Link for github repo:

<https://github.com/AndreiFoidas/LFTC>

<https://github.com/AndreiFoidas/LFTC/tree/main/src/main/java>

# Grammar

Is a class containing 4 fields: terminals (List of Strings), nonTerminals (List of Strings), startingSymbol (String) and productions (Hashmap were the key is a List of Strings and the value is a List containing a List of Strings), where each field is equivalent to the theoretical definition.

## Operations

* isCFG(): verifies if the grammar is context free by verifying if the left hand side of the productions (the key) includes a single nonTerminal and that the right hand side of the productions only contain terminals, nonTerminals or epsilon
* isValid(): verifies that the grammar is valid by checking to see if the starting symbol is a nonterminal, the left hand side (the keys) of the productions are all nonTeminals and that the right hand side (the values) of the productions are terminals, nonTerminals or epsilon
* getProductionsForNonTerminal(nonTerminal): it will return a List containing a List of Strings representing all the productions that have that nonterminal in the left hand side (as the key)

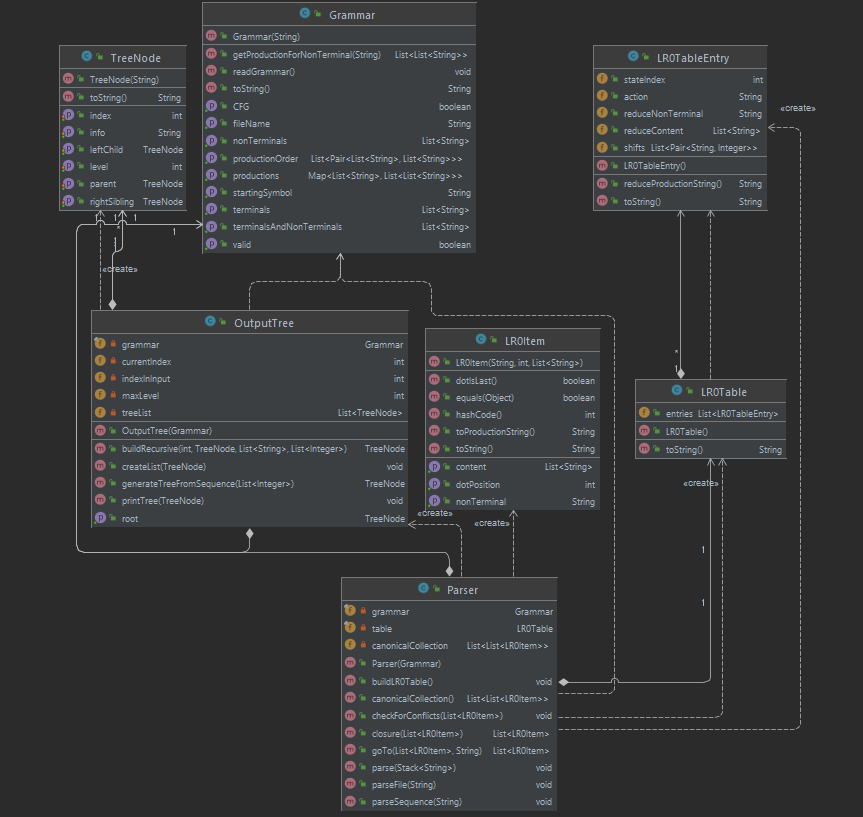
# Parser

Is a class containing 3 fields: grammar (the Grammar class), a table (LR0Table class) and canonicalCollection (a List of List of LR0Items). A LR0Item is a class that represents the Item from the theoretical definition: [A → α.β]. It has the fields: nonTerminal (here is A), dotPosition (position of the dot on the right hand side, here 1) and content (here αβ).

## Operations

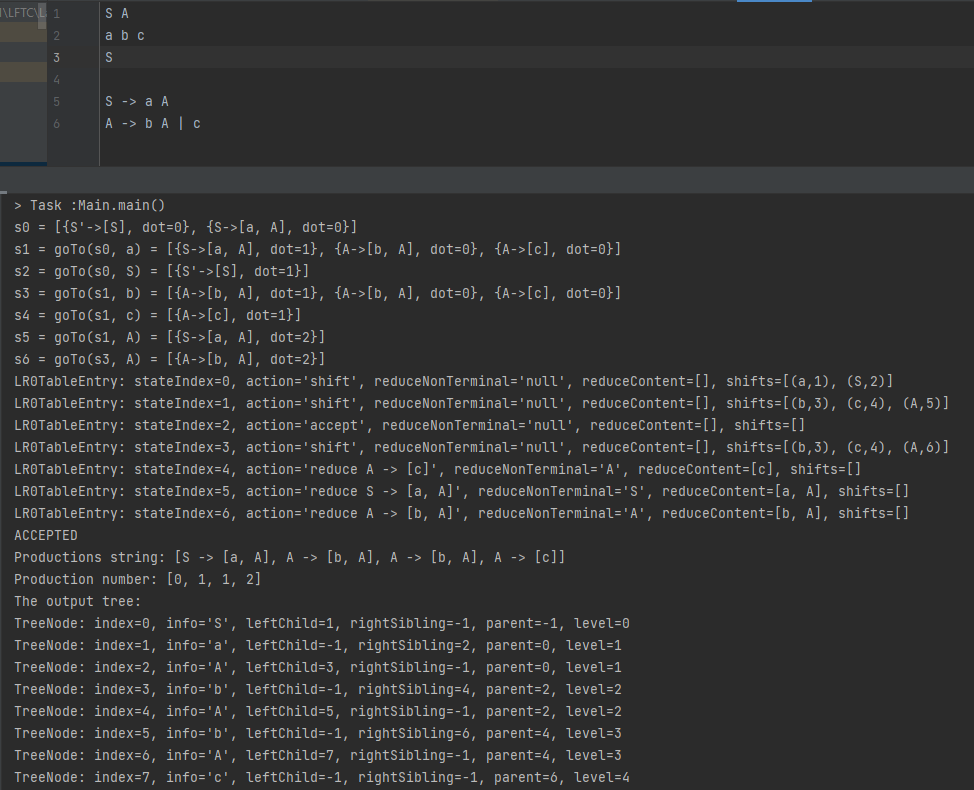
* goto(state, symbol): for every item in the state that has the symbol “symbol” where the dot indicates the position, it returns the closure of that item but with the dot position shifted one position to the right
* closure(items): for every LR0Item in the list of items, if the dot position shows a nonTermial, we search and add all the productions of that nonTerminal to the closure set (we do not add duplicates)
* canonicalCollection(): we will keep adding to LR0Items the canonicalCollection until there are no new elements we can add by the following rule: for every state in the collection, we compute the goTo of that state and every symbol of the Grammar, if the result is not empty or the collection doesn’t have the result already, we add it
* buildLR0Table(): first we build the canonicalCollection, then for every LR0Item from the collection we create a table entry in the following way: if the dot is not in the last position we mark its action as: “shift”, if it is in the last position and the nonTerminal is S’ we mark its action with: “accept”, if the dot is in the last position and the nonterminal is not S’ we mark its action with: “reduce” + the production that has the dot in the last position and we also save the nonTerminal and the content, if none then we mark it as an “error”; we follow up by creating a list of Pairs of goTos (or shifts) for this entry containing the symbol and the index of the state it goes into; also in this step we will check to see if we have reduce-reduce or shift-reduce conflicts in the LR0Table
* parse(inputStack): this function is using a “working stack” where we will save the last symbol that was parsed and the index of the state we are in; we begin with an empty symbol and the index 0. Until we accept the sequence or we find an error we do the following: we get the LR0Table entry corresponding to the state we are in, if its action is a shift, we will find from the list of gotos in that entry the one that has the same symbol as the one on top of the input stack and push on the working stack that symbol and the state the goto goes into; if it’s a reduce, we will eliminate from the working stack the symbols found in production of that LR0TableEntry, after this it will push onto the working stack the symbol found on left hand side of the production and the state index of where this symbol is found in the table in the reduceNonTerminal field, we also push onto the output stack the production we reduced with; if the action is accept the algorithm is over and we will print the output stack which represents the production string but we will also build a parse tree
* the parse tree will be built recursively: each node will save its index, info, parent, left-most child (if it has one), right sibling (if it has one) and the level of tree where it’s found; first we set the root of the tree (the first production of the output), after this we will build in the following way: we advance with the content of the last production and we traverse the content symbol by symbol, if it’s a terminal symbol, we create a new node and we will create it’s right sibling next; if it’s a nonTerminal symbol, we create the node, we get the production of the nonTerminal and create a left child for this current node but we also try to create it’s right child if possible so we don’t miss any nodes

## Class Diagram

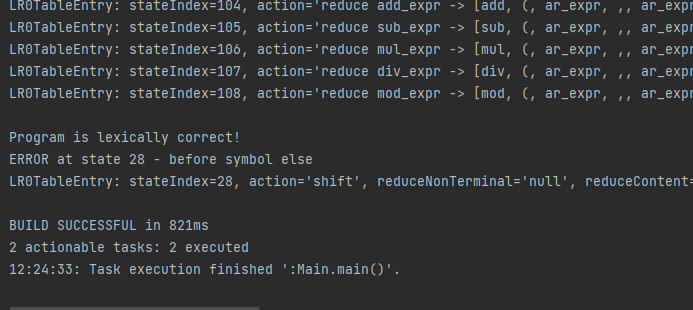
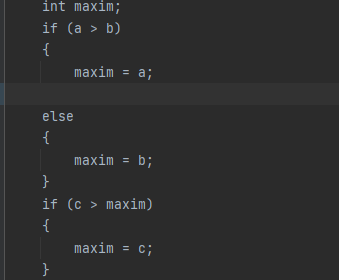


## Tests

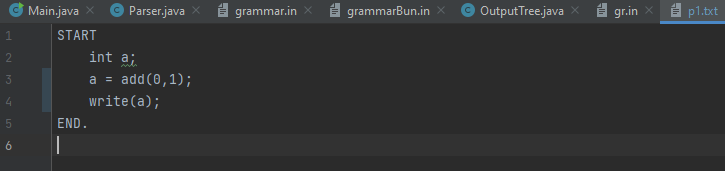
* with smaller grammar and a string sequence to parse

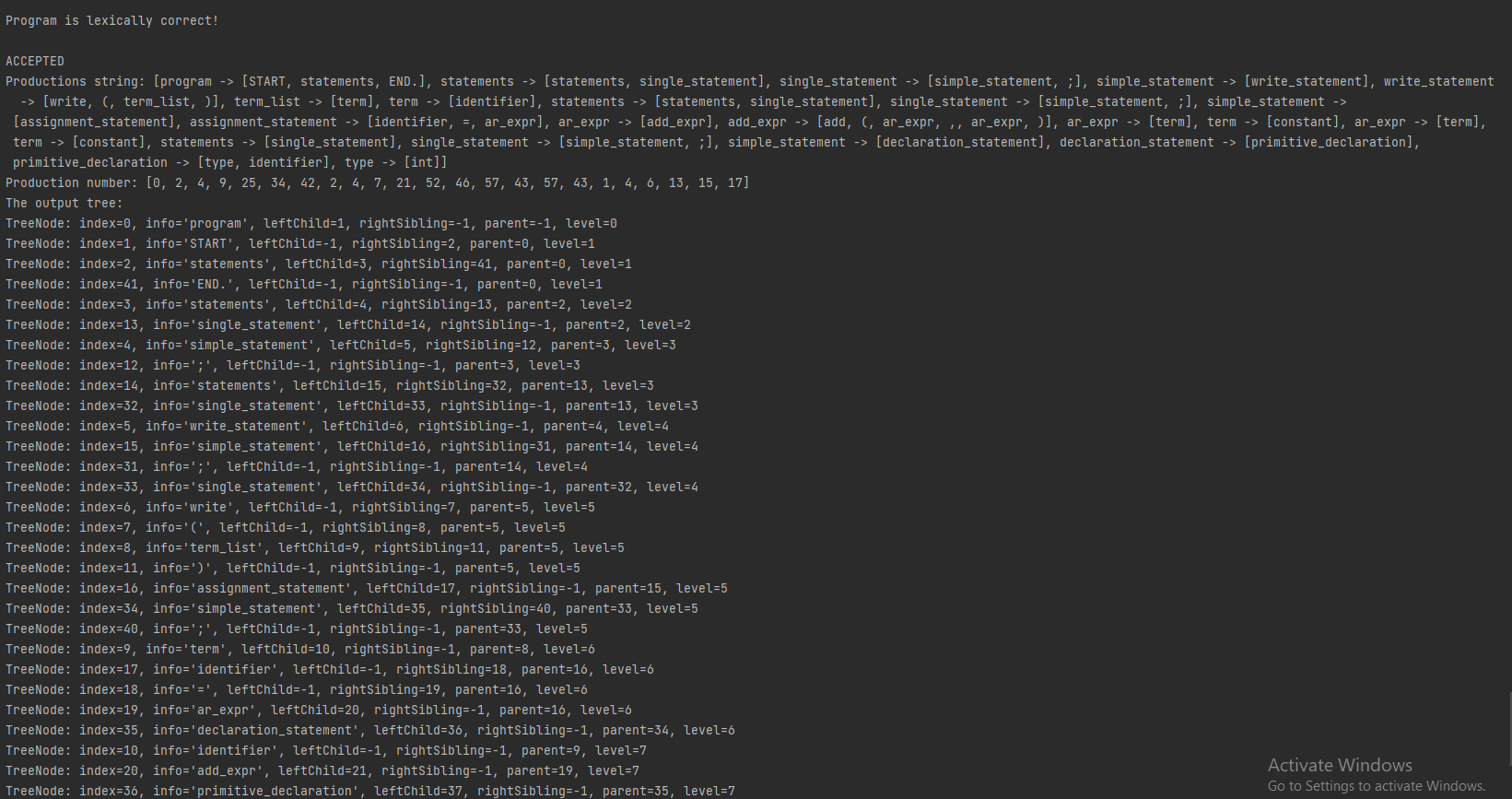


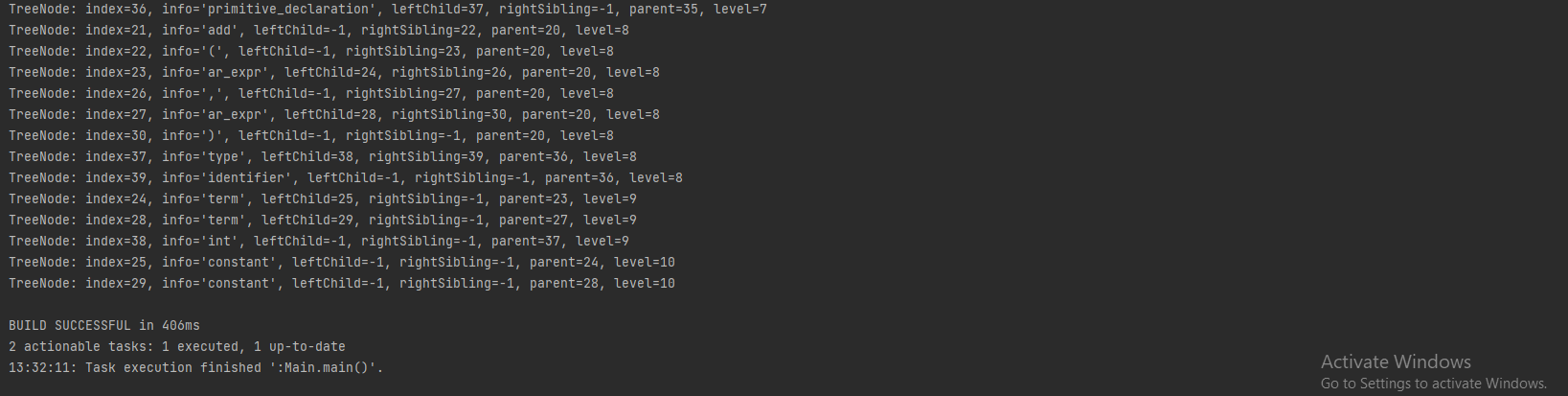
* with a grammar of a coding language and a program to parse which has errors



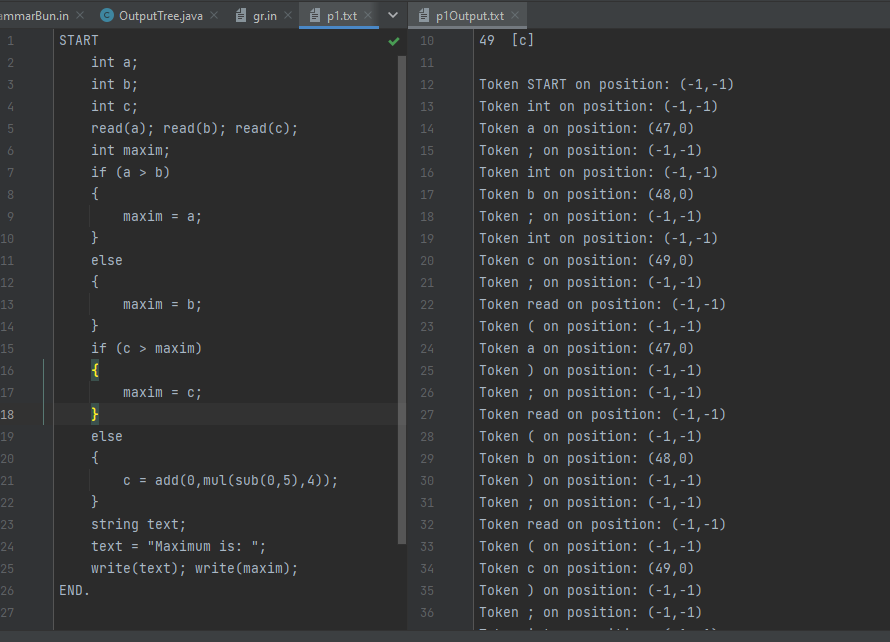
* with a grammar of a coding language and a very simple program to parse

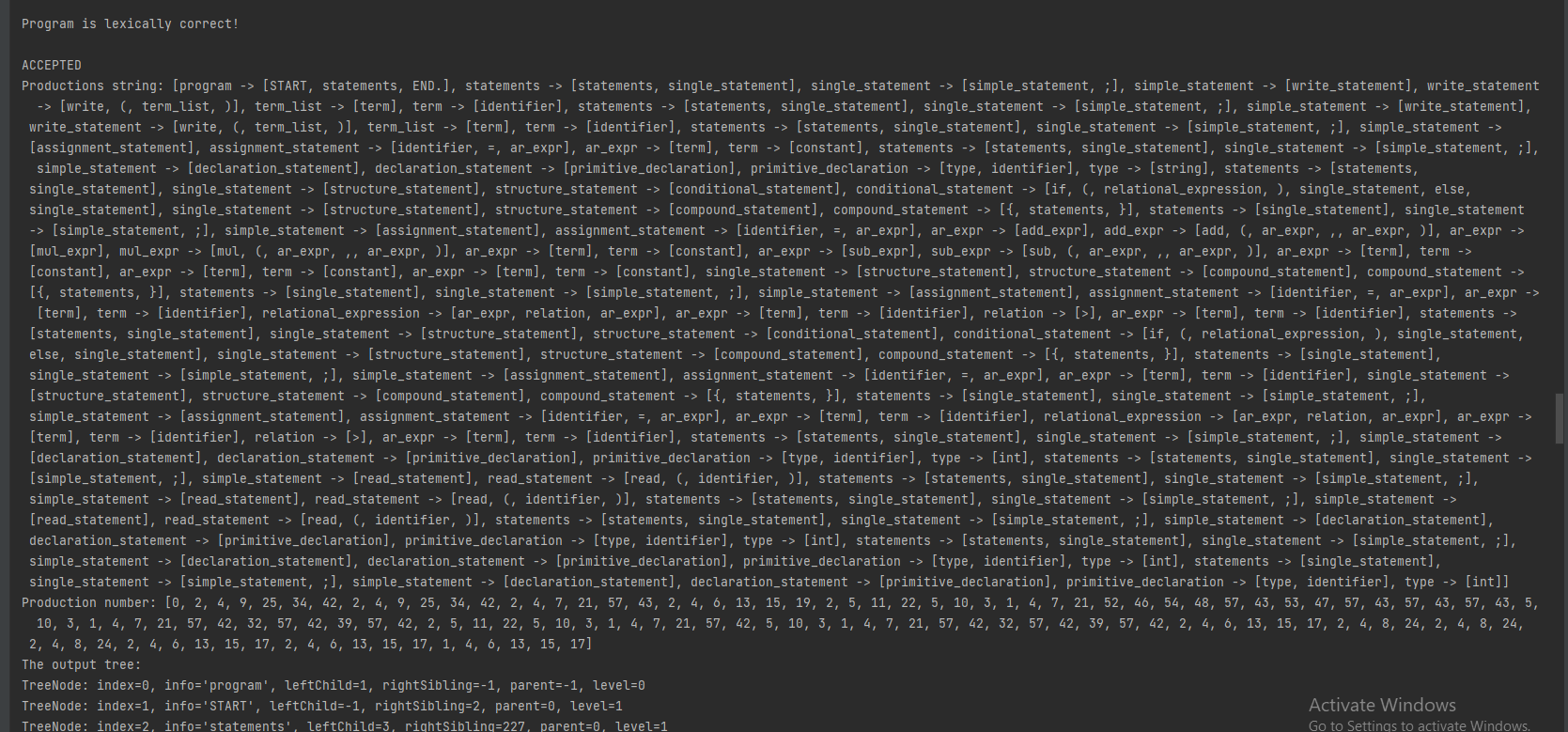






* with a grammar of a coding language and a more complex program to parse





* the grammar of the coding language is

