Github repo: https://github.com/915-Petruta-Razvan/LFTC_Labs

Documentation

Pair

Details: a generic key value pair class.

Node

Details: class used in the representation of the tree table of the parsing result.
Members:
- Index: int (the index of the node in the table)
- Info: string
- Parent: int (the index of the parent)
- RightSibling: int (the index of the right sibling, 0 if none)

Production

Details: this class is used to represent a production having a left hand side and a right hand side.
Members:
- LeftHandSide: string
- RightHandSide: string

Grammar

Details: class used to represent a grammar read from a file.
Members:
- NonterminalSymbols: List<string> (the list of nonterminals)
- TerminalSymbols: List<string> (the list of terminals)
- Productions: List<Production> (the list of productions)
- StartingSymbol: string
- PathToFile: string (the path to the file in which the grammar information is held)

Methods:
- PrintNonterminalSymbols(): prints the nonterminals of the grammar
- PrintTerminalSymbols(): prints the terminals of the grammar
- PrintAllProductions(): prints the set of productions of the grammar
- CheckCFG(): checks if a grammar is a context free grammar (the left hand side of a production is a nonterminal and the right hand side is a combination of terminal and nonterminal symbols)

- PrintProductionLHSsForANonterminal(nonterminal: string): prints the productions in which the nonterminal appears in the left hand side
- GetProductionLHSsForANonterminal(nonterminal: string): gets the productions in which the nonterminal appears in the left hand side
- PrintProductionRHSsForANonterminal(nonterminal: string): prints the productions in which the nonterminal appears in the right hand side
- GetProductionRHSsForANonterminal(nonterminal: string): gets the productions in which the nonterminal appears in the right hand side
- PrintStartingSymbol(): prints the starting symbol of the grammar
- InitFromFile(): read the grammar from a file

LL1Parser

Details: parser implementation for LL(1)
Members:
- grammar: Grammar (the grammar we are working with)
- _firstDictionary: Dictionary<string, HashSet<string>> (a dictionary in which we will store for each nonterminal the FIRST set)
- _followDictionary: Dictionary<string, HashSet<string>> (a dictionary in which we will store for each nonterminal the FOLLOW set)
- _parseTable: Dictionary<Pair<string, string>, Pair<string, int>> (a dictionary used to create the parsing table, for each pair (row, column) we will have an associated pair (action, index), where action can be a move, pop, acc or err)

Methods:
- PerformConcatenationOfSizeOne(nonterminals: List<string>, terminal: string):
  - If a nonterminal can derive ε, the first symbol of a string derived from the sequence could come from the FIRST set of the next nonterminal.
  - The process continues until a nonterminal that cannot derive ε is encountered, or all nonterminals have been checked.
  - if all nonterminals can derive ε, and there's a terminal, the terminal is also included in the resulting set (as the entire nonterminal sequence can derive ε, leaving the terminal as the first symbol).
- GenerateFirstDictionary():
  - Initial Pass: The method first adds terminals or ε that are directly derivable from each nonterminal.
  - Iterative Refinement: The method then iteratively refines these sets. This is necessary because the FIRST set of a nonterminal may depend on the FIRST sets of other nonterminals. For example, if a nonterminal A has a production A → B C, then the FIRST set of B (and possibly C, if B can derive ε) contributes to the FIRST set of A.

    - The loop continues until no more changes occur in the FIRST sets
- GenerateFollowDictionary():
    - Initial Setup: The method starts by adding ε to the FOLLOW set of the starting symbol, as it's the first symbol in the derivation process.
    - Handling Productions: The method examines each production where a nonterminal appears on the RHS. Depending on the position and the symbol following the nonterminal, different rules are applied to update the FOLLOW set.
    - Terminal and Nonterminal Handling: If a terminal follows the nonterminal, it's added directly to the FOLLOW set. If another nonterminal follows, the method adds all terminals from its FIRST set (except ε) to the FOLLOW set. If ε is in the FIRST set of this following nonterminal, the FOLLOW set of the LHS nonterminal is also included.
    - Iterative Refinement: The method iteratively updates the FOLLOW sets. Since the FOLLOW set of one nonterminal can depend on others, multiple iterations are needed until no further changes occur.
- GenerateParseTable():
    - Initial Setup: The rows are the set of nonterminals and terminals + "$". The columns are the set of terminals + "$". ($, $) will be filled with acc. For every pair (a, a), where a is a terminal, the cell will be filled with pop. The others cells will be filled with the default value: err.
    - We loop through every production and we get the FIRST set of the right hand side. For every symbol from this set, the cell (LHS, symbol) will be completed with (RHS, i). We also check for the conflicts (if we already added a value to the parse table).
- PrintParseTable():
    - Function used to print the parse table in a table like manner
- ParseSequence(sequence: List<string>):
    - Using the table computed above, we then want to parse a sequence, given in a list. For this we use 2 stacks: alpha (input stack) and beta (working stack) and the final result which will be stored in pi. We push in alpha the sequence and in beta the starting symbol. Then while we did not get to acc, we check the pair formed by the top of the stacks for the specific value and we treat them accordingly.

ParserOutput

Details: class used to build the tree table of the parsing result.
Members:

- _parser: LL1Parser (the LL(1) parser instance)
- _head: Node (the head of the tree)
- _nodeList: List<Node> (the list of nodes in the tree table representation)
- _parsingResult: List<int> (the result of parsing)

Methods:

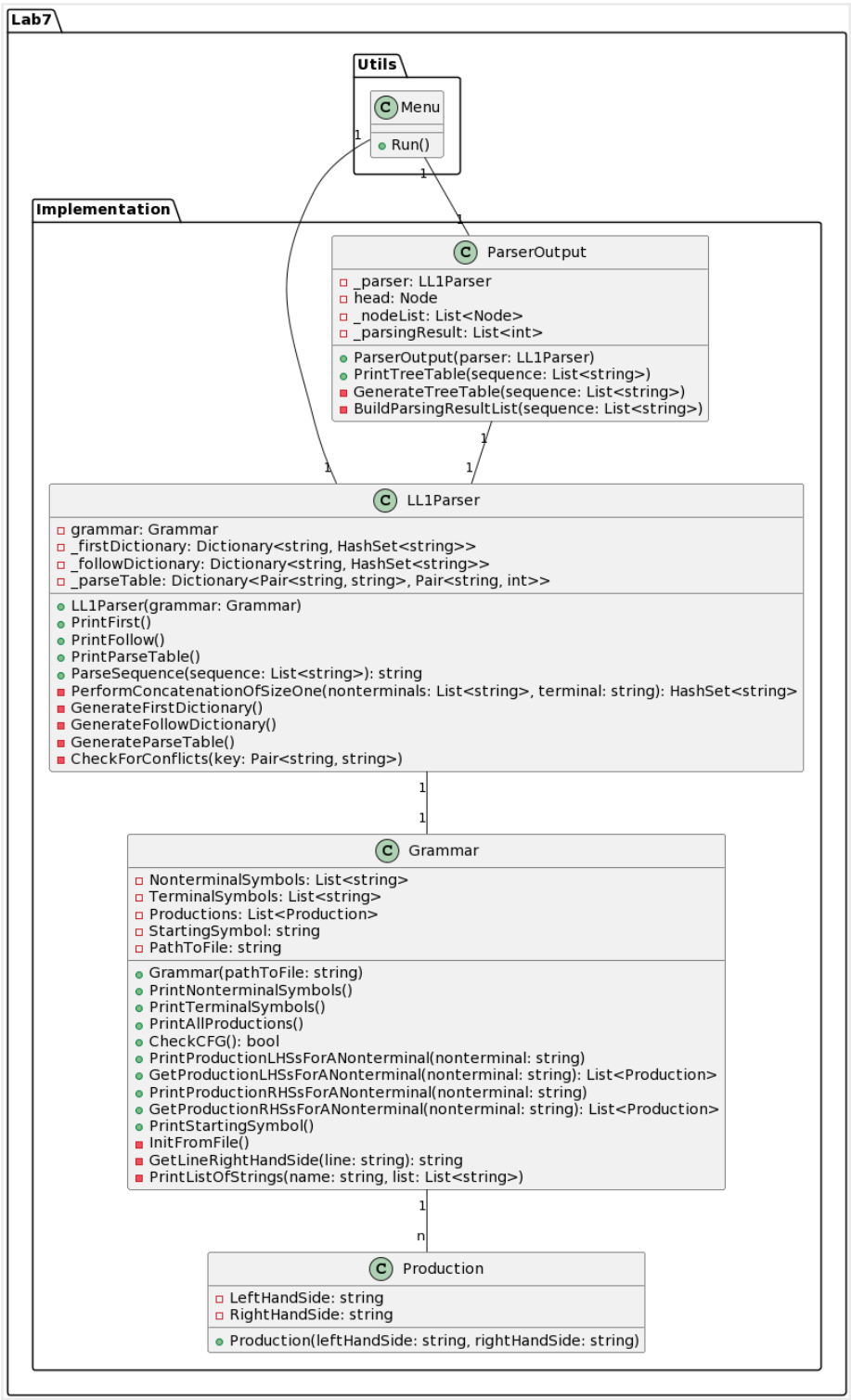
- GenerateTreeTable(sequence: List<string>):
  - Generates the tree table from the given sequence. It builds the parse tree and populates the node list. We go through the productions given by the index in the _parsingResult using a stack. If it is a terminal or epsilon, we just pop it from the stack, otherwise we go through the children and build the necessary associations.
- PrintTreeTable(sequence: List<string>):
  - Generates the tree table for a given sequence using the parser, then after getting the parse tree table, it prints it to the console and in the “TreeTableOutput.txt” file.

Menu

Details: this class is used to create a menu for the user to interact with.
Possible options:

- EXIT
- Show nonterminals
- Show terminals
- Show all productions
- Show productions for a given nonterminal (LHS)
- Show productions for a given nonterminal (RHS)
- Show starting symbol
- Is the grammar a context free grammar?
- Show FIRST
- Show FOLLOW

Class Diagram



Testing the algorithm:

Grammar from seminar 8:

Ex.: Given the CFG $G = (\{S, A, B, C, D\},\ \{+, *, a, (,)\},\ P,\ S)$,

P :

(1) $S \rightarrow BA$

(2) $A \rightarrow +BA$

(3) $A \rightarrow \varepsilon$

(4) $B \rightarrow DC$

(5) $C \rightarrow *DC$

(6) $C \rightarrow \varepsilon$

(7) $D \rightarrow (S)$

(8) $D \rightarrow a$,

FIRST(S) = { (, a }
FIRST(A) ={+, ε}
FIRST(B) ={ (, a }
FIRST(C) ={*, ε}
FIRST(D) ={ (, a }

FOLLOW(S) = {ε,) }
FOLLOW(A) = {ε,) }
FOLLOW(B) = {+, ε,) }
FOLLOW(C) = {+, ε,) }
FOLLOW(D) = {*,+, ε,) }

	a	+	*	(	)	$
S	BA, 1			BA, 1		
A		+BA,2			ε ,3	ε ,3
B	DC,4			DC,4		
C		ε ,6	*DC,5		ε ,6	ε ,6
D	a,8			(S),7		
a	pop					
+		pop				

*			pop			
(				pop		
)					pop	
$						acc

$(a*(a+a)\$, S\$, \varepsilon) \vdash$
$(a*(a+a)\$, BA\$, 1) \vdash (a*(a+a)\$, DCA\$, 14) \vdash$
$(a*(a+a)\$, aCA\$, 148) \vdash (*(a+a)\$, CA\$, 148) \vdash (*(a+a)\$, *DCA\$, 1485) \vdash$
$((a+a)\$, DCA\$, 1485) \vdash ((a+a)\$, (S)CA\$, 14857) \vdash (a+a)\$, S)CA\$, 14857) \vdash$
$(a+a)\$, BA)CA\$, 148571) \vdash (a+a)\$, DCA)CA\$, 1485714) \vdash$
$(a+a)\$, aCA)CA\$, 14857148) \vdash (+a)\$, CA)CA\$, 14857148) \vdash$
$(+a)\$, A)CA\$, 148571486) \vdash (+a)\$, +BA)CA\$, 1485714862) \vdash$
$(a)\$, BA)CA\$, 1485714862) \vdash (a)\$, DCA)CA\$, 14857148624) \vdash$
$(a)\$, aCA)CA\$, 148571486248) \vdash ()\$, CA)CA\$, 148571486248) \vdash$
$()\$, A)CA\$, 1485714862486) \vdash ()\$,)CA\$, 14857148624863) \vdash$
$(\$, CA\$, 14857148624863) \vdash (\$, A\$, 148571486248636) \vdash$
$(\$, \$, 1485714862486363)$

Program execution:

```
0: EXIT
1: Show nonterminal symbols
2: Show terminal symbols
3: Show set of productions
4: Show productions for a given nonterminal (LHS)
5: Show productions for a given nonterminal (RHS)
6: Show starting symbol
7: Is the grammar a context free grammar?
8. Show FIRST
9. Show FOLLOW
> 1
N = {S,A,B,C,D}
> 2
E = {+,*,a,(,)}
> 3
P = {
        S -> B A
        A -> + B A
        A -> ε
        B -> D C
        C -> * D C
        C -> ε
        D -> ( S )
        D -> a
}
> 6
S = S
```

```
> 8
S: (, a
A: +, ε
B: (, a
C: *, ε
D: (, a
> 9
S: ε, )
A: ε, )
B: +, ε, )
C: +, ε, )
D: *, ε, +, )
```

Key	+	*	a	(	)	$
S			B A, 1	B A, 1		
A	+ B A, 2				ε, 3	ε, 3
B			D C, 4	D C, 4		
C	ε, 6	* D C, 5			ε, 6	ε, 6
D			a, 8	(S), 7		
+	pop, -1					
*		pop, -1				
a			pop, -1			
(				pop, -1		
)					pop, -1	
$						acc, -1

```
sequence (space separated symbols): a * ( a + a )
1485714862486363
```

Index	Parent	Info	Right sibling
1	0	S	0
2	1	B	3
3	1	A	0
4	2	D	5
5	2	C	0
6	4	a	0
7	4	*	8
8	4	D	9
9	4	C	0
10	8	(	11
11	8	S	12
12	8	)	0
13	11	B	14
14	11	A	0
15	13	D	16
16	13	C	0
17	15	a	0
18	15	ε	0
19	15	+	20
20	15	B	21
21	15	A	0
22	20	D	23
23	20	C	0
24	22	a	0
25	22	ε	0
26	22	ε	0
27	22	ε	0