Link to github: https://github.com/CopsiMan/FLCD-1

Requirement :

Statement: Implement a parser algorithm

1. One of the following parsing methods will be chosen (assigned by teaching staff):

    1.c. lr(0)

2. The representation of the parsing tree (output) will be (decided by the team):

    2.c. table (using father and sibling relation) (max grade = 10)

PART 1: Deliverables

Class Grammar (required operations: read a grammar from file, print set of nonterminals, set of terminals, set of productions, productions for a given nonterminal, CFG check)

Input files: *g1.txt* (simple grammar from course/seminar), *g2.txt* (grammar of the minilanguage - syntax rules from Lab 1b)

PART 2: **Deliverables**

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

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

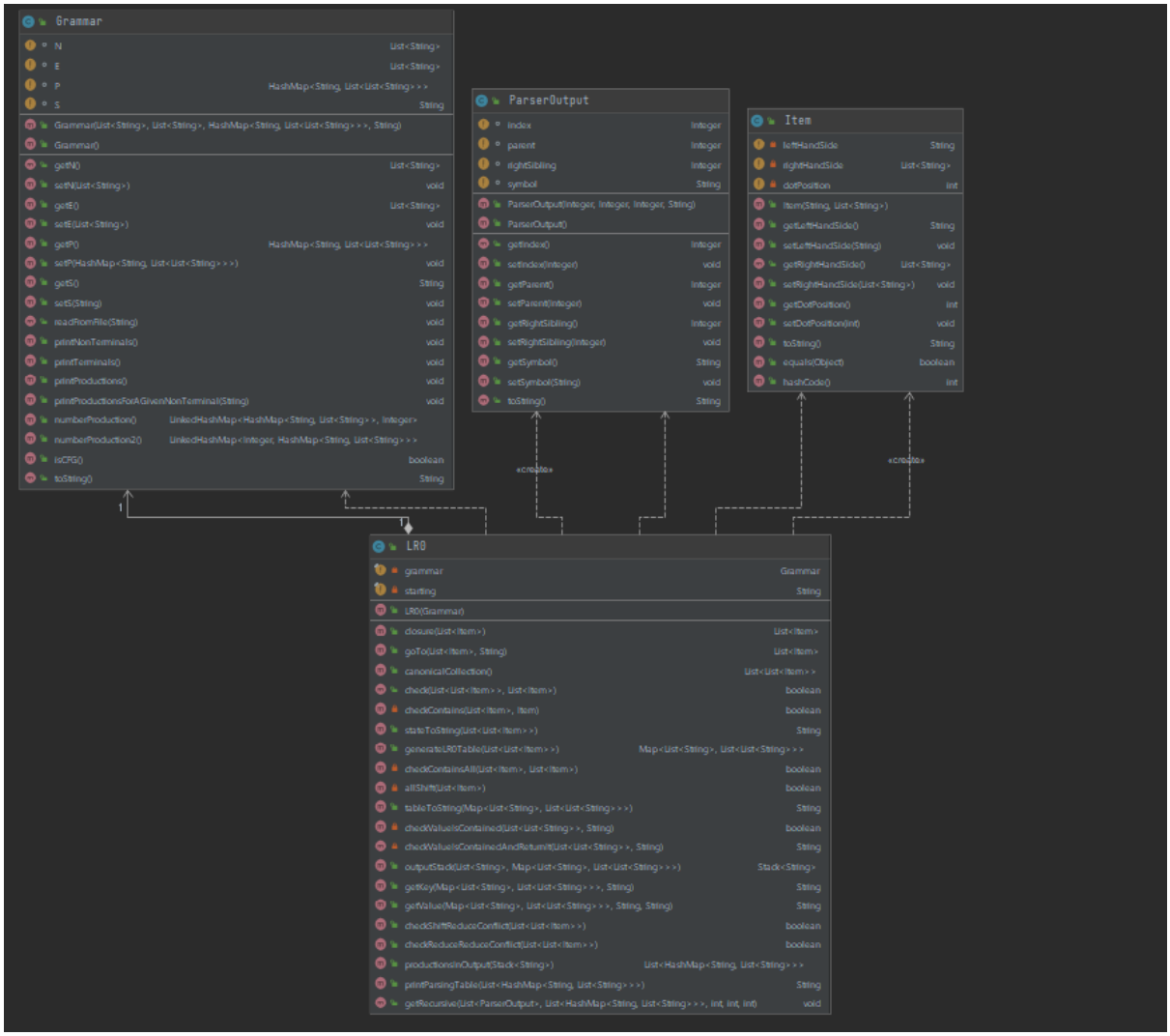
PART 3: Deliverables

1. Algorithms corresponding to *parsing table* (if needed) and *parsing strategy*

2. Class *ParserOutput* - DS and operations corresponding to choice 2.a/2.b/2.c (Lab 5) (required operations: transform parsing tree into representation; print DS to screen and to file)

Remark: If the table contains conflicts, you will be helped to solve them. It is important to print a message containing row (symbol in LL(1), respectively state in LR(0)) and column (symbol) where the conflict appears. For LL(1), values (α,i) might also help.

Class Diagram:



Analysis and design:

Grammar-N->List of strings, the non terminals

-E->List of strings, the terminals

-P->hash map that has as key a string, the non terminal, and a list of list of strings, the right hand side of a production

-S->the initial non terminal

Methods:

-readFromFile(String file)->read the grammar for a file and initializes every element of the grammar with its corresponding value

-printNonTerminals()-> prints all the non terminals

-printTerminals()-> prints all the terminals

-printProductions ()-> prints all the productions

-printProductionForAGivenNonTerminals()-> prints all the productions for a given non terminal

-numberProduction()->associates a number to each production(key->production,value->number)

-numberProduction2()->associates a number to each production(key->number,value->production)

-isCFG()->checks if the grammar is context-free

Item:

-leftHandSide->String

-rightHandSide->List of Strings

-dotPosition->int, the position of the dot in the right hand side

ParserOutput: (we have chosen the tree with parent, right sibling representation)

-index->int, the index of the symbol

-parent->int, the index of the symbol the parent has

-rightSibling->int, the index of the symbol the right sibling has

-symbol->string, the symbol

LR0:

-grammar-> the Grammar used in the parsing

-starting->a String, the starting point

Methods:

-closure(List<Item> items)->performs the closure for a list of items

-goTo(List<Item> state, String element)->performs goTo for a state and an element

-canonicalCollection()->return the canonical collection for the lr0 parsing

-generateLR0Table(List<List<Item>> canonicalCollection)-> generates the table used for parsing, if there are no conflicts in the grammar)

-allShift(List<Item> items)-> checks if all the items in a state are able to perform a shift

-outputStack(List<String> w, Map<List<String>, List<List<String>>> table)->does the actual lr0 parsing, for a word w and using the table obtained, return the iutput state containing the number of the productions and their order

-checkShiftReduceConflict(List<List<Item>> canonicalCollection)->check if there are shift reduce conflicts

-checkReduceReduceConflict(List<List<Item>> canonicalCollection)->check if there are reduce reduce conflicts

-productionsInOutput(Stack<String> outputStack)-> creates a list of the productions used in the parsing, in the order they are presented in the output stack

-printParsingTable(List<HashMap<String,List<String>>> production)->function called for the printing of the output as a tree, calls the first recursive function for this and then prints the final tree at the end

-getRecursive(List<ParserOutput> tree, List<HashMap<String,List<String>>> productions, int parent, int rowIndex, int currentProductionIndex)->recursive function used for building the tree gradually, using the list of productions obtained in productionsInOutput() function

Testing:

For the following grammar(g4.txt):

```
N={S,A}
E={a,b,c}
S={S}
P={S->a A,A->b A|c}
```

I get the following output:

Canonical Collection:

```
state 0 [S'->[S], dot position:0, S->[a, A], dot position:0]
state 1 [S'->[S], dot position:1]
state 2 [S->[a, A], dot position:1, A->[b, A], dot position:0, A->[c], dot position:0]
state 3 [S->[a, A], dot position:2]
state 4 [A->[b, A], dot position:1, A->[b, A], dot position:0, A->[c], dot position:0]
state 5 [A->[c], dot position:1]
state 6 [A->[b, A], dot position:2]
```

The table:

	Action	S	A	a	b	c
0	Shift	1	-	2	-	-
1	Accept	-	-	-	-	-
2	Shift	-	3	-	4	5
3	Reduce 3	-	-	-	-	-
4	Shift	-	6	-	4	5
5	Reduce 2	-	-	-	-	-
6	Reduce 1	-	-	-	-	-

We chose to verify abbc, which can be parsed by the grammar and we got the following output stack:

```
[3, 1, 1, 2]
```

Where the productions were numbered as:

```
{{A=[b, A]}=1, {A=[c]}=2, {S=[a, A]}=3}
```

The representation as a tree of the result is:

```
ParserOutput{index=0, parent=-1, rightSibling=-1, symbol='S'}
ParserOutput{index=1, parent=0, rightSibling=2, symbol='a'}
ParserOutput{index=2, parent=0, rightSibling=-1, symbol='A'}
ParserOutput{index=3, parent=2, rightSibling=4, symbol='b'}
ParserOutput{index=4, parent=2, rightSibling=-1, symbol='A'}
ParserOutput{index=5, parent=4, rightSibling=6, symbol='b'}
ParserOutput{index=6, parent=4, rightSibling=-1, symbol='A'}
ParserOutput{index=7, parent=6, rightSibling=-1, symbol='c'}
```

We also tested a program in the mini language(g2.txt), and eliminated the conflicts by simplifying the language.