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**

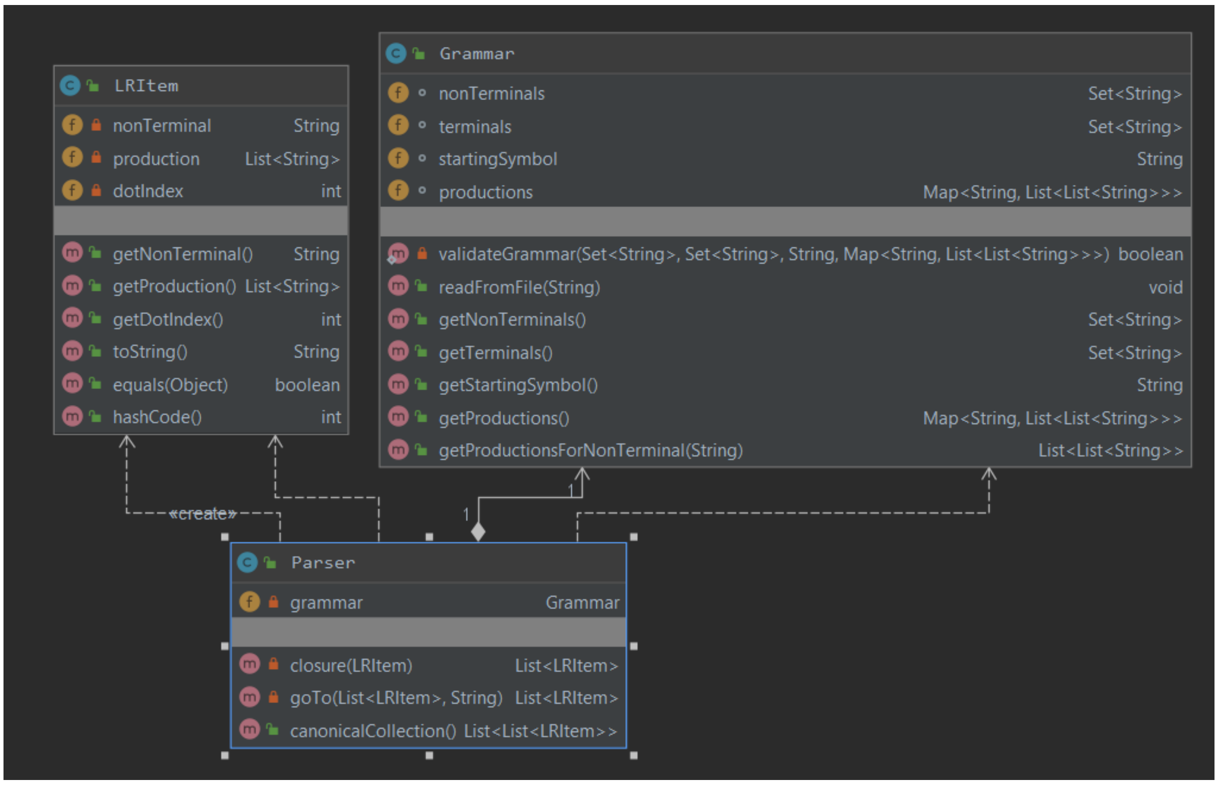
➢ **addChild**: adds a child/sibling to a given parent
- **check if the parent already has a left child, if not we add the current node as a left child to the parent**
- **if the parent already has a left child, we parse all the siblings until we reach the last one and add the node as a right sibling 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.

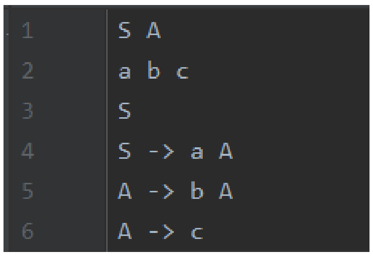
Class diagram


LRItem
nonTerminal String
production List<String>
dotIndex int
getNonTerminal() String
getProduction() List<String>
getDotIndex() int
toString() String
equals(Object) boolean
hashCode() int
Grammar
nonTerminals Set<String>
terminals Set<String>
startingSymbol String
productions Map<String, List<List<String>>>
validateGrammar(Set<String>, Set<String>, String, Map<String, List<List<String>>>) boolean
readFromFile(String) void
getNonTerminals() Set<String>
getTerminals() Set<String>
getStartingSymbol() String
getProductions() Map<String, List<List<String>>>
getProductionsForNonTerminal(String) List<List<String>>
«create»
1
1
Parser
grammar Grammar
closure(LRItem) List<LRItem>
goTo(List<LRItem>, String) List<LRItem>
canonicalCollection() List<List<LRItem>>

Examples:

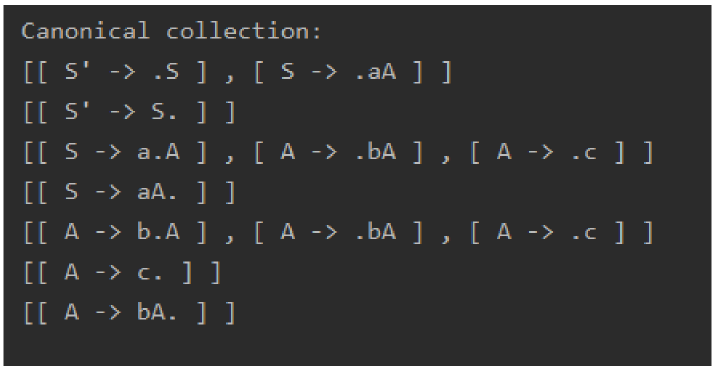
1.

- grammar.txt:



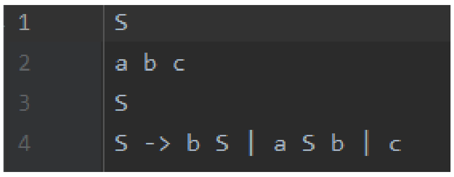
```
S A
a b c
S
S -> a A
A -> b A
A -> c
```

- Canonical Collection:

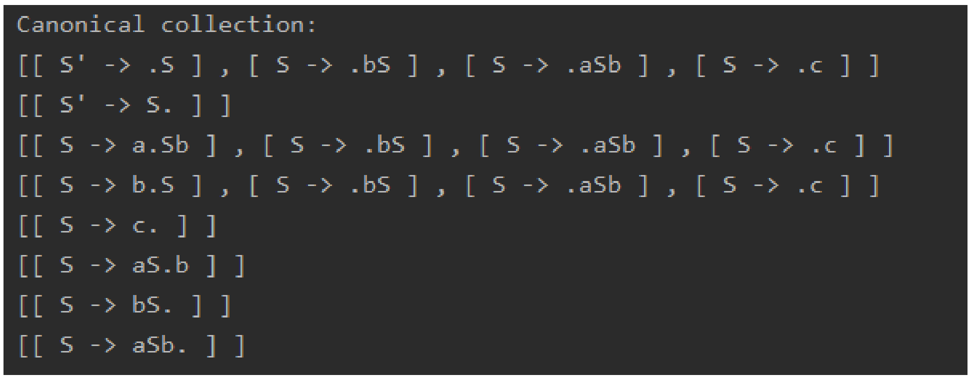

```
Canonical collection:
[[ S' -> .S ] , [ S -> .aA ] ]
[[ S' -> S. ] ]
[[ S -> a.A ] , [ A -> .bA ] , [ A -> .c ] ]
[[ S -> aA. ] ]
[[ A -> b.A ] , [ A -> .bA ] , [ A -> .c ] ]
[[ A -> c. ] ]
[[ A -> bA. ] ]
```

2.

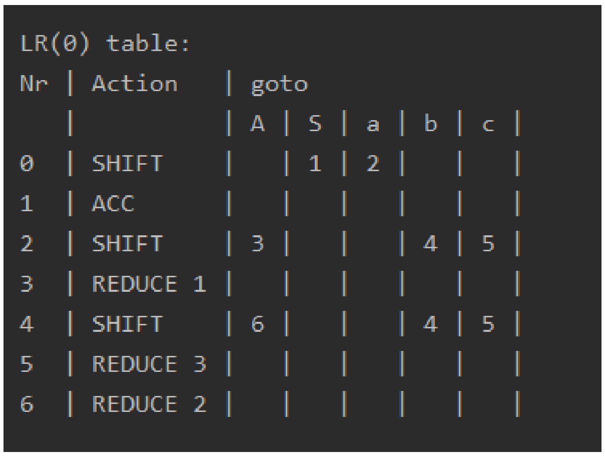
- grammar.txt:


```
S
a b c
S
S -> b S | a S b | c
```

- Canonical Collection:

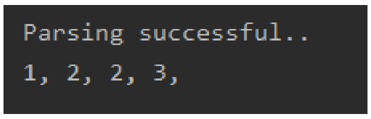

```
Canonical collection:
[[ S' -> .S ] , [ S -> .bS ] , [ S -> .aSb ] , [ S -> .c ] ]
[[ S' -> S. ] ]
[[ S -> a.Sb ] , [ S -> .bS ] , [ S -> .aSb ] , [ S -> .c ] ]
[[ S -> b.S ] , [ S -> .bS ] , [ S -> .aSb ] , [ S -> .c ] ]
[[ S -> c. ] ]
[[ S -> aS.b ] ]
[[ S -> bS. ] ]
[[ S -> aSb. ] ]
```

3. LR0Table



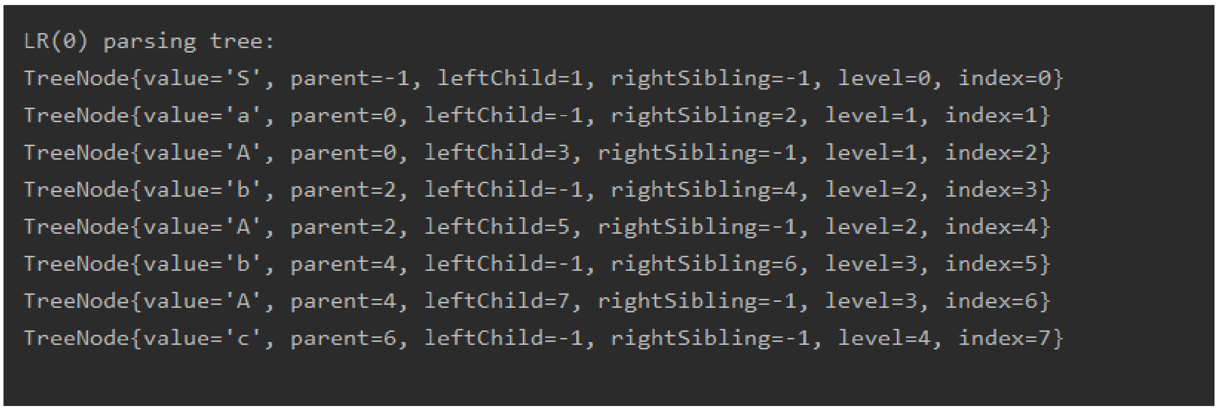
```
LR(0) table:
Nr | Action    | goto
   |           | A | S | a | b | c |
0  | SHIFT     |   | 1 | 2 |   |   |
1  | ACC       |   |   |   |   |   |
2  | SHIFT     | 3 |   |   | 4 | 5 |
3  | REDUCE 1  |   |   |   |   |   |
4  | SHIFT     | 6 |   |   | 4 | 5 |
5  | REDUCE 3  |   |   |   |   |   |
6  | REDUCE 2  |   |   |   |   |   |
```

Sequence: a b b c



```
Parsing successful..
1, 2, 2, 3,
```

Parsing tree:



```
LR(0) parsing tree:
TreeNode{value='S', parent=-1, leftChild=1, rightSibling=-1, level=0, index=0}
TreeNode{value='a', parent=0, leftChild=-1, rightSibling=2, level=1, index=1}
TreeNode{value='A', parent=0, leftChild=3, rightSibling=-1, level=1, index=2}
TreeNode{value='b', parent=2, leftChild=-1, rightSibling=4, level=2, index=3}
TreeNode{value='A', parent=2, leftChild=5, rightSibling=-1, level=2, index=4}
TreeNode{value='b', parent=4, leftChild=-1, rightSibling=6, level=3, index=5}
TreeNode{value='A', parent=4, leftChild=7, rightSibling=-1, level=3, index=6}
TreeNode{value='c', parent=6, leftChild=-1, rightSibling=-1, level=4, index=7}
```