[Former-languages-and-compiler-design/Lab7 at main · 915-Nistor-Anca/Former-languages-and-compiler-design (github.com)](https://github.com/915-Nistor-Anca/Former-languages-and-compiler-design/tree/main/Lab7)

**Lab 7 – documentation**

*Grammar class:*

New functions:

*productionIsNull(self, production):* Checks if a production is null, meaning that the right-hand side is epsilon.

*getProductionRulesForNonTerminal(self, non\_terminal):* For the given non-terminal, it returns a list of productions in which the left-hand side is the non-terminal.

*findRightChoice(self, terminal, non\_terminal):* It returns the right production to add in the parsing table, on the terminal column and non-terminal line. The right production means the terminal is in the first set of the non-terminal and it’s on the right-hand side of the production. Also, if this case has no solution, a good production can also mean the terminal is in the follow set of the non-terminal and the production is null. Otherwise, a right choice doesn’t exist.

*constructParsingTable(self):* Constructs the parsing table, which is a list of lists. The first row is filled with all the terminals, including epsilon. The first column contains both the non-terminals and the terminals and also the dollar sign. In case of a pair table(terminal,non-terminal), the element is the one given from the findRightChoice function. If the pair is (x,x), where x is a non-terminal, it is completed with ”pop”, and if the pair is ($, epsilon), the space is completed with ”accept”.

*ParserOutput class:*

*\_\_init\_\_(self, grammar):* It initializes a new ParserOutput object having the grammar it works on.

*getAction(self, row, column):* It takes the value of a cell in the table, either a production rule, ”pop” or ”accept” (or ”” if the cell is empty).

*parse(self, input):* It performs the parsing operation on the input string and stores actions that occur during the parsing process. It has an input stack, an output stack and an action list. The input stack is initialized with the input string split into tokens and a '$' marker appended to signify the end of the input. The work stack is initialized with the start symbol of the grammar and '$' to represent the bottom of the stack. The method enters a while loop to iterate through the parsing steps until it reaches an "accept" state or an empty action. At each step, it identifies the symbols at the top of both stacks and it gets the next action to perform based on these symbols using the getAction method. The possible actions might include "accept", "pop", or a production rule to apply. At the end, it creates the parsing tree.

*createTable(self, actions*): ”actions” is a list of lists in which each list is of type [left-hand side of a production, [each symbol from the right-hand side of a production]]. The left-hand side represents the father for everything there is on the right (and those symbols are siblings). In order to construct a list in which each element is of type (id, symbol, father, right sibling), it takes each action and adds, for each symbol in the right hand side, the corresponding pair. The parent is the left-hand side and the id of the sibling will be just its id + 1, because it is the next which comes in line.