## Requirement:

**Statement: Implement a parser algorithm (final tests)**

Input: 1) g1.txt + seq.txt

2) g2.txt + PIF.out (result of Lab 3)

Output: out1.txt, out2.txt

Run the program and generate:

- out1.txt (result of parsing if the input was g1.txt);

- out2.txt (result of parsing if the input was g2.txt)

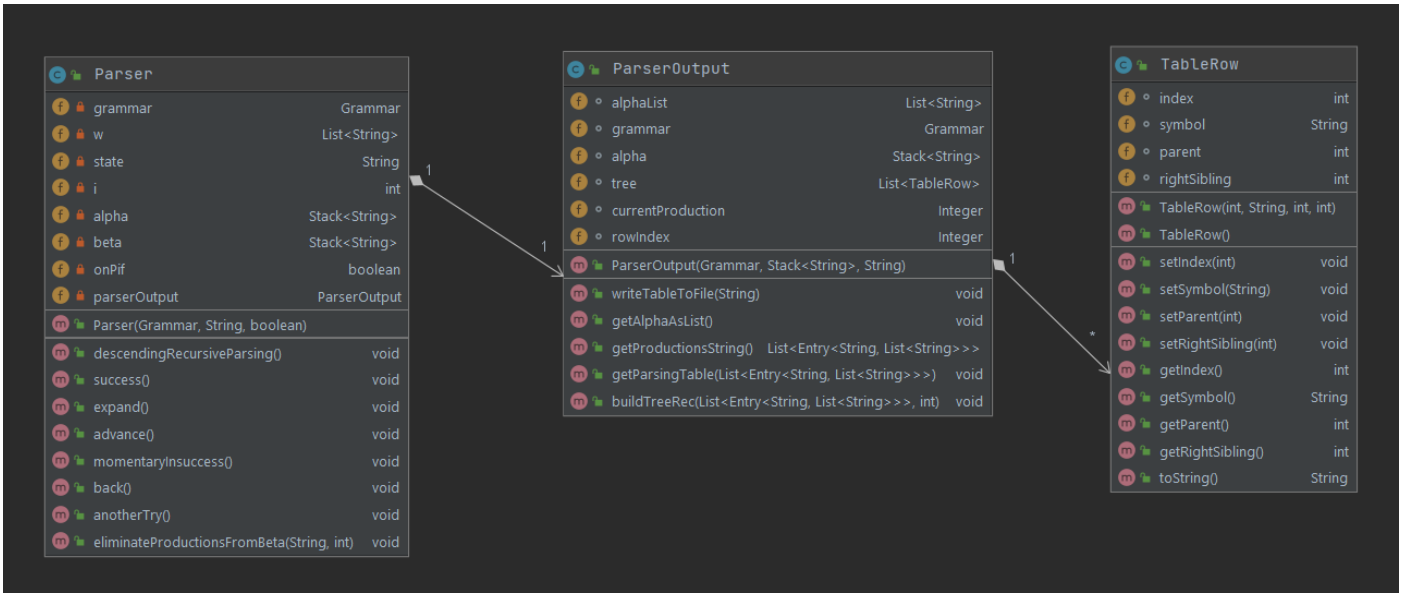
-messages (if conflict exists/if syntax error exists - specify location if possible)

**PART 4: Deliverables**

Source code for the parser + in/out files + documentation

Code review

## Analysis and Design:




The **Parser** is a class which uses Descendent Recursive Parsing Algorithm.

Its attributes are:

- grammar: Grammar – the grammar according to which the parsing is done
- w: List<String> – the word to be parsed, represented as a List of Strings (which represent terminal symbols)
- state: String – the current state of the parsing
- i: Int – the position of the current symbol in the input sequence w
- alpha: Stack<String> – the working stack

- beta: Stack<String> – the input stack
- alphaList: List<String> − a list containing all the symbols from the working stack, in the right order
- onPif: Boolean – a flag indicating if the parsing is done on the minilanguage or on a simple grammar (it is used to write in the propriate file)

Its methods are:

- descendingRecursiveParsing() – parses the input sequence using the descending recursive parsing; displays an appropriate message denoting if the sequence is accepted or not
- expand()
    - Pre: head of beta is a nonterminal A; state = “q”
    - Post: A is pushed to alpha; all the symbols from the rhs of the first production of A are pushed to beta
- advance()
    - Pre: head of beta is a terminal a, equal to the current symbol from input; state = “q”
    - Post: i = i+1; a is pushed to alpha and removed from beta
- momentaryInsuccess()
    - Pre: head of beta is a terminal a, different from the current symbol from input
    - Post: state = “b”
- back()
    - Pre: head of alpha is a terminal a, state = “b”
    - Post: i = i-1, a is pushed to beta and removed from alpha
- anotherTry()
    - Pre: head of the working stack is a nonterminal A, state=”b”
    - Post: if A still has a production A -> Gamma that was not used => state = “q”; A is pushed to alpha; all the symbols from Gamma are pushed to beta // else if i=1 and A=S, state=”e”// else state = “b”; A is pushed to beta and removed from alpha
- eliminateProductionsFromBeta(String A, int i) – eliminates from beta all the symbols from the top, that were added using the production i of the nonterminal A

The **ParserOutput** is a class that generates the parser tree represented as a table (using father and sibling relation).

Its attributes are:

- alpha: Stack<String> – the working stack
- grammar: Grammar – the grammar
- alphaAsList: List<String> – the working stack represented as a list, with all the symbols in the right order
- tree: List<TableRow> – the table
- currentProduction: Int – the index of the current used production in the list of all the productions from the working stack
- rowIndex: Int – the index of the row that is currently created

Its methods are:

- writeTableToFile(filename: String) – writes the parsing tree in the file given as parameter
- getAlphaAsList() – transforms the working stack, from stack to a list
- getProductionsString(): List<Map.Entry<String, List<String>>

    - Post: a list with all the productions used in parsing, in the right order. A production is represented as a map entry, that maps a string (the lhs) to a list of strings (a list of all the symbols from the rhs)
- getParsingTree(usedProductions: List<Map.Entry<String, List<String>>) – constructs and displays the parsing tree represented as a table
- buildTreeRec(usedProductions: List<Map.Entry<String, List<String>>, parent: Int)
    - Pre: usedProductions – a list with all the productions used in parsing, in the right order; parent – the index in the parsing table of the parent of the elements that will be added in the current iteration
    - Post: adds to the table the rows corresponding to all the symbols from the current productions

The **TableRow** class represents a row in the parsing tree represented as a table.

Its attributes are:

- index: Int – the index of the row in the table (the id)
- symbol: String – the symbol in the row
- parent: Int – the index of the parent in the table
- rightSibling: Int – the index of the rightSibling in the table

## Implementation:

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

## Testing:

1. Grammar =

```
S A B C
a b c v x epsilon
S
S -> a A C | b B
A -> x A | epsilon
B -> b B A v | b B | v
C -> c
```

a) Input sequence = “b b v v”

Result =

```
Sequence accepted.
0       S               -1      -1
1       b               0       2
2       B               0       -1
3       b               2       4
4       B               2       5
5       A               2       6
6       v               2       -1
7       v               4       -1
8       epsilon         5       -1
```

b) Input sequence = “b b c v”

Result =

```
Error.
```

1. Grammar = (left recursive grammar)

```
1       S A B C
2       a b c v x1 x2 s
3       S
4       S -> A a | b B
5       A -> A c | A s | x1 | x2
6       B -> b B A v | b B | v
7
```

The grammar is transformed:

```
GRAMMAR:
NonTerminals: A B S C AAux
Terminals: epsilon a b c s v x1 x2
Starting symbol: S
Productions:
A->x1 AAux | x2 AAux |
B->b B A v | b B | v |
S->A a | b B |
AAux->c AAux | s AAux | epsilon |
```

a) Input sequence = “x1 c s”

Result =

```
Sequence accepted.
0       S               -1      -1
1       A               0       2
2       a               0       -1
3       x1              1       4
4       AAux            1       -1
5       c               4       6
6       AAux            4       -1
7       s               6       8
8       AAux            6       -1
9       epsilon         8       -1
```

b) Input sequence = “x1”

```
Error.
```