Parser

Github link:

https://github.com/CimpeanAndreea/FLCD/tree/master/Lab5

Problem Statement:

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

Remark: Lab work evaluation is not done per project/team, but per team member (reflecting the individual contribution to what has been delivered). Please make sure that you split the tasks in a balanced way among team members! In case you decide to do pair programming, each team member is supposed to know all the details of what has been implemented!

PART 2: **Deliverables**

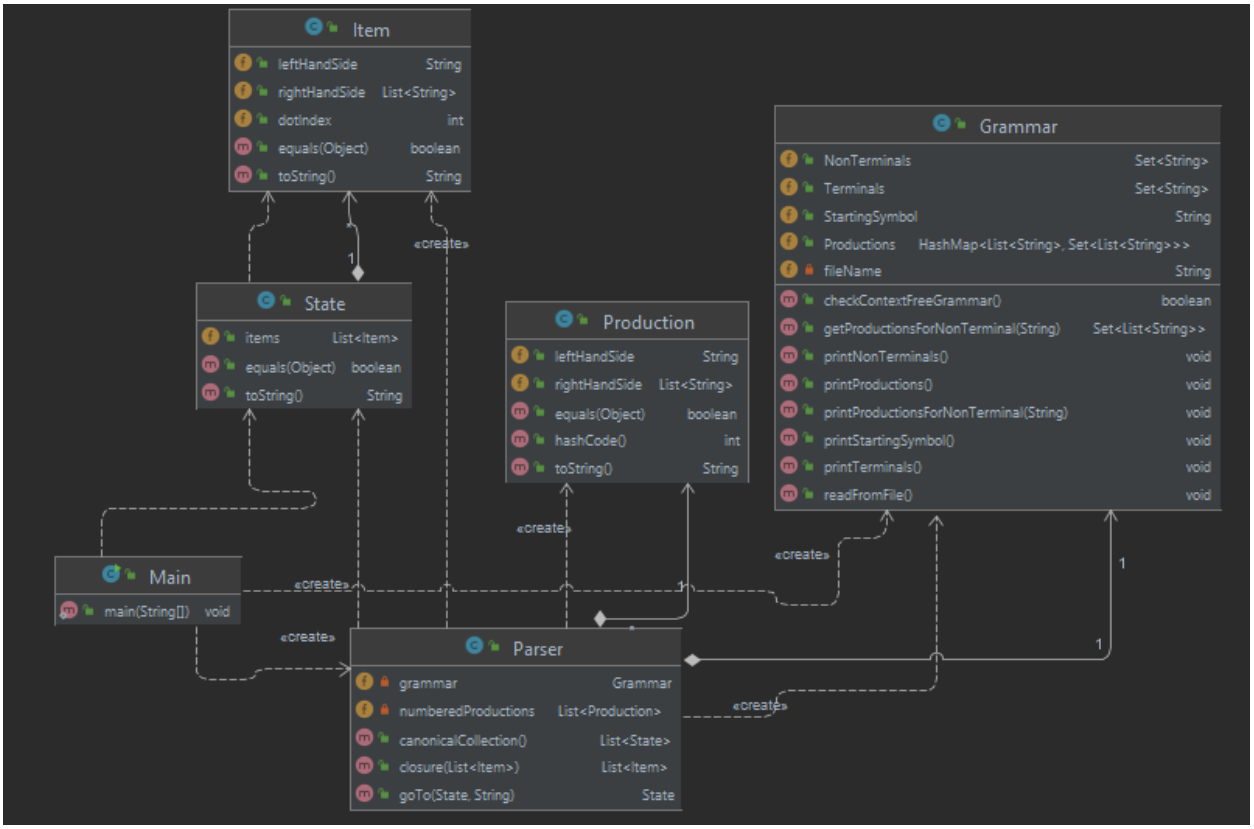
Functions corresponding to the assigned parsing strategy + appropriate tests, as detailed below:

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

LL(1) - functions *FIRST*, *FOLLOW*

LR(0) - functions *Closure*, *GoTo*, *CanonicalCollection*

Implementation:



LR(0) parsing follows the following steps:

1. Define LR(0) item: [A -> alpha . beta]
2. Construct set of states
    a. What does a state contain: algorithm CLOSURE
    b. How to move from a state to another: function GOTO
    c. Construct set of states: CANONICAL COLLECTION
3. Construct parsing table
4. Parse input sequence based on moves between configurations

Continuing previous work, more classes and functions were added to represent the first 2 steps of the LR(0) parsing algorithm. The new classes are:

Item: represents the LR(0) item construct [A -> alpha . beta]

leftHandSide: string (A)

rightHandSide: String[] (alpha . beta – represented as a list of strings)

dotIndex: int (the index of the dot inside the rightHandSide list of strings)

Production: additional way to represent a production of the form (A -> alpha B) to ease inclusion checking of productions in the collections used to construct the set of states

leftHandSide: String

rightHandSide: String[]

State: represent states used by the parser, it only contains a list of Items and function to check equality between states, which is useful in checking inclusion of a state in the canonical collection

items: Item[]

To the **Parser** class the following functions were added:

function closure(Item[] items):

description: takes a list of items and constructs the closure of them (the items of the state that results from given items) following the logic:

let C be the result, initially empty list

for every item in the list

- add it to C
- until C does not modify:
  - get items from C that have dot in front of a non terminal (i.e.[A -> alpha . B beta])
  - for every B -> omega belonging to the set of productions of the augmented grammar
    - if [B -> . omega] does not belong to C, add it

pre: constructed the augmented grammar (i.e. added production S` -> S, where S is the starting symbol)

post: return a list of Items

function goTo(State state, String symbol):

description: go from one state from another -> search in the state the items that contain the symbol and have dot exactly in front of it, after that in every such item move the dot after the symbol and apply closure algorithm on this new list of items to obtain the new State

post: returns a State or null if no item described above is found

function canonicalCollection():

description: get the set of all states of the parsing algorithm

- starts with the item S`->.S and compute its closure => state S0
- add S0 to the collection
- until collection does not modify
    - take all states from the collection and try to obtain new states by applying goTo(state, symbol), where symbol takes one by one the values from the list of terminals and nonterminals of the grammar

pre: constructed the augmented grammar

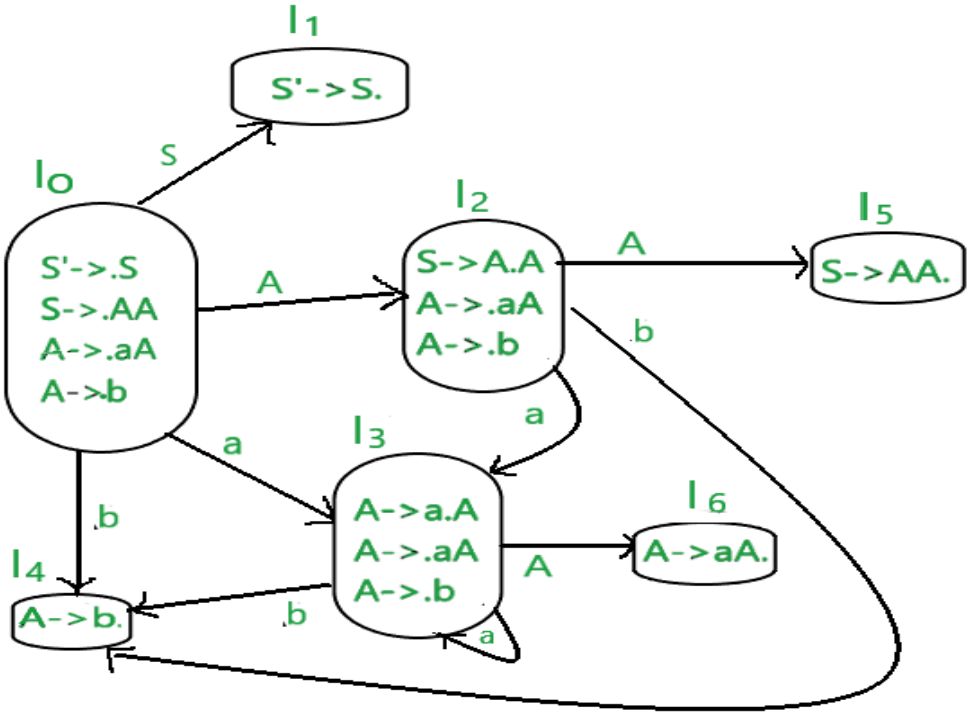
post: returns the set of states

Tests:

To check the algorithm I printed the set of states obtained. The following image shows the expected result for the grammar:

S->AA

A->aA | b



Although the indexing of the states is slightly different they are the same set of states:

State 0

[S_ -> . S]

[S -> . A A]

[A -> . a A]

[A -> . b]

State 1

[S -> A . A]

[A -> . a A]

[A -> . b]

State 2

[S_ -> S .]

State 3

[A -> a . A]

[A -> . a A]

[A -> . b]

State 4

[A -> b .]

State 5

[S -> A A .]

State 6

[A -> a A .]