FLCD - LAB 7 - PARSER ALGORITHM
POP VLAD - PURCEL LOREDANA

https://github.com/Pop-Vlad/FLCD-Parser

Parsing Method: RECURSIVE DESCENDENT

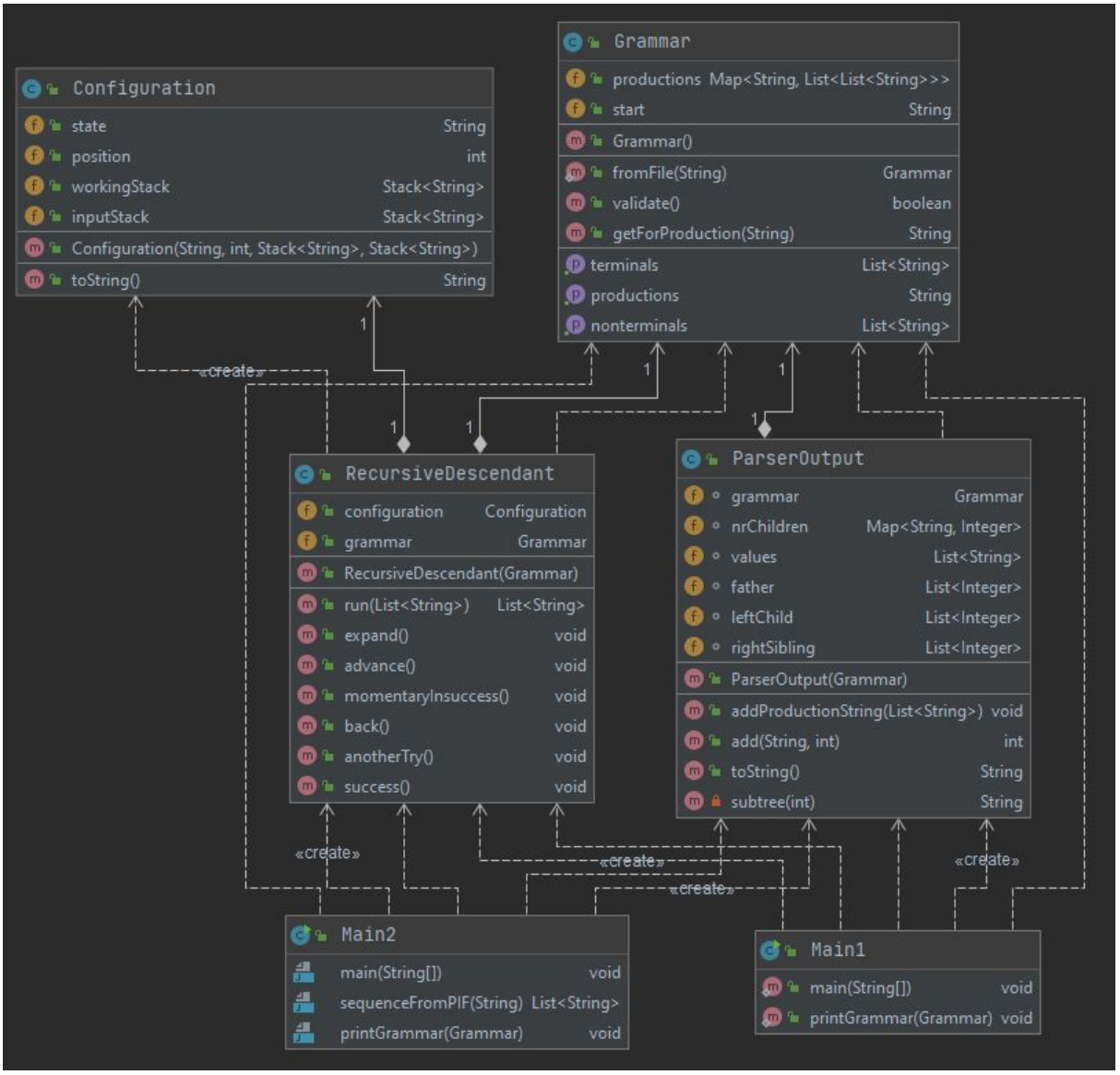
The representation of the parsing tree (output) will be: TABLE (using father and sibling relation)

1. Class grammar (required operations: read a grammar from file, print set of nonterminals, set of terminals, set of productions, production for a given nonterminal)
2. *Input*: g1.txt (grammar from seminar), seq.txt, g2.txt (grammar of the mini-language with syntax rules from Lab1), PIF.out (result of lab 3)
3. *Output*: out1.txt, out2.txt
4. Functions corresponding to parsing strategy
5. Functions corresponding to moves: **expand, advance, momentary insucces, back, another try, succes**
6. Algorithm corresponding to parsing tables (if needed) and parsing strategy
7. Class ParserOutput - DS and operations corresponding to table (lab 5) (required operations: transform parsing tree into representation; print DS to screen and to file)

Input File Format:

1. Nonterminals
2. Terminals
3. Starting symbol
4. 5. 6. ... Productions

Class Diagram:



Grammar class:

```
public static Grammar fromFile(String fileName) :
```

- Populates the “nonterminals” list, “terminals” list, productions and starting symbol from the file.

Configuration class:

Configuration: (s, i, **α**, ***β***)
Where:

- s(state) can be:
  - q: normal state
  - b: back state
  - f: final state

- e: error state
- i: position of current symbol in input sequence(w)
- **α**: working stack: stores the way the parser is built
- ***β***: input stack: part of the tree to be built

RecursiveDescendant class:

```
public List<String> run(List<String> w):
```

The function “run” stops when the state is either final or error so:
If the state is normal:

We EXPAND: when the head of the input stack is a nonterminal.

We ADVANCE: when the head of the input stack is a terminal AND this terminal is equal with the current symbol from input sequence.

Or we call MOMENTARY INSUCCESS: when the head of the input stack is a terminal AND this terminal is different from the current symbol from input sequence.

Otherwise if the state is back:

We call ANOTHER TRY: when the head of the input stack is a nonterminal.

Or we go BACK if the head of the input stack is a terminal.

Tests:

Input 1: **g1.txt**

- Grammar:

  S
  a b c
  S
  S->a#S#b#S
  S->a#S|c
- Input sequence(w): "a", "a", "c", "b", "c"

Output1:

- Working stack(**α**): S#1, a, S#2, a, S#3, c, b, S#3, c
- Tree:

```
\-- S#1
|-- a
|-- S#2
|          |-- a
|          \-- S#3
|                    \-- c
```

|-- b
\-- S#3
\-- c

Input 2: **g2.txt**

- Grammar:

program stmt_list stmt simple_stmt struct_stmt assign_stmt decl_stmt io_stmt
identifier constant expression array_elem term factor arithmetic_operand decl_stmt
type simple_type comp_type read_operand write_operand if_stmt while_stmt
condition relation pif_symbol
:= - + = * / % ( ) [ ] [] read write if else while int char { } ; == < > <= >= != 0 1
program
program->stmt_list
stmt_list->stmt#;|struct_stmt|stmt#;#stmt_list|struct_stmt#stmt_list
stmt->simple_stmt|struct_stmt
simple_stmt->assign_stmt|decl_stmt|io_stmt
assign_stmt->identifier#:=#expression|array_elem#:=#expression
expression->term#+#expression|term#-#expression|term
term->factor#*#term|factor#/#term|factor#%#term|factor
factor->arithmetic_operand|(#expression#)
arithmetic_operand->identifier|constant|array_elem
decl_stmt->type#identifier
type->simple_type|comp_type
simple_type->int|char
comp_type->simple_type#[#constant#]
io_stmt->read#(#read_operand#)|write#(#write_operand#)
read_operand->identifier|array_elem
write_operand->identifier|array_elem|constant
struct_stmt->{#stmt_list#}|if_stmt|while_stmt
if_stmt->if#(#condition#)#struct_stmt|if#(#condition#)#struct_stmt#else#stmt
while_stmt->while#(#condition#)#struct_stmt
condition->expression#relation#expression
relation->==|<|>|<=|>=|!=
identifier->pif_symbol
array_elem->identifier#[#expression#]
constant->pif_symbol
pif_symbol->0|1

- Input sequence(w): int 0 ; int 0 ; read (0) ; read (0) ; while (0 != 0) { if (0 > 0) { 0 := 0 - 0 ; } else { 0 := 0 - 0 ; } } write (0) ;