Lab 5-7

Link Github: https://github.com/913-groza-vlad/Formal-Languages-and-Compiler-Design/tree/main/Lab5-7

The Grammar class represents a grammar and provides methods to initialize the grammar from a file, parse its content and check whether it is a context-free grammar.

The Grammar class has the following attributes:
- nonTerminals, which is a set containing the non-terminal symbols of the grammar
- terminals, which is a set containing the terminal symbols of the grammar
- productions, which is a map representing production rules, where each key is a non-terminal symbol, and the corresponding value is a set of production rules
- startSymbol, the start symbol of the grammar
- filename, the name of the file containing the grammar rules

The class contains methods for representing each of its fields in string format and the also the whole grammar (so we can display the grammar and its content) and the methods:

private Set<String> parseLine(String line):
This method parses a line from the grammar file to extract a set of elements enclosed in curly braces.

public boolean checkCFG():
Checks whether the grammar adheres to the rules of a context-free grammar. In order to return true, the following conditions must be met: the start symbol of the grammar must appear on the left-hand side of at least one production rule, all left-hand side symbols of production rules must be non-terminals and all symbols in the right-hand side must be either non-terminals, terminals, or the symbol epsilon.

public void readGrammarFromFile():
It reads the grammar rules from a specified file, initializes the grammar attributes (non-terminals, terminals, start symbol and productions) and populates the

Grammar object with the parsed information. The method reads the first two lines from the file, extracting non-terminals and terminals from lines enclosed in curly braces. Then, on the third line there is the start symbol of the grammar. Then, the productions are read by: skipping the line containing P = { header, then iterate through the remaining lines, parsing production rules and adding them to the set of productions for each non-terminal.

The Parser class is designed to implement a predictive parsing table and perform parsing on a given sequence using the LL(1) parsing technique. The class contains the following attributes:
- grammar: Grammar, which represents the grammar object on which we parse the sequence to see if it is accepted
- first: a dictionary which maps a non-terminal with the set of symbols representing the first set corresponding to that non-terminal
- follow: a dictionary which maps a non-terminal with the set of symbols representing the follow set corresponding to that non-terminal
- parseTable: the table is represented using a dictionary mapping the value (this is a pair consisting in a string and an integer) to a position (this is a pair of strings representing the row and the column in the table)
-rhsOfProductions: a list of list of strings containing the right hand side of all productions of the grammar

There are implementations of the first and follow sets associated to each non-terminal of a grammar, which are essential for constructing the parsing table. Thus, these methods are described as:

public void computeFirst():
This method calculates the FIRST set for each non-terminal in the grammar and the algorithm iteratively refines the FIRST sets until no further changes occur. Initially, for each non-terminal A of the grammar, the set $F_0(A)$ is formed by checking those productions for the non-terminal A, whose right hand side starts with a terminal, and that terminal is added to $F_0(A)$. If there is a production for A that contains epsilon, it will also be added to the first set of A. Then, the method iterates over each non-terminal, updating its FIRST set based on the FIRST sets of

its production symbols. For each production of the non-terminal, symbols are added to the FIRST set until a terminal or a symbol with a non-empty FIRST set is encountered. If the symbol has an epsilon in its FIRST set, it continues to the next symbol in the production. The iterations continue until $F_{i-1}(A) = F_i(A)$ for each non-terminal A and the last configuration of the FIRST is kept.

public void computeFollow():
It calculates the FOLLOW set for each non-terminal in the grammar. The FOLLOW set of a non-terminal is the set of terminals that can appear immediately to the right of instances of that non-terminal in some sentential form. The algorithm iteratively refines the FOLLOW sets until no further changes occur. We initialize for each non-terminal A of the grammar the set $L_0(A)$ with an empty set, excepting the follow set of the starting symbol, which contains epsilon. Then, for each non-terminal we identify all the productions which contains that terminal in the right hand side of the production and add in the $L_i(A)$ previous follow set: $L_{i-1}(A)$. If beta is the sequence that follows A in the right side of a production, the symbols in FIRST(beta), excepting epsilon, are added to $L_i(A)$. If FIRST(beta) contains epsilon, or there is no sequence that follows A, in $L_i(A)$, the follow of the left hand side non-terminal is added. The iterations end when two columns are equal(the follow sets for all non-terminals doesn't change).

public void createParseTable():
This method creates the LL(1) parsing table. It initializes the table with error and pop entries and then fills in the table entries based on the first and follow sets of non-terminals. It also checks for conflicts and handles them appropriately.
In more detail, the table has rows corresponding to all the symbols of the grammar and the $ symbol and columns corresponding to the terminal symbols + the $ symbol. The cells of the table associated with the pairs (a, a), where a is a terminal, are filled in with the pair ("pop", -1), the cell corresponding to the pair ($, $) is completed with ("acc", -1) and the others cell are filled with ("err", -1). Then for each non-terminal A, each production is processed, in the following way: if A -> alpha is the production, numbered with i, we get the first of alpha, and for all symbols corresponding to this set, we fill in the cell (A, x) with the pair (alpha, i) If there is epsilon in the first(alpha), all cells (A, y) are completed with (alpha, i), where is y is a symbol in the follow set of A.

public List<Integer> parseSequence(String sequence):
This method performs the parsing of a given sequence. It uses two stacks, inputStack and workingStack, along with the LL(1) parsing table. The input stack contains the $ symbol and the symbols of the sequence, the top of the input stack being the first symbol of the sequence. The working stack contains $ and the starting symbol of the grammar. At each iteration we retrieve from the parse table the value corresponding to the position consisting of the peeks of both stacks, then if we have “pop”, we perform two pop operations from the stacks, otherwise, if we have a production, a pop is made from the working stack and the symbols of the productions are pushed to it. It continues to iterate until it either accepts the sequence or encounters an error and generates a list of production numbers during the parsing process which is returned if the sequence is accepted, otherwise, if an error occurred, a message is displayed and an array containing the value -1 is returned.

The ParserOutput class is designed to capture the output of the parsing process, particularly the sequence of derivations. As an overview of its structure and functionality, we have:
- parser: holds an instance of the Parser class, providing access to parsing-related functionality
- productionSequence: represents the list of production numbers generated during parsing
- sequenceOfDerivations: a list of strings that keeps track of the sequence of derivations
- hasError: indicates whether an error occurred during parsing
- fileName: stores the name of the file to which the sequence of derivations are written

public void buildSequenceOfDerivations():
It constructs the sequence of derivations based on the production numbers and the grammar's associated non-terminals and right-hand sides of productions.

public void printSequenceOfDerivations():
Writes the string representation of the sequence of derivations to a file and prints it to the console.

Class diagram: