https://github.com/ComanacDragos/ToyLanguageCompiler

Statement: Implement a parser algorithm

1. One of the following parsing methods will be chosen (assigned by teaching staff):

   1.a. recursive descendent

   1.b. ll(1)

   1.c. lr(0)

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

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

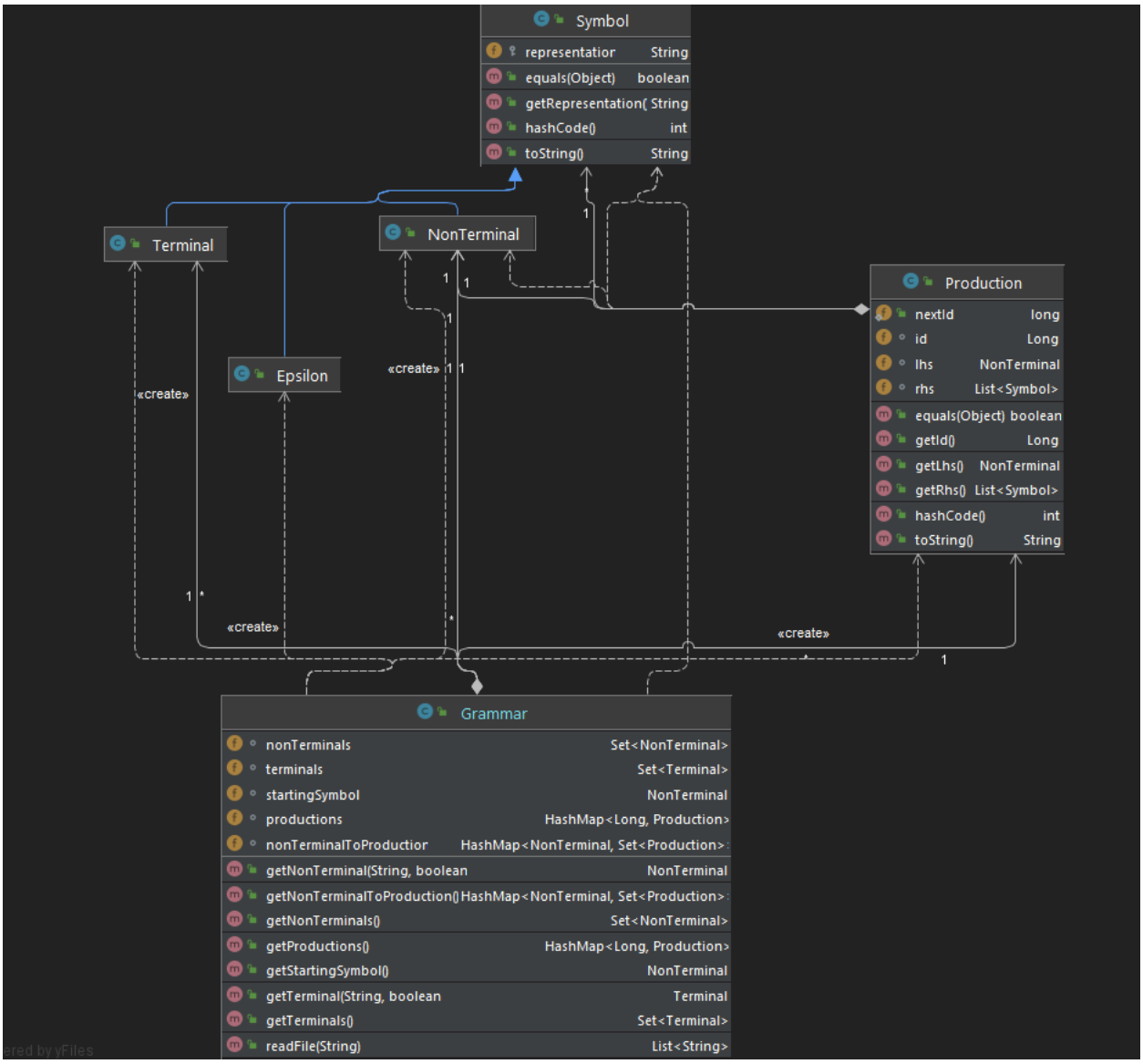
   2.b. derivations string (max grade = 9)

   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)

Implementation:



public class Grammar Used to represent the grammar
Set<NonTerminal> nonTerminals = new HashSet<>(); // set of non-terminals
Set<Terminal> terminals = new HashSet<>(); // set of terminals
NonTerminal startingSymbol; // starting symbol
HashMap<Long, Production> productions = new HashMap<>(); // maps the id of a production to the respective production
HashMap<NonTerminal, Set<Production>> nonTerminalToProduction = new HashMap<>(); // maps a non-terminal to all it's productions

```
/*
The file containing a grammar is processed
First line: the non-terminals separated by space
Second line: terminals separated by space
Third line: starting symbol
Remaining lines: productions: lhs is separated by rhs by ::= and | is used to separate rhs. List of symbols is separated by space
 */
public Grammar(String file)

/*
Returns the non-terminal with the given representation from the list of non-terminals
If the non-terminal does not exist and throwException is true, an exception is thrown
If the non-terminal does not exist and throwException is false null is returned
 */
public NonTerminal getNonTerminal(String representation, boolean throwException)

/*
Returns the terminal with the given representation from the list of terminals
If the terminal does not exist and throwException is true, an exception is thrown
If the terminal does not exist and throwException is false null is returned
 */
public Terminal getTerminal(String representation, boolean throwException)
```

The following classes are used to represent the symbols and productions

```
public class Symbol {
    protected String representation;

    protected Symbol(String representation)
```

public class Terminal extends Symbol
  public Terminal(String representation)

public class NonTerminal extends Symbol
  public NonTerminal(String representation)

public class Epsilon extends Symbol
  public Epsilon()

public class Production

public static long nextId = 0; // global id
Long id; // local id
NonTerminal lhs; // left-hand side of the production
List<Symbol> rhs; // right-hand side of the production

public Production(NonTerminal lhs, List<Symbol> rhs)

Testing

simple_grammar.in

S A
a b c d e
S
S ::= a S | b S c | d A
A ::= d c | e | e
A ::= d | e

syntax.in

program statement_list statement simple_statement compound_statement simple_type array_type type expression binary_operator unary_operator declaration_statement iostatement assignment_statement if_statement else_branch while_statement
id constant int char bool string float >> << while if else and or ! + - * / % > < >= <= != == = ; [ ] { } ( ) ,
^
program
program ::= statement_list
statement_list ::= statement | statement statement_list
statement ::= simple_statement | compound_statement

simple_statement ::= assignment_statement ; | iostatement ; | declaration_statement ;

compound_statement ::= if_statement | while_statement

simple_type ::= bool | char | int | string | float

array_type ::= simple_type [constant]

type ::= simple_type | array_type

expression ::= constant | id | id [constant] | expression binary_operator expression | unary_operator expression | (expression)

declaration_statement ::= type id | type id = expression

iostatement ::= << id | >> expression

assignment_statement ::= id = expression

if_statement ::= if (expression) { statement_list } else_branch
else_branch ::= epsilon | else { statement_list }

while_statement ::= while (expression) { statement_list }

unary_operator ::= !
binary_operator ::= + | - | * | / | ^ | % | and | or | > | < | >= | <=| != | ==