<https://github.com/ComanacDragos/ToyLanguageCompiler>

**Statement: Implement a parser algorithm (cont.)**- as assigned by the coordinating teacher, at the previous lab

**PART 2**: **Deliverables**

Functions corresponding to the assigned parsing strategy + appropriate tests,  as detailed below:

Recursive Descendent - functions corresponding to moves (*expand*, *advance*, *momentary insuccess*, *back*, *another try*, *success*)

**Implementation**

In grammar class the following function is added

For a given production id returns the next production of the left-hand side if it exists

Otherwise return null  
public Production getNextProductionForNonTerminal(Long previousId)

The following classes are introduced:

public class NonTerminalWithProduction extends NonTerminal   
 Long productionId;

public enum ParserState  
 NormalState,  
 BackState,  
 FinalState,  
 ErrorState

public class Parser   
 Grammar grammar;  
 ParserState currentState = ParserState.NormalState; // current state of the parser  
 int i = 0; // position of the current symbol in input sequence  
 Deque<Symbol> workingStack = new ArrayDeque<>();  
 Deque<Symbol> inputStack = new ArrayDeque<>();

// performs Recursive descendent algorithm on the given sequence  
public void parse(List<String> sequence)

The parse function uses the following functions corresponding to the Descendent Recursive algorithm’s actions:

// initializes the parser state  
public void init()

public void expand()

public void advance()

public void epsilonAdvance()

public void momentaryInsuccess()

public void back()

public void anotherTry()

public void success()

Graphical user interface, text

Description automatically generatedImage is also available on GitHub in documentation/lab9 folder

**Testing**

TestParser is a class that tests the Parser functions

S A B  
a b  
S  
S ::= a B | b A  
A ::= a | a S | b A A  
B ::= epsilon | b | b S | a B B

int[256] a;  
int i=0;  
int n;  
<<n;  
if (i<n){  
 <<a[i];  
 i=i+1;  
}  
n=0;

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 expression' expression\_simple  
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\_simple ::= constant | id | id [ constant ] | id [ id ] | unary\_operator expression | ( expression )  
  
expression' ::= binary\_operator expression expression' | epsilon  
  
expression ::= expression\_simple expression'  
  
declaration\_statement ::= type id | type id = expression  
  
iostatement ::= << id | << id [ constant ] | << id [ 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 | > | < | >= | <=| != | ==