Github Link: https://github.com/916RasnitaRadu/SEM5-FLCD/tree/main/Lab6-7

Documentation

Grammar

The Grammar class implements the grammar for a formal language. It has a field for each (N, E, P, S) set of the grammar and those are: *non-terminals*, *terminals*, *productions* and *the starting symbol*. The starting symbol is kept as a map between, where the key is the left-hand side of the production and the value is the right-hand side of the production represented as a list of strings.

Most of the methods are for parsing, such as:

- parseLine(String line) – parse a line of the file and return a list of words
- fromFile(String fileName) – read the input grammar file, creates the (N, E, P, S) tuple and verifies if the grammar is context free by calling the validate() function
- parseRules(List<String> rules) – takes the set of the productions and creates the map from the class

There are also other methods:

- validate(List<String> N, List<String> E, Map<String, List<String>> P, String S) – checks if the grammar is context free
- isNonTerminal(String value)
- isTerminal(String value)
- getProductionsFor(String nonterminal) – returns the list of productions for a non-terminal

Parser

The Parser class is implementing the LL(1) algorithm, with the FIRST and FOLLOW algorithms. The class is composed of: one object of type Grammar, the FIRST set of terminals and the FOLLOW set of terminals.

Operations:

- generateFirst( ) – method that generates a set for each non-terminal that contains all terminals from which we can start a sequence (from the given non-terminal)

- generateFollow( ) – method that builds for each non-terminal which contains the "first of what's after", namely all the non-terminals into which we can go from the given non-terminal

- innerLoop(Set initialSet, List items, Set additionalSet) – method for computing the FIRST set of a sequence of symbols within a production rule

- generateTable( ) : void - function creates the LL(1) parse table following the corresponding rules: we build a table that has rows and columns for all non-terminals and terminals plus the '$' sign in both rows and columns.

- evaluateSequence(List<String> sequence) : String - responsible for the parsing of a given input sequence using the parsing table, 2 stacks alpha and beta and push/pop rules from the lecture.

Tree

Tree class provides the parser output in a tree format, using a child and sibling relation, in BFS format.

Attributes: A root Node, a grammar, an index <crt> for the current node in the tree (used for printing), ws a list of words from the sequence;

- build(sequence): node -> calls the buildTree method
- buildTree(currentSymbol): node -> recursive function that builds the tree, starting from the start symbol of the grammar: at first we check if there is any relevant symbol left in the sequence, then we create the new node, it it is terminal it can be returned, otherwise we check for the next valid production that matches the symbols in the sequence, and we create a new child node for each and, implicitly, a new branch; going back down the stack, we make the child and sibling relations
- printTable() -> calls the method that prints the table in bfs format
- bfs(node, parentIndex, rightSiblingIndex) -> performs a bfs on the result tree from buildTree method, assigning indexes to the nodes, for each node we print the index, symbol, parentIndex and rightSiblingIndex

Node

Helper class for defining the nodes of the tree

Attributes: String value, Node child, Node rightSibling

TableEntry

Helper class for defining the entries in the output table from the bfs method

Attributes: Node item, int parentIndex, rightSiblingIndex