Laboratory 7 – FLCD

Pop Ruxandra Paula : <https://github.com/ruxipaula/flcd>

Popa Mihai Adrian: <https://github.com/adrian-popa/flcd>

Grammar class

#### Class that contains 4 fields:

* nonTerminals : list of non-terminals
* terminals: list of the terminals
* startingSymbol: starting symbol
* productions: map (string -> list of list of strings)

LRItem class

#### Class that contains 3 fields:

* nonTerminal : string
* production: list of strings
* dotIndex: int – index that shows the position of the dot in the production

Parser class

#### Functions:

* **closure**: takes a list of LRItems as input and returns the closure of that list
* **goTo**: calls the closure function with the LRItems that match the corresponding state and symbol
* **canonicalCollection**: returns the canonical collection of the given grammar

LR0Table class

#### Class that contains 2 fields:

* table : list of pairs containing a string(action) and a map that maps the symbol to the result of the goTo function(state index)
* list: list of strings containing all symbols from the grammar

#### Functions:

* **cannonicalCollectionToLR0Table**: takes the cannonicalCollection and return the lr0table
* **we take each state from the cannonicalCollection and check if only one action(shift, reduce, acc) is appropriate**.
* **then we add the corresponding action to the table, mapping also the goTo symbol to the next state**

TreeNode class

#### Class that contains 6 fields:

* value: value of the current node
* parent: reference to the node’s parent
* leftChild: reference to the node’s leftChild
* rightSibling: reference to the node’s rightSibling
* index: unique index of the node
* representation used for the parser tree

ParserOutput class

#### Class that contains 1 field:

* root : root of the parsing tree

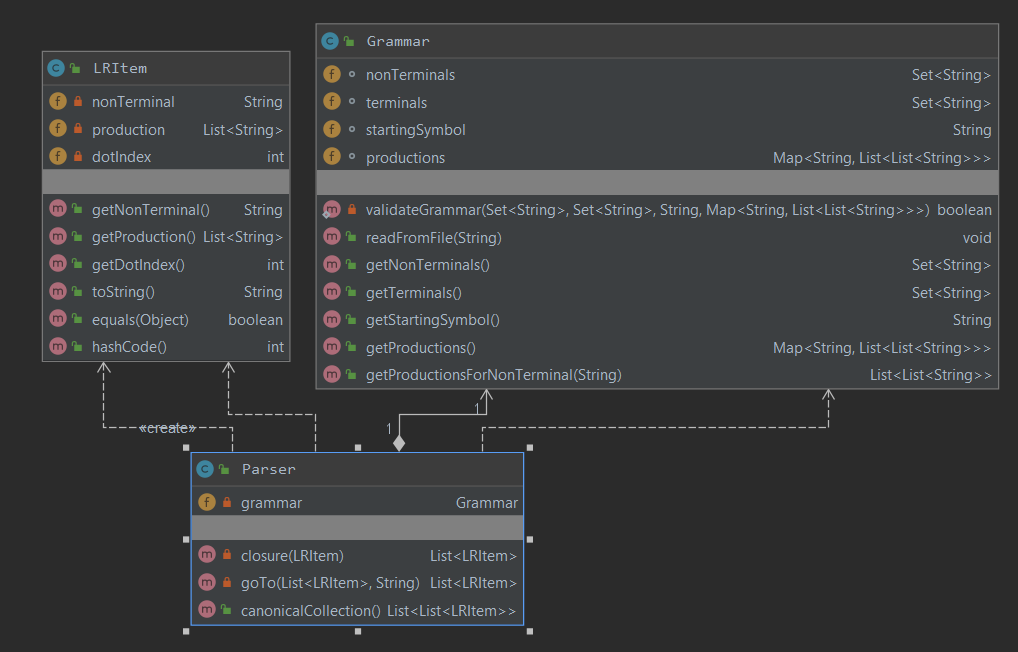
#### Functions:

* **addParsedSequence**: transforms the parsed sequence into a parsing tree
* **we take each non-terminal from the corresponding production and add its transitions to the parsing tree**
* **if the parsing tree is empty then we add the non-terminal as the tree’s root**
* **addChild**: adds a child/sibling to a given parent
* **check if the parent already has a left child, if not we add the current node as a left child to the parent**
* **if the parent already has a left child, we parse all the siblings until we reach the last one and add the node as a right sibling to the last one**
* **returns the added node**
* **addSibling**: adds a sibling to the last sibling of the given node
* **parse all the siblings and add the new node as a right sibling to the last one**
* **if the parent already has a left child, we parse all the siblings until we reach the last one and add the node as a right sibling to the last one**
* **returns the added node**

productions

* We used a hashmap to represent the productions, keeping the non-terminals as the keys and the corresponding production as a list of lists that contain every symbol of the production.

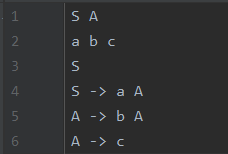
Class diagram



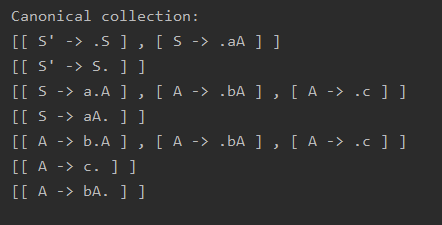
Examples:

1.

* grammar.txt:

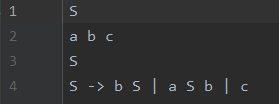


* Canonical Collection:

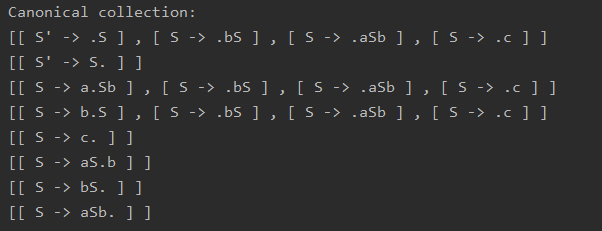


2.

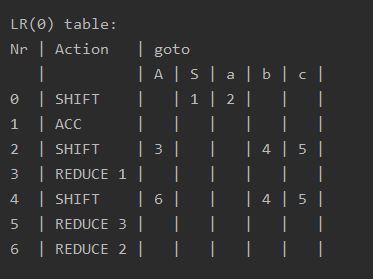
* grammar.txt:



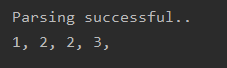
* Canonical Collection:



3. LR0Table



Sequence: a b b c



Parsing tree:

