Formal Languages and Compiler Design
Sixth laboratory
LL(1) Parser
Part 2

Link to GitHub repository: https://github.com/albcristi/flcd-team-work

Implemented class:

1. ParserOutput:
   1.1 Constructor
   - takes as input a Parser object and a list of productions
   - constructs the tree corresponding to a sequence

   1.2 Main Methods:

   1.2.1 generateTree(): builds a tree for a sequence

   1.2.2 generateSequence(): builds a sequence from the corresponding nodes

   1.2.3 printTree(): prints the list of existing nodes from the tree, providing informations like: index, value, sibling and father

   1.2.4 writeToFIle(): writes the corresponding tree to a file, where each line represents a node that is structured after the following rule:

   index <space> value <space> father <space> sibling <space> hasright

   EBNF:

   file :: = line "\n" | line "\n" file

   line ::= index " " value " " father " " sibling " " hasright

   hasright ::= 0 | 1

   value ::= terminal | nonTerminal

   terminal ::= string

   nonTerminal ::= string

Bacotiu Denisa-Cristina
Alb Dan-Cristian
931

index ::= posNumber

father ::= posNumber | -1

sibling ::= posNumber | -1

posNumber ::= nonzero{digit}
digit ::= "0" | nonzero nonzero:= “1” | “2”| … | “9”

1.2.5 Output file example:

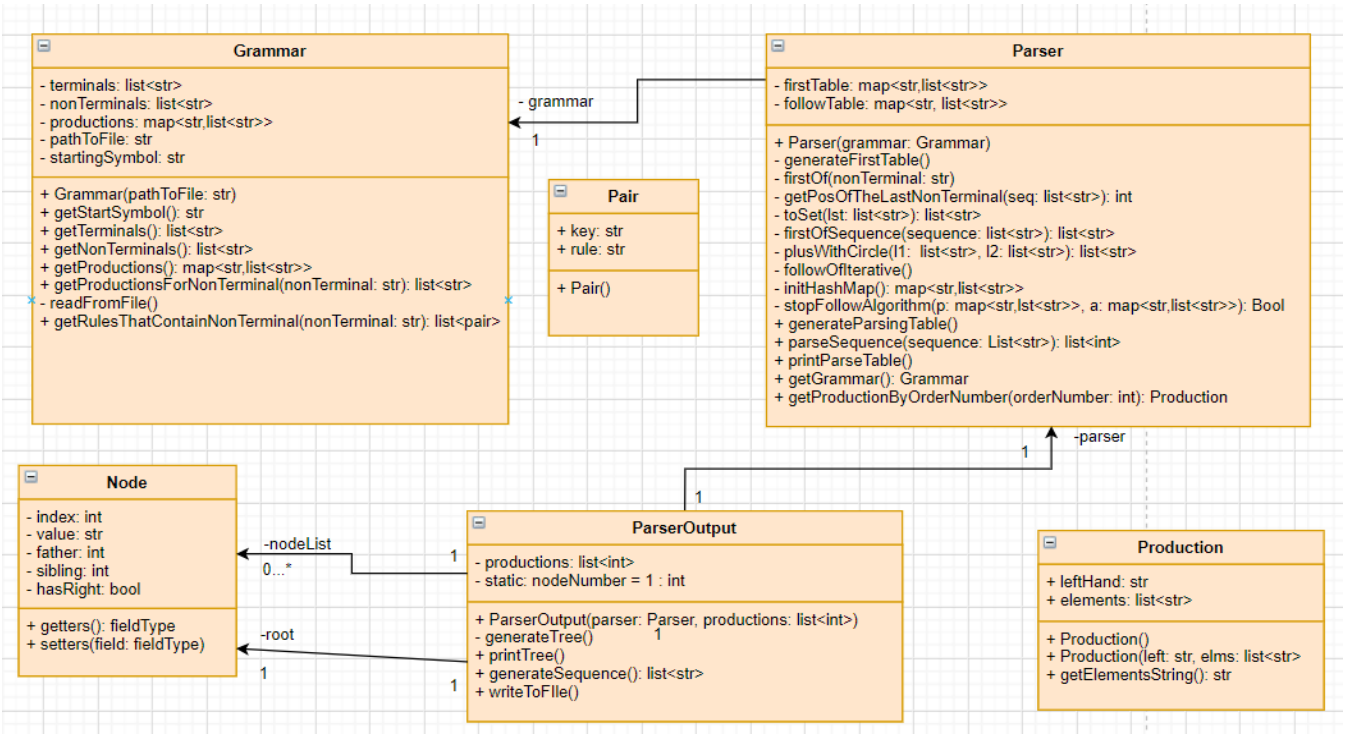
```
1 S -1 -1 0
3 A 1 2 0
2 B 1 -1 1
5 C 2 4 0
4 D 2 -1 1
6 a 4 -1 0
7 ε 5 -1 0
10 A 3 9 0
9 B 3 8 1
8 + 3 -1 1
12 C 9 11 0
11 D 9 -1 1
13 a 11 -1 0
14 ε 12 -1 0
15 ε 10 -1 0
```

2. Node:
   2.1 Constructor: default
   2.2 Description and usage:
      2.2.1 Used for representing nodes in a tree alike structure
      2.2.2 Fields:
         - index: unique identifier, like a node number
         - value: value encapsulated in the node (e.g: terminal, nonterminal)
         - father: index of the father node
         - sibling: index of left brother node
         - hasRight: indicates if there is a right brother for a node

Tree generation for a sequence algorithm:

- we set the root of the tree as being the nonTerminal from the first element form the list of productions resulted from the string of productions algorithm from Parser class
- we create a stack object and add to it the root node
- we loop until there are no elements on the stack or we fully parsed the list of productions based on production order indexes
- we take the top of the stack, peek operation, not pop
- if we have a terminal or epsilon:
  we pop elements from stack till we get to an element that can have children that were not yet processed
- we get the production corresponding to the current node and add its children. Production is obtained from the current element we point at in the list of productions (we keep an index for parsing this list)

Class diagram:



Bacotiu Denisa-Cristina
Alb Dan-Cristian
931