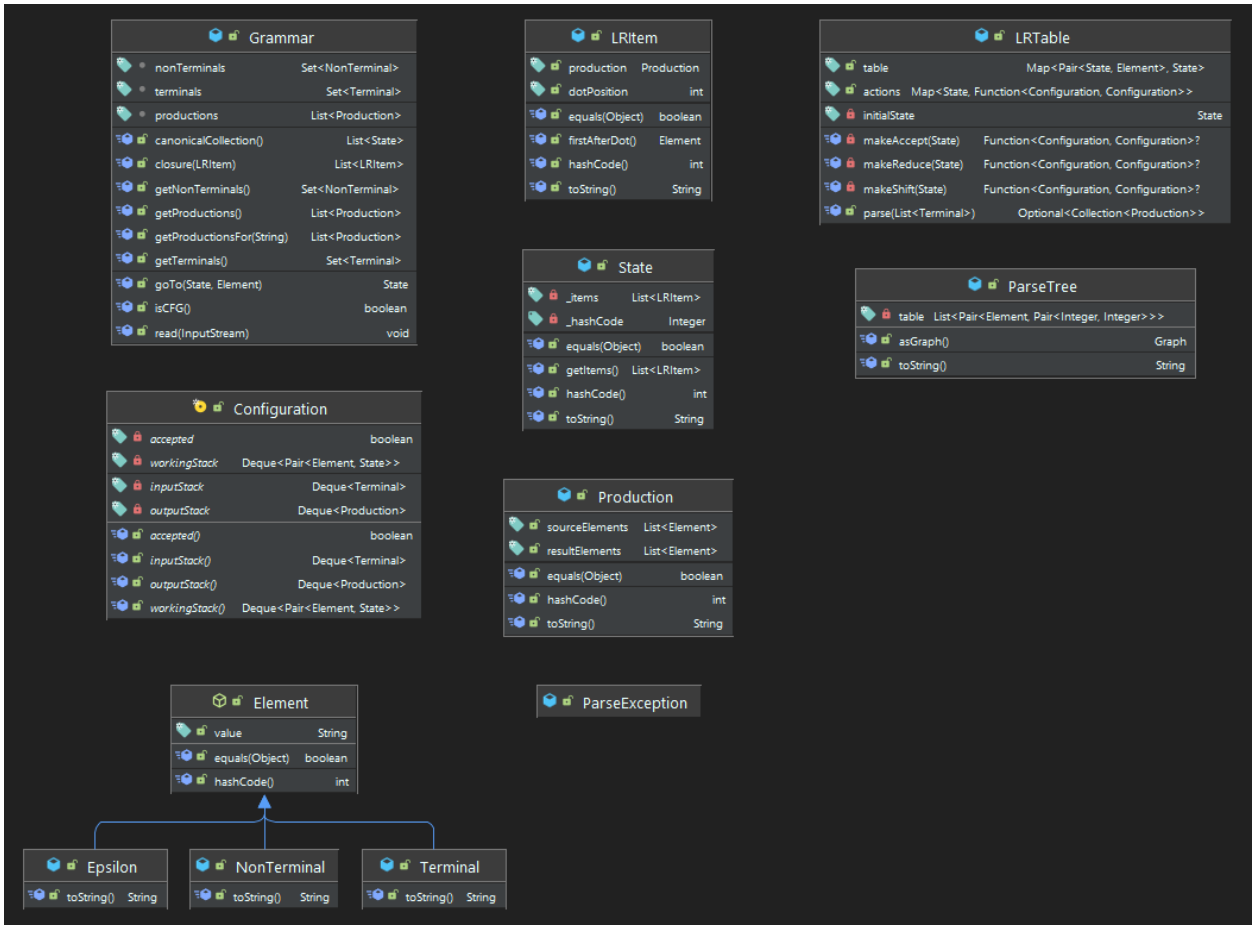
Lab 8

Catalin Borza, Darius Calugar

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.

LRTable class

Represents the LR(0) table.

For "goto" a pair of LR(0) State and Element are mapped to the result State; a Map is used for this purpose, and it is set up according to the Grammar's goto method.

The actions column in the LR(0) table is represented by a Map, where a State is mapped to a Function that operates on a Configuration. The functions are created using the corresponding methods makeShift, makeReduce, makeAccept, and two or more of these methods succeeding on a state usually means there is a conflict (e.g. makeShift and makeReduce return non-null Functions => shift-reduce conflict).

Configuration class

Encapsulates the working stack, input stack and output stack

ParseTree class

Contains the representation of a parse tree, in father-sibling format.

- Represented as a list of entries with an Element and two Integers, where the integers point to the positions of the father and the right sibling in the same list

public ParseTree(NonTerminal initial, List<Production> productionString)

- Creates the parse tree from a string of productions, starting with the initial (start) symbol

public String toString()

- Returns the parse tree as a csv string

public Graph asGraph()

- Creates a top-down Graphviz graph representing the parse tree

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/Lab8

Tests

Tests were ran on the following grammars as inputs:

Test 1

Input:

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

Output:

parse exception at ([S' -> E ., E -> E . + T]): Shift - Reduce conflict

Test 2

Input:

```
program ::= cmpdStmt
type ::= "integer"|"string"|type "list"|userType
var ::= "identifier" | "the" "identifier" "of" var | "that" var
"at" expression
userType ::= "my" "identifier"
condition ::= expression "is" relation "compared" "to"
expression
expression ::= to_operator expression "to" expression |
from_operator expression "from" expression | with_operator
expression "with" expression | var | "const"
to_operator ::= "add"
from_operator ::= "subtract"
with_operator ::= "multiply"|"divide"|"mod"
relation ::= "smaller"|"greater"|"equal"
defStmt ::= "define" userType cmpdDeclStmt
cmpdDeclStmt ::= "with" declStmts "end"
declStmts ::= declStmts declStmt | declStmt
declStmt ::= type "identifier"
stmt ::=
cmpdStmt|assignStmt|ioStmt|ifStmt|declStmt|stopStmt|defStmt|loo
pStmt
stmts ::= stmt | stmts stmt
cmpdStmt ::= "begin" stmts "end"
assignStmt ::= var "becomes" expression
ioStmt ::= readStmt|printStmt
readStmt ::= "read" var
printStmt ::= "print" expression
```

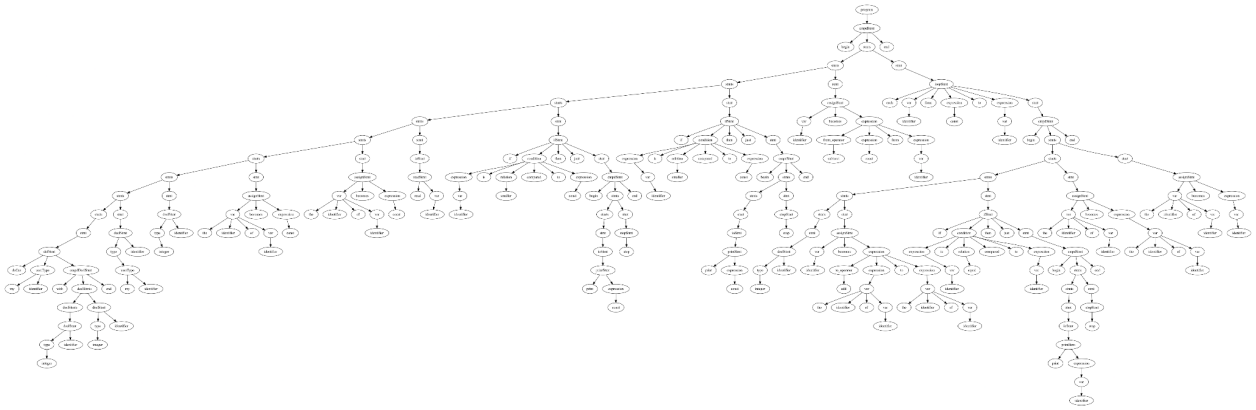
```
ifStmt ::= "if" condition "then" "just" stmt | "if" condition
"then" stmt "else" stmt
loopStmt ::= "each" var "from" expression "to" expression stmt
stopStmt ::= "stop"
```

Input:

```
begin
   define my type
   with
       integer n1
       integer n2
   end

   my type obj
   integer k
   the n1 of obj becomes 1
   the n2 of obj becomes 1
   read k
   if k is smaller compared to 1 then just
   begin
       print "Error this will not work"
       stop
   end
   if k is smaller compared to 3 then just
   begin
       print 1
       stop
   end
   k becomes subtract 2 from k
   each i from 1 to k
   begin
       integer new
       new becomes add the n1 of obj to the n2 of obj
       if i is equal compared to k then just
       begin
           print new
           stop
       end
       the n1 of obj becomes the n2 of obj
       the n2 of obj becomes new
   end
end
```

Output:



Test 3

Grammar:

```
S ::= "a" A
A ::= "b" A
A ::= "c"
```

Input:

abbbbc

Output: