<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](https://moodle.cs.ubbcluj.ro/mod/assign/view.php?id=2562))

**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*)

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

Input: 1) g1.txt + seq.txt

          2) g2.txt + PIF.out (result of [Lab 3](https://moodle.cs.ubbcluj.ro/mod/assign/view.php?id=2657))

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 3:** **Deliverables**

1.Algorithms corresponding to *parsing table* (if needed) and *parsing strategy*

2. Class *ParserOutput*- DS and operations corresponding to choice 2.a/2.b/2.c ([Lab 5](https://moodle.cs.ubbcluj.ro/mod/assign/view.php?id=2841)) (required operations: transform parsing tree into representation; print DS to screen and to file)

**PART 4: Deliverables**

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

Code review

**Implementation**

**A picture containing text

Description automatically generated**

Picture also available at <https://github.com/ComanacDragos/ToyLanguageCompiler/tree/main/documentation/lab11>

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)

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 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)

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()

Node  
 Long id = nextId++;  
 Symbol value;  
 List<Node> children;

public Node(Symbol value, List<Symbol> symbols) // creates recursively the tree

public class TreeGenerator   
 Node root; // root of the tree  
 Long nextId = 0L; // id generator for nodes  
 Deque<NonTerminalWithProduction> productionStack;  
 Grammar grammar;

public TreeGenerator(Deque<Symbol> workingStack, Grammar grammar) // instantiate the productionStack from a given workingStack so that it contains only the NonTerminalWithProduction classes

public void generateTree() // begins the generation of the tree

public void toFile(String outputDir) // writes to an output directory the table

private List<String> constructTableRecursive(Node currentNode, Node fatherNode, Integer positionRelativeToFather) // constructs the string representation of the table recursively

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

TestParser is a class that tests the Parser functions

simple\_grammar

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 | > | < | >= | <=| != | ==

unary\_operator ::= !  
binary\_operator ::= + | - | \* | / | ^ | % | and | or | > | < | >= | <=| != | ==

Output for simple\_grammar

Table

Description automatically generated with low confidence

Output for syntax grammar

Graphical user interface

Description automatically generated with medium confidence