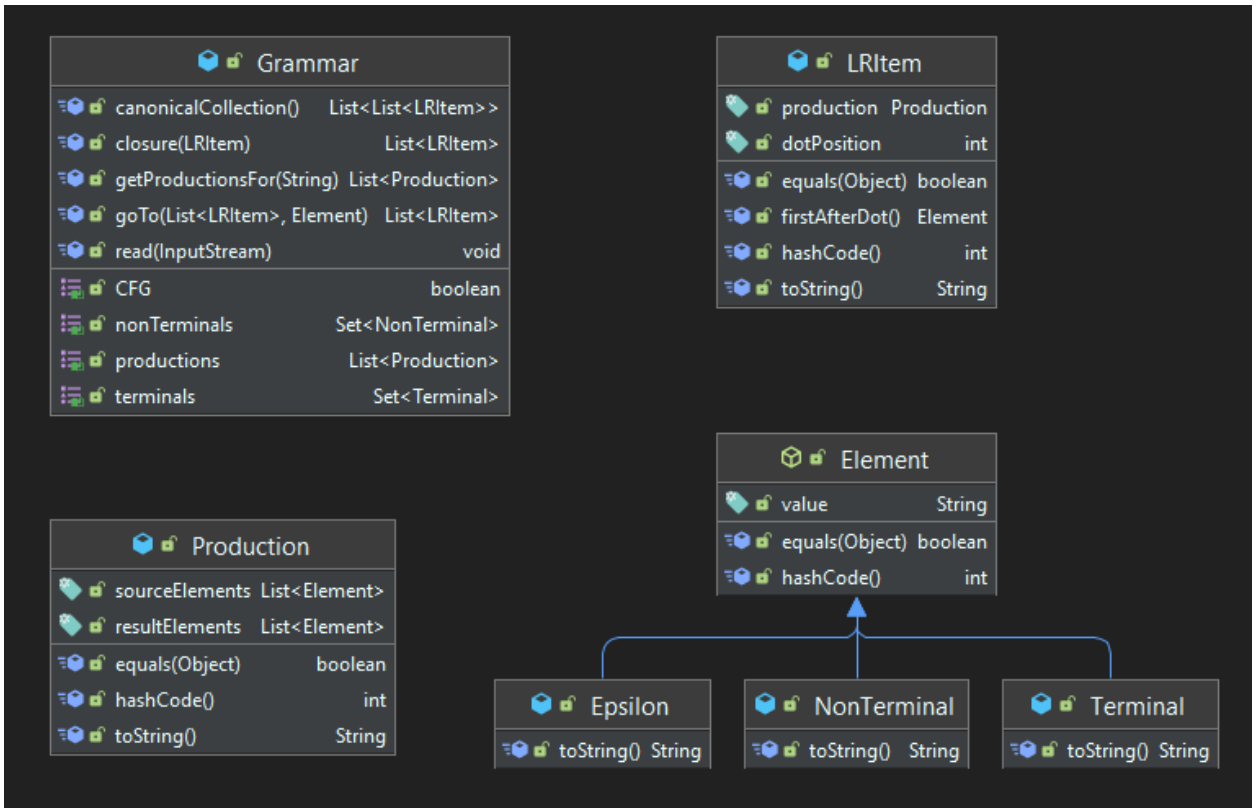
Lab 7

Requirements

- Algorithms corresponding to parsing table (if needed) and parsing strategy
- 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)

-

Architecture



Grammar class

void read(InputStream input)

- Initializes a grammar instance based on an input stream.
- The input stream should contain a list of productions separated by newline.
- Throws:
    - IOException - Production source is empty
    - IOException - Production source only contains terminals
    - IOException - Production result is empty

boolean isCFG()

- Checks if each production has exactly one non-terminal symbol as source
- Returns:
    - True - the grammar is CFG
    - False - the grammar is not CFG

List<LRItem> closure(LRItem item)

- Generates the closure of an input production
- Recursively calls itself on each production in order to generate the complete collection (ignores duplicates)

List<LRItem> goto(List<LRItem> state, Element element)

- Computes the state generated by the goto LR function.
- Takes the state productions where the symbol after the dot is equal to the input element, shifts the dot one position to the right, and computes the closure of each resulting production
- Parameters:
    - state - the goto input state for the goto LR function
    - element - the input symbol for the goto LR function

List<List<LRItem>> canonicalCollection()

- Computes the canonical collection of the grammar
- The resulting collection is represented by a list of states
- Generates a starting production (S' -> .S) and recursively calls goto on each valid combination of states and symbols unti all states are generated

Production class

Contains a list of source symbols (Element) and a list of result symbols

LRItem class

Encapsulates a Production object and also holds the dot position.

Design Details

- A state is represented by a collection of LRItems (where LRItems are productions with dots).
- Classes Element, Terminal, NonTerminal, which just encapsulate strings were used in order to make it easy to determine if a symbol is a terminal or a non terminal, in conjunction with using double quotes (“) in input files to denote terminal symbols;
- A Grammar contains a set of terminals, another one for nonterminals, and a list of productions; the first symbol of the first production (as entered in the input file) is considered to be the starting symbol;
- There is a reserved character (“$”) for denoting ε in the input file;
- “::=” is used as a separator between a production’s left side and right side, and a space is used to separate symbols

Implementation

Github url: https://github.com/darius-calugar/flcd-parser-1/tree/Lab7

Tests

Tests were ran on the following inputs:

Test 1

```
E ::= E "+" T | T
T ::= T "*" F | F
F "a" ::= "(" E ")" | "a"
```

Test 2

```
program ::= cmpdStmt
type ::= "integer"|"string"|userType|arrayType
arrayType ::= type "list"
userType ::= "identifier"
var ::= "identifier" | "identifier" "of" var | var "at" "const"
expression ::= numExpression|"const"
numExpression ::= expression operator expression|var|"const"
condition ::= expression relation expression
operator ::= "plus"|"minus"|"times"|"divided"|"mod"
relation ::= "smaller"|"larger"|"is"
def ::= "define" userType
declStmt ::= type "identifier"
declListStmt ::= arrayType "identifier"
stmt ::= cmpdStmt|assignStmt|ioStmt|ifStmt|declStmt|declListStmt|stopStmt
stmts ::= $ | stmt stmts
cmpdStmt ::= "begin" stmts "end"
assignStmt ::= var "becomes" expression
ioStmt ::= readStmt|printStmt
readStmt ::= "read" var
printStmt ::= "print" expression
ifStmt ::= "if" condition "then" stmt | "if" condition "then" stmt "else" stmt
loopStmt ::= "each" var "from" numExpression "to" numExpression stmt
stopStmt ::= "stop"
```