Harangus Vlad, Jercan Ioana – https://github.com/jercanioana/FLCD
https://github.com/vladharangus/Lab2

Statement: Implement a parser algorithm

1.b. ll(1)

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

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

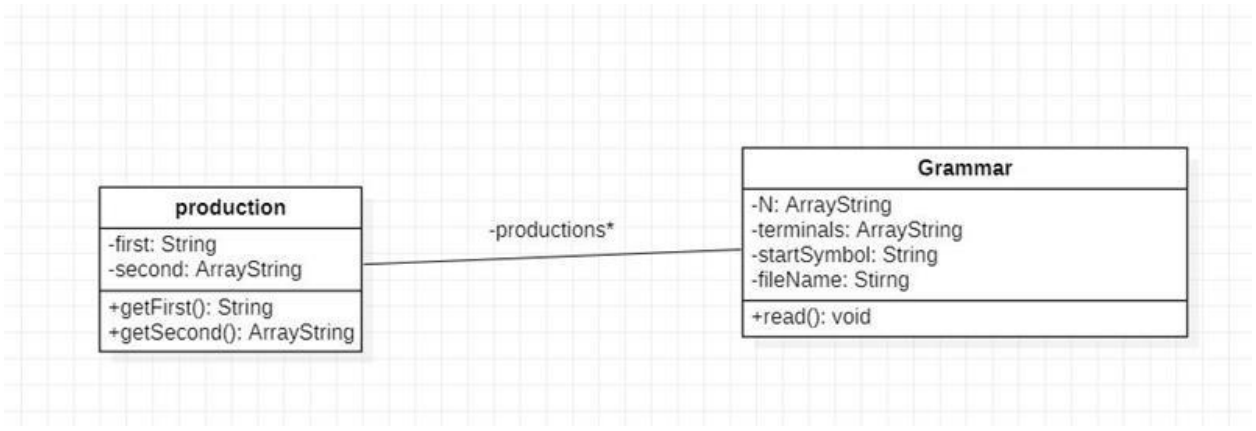
Input: g1.txt seq.txt

g2.txt, PIF.out

For this laboratory we created the Grammar class, which contains as attributes a list of strings for non terminals, a list of strings for terminals, a string for starting symbol and a list of productions. In order to represent the list of productions we created a new class called Production. The Production class has two fields: a string for the left-handside and a list of strings for the right-handside.

g1.txt (grammar from seminar):

```
S;A;B;C;D
+;*;a;(;)
S
S;B A
A;+ B A;#
B;D C
C;* D C;#
D;( S );a
```



For implementing the parser algorithm, we created a Parser class, which has as attributes the grammar used, the set of follows and of firsts, the parse table and the three stacks: alpha, beta, pi.

The first algorithm: We take all the productions for the given nonterminal and take the first element of the right hand side of each production. If it is a terminal or epsilon (#), we put it in the set, otherwise the first element is a nonterminal and we call the first function for it.

The follow algorithm: next k symbols generated after/ following A. As parameters we have the non-terminal A and an array for keeping the track of visited symbols. We take the productions containing A. for each right-hand-side we check whether the rule contains A. If so, if A is visited we continue to the next rule. If A is not visited we take the index of A and the next index. If the next index is at the end of the string, we add A in visited and add to the follow array of A all the symbols in the follow array of left-hand-side symbol in that production. If we are not at the end of the rule, we consider a temporary array of arrays of strings. From that index incremented to the final of the rule, at each step, we take the next symbol and add in the temporary array the array of firsts for that next symbol. The result is an array that we obtain by calling the "concat" function for temporary array which makes concatenation of length 1 for that temporary array of arrays. If the result array does not contains "#" we add every element in result to the follow set of A. If result contains "#", we still add every element in result to the follow set of A. Then we add A to visited and again add each element of follow set of left-hand-side symbol to the follow set of A.

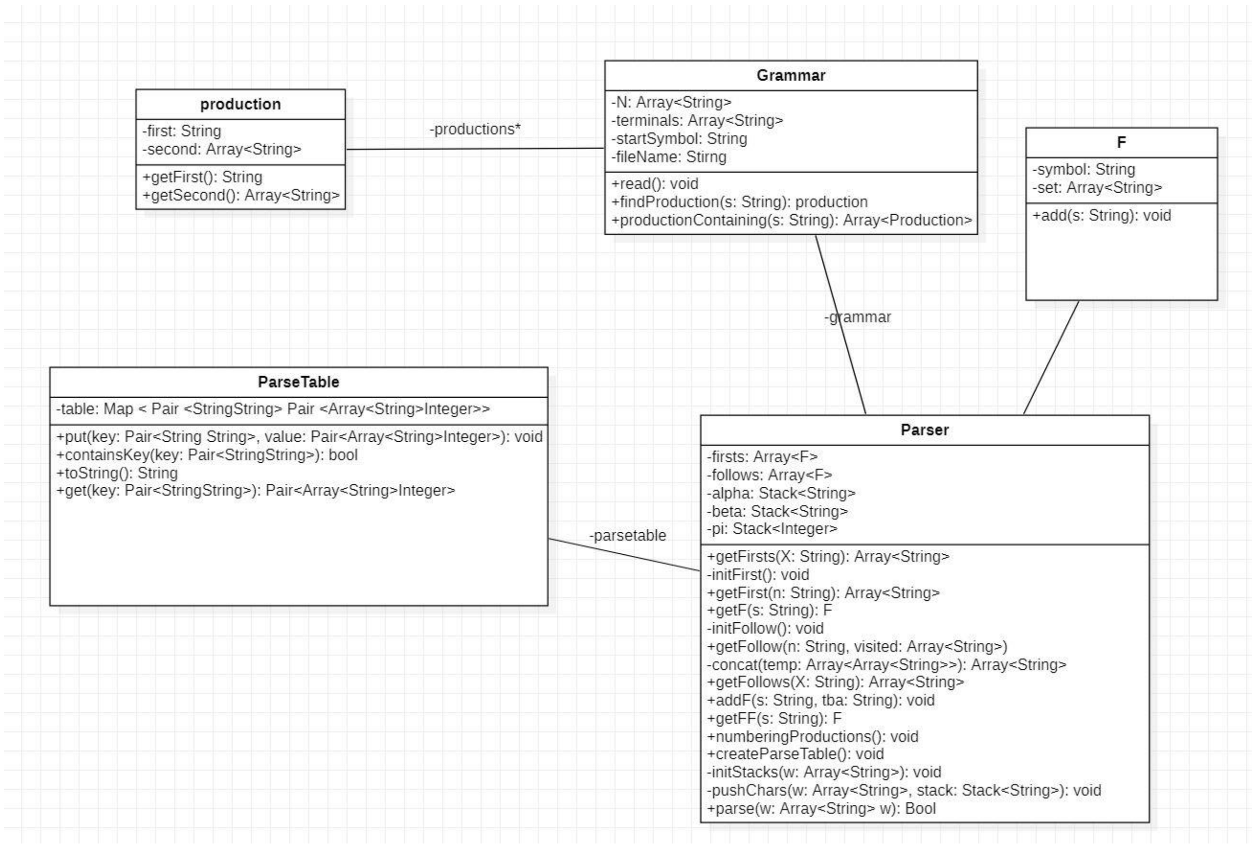
For representing the parse table, we created a new class: ParseTable. In order to represent it, we used a LinkedHashMap, which takes as key a pair of two strings(row and column) and as value a pair of an array of strings and an integer (the left handside of the production and it's corresponding number).

Algorithm for creating the table:

We initialize the columns array with the terminals, as well as the "$" symbol. Then, according to the rules, we populate the table. First, we put the value ("accept", -1) for the ("$", "$") cell and the ("pop", -1) values for the (terminal, terminal) cells. Then, for each production, we take the starting symbol which will represent the row and the Array of string from the right hand side. Using the values from the columns array, we create a Pair which takes as key the row and the column. If the first symbol of the right hand side of the production is the element from the current column and it is different from epsilon, we put in the table the following values: (row, column) -> (production, number of production). Otherwise if the first symbol is a nonterminal and the column symbol is in the First set of the nonterminal, then we put in the table the values (row, column) -> (production, number of production). If the first symbol of the right hand side is epsilon, then we use the follow set for the left handside of the production, and put in the table: (startingSymbol, symbol from follow) -> (production, number of production).

Algorithm for parse:

We initialize the two stacks: alpha(input stack), beta(working stack). According to the rules, we perform the following operations: We push onto the output stack the number of the production if we have a production from the head of the working stack which leads to the terminal from the head of the input stack. We pop if the head of alpha and beta are the same terminal. We accept if the heads of alpha and beta are both empty stack symbol ($), and error otherwise.


production
-first: String
-second: Array<String>
+getFirst(): String
+getSecond(): Array<String>
-productions*
Grammar
-N: Array<String>
-terminals: Array<String>
-startSymbol: String
-fileName: Stirng
+read(): void
+findProduction(s: String): production
+productionContaining(s: String): Array<Production>
F
-symbol: String
-set: Array<String>
+add(s: String): void
-grammar
ParseTable
-table: Map < Pair <StringString> Pair <Array<String>Integer>>
+put(key: Pair<String String>, value: Pair<Array<String>Integer>): void
+containsKey(key: Pair<StringString>): bool
+toString(): String
+get(key: Pair<StringString>): Pair<Array<String>Integer>
-parsetable
Parser
-firsts: Array<F>
-follows: Array<F>
-alpha: Stack<String>
-beta: Stack<String>
-pi: Stack<Integer>
+getFirsts(X: String): Array<String>
-initFirst(): void
+getFirst(n: String): Array<String>
+getF(s: String): F
-initFollow(): void
+getFollow(n: String, visited: Array<String>)
-concat(temp: Array<Array<String>>): Array<String>
+getFollows(X: String): Array<String>
+addF(s: String, tba: String): void
+getFF(s: String): F
+numberingProductions(): void
+createParseTable(): void
-initStacks(w: Array<String>): void
-pushChars(w: Array<String>, stack: Stack<String>): void
+parse(w: Array<String> w): Bool