Requirement: Statement: Implement a parser algorithm

1. The parsing method:

   1.a. recursive descendent

2. The representation of the parsing tree (output) will be (decided by the team):

   2.a. productions string (max grade = 8.5)

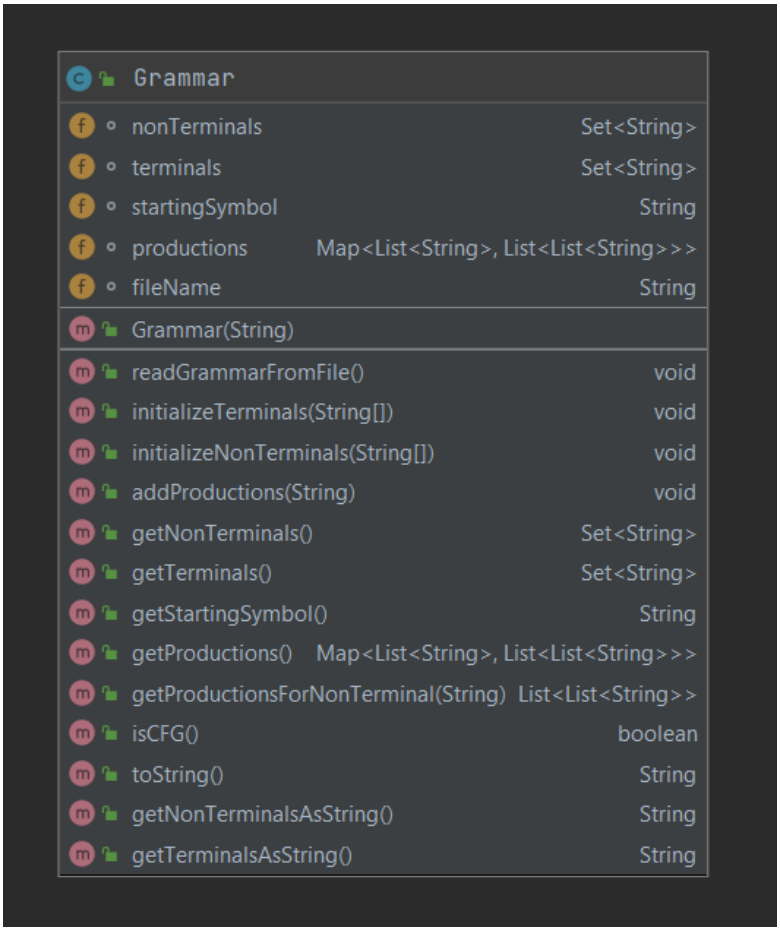
   2.b. derivations string (max grade = 9)

   2.c. table (using father and sibling relation) (max grade = 10)

PART 1: Deliverables

1. *Class Grammar* (required operations: read a grammar from file, print set of nonterminals, set of terminals, set of productions, productions for a given nonterminal, CFG check)
2. Input files: *g1.txt* (simple grammar from course/seminar), *g2.txt* (grammar of the minilanguage - syntax rules from Lab 1b)

Analysis and Design:



Grammar – the class for the Grammar entity.

- **Attributes**:
  - nonTerminals: Set<String> – the set of all nonTerminals, which are represented as Strings.
  - terminals: Set<String> – the set of all terminals, which are represented as Strings.
  - startingSymbol: String – the starting symbol of the grammar.
  - productions: Map<List<String>, List<List<String>>> – the productions of the grammar:

        - it is represented as a map, which has, as key, the left-hand side of a production, and as value – the right-hand side of all the productions of the key.
        - The key -> the left-hand side of a production is represented as a List of Strings, consisting of all the symbols from that lhs.
        - The value -> the List of all right-hand sides corresponding to the productions of the key, and each right-hand side is represented as a List of Strings, consisting of all the symbols from that rhs.
    - fileName: String – the name of the file in which the grammar is represented.

- **Main Methods:**
    - readGrammarFromFile() – scans the file in which the grammar is represented and finds the grammar's attributes.
    - initializeNonTerminals(String[] nonTerminals)
        - Pre: nonTerminals = the set of nonTerminals, read from file
        - Post: grammar. nonTerminals = nonTerminals
    - initializeTerminals(String[] terminals)
        - Pre: terminals = the set of terminals, read from file
        - Post: grammar.terminals = terminals
    - addProductions(String line)
        - Pre: line = a line from file, containing one or more productions for the same lhs, from which we extract the new_productions.
        - Post: grammar.productions = grammar.productions U {new_productions}

    - getProductionsForNonTerminal(String nonTerminal):
        - Pre: nonTerminal = a String, denoting the nonTerminal for which we want to get the productions.
        - Post: a List of the right-hand sides of all the productions of the nonTerminal.
    - isCFG():
        - Post: true, if the grammar is context-free; false otherwise.

Implementation:

https://github.com/LaviniaGalan/FLCD/tree/master/Lab5

Testing:

1) For the following CFG:

```
1  S A B
2  a b epsilon
3  S
4  S -> epsilon | a B | b A
5  A -> a | a A
6  A -> a A | b A
7  B -> b | b B
```

The results are:

```
-----------------------------------------
GRAMMAR:
NonTerminals: A B S
Terminals: epsilon a b
Starting symbol: S
Productions:
A->a | a A | b A |
B->b | b B |
S->epsilon | a B | b A |
-----------------------------------------
Productions for the nonterminal S: [[epsilon], [a, B], [b, A]]
Productions for the nonterminal A: [[a], [a, A], [b, A]]
Productions for the nonterminal B: [[b], [b, B]]

Is cfg: true
```

2) For the following grammar, not context free:

```
1  S A B
2  a b epsilon
3  S
4  S -> epsilon | a B | b A
5  A -> a | a A
6  A -> a A | b A
7  B -> b | b B
8  A B -> a
```

The results are:

```
-----------------------------------------
GRAMMAR:
NonTerminals: A B S
Terminals: epsilon a b
Starting symbol: S
Productions:
A->a | a A | b A |
B->b | b B |
S->epsilon | a B | b A |
AB->a |
-----------------------------------------
Productions for the nonterminal S: [[epsilon], [a, B], [b, A]]
Productions for the nonterminal A: [[a], [a, A], [b, A]]
Productions for the nonterminal B: [[b], [b, B]]

Is cfg: false
```