Lab 6

https://github.com/adrianPascan/FLCD_Parser.git

The application is structured in the following classes:

- Class Grammar
- Class State
- Class Parser
- Class ParserOutput

Class Grammar

Contains the following attributes:

- Terminals – a list containing the terminals
- Nonterminals – a list containing the nonterminals
- Productions – a dictionary of productions
- StartSymbol – the starting symbol

And the following methods:

- __init__ - receives the name of the file containing the data to be initialised, initialises the attributes with None and calls the function fromFile to decide the values of the attributes. Params: file – name of the file ; Returns: -
- fromFile - Reads line by line from the file:, first line should contain terminals separated by, second line should contain the nonterminals(separated by ,); third line should contain the start symbol; from the forth line til the end of the file should be the productions under the form : the nonterminal + ':' right side + "|" + right side or just right side. Params: filename – name of the file; Returns: -
- parseLine – Splits the given string by , and deletes the useless spaces. Params: line- the string containing the line from the file; Returns: The list modified as explained before.
- parseProductions - It creates a dictionary that has as keys the states, and as value, a list with dictionaries, having as keys values from the alphabet, and as values the states in which we will arrive. Params: parts – a list of strings that represent the lines of the file containing the productions; Returns: The dictionary
- getStartSymbol – Returns the start symbol. Params: - ; Returns: start symbol.
- getNonTerminals – Returns the list containing the nonterminals. Params: - ; Returns: nonterminals.
- getTerminals – Returns the list containing the terminals. Params: - ; Returns: terminals.
- getProductions – Returns the dictionary containing the productions. Params: - ; Returns: the productions.
- getProductionsN – Returns the list with the productions of a given nonterminal. Params: non – the nonterminal for which we search the productions. Returns: the list of productions
- getFirstProduction – Returns the first production of a given nonterminal. Params: non- the nonterminal for which we search the first production; Returns: the first production.

- getNextProduction- Returns the first production after the given one from the given list of productions. Params: production – the production previously to the one that we search, productions- the list of productions; Return – None if there is no next, the next production otherwise
- printNonterminals – Prints on the console the list of nonterminals. Params: -; Return: -.
- printTerminals – Prints on the console the list of terminals. Params: -; Return: -.
- printProductions - Prints on the console the dictionary of productions. Params: -; Return: -.
- printProductionsN – Prints the productions of a given nonterminal. Params: non – the nonterminal for which we are printing the productions. Return: -
- main – Prints the available operations and lets the user make a choice, then it calls the corresponding function. Params: -. Return: -.

Class State

Contains the following values:

- NORMAL = 'Q'
- ERROR = 'E'
- BACK = 'B'
- FINAL = 'F'

Class Parser

Contains the following attributes:

- State – the current state
- Index - the current index
- MaxIndex - the index from which we started making back (so we can find where the error occurred)
- WorkingStack - the working stack
- InputStack - the input stack
- G - the grammar that we are using
- Sequence - the sequence that we are checking
- Left_recursive – keeps True or False depending if the grammar is left recursive or not

And the following methods:

- __init__ – Initialises the state with NORMAL, sets the index & maxIndex to 0, creates an empty list for the working stack, puts the start symbol in the inputStack, sets the sequence to None and sets the value of g to grammar(the received value through the parameter). Also, checks if the grammars is left recursive or not and puts it in left_recursive. Params: grammar – the grammar of the mini language. Returns: -
- Momentary_insucces – Makes the expected prints and changes the state to BACK. Params: - Returns: -

- Succes – Makes the expected prints and changes the state to FINAL. Params: - Returns: -
- Expand – Pops the nonterminal from the input stack, searches for its first productions and adds the list of [nonterminal, production] into the working stack, the pushes into the input stack the production. Params: - Return: -
- Advance – It increases the value of the index, then pushes the element from the top of the input stack into the working stack and pops the current element from the input stack. Params: - Returns: -
- Back - Does the opposite of advance, it decreases the value of the index, pops the element from the working stack and pushes it to the top of the input stack. Params: - Returns: -
- Another_try – It takes the top element from the working stack and using the nonterminal it searches for the next production. If there is one, we change the state to normal, we pop the element from the workingStack and pushes the new production, also in the input stack. If we do not find a next production, the index is 0 and the nonterminal extracted is the startSymbol, we change the state to ERROR. Else, we pop from the working stack and add it to the input stack.Params: - Returns: -
- CheckSequence – While we don't get into the State ERROR or FINAL, we try to check if the received sequence is accepted or not. If the state is Final, we call the parsing_production_string function. Params: seq – the sequence to be checked. Returns: True and the rules if the sequence is accepted, else False and empty set.
- Check_grammar_left_recursive – Checks if the received grammar is left recursive by verifying if the nonterminal from the lefthand side of the production appears in the righthand side. Params: -. Returns: True if it is left recursive, False otherwise.

Class ParserOutput

Contains the following attributes:

- Grammar - the grammar that we are using
- WorkingStack - the working stack
- Root - the root of the tree

And the following methods:

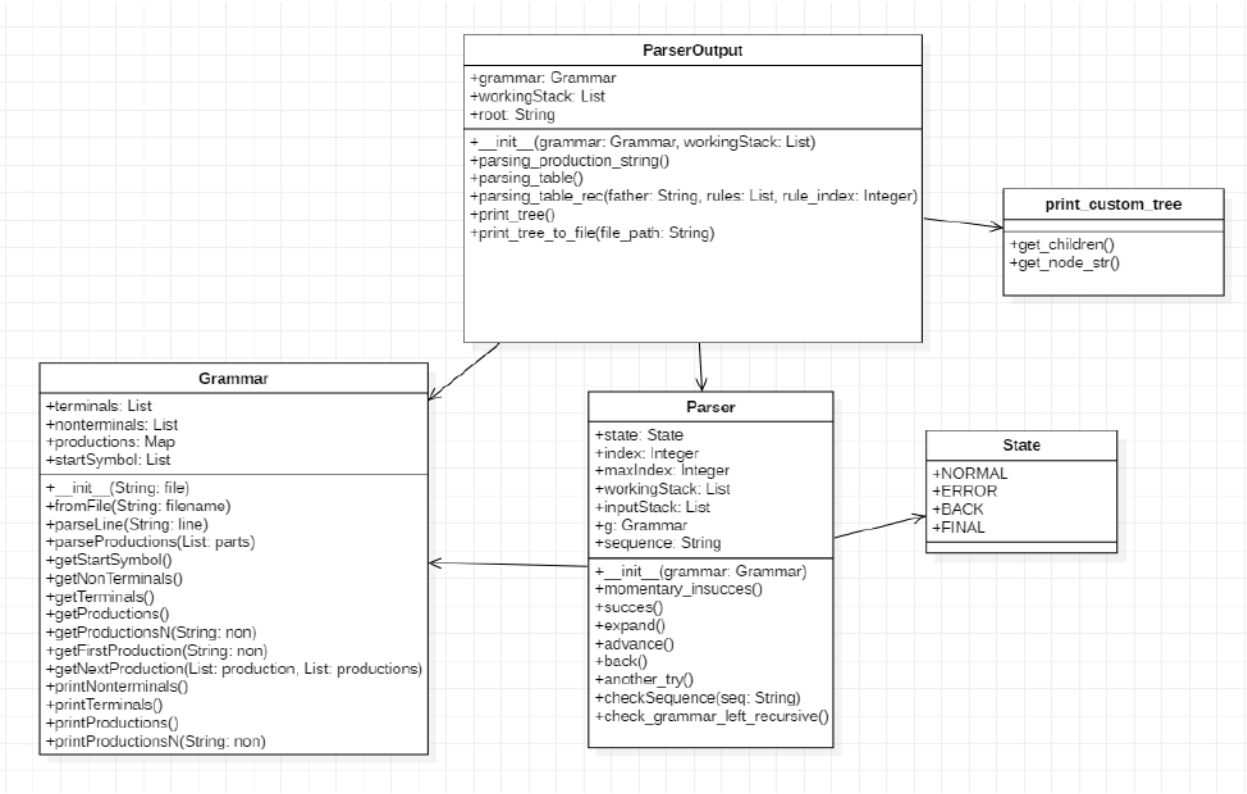
- __init__ – Initiates the value of grammar with the corresponding value received through parameter, does the same to the workingStack and sets root to None. Params: grammar – the grammar to be used, workingStack – the working stack from the parser. Returns: -
- Parsing_production_string – Creates a list with the production strings. Params: - Returns: the list
- Parsing_table – Sets the root of the tree and calls the parsing_tree_rec to decide the children. Params: - Returns: -
- Parsing_table_rec – Computes recursevely the children of a node and puts them in place in the tree. Params: father – the node for which we search the children, rules – the rules obtained from the function parsing_production_string, rule_index – the index of the rule. Returns: the rule index for the tree.
- Print_tree – Calls the print_custom_tree method. Params: - Return: the tree printed
- Print_tree_to_file – Calls the print_tree function and writes it into a file. Params: file_path – the name of the file; Returns: -

Class print_custom_tree that inherits from print_tree(class from a library)

Has the following methods:

- get_children – Returns the children of a node. Params: node- the parent node. Return: the children.
- get_node_str – Returns the pretty string that represents a node. Params: node-the node to print. Return: the string representing the node.

UML Diagram for the described application:



Example:

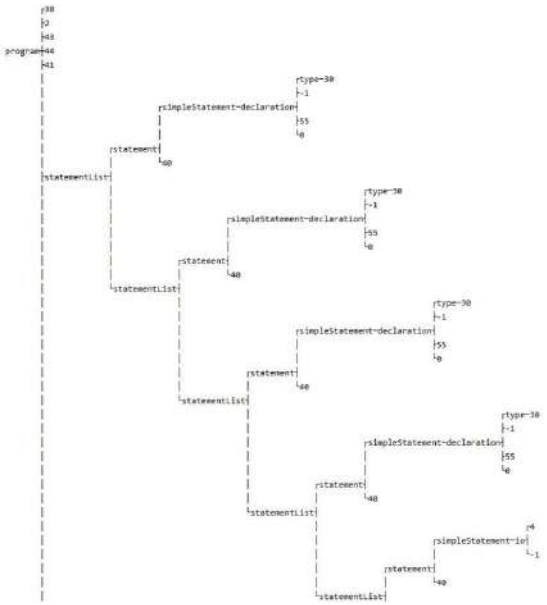
1.Happy flow

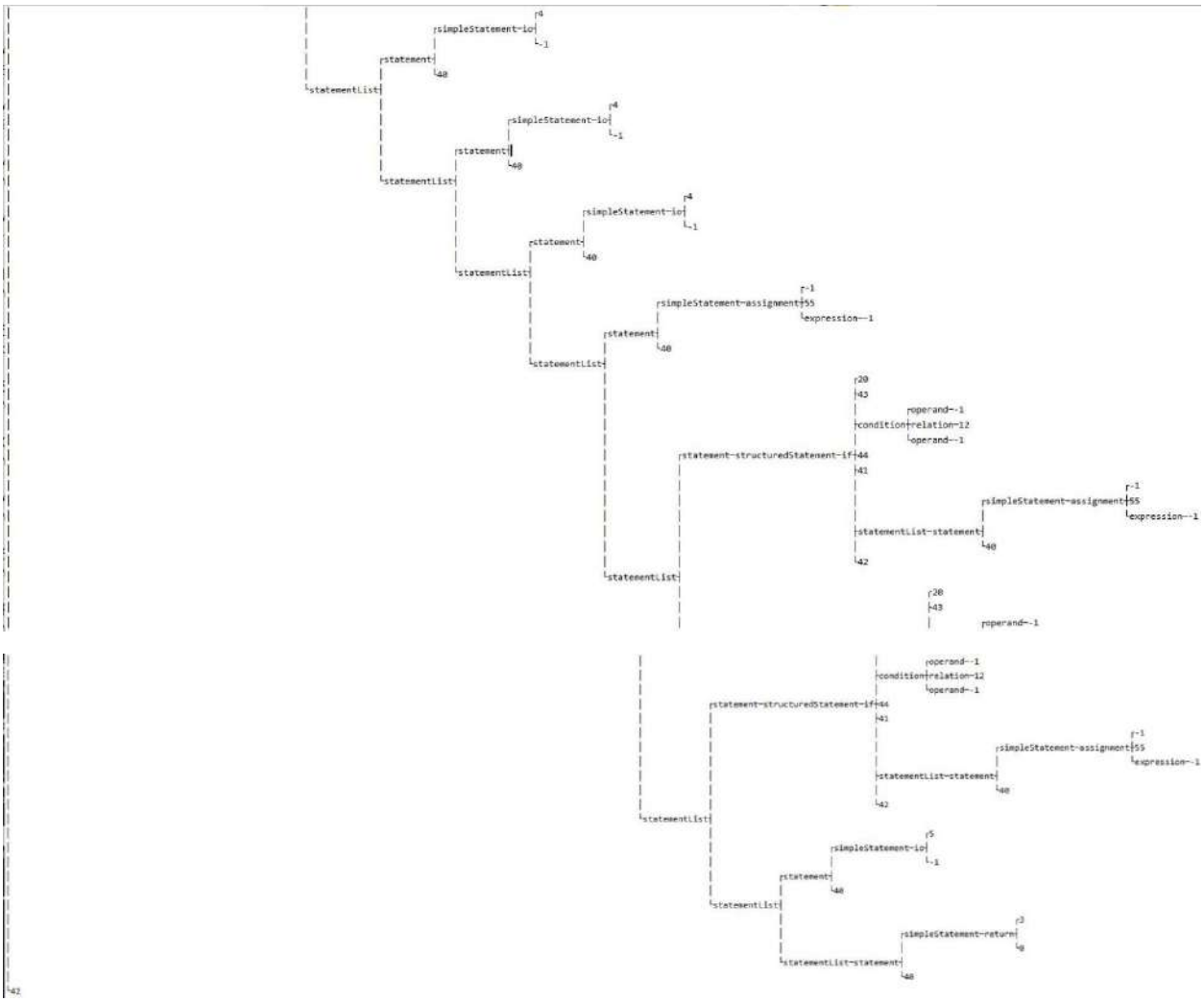
Input:

```
program,type,relation,operand,operator,size,statementList,statement,simpleStatement,structuredStatement,declaration,io,assignment,expression,condition,return,if,for
-1,0,2,3,4,5,6,7,10,11,12,13,14,20,21,22,30,31,40,41,42,43,44,45,46,50,51,52,53,54,55
program

program: 30 2 43 44 41 statementList 42
type: 30 | 31
relation: 10 | 11 | 12 | 13 | 14
operand: -1 | 0 | -1 45 -1 46 | -1 45 0 46
operator: 50 | 51 | 52 | 53 | 54 | 55
size: 0
statementList: statement | statement statementList
statement: simpleStatement 40 | structuredStatement
simpleStatement: declaration | io | assignment | condition | return
declaration: type -1 | type -1 55 0 | type -1 45 size 46 | type -1 45 size 46 55 0
io: 4 -1 | 4 -1 45 -1 46 | 4 -1 45 0 46 | 5 -1 | 5 0 | 5 -1 45 -1 46 | 5 -1 45 0 46 |
assignment: -1 55 expression
expression: operand operator operand | -1 | 0
condition: operand relation operand | expression relation expression | operand relation expression | expression relation operand
return: 3 0
structuredStatement: if | for
if: 20 43 condition 44 41 statementList 42
for: 22 43 assignment 40 condition 40 assignment 44 41 statementList 42
```

Output:





2.Left recursive

Input:

```
program,type,relation,operand,operator,size,statementList,statement,simpleStatement,structuredStatement,declaration,io,assignment,expression,condition,return,if,for
-1,0,2,3,4,5,6,7,10,11,12,13,14,20,21,22,30,31,40,41,42,43,44,45,46,50,51,52,53,54,55
program

program: 30 2 43 44 41 statementList 42
type: 30 | 31
relation: 10 | 11 | 12 | 13 | 14
operand: -1 | 0 | -1 45 -1 46 | -1 45 0 46
operator: 50 | 51 | 52 | 53 | 54 | 55
size: 0
statementList: statement | statementList statement
statement: simpleStatement 40 | structuredStatement
simpleStatement: declaration | io | assignment | condition | return
declaration: type -1 | type -1 55 0 | type -1 45 size 46 | type -1 45 size 46 55 0
io: 4 -1 | 4 -1 45 -1 46 | 4 -1 45 0 46 | 5 -1 | 5 0 | 5 -1 45 -1 46 | 5 -1 45 0 46 |
assignment: -1 55 expression
expression: operand operator operand | -1 | 0
condition: operand relation operand | expression relation expression | operand relation expression | expression relation operand
return: 3 0
structuredStatement: if | for
if: 20 43 condition 44 41 statementList 42
for: 22 43 assignment 40 condition 40 assignment 44 41 statementList 42
```

Output:

```
Grammar is LEFT-RECURSIVE
P1:


empty
```