Documentation for Lab7

LINK TO GIT: https://github.com/radutalaviniaelena/FLCD

REQUIREMENT:

Statement: **Implement a parser algorithm (cont.)**

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.

For implementing the laboratory, I have the following:

1) **Grammar** class – described in documentation for the fifth laboratory;

2) **State** enum – it contains all possible states a configuration can receive during the execution of the parser algorithm:

a) **NORMAL_STATE** – it is found in success, expand, advance and momentary insuccess moves;

b) **BACK_STATE** – it is found in back and another try moves;

c) **FINAL_STATE** – it is received in the end, when the algorithm is successfully finished;

d) **ERROR_STATE** – it is received when something wrong is happening during the execution of the parser algorithm (e.g.: when the given sequence cannot be parsed)

3) **Move** enum – it contains all possible moves that the parser algorithm can assign:

a) **SUCCESS** – accessed when the state of the parsing is normal, the sequence is entirely processed and the input stack is empty;

b) **EXPAND** – accessed when the state of parsing is normal and the top of the input state is a nonterminal;

c) **ADVANCE** – accessed when the state of parsing is normal and the top of the input state is a terminal which is equal to the current element in the sequence (if it exists);

d) **MOMENTARY INSUCCESS** – accessed when the state of parsing is normal and the top of the input state is a terminal which is not equal to the current element in the sequence;

e) **BACK** – accessed when the state of parsing is back and the top of the working stack is a terminal;

f) **ANOTHER TRY** - accessed when the state of parsing is back and the top of the working stack is a nonterminal;

4) **Configuration** class – contains all fields needed in order to keep the evidence of steps during the execution of the parsing algorithm:

a) **private Move move** – the move needed to achieve the current configuration;

b) **private State stateOfParsing** – the state of the parsing;

c) **private Integer positionCurrentSymbol** – the position of the current element from the given sequence;

d) **private Stack<String> workingStack** – keeps all elements (terminals + nonterminals) processed so far;

e) **private Stack<String> inputStack** - keeps all elements (terminals + nonterminals) that will have to be processed;

5) **DescendantRecursiveParser** class – contains the entire logic to be able to implement the parser algorithm:

a) /**
* This function implements the logic of EXPAND move: creates the next configuration of the algorithm, removing the nonterminal from the top of the input stack and processing the production indicated by the count parameter – adds the nonterminal to the working stack to know which one is used and its corresponding production to the input stack.
* **@param** configuration : the current configuration (which will be changed)
* **@param** count : an integer number representing the number of the production to be processed for the current nonterminal
* **@param** grammar : the grammar of the language (the one from the fifth laboratory)
* **@return** : the new (modified) configuration
*/
public Configuration expand(Configuration configuration, int count, Grammar grammar)

b) /**
* This function implements the logic of ADVANCE move: creates the next configuration of the algorithm, removing the terminal from the top of the input stack and adding it to the working stack. In the case the removed terminal is epsilon, it will not be added to the working stack (because it will never be equal to any element from the sequence).
* **@param** configuration : the current configuration (which will be changed)
* **@return** : the new (modified) configuration
*/
public Configuration advance(Configuration configuration)

c) /**
* This function implements the logic of MOMENTARY INSUCCESS move: creates the next configuration of the algorithm, changing the state of parsing to BACK_STATE.
* **@param** configuration : the current configuration (which will be changed)
* **@return** : the new (modified) configuration
*/
public Configuration momentaryInsuccess(Configuration configuration)

d) /**
* This function implements the logic of BACK move: creates the next configuration of the algorithm, removing the terminal from the top of the working stack and adding it to the input stack (it decreases the position of the current symbol table because we go back with the process of searching for a position).
* **@param** configuration : the current configuration (which will be changed)
* **@return** : the new (modified) configuration
*/
public Configuration back(Configuration configuration)

e) /**
* This function implements the logic of ANOTHER TRY move: creates the next configuration of the algorithm, trying another production for the nonterminal from the top of the working stack if it exists or adding the nonterminal back to the input stack otherwise. In case we return to the start symbol, it is an error.
* **@param** configuration : the current configuration (which will be changed)
* **@param** grammar : the grammar of the language (the one from the fifth laboratory)
* **@return** : the new (modified) configuration
*/
public Configuration anotherTry(Configuration configuration, Grammar grammar)

f) /**
* This function implements the logic of SUCCESS move: creates the next configuration of the algorithm, changing the state of parsing to FINAL_STATE.
* **@param** configuration : the current configuration (which will be changed)
* **@return** : the new (modified) configuration

*/
public Configuration success(Configuration configuration)

g) /**
* This function implements the parser algorithm, calling the above described functions when they meet their conditions (which are described at 3). This function uses the following methods:
-> **private boolean verifyAdvance(String[] sequence, Grammar grammar, Configuration configuration)** – contains all conditions needed to access advance move;
-> **public boolean isIdentifier(String identifier)** – verifies if the given string is an identifier;
-> **public boolean isConstant(String constant)** – verifies if the given string is an identifier;
-> **private void constructWorkingAndInputStacks(List<Configuration> configurations, Configuration configuration)**

* **@param** sequence : the sequence given by user
* **@param** grammar : the grammar of the language (the one from the fifth laboratory)
*/
public void descendantRecursiveParserAlgorithm(String[] sequence, Grammar grammar)

h) /**
* This function writes to a file the content of a java List.
* **@param** path : the location of the file
* **@param** configurations : the list of configurations
*/
public void writeConfigurationsToFile(String path, List<Configuration> configurations)

i) /**
* This function extracts all the non-terminals from a given working list and adds each of them to the tree along with their generated productions. For each parsed symbol, a new Node will be generated containing the following fields:
- **index**: will keep increasing by 1 as each node is added to the tree
- **info**: the String representation of the current symbol
- **leftSibling**: for each generated production, the first element will not have a left slibing so the field will be marked as -1, as for the following elements, their left slibing will be the previous index value
- **parent**: this field contains the index of the non-terminal which generated the current production. In order to remember the index of each parent, a new Dictionary is created which has as a key the non-terminal String representation, and as the corresponding value, a queue which contains the indexes of the non-terminals whose productions have not been parsed yet. Once the production is parsed, their parent index will be eliminated from the queue, as we will not need it anymore for the following iterations.
* **@param** workingList: a configuration's final working list
* **@param** grammar : the grammar of the language (the one from the fifth laboratory)
*/

public List<Node> constructTree(List<String workingList, Grammar grammar)

j)/**
* This function writes in a table format a tree representation given as a list of objects of type Node
* **@param** path : the location of the file
* **@param** tree : the list of nodes
*/
public writeTableToFile(String path, List<Node> tree)

6) **Node** class – it contains the structure of each element from the parsing tree:

a) **private int index** – the position of the node in the parsing tree
b) **private String info** – the value corresponding to the node
c) **private int parent** – the index of the node's parent (its parent is the non-terminal which generated the current production)
d) **private int leftSlibing** – the index of the previous term (terminal or non-terminal) from the current production. If the current symbol is the first in the production, it does not have a left slibing, so the leftSlibing will be marked as -1.

7) **Main** class – same as in the fifth laboratory with one exception: the menu contains a new item, writing a sequence to check whether or not it can be parsed by grammar