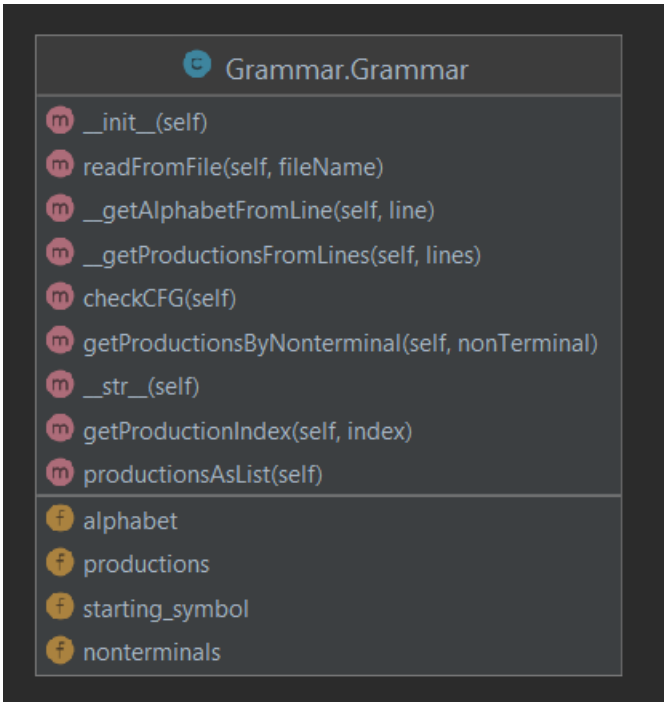
https://github.com/CosminHolcan/FLCD/tree/main/Lab7

Team members :

Gorbatai Cristian

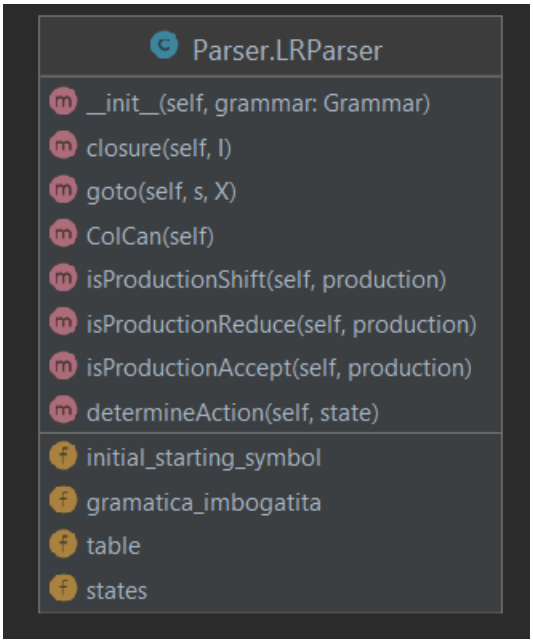
Holcan Cosmin

Grammar Class



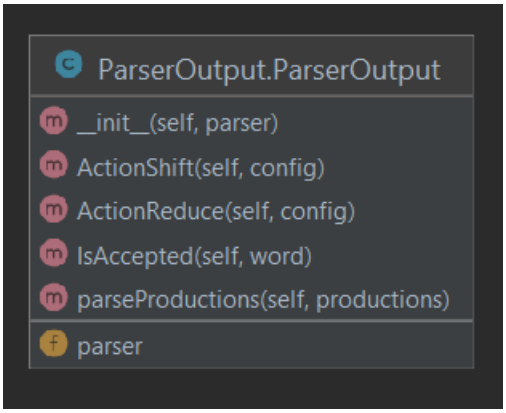
I represented the alphabet as a list of strings, productions as a dictionary, starting_symbol as a simple string and nonterminals as a list of strings. The dictionary of productions has as key a tuple of strings which represents the nonterminal from the left hand side of a production and as a value a list of lists of strings from the right hand side (for example, SA -> abS | Acb, then the key is (S,A) and the value is [[a,b,S], [A,c,b]]). In this way, to check if the grammar is a cfg all I have to do is to check if for every key in dictionary of productions the length of respective key is 1 (otherwise is not a cfg). Also, I checked for every part of the right hand side of the production that all the strings are either symbols of alphabet or nonterminals. I used the methods getProductionIndex and productionsAsList for the next part of the project, more precisely, when we need to establish for each state in ColCan function which is its action, for reduce cases we need to know the number of each production.

Parser Class



In init function, we compute the enriched grammar (changing the starting symbol to “start” and adding a new move, start -> initial starting symbol of the grammar). The closure function computes the closure of an element to be analyzed after the lecture algorithm: we parse each transition from the input and if we find a point in the middle of the right hand side member then we see what is the next symbol after point; if it is a nonterminal we add all productions with that nonterminal in left hand side part which are not already in the element to be analyzed. The next function is GoTo which takes as parameter a state s and a symbol X which can be either a nonterminal or part of alphabet and computes closure({[A-> alpha X . beta] where (A -> alpha . X beta) is from s }) (practically, we “move” the point with one position from left to right if it is before X and we compute the closure of these new transitions). A transition is as represented as a list with two elements : first, is a tuple with only one element, the second is a list of strings (ex : A -> alpha . beta => [(A), [alpha, ., beta]]). The function ColCan is very important because we also determine here the table of states. We start with the initial state s0 which is the closure of start -> . initial_starting_symbol. After that, we have a set C of states, initially has just state s0. Then, we compute for any state ‘s’ of C and any symbol ‘symbol’ (which is nonterminal or part of alphabet) goto(s, symbol) and if we find a new state, we added to C and compute its correspondent action. The algorithm stops when we don’t find new state to add in C. To establish the type of a state’s action we have 3 functions: isProductionShift, isProductionReduce, isProductionAccept which verifies for parameter s (a state) if it’s a shift, a reduce or accept according to rules from the lecture.

ParserOutput class



ParserOutput received as a parameter a parser and verify if a sequence is accepted. According to the rules from lecture, we compute the 3 parts : alpha, beta and phi of the algorithm LR(0) Parsing Algorithm and for the cases in which the last state is an 'action' or a 'shift' we implemented separated functions to make the moves. If the sequence is accepted, a list with the needed productions to be made in order to obtain the corresponding sequence will be returned, otherwise a message will be displayed. Function parseProductions is used to compute a tree of the productions for a more meaningful print of the results and it uses the classes Node and Table.