files/lex_negative_grading/lex_negative_grading.src",
                "test_source_files/lex_positive_grading/lex_positive_grading.src",
                "test_source_files/polynomial/polynomial.src"
        };

        for (String sourceFilePath : sourceFilePaths)
            tokenPrinter.printTokens(sourceFilePath);
    }
}
```

Figure 3.1 – The Driver Class

Line 11 of the Driver creates a new Lexer class (located in src/lexical_analysis/Lexer.class).



```
5 usages  Adrien Tremblay
public class Lexer {
    2 usages
    private NdfaGenerator ndfaGenerator;
    3 usages
    private DfaState dfa;
    4 usages
    private HashMap<String, Token> reservedWordMap;
    21 usages
    private String sourceCode;
    27 usages
    private int sourceIndex;
    9 usages
    private int curLine;
    4 usages
    private int curChar;

    1 usage  Adrien Tremblay
    public Lexer() throws IOException {
        ndfaGenerator = new NdfaGenerator();
        State ndfa = ndfaGenerator.generate();
        AutomataVisualizer.visualize(ndfa,  outputFileName: "ndfa");

        dfa = NdfaToDfaConvertor.convert(ndfa);
        AutomataVisualizer.visualize(dfa,  outputFileName: "dfa");

        reservedWordMap = new HashMap<String, Token>();
        for (Token token : Token.values()) {
            if (!token.isReservedWord())
                continue;

            reservedWordMap.put(token.getName(), token);
        }
    }
```

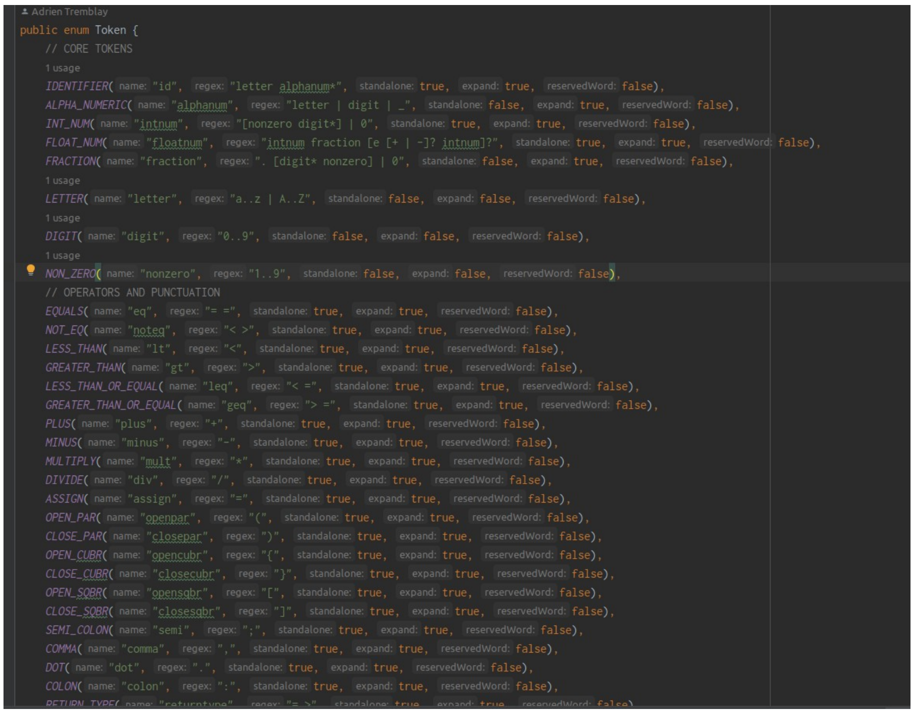
Figure 3.2 – The constructor of the Lexer class

The Lexer class' constructor creates an NdfaGenerator class and then generates an NDFA. The details of the NdfaGenerator won't be described in this document but can be explained by me during the demo if needed.

Adrien Tremblay
40108982

29th January 2023

The NdfaGenerator reads the lexical specification provided by the Token ENUM and combines it into one big NDFA using my implementation of *Thompson's Construction.* The resulting NDFA is a graph data structure comprised of State and Edge objects.



```
Adrien Tremblay
public enum Token {
    // CORE TOKENS
    1 usage
    IDENTIFIER( name: "id",  regex: "letter alphanum*",  standalone: true,  expand: true,  reservedWord: false),
    ALPHA_NUMERIC( name: "alphanum",  regex: "letter | digit | _",  standalone: false,  expand: true,  reservedWord: false),
    INT_NUM( name: "intnum",  regex: "[nonzero digit*] | 0",  standalone: true,  expand: true,  reservedWord: false),
    FLOAT_NUM( name: "floatnum",  regex: "intnum fraction [e [+ | -]? intnum]?",  standalone: true,  expand: true,  reservedWord: false),
    FRACTION( name: "fraction",  regex: ". [digit* nonzero] | 0",  standalone: false,  expand: true,  reservedWord: false),
    1 usage
    LETTER( name: "letter",  regex: "a..z | A..Z",  standalone: false,  expand: false,  reservedWord: false),
    1 usage
    DIGIT( name: "digit",  regex: "0..9",  standalone: false,  expand: false,  reservedWord: false),
    1 usage
    NON_ZERO( name: "nonzero",  regex: "1..9",  standalone: false,  expand: false,  reservedWord: false),
    // OPERATORS AND PUNCTUATION
    EQUALS( name: "eq",  regex: "= =",  standalone: true,  expand: true,  reservedWord: false),
    NOT_EQ( name: "noteq",  regex: "< >",  standalone: true,  expand: true,  reservedWord: false),
    LESS_THAN( name: "lt",  regex: "<",  standalone: true,  expand: true,  reservedWord: false),
    GREATER_THAN( name: "gt",  regex: ">",  standalone: true,  expand: true,  reservedWord: false),
    LESS_THAN_OR_EQUAL( name: "leq",  regex: "< =",  standalone: true,  expand: true,  reservedWord: false),
    GREATER_THAN_OR_EQUAL( name: "geq",  regex: "> =",  standalone: true,  expand: true,  reservedWord: false),
    PLUS( name: "plus",  regex: "+",  standalone: true,  expand: true,  reservedWord: false),
    MINUS( name: "minus",  regex: "-",  standalone: true,  expand: true,  reservedWord: false),
    MULTIPLY( name: "mult",  regex: "*",  standalone: true,  expand: true,  reservedWord: false),
    DIVIDE( name: "div",  regex: "/",  standalone: true,  expand: true,  reservedWord: false),
    ASSIGN( name: "assign",  regex: "=",  standalone: true,  expand: true,  reservedWord: false),
    OPEN_PAR( name: "openpar",  regex: "(",  standalone: true,  expand: true,  reservedWord: false),
    CLOSE_PAR( name: "closepar",  regex: ")",  standalone: true,  expand: true,  reservedWord: false),
    OPEN_CUBR( name: "opencubr",  regex: "{",  standalone: true,  expand: true,  reservedWord: false),
    CLOSE_CUBR( name: "closecubr",  regex: "}",  standalone: true,  expand: true,  reservedWord: false),
    OPEN_SQBR( name: "opensqbr",  regex: "[",  standalone: true,  expand: true,  reservedWord: false),
    CLOSE_SQBR( name: "closesqbr",  regex: "]",  standalone: true,  expand: true,  reservedWord: false),
    SEMI_COLON( name: "semi",  regex: ";",  standalone: true,  expand: true,  reservedWord: false),
    COMMA( name: "comma",  regex: ",",  standalone: true,  expand: true,  reservedWord: false),
    DOT( name: "dot",  regex: ".",  standalone: true,  expand: true,  reservedWord: false),
    COLON( name: "colon",  regex: ":",  standalone: true,  expand: true,  reservedWord: false),
```

Figure 3.3 – The Token ENUM

Line 24 of the Lexer uses the AutomataVisualizer class to generate an image of the NDFA using the *graphviz-java* library. Again, I won't go into the details of this class, but it essentially converts my representation of an automata to the library specific one, and then calls a library method to draw it to an image. Line 26 of the Lexer uses the NdfaToDfaConverter class to convert the NDFA to a DFA using my implementation of *Rabin-Scott Powerset Construction.* Line 27 of the Lexer then again calls the AutomataVisualizer class to draw the DFA. The Lexers constructor also populates the reservedWordMap starting on line 29 which is a HashMap containing all keywords. It is used during the lexing process to identifies keywords once an identifier is found.

The most important method of the Lexer class is the nextToken() method. It returns a value of type FoundToken which besides the Token, also contains the lexeme, and the line and char numbers that the token was found in the source code. The nextToken method is quite long and complicated so I will describe it according to line number. Lines 47-48 skips whitespace. Lines 54-62 detects incline comments and returns their tokens. Lines 65 to 91 comprises the logic to detect and return tokens for block comments. This will not cause errors if the block comment contains nested block comments. Lines 96-145 comprise the code that traversed the DFA starting from the root state. If a character is read and the current state has edges for that character, then the algorithm travels along this edge. This process continues until a character is read and the current state does not have edges for this character. In this case, if the the current state is terminal then the token for this state is returned along with it's lexeme and location. Otherwise an error is returned because the algorithm failed to identify the token.

An important thing to note is that my algorithm is actually a bit more complicated than this. Instead of keeping track of only one current state, the algorithm keeps track of all possible states for the current character. This is done because these is some ambiguity in the DFA. For example the character 0 can be interpreted as a Digit but also as the character '0'. Another example are non-zero digits. 3 could be interpreted as Nonzero or as a Digit. The ArrayList currentGeneration is used to keep track of these states. A second variable lastSeenTerminalState is then used to keep track of the last visited terminal state. When the algorithm fails to process a character, it returns the lastSeenTerminalState then backtracks to the location in the source code where this state was found. A more detailed line-by-line description of this method can be given during the demo.

```java
96          ArrayList<DfaState> currentGeneration = new ArrayList<DfaState>();
97          currentGeneration.add(dfa);
98
99          DfaState lastSeenTerminalState = null;
100         int lastSeenTerminalStateSourceIndex = 0;
101
102         while (currentGeneration.size() > 0 && sourceIndex < sourceCode.length()) {
103             char next = peekChar();
104             ArrayList<DfaState> offspring = new ArrayList<DfaState>();
105
106             for (DfaState survivor : currentGeneration) {
107                 if (survivor.isTerminal()) {
108                     lastSeenTerminalState = survivor;
109                     lastSeenTerminalStateSourceIndex = sourceIndex;
110                 }
111
112                 for (State.Edge edge : survivor.getEdges()) {
113                     if ((edge.label.equals(Token.LETTER.getName()) && isAlpha(next)) ||
114                             (edge.label.equals(Token.DIGIT.getName()) && isDigit(next)) ||
115                             (edge.label.equals(Token.NON_ZERO.getName()) && isNonZero(next)) ||
116                             (edge.label.length() == 1 && edge.label.charAt(0) == next)) {
117                         offspring.add((DfaState) edge.destination);
118                     }
119                 }
120             }
121
122             currentGeneration = offspring;
123             if (offspring.size() != 0)
124                 nextChar();
125         }
```

Figure 3.4 -Lines 96-125 of Lexer.nextToken()

```java
127         FoundToken ret;
128         if (lastSeenTerminalState != null) {
129             // backtracking
130             sourceIndex = lastSeenTerminalStateSourceIndex;
131             String lexeme = sourceCode.substring(foundTokenStartIndex, lastSeenTerminalStateSourceIndex);
132
133             if (lastSeenTerminalState.getPathToken() == Token.IDENTIFIER && reservedWordMap.containsKey(lexeme))
134                 return new FoundToken(reservedWordMap.get(lexeme), lexeme, curLine, foundTokenStartChar);
135
136             return new FoundToken(lastSeenTerminalState.getPathToken(), lexeme, curLine, foundTokenStartChar);
137         } else if (foundTokenStartIndex == sourceIndex) {
138             ret =  new FoundToken(Token.INVALID_CHAR, sourceCode.substring(sourceIndex, sourceIndex + 1), curLine, foundTokenStartChar);
139             nextChar();
140             return ret;
141         } else {
142             ret = new FoundToken(Token.ERROR, sourceCode.substring(foundTokenStartIndex, sourceIndex), curLine, foundTokenStartChar);
143             nextChar();
144             return ret;
145         }
```

Figure 3.5 -Lines 127-145 of Lexer.nextToken()

Adrien Tremblay
40108982

29th January 2023

Now, back in main method of the Driver class, line 12 creates a TokenPrinter class and passes it the Lexer that was created on line 11. The TokenPrinter class' job is simple. It reads a source code file, then uses the Lexer it has been passed to extract it's tokens by repeatedly calling the nextToken() method (line 38). It then prints the tokens in the appropriate .outlextokens file, and the errors to the appropriate .outlexerrors file.

```
17  public void printTokens(String sourceFilePath) throws IOException {
18      // reading and loading source
19      String source = readFileAsString(sourceFilePath);
20      lexer.loadSource(source);
21
22      // creating files
23      Path sourceFilePathAsPath = Path.of(sourceFilePath);
24      Path outputDir = sourceFilePathAsPath.getParent();
25      String sourceFileName = outputDir.getFileName().toString();
26
27      String tokensFileName = sourceFileName + ".outlextokens";
28      File tokensOutputFile = new File( pathname: outputDir + "/" + tokensFileName); // todo: make sure this works on windows
29      BufferedWriter tokensWriter = new BufferedWriter(new FileWriter(tokensOutputFile));
30
31      String errorsFileName = sourceFileName + ".outlexerrors";
32      File errorsOutputFile = new File( pathname: outputDir + "/" + errorsFileName); // todo: make sure this works on windows
33      BufferedWriter errorsWriter = new BufferedWriter(new FileWriter(errorsOutputFile));
34
35      // reading tokens and writing to file
36      int currentLine = 1;
37      FoundToken t;
38      while ((t = lexer.nextToken()).getToken() != Token.END_OF_FILE) {
39          if (t.getFoundOnLine() > currentLine) {
40              tokensWriter.write( str: "\n");
41              currentLine = t.getFoundOnLine();
42          }
43          tokensWriter.write(t.toString());
44          tokensWriter.write( c: ' ');
45
46          if (t.getToken() == Token.ERROR || t.getToken() == Token.INVALID_CHAR) {
47              errorsWriter.write( str: "Lexical error: ");
48              switch (t.getToken()) {
49                  case ERROR -> errorsWriter.write( str: "Error: ");
50                  case INVALID_CHAR -> errorsWriter.write( str: "Invalid character: ");
51              }
52              errorsWriter.write( str: "\"" + t.getLexeme()  +"\": ");
53              errorsWriter.write( str: "line " + String.valueOf( obj: t.getFoundOnLine() + ", "));
54              errorsWriter.write( str: "character " + String.valueOf( obj: t.getFoundOnChar() + ".\n"));
55          }
```

Figure 3.6 - The printTokens() method of the TokenPrinter class

Lines 14-20 of the Driver class specify a list of source code files. Then on lines 22-23 these files are looped through and fed to the tokenPrinter. Note that that the file test_source_files/my_test/my_test.src contains my test cases.

Adrien Tremblay
40108982

29th January 2023

Section 4 – Use Of Tools

The only tool used was the *graphviz-java* (https://github.com/nidi3/graphviz-java) library to generate image visualizations of the NDFA and DFA generated by my program. Since I did not use any online tool to create my automata, using this library which generates an image of the automata from code was convenient for me because It saved me from manually drawing my DFA. It also allowed me to verify the correctness of the NDFA and DFA that was generated by my code through visual inspection.