Documentation

GitHub: https://github.com/917tapoimarius/LFTC_Parser

Team members:

Taravinas Iannis

Tapoi Marius-Stefan

Class Grammar: A class representing the grammar of the programming language:

Attributes:

nonTerminals: List<String> – a list of Strings representing the nonterminals of the grammar.

terminals: List<String> - a list of Strings representing the terminals of the grammar.

startSymbol: String – a String which represents the starting symbol of the grammar.

productions: Map<String, List<List<String>>> – a table like representation of the productions of the grammar; the key is represented by a nonterminal, and for each key, the value is represented by a list of lists of strings, so that a nonterminal can be represented in multiple ways by one or more symbols.

Methods:

Grammar (): Constructor of the class; initializes the nonTerminals List as an ArrayList, the terminals List as an ArrayList and the productions table as a HashMap.

params: -

returns: -

readFile (String filename): Method used to read through each line of the input file. The first line represents the nonterminals. When it is read, the strings are stored in the *nonTerminals* list. The second line represents the terminals which are stored in the *terminals* list. The third line represents the *startSymbol*. The following lines represent the *productions.* Each line is split into a left side and a right side, separated by an arrow. The left side represents a nonterminal, while on the right side there can be another nonterminal, a terminal, or a list of nonterminals. If the left side already exists in the productions list as a key, then the right side is added at the key's address, otherwise the left side is stored as a key, and the right side as an ArrayList at the key's address.

params:

filename: String – the location of the file from which the data is read.

returns: -

checkCFG(): Method used to check if the read grammar is a context free grammar. At first, it checks if all the keys (represented by nonterminals) are stored in the productions table. If one is missing, then the grammar is not context free. Then, it checks all the values from the table. It checks every element from each production list and if the one of the elements is not found in the nonterminals or terminals lists then the grammar is not context free. If these two checks pass, then the grammar is context free.

params: -

returns: true if the grammar is context free or false otherwise.

printNonTerminals(): Method used to print the nonterminals list.

params: -

returns: -

printTerminals(): Method used to print the terminals list.

params: -

returns: -

printProductions(): Method used to print the productions list in a structured manner. Each key (nonterminal) is followed by an arrow and the list of all possible representations of the nonterminal, each representation being separated by the '|' character.

printStartSymbol(): Method used to print the start symbol.

params: -

returns: -

printProductionsForNonterminal(String nonTerminal): Method used to print all the productions for a given nonterminal. The nonterminal is printed (key), followed by the production (value), separated by an arrow. If the production is represented by multiple lists, the key is printed with each of its productions.

params:

nonTerminal: String – given string representing a nonterminal for which all the productions are printed.

returns: -

getNextProduction(String production, String nonTerminal): Method used to retrieve the production which follows the given *production* for the given nonterminal. It gets all the productions for the given nonterminal and iterates through the list of productions until it finds the matching one. If it is not the last from the list, it returns the next production. If it is the last one, it returns null.

params:

production: String – a string composed of one of the nonterminal's productions concatenated.

nonTerminal: String - the nonterminal for which the productions list is used to find the next production after the given one.

returns: the production which follows the given one or null if the given production is the last from the productions list.

getStartSymbol(): Method used to get the start symbol of the grammar.

params: -

returns: a string representing the start symbol of the grammar.

getNonTerminals(): Method used to get the list of nonterminals of the grammar.

params: -

returns: a list of strings representing the nonterminals of the grammar.

getTerminals(): Method used to get the list of terminals of the grammar.

params: -

returns: a list of strings representing the terminals of the grammar.

getProductions(): Method used to get the productions of the grammar.

params: -

returns: a map (table) for which the key is represented by a nonterminal (string), and for each key, the value is represented by a list of lists of strings; this map represents the productions of the grammar.

getProductionForNonterminal(String nonterminal): Method used to get productions list of a given nonterminal.

params: -

returns: a list of lists of strings representing the productions list of the nonterminal.

Class RecursiveDescendentParser: A class representing the parser of the programming language. It is implemented using the recursive descendent parsing method:

Attributes:

grammar: Gramamar – a Grammar object representing the grammar of the programming language.

working: List<List<String>> - a list of lists of strings representing the working stack of the parser.

input: List<String> – a list of strings representing the input stack of the parser.

state: String – a string which represents the current state of the parsing.

sequence: List<String> - a list of strings representing the sequence which needs to be parsed.

index: int – an integer representing the position of current symbol in input sequence.

depth: int – an integer representing the depth at which the parsing got.

leftRecursive: boolean – a boolean which shows whether the used grammar is left recursive or not.

Methods:

RecursiveDescendentParser(Grammar grammar): Constructor of the class; it sets the *grammar*, initializes the *working* and *input* stacks as ArrayLists and pushes in the *input* stack the starting symbol of the *grammar*, the *state* to "q" (normal state), the *index* and *depth* to 0, the *sequence* to null and the *leftRecursive* to the value returned by the left recursion checker method

params:

grammar: Grammar – the grammar of the programming language.

returns: -

checkGrammarLeftRecursive(): Method used to check if the parser's grammar is left recursive. It iterates through the productions list, checks if each nonterminal (the key of each produiction) is the first element of its productions and returns true if so or false otherwise.

params: -

returns: - true if the grammar is left recursive or false otherwise.

getModel(): Method used to print the configuration model in a structured manner.

params: -

returns: -

momentaryInsuccess(): Method used to set the state to "b" (back).

params: -

returns: -

success(): Method used to set the state to "f" (final).

params: -

returns: -

advance(): Method used to advance in the recursion tree. It pushes the top element of the input stack into the working stack and then pops the input. It increments the index, and if it is higher than the depth, the depth takes its value.

params: -

returns: -

expand(): Method used to expand. It takes the non-terminal from the input and searches for its first production and pushes it on the working stack as a list containing the nonterminal and the found production. It also removes the non-terminal from the input and puts the first production into the input.

params: -

returns: -

back(): Method used pop the head from the working stack which is a terminal and push it on the input stack. The index is also reduced.

params: -

returns: -

anotherTry(): Method used try the next production from the production list of the current nonterminal. If the next production exists, the state is set to normal, the working stack head is replaced by the found production, the input stack is cleared, and the new sequence is pushed. If it is not found, the index is 0 and the last production's symbol is the starting symbol, the try also failed and the error state is set. Otherwise, the head of the working state is popped, the input stack is cleared, and the last production is pushed on it.

params: -

returns: -

checkSequence(List<String> sequence): Method used to check a if a given sequence is accepted or rejected by the parser's grammar. If the grammar is left recursive, it cannot be checked. Then, it checks if every element of the sequence is found in the terminals list. If one of them is not found, the state is set to error, and the user is notified. If these two checks are passed, the sequence is checked. While the state is not set to error or final, the model is printed. If the state is set to normal and if the input stack is empty and the index reached the sequence size, the *success()* method is called. If these last two conditions are not met it is checked if the head of input stack is a nonterminal. If this is true, *expand()* method is called. If this is false, but the head of the input stack is a terminal which is equal to the current symbol from the input the *advance()* method is called. Else, the algorithm failed to find the symbol so *momentaryInsuccess()* is called. If the state is set to back and the head of the working stack is a terminal, the *back()* method is called, otherwise *anotherTry()* is called. If at some point the state is set to error, the algorithm failed, so the sequence is not accepted. Otherwise, the sequence is valid.

params:

sequence: List<String> - a list of strings representing the sequence which needs to be checked if is valid for the given programming language.

returns: true if the sequence is valid for the given programming language (accepted) or false otherwise (rejected)

Class Diagram: