CS 410 Automata Theory and Formal Languages

Project 2 Report

Bora Özügüzel

S014465

1. Introduction

The problem that is needed to solve in this homework is converting a Context-Free Grammar (CFG) to the Chomsky Normal Form (CNF). The CFG's are given as text files named G1.txt and G2.txt.

For Chomsky Normal Form, there cannot be a start variable on the right hand side of any rule. If thr righ side contains non terminals its length must be two, if it contains terminals its length must be one.

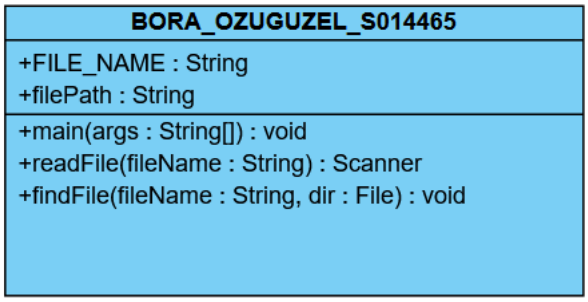
The language chosen for this project is Java.

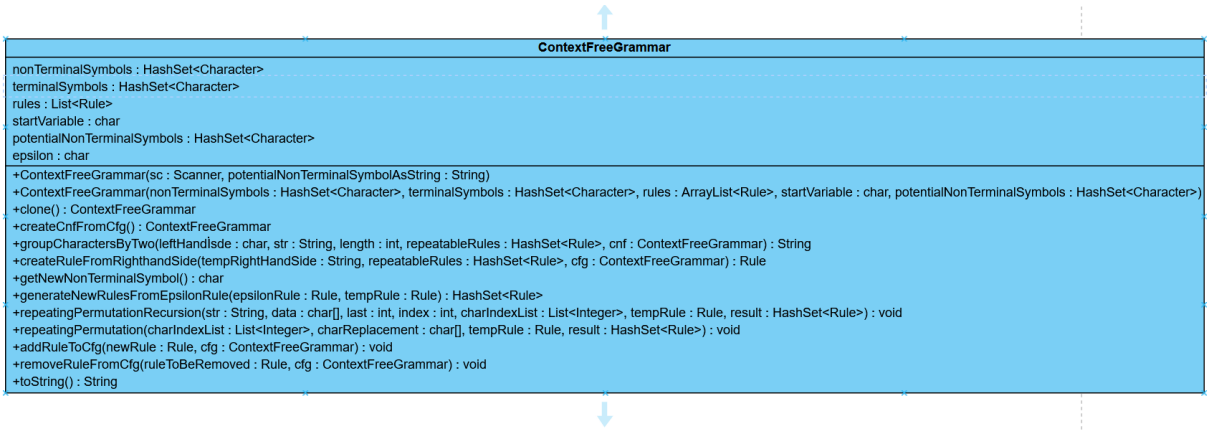
The file format for the input files:

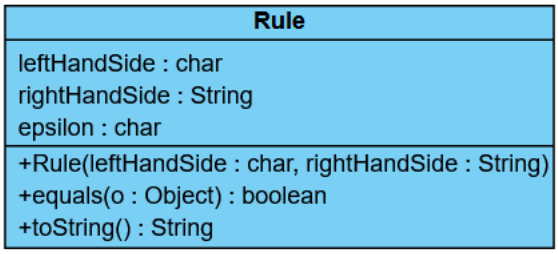
NON-TERMINAL
S
X
…
Y
TERMINAL
0
…
RULES
S:XY
X:0
…
Y:XX
START
S

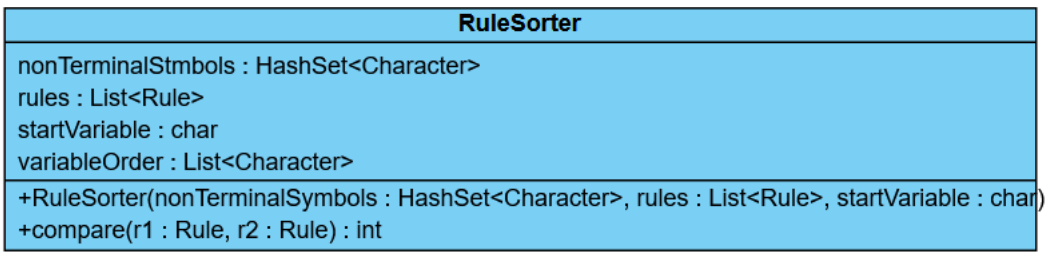
2. Classes and Methods

Class Diagram of the project is given below:


BORA_OZUGUZEL_S014465
+FILE_NAME : String
+filePath : String
+main(args : String[]) : void
+readFile(fileName : String) : Scanner
+findFile(fileName : String, dir : File) : void


ContextFreeGrammar
nonTerminalSymbols : HashSet<Character>
terminalSymbols : HashSet<Character>
rules : List<Rule>
startVariable : char
potentialNonTerminalSymbols : HashSet<Character>
epsilon : char
+ContextFreeGrammar(sc : Scanner, potentialNonTerminalSymbolAsString : String)
+ContextFreeGrammar(nonTerminalSymbols : HashSet<Character>, terminalSymbols : HashSet<Character>, rules : ArrayList<Rule>, startVariable : char, potentialNonTerminalSymbols : HashSet<Character>)
+clone() : ContextFreeGrammar
+createCnfFromCfg() : ContextFreeGrammar
+groupCharactersByTwo(leftHandIsde : char, str : String, length : int, repeatableRules : HashSet<Rule>, cnf : ContextFreeGrammar) : String
+createRuleFromRighthandSide(tempRightHandSide : String, repeatableRules : HashSet<Rule>, cfg : ContextFreeGrammar) : Rule
+getNewNonTerminalSymbol() : char
+generateNewRulesFromEpsilonRule(epsilonRule : Rule, tempRule : Rule) : HashSet<Rule>
+repeatingPermutationRecursion(str : String, data : char[], last : int, index : int, charIndexList : List<Integer>, tempRule : Rule, result : HashSet<Rule>) : void
+repeatingPermutation(charIndexList : List<Integer>, charReplacement : char[], tempRule : Rule, result : HashSet<Rule>) : void
+addRuleToCfg(newRule : Rule, cfg : ContextFreeGrammar) : void
+removeRuleFromCfg(ruleToBeRemoved : Rule, cfg : ContextFreeGrammar) : void
+toString() : String


Rule
leftHandSide : char
rightHandSide : String
epsilon : char
+Rule(leftHandSide : char, rightHandSide : String)
+equals(o : Object) : boolean
+toString() : String


RuleSorter
nonTerminalStmbols : HashSet<Character>
rules : List<Rule>
startVariable : char
variableOrder : List<Character>
+RuleSorter(nonTerminalSymbols : HashSet<Character>, rules : List<Rule>, startVariable : char)
+compare(r1 : Rule, r2 : Rule) : int

3. Implementation Details

In the main method, a method to read the G1.txt file will be called named readFile. The method tries to find the file in the project folder with the given filename defined in FILE_NAME which is assumed to be G1.txt. If it cannot find it directly, it will try to search the file in the whole project with the method named findFile. This file will be accessed by the Scanner class from the Java.Util Package. The result will be sent to instantiate a new ContextFreeGrammar object, which can hold the information of both a CFG in non-Chomsky Normal Form or in Chomsky normal Form. A string named potentialTerminalSymbolsAsString will be also needed by the constructor. potentialNonTerminalSymbolsAsString holds the English Alphabet in upper case so it can be used to name new non-terminals. The constructor shown as below will be called:

public ContextFreeGrammar(Scanner sc, String potentialNonTerminalSymbolsAsString) {…}

When the construction of the CFG starts each char of potentialNonTerminalSymbolsAsString will be added to a Character HashSet called potentialNonTerminalSymbols.
In a while loop the scanned filed will be parsed line by line.
While constructing the object, an integer called partitionState is kept holding which partition between the titles in the txt file in. If the partitionState is 1, which means that it is currently reading the lines between NON-TERMINAL and TERMINAL. It will add every line as a char data type to a Character HashSet called nonTerminalSymbols. The same will happen for terminalSymbols. When a non-terminal is read, the corresponding non-Terminal in the potentialNonTerminalSymbols will be removed. For the RULES, a list named rules will be used. The List will use a new object called Rule. In the file, for rules a string that is separated to two with the colon symbol. The whole line is tokenized with StringTokenizer from Java.Util. The character at left side of the colon will be a char named leftHandSide and the right side of the colon will be a string named rightHandSide. After this, all value in the G1.txt file is assigned to the ContextFreeGrammar Object instantiated with the name cfg. Both the ContextFreeGrammar and Rule object hava a static variable to define epsilon as char e.

ContextFreeGrammar has a method called createCnfFromCfg as shown below:

public ContextFreeGrammar createCnfFromCfg() {…}

createCnfFromCfg method first clones the original context-free grammar, by using the clone method. To override the clone method the object needs to implement Cloneable interface then override the clone method. The clone method uses a constructor as shown below:

public ContextFreeGrammar(HashSet<Character> nonTerminalSymbols, HashSet<Character> terminalSymbols,
ArrayList<Rule> rules, char startVariable, HashSet<Character> potentialNonTerminalSymbols) {

The constructor clones all the inputs, so that the original cfg is not modified. So, only the new object named cnf is modified.

A method called getNewNonTerminalSymbol is needed, which generates a random index for the potentialNonTerminalSymbols HashSet. Then it uses the index to return a new char value to use as a new non-terminal symbol. It also removes the value from the HashSet.

getNewNonTerminalSymbol is used to create a new start variable. The start variable of cnf is changed and a new rule is added. The left side of the rule is the new start variable, while the right side is the start variable of the original object.

Epsilon rules are dealt with in a while loop. A HashSet of all the currently visible epsilon rules are found with a java stream lambda expression. It collects all rules where the right side is epsilon and the left side is not the start variable.
In a for loop for each epsilon rule, every rule of the cnf Is filtered where the right side contains the left side of the epsilon rule. The pair of matching rule and epsilon rule are sent to a method called generateNewRulesFromEpsilonRule.

public HashSet<Rule> generateNewRulesFromEpsilonRule(Rule epsilonRule, Rule tempRule) {…}

In generateNewRulesFromEpsilonRule, indexes of the left side of the epsilon rule are found in the right side of the matching rule, then they are collected to an integer list named charIndexList. A char array is created named charReplacement, which contains the epsilon value and the left side of the epsilon rule. They are sent to a method named repeatingPermutation to find all the new rules generated from the matching rule, where the left side of the epsilon rule is replaced with the epsilon value.

public void repeatingPermutation(List<Integer> charIndexList, char[] charReplacement, Rule tempRule, HashSet<Rule> result) {…}

It creates a char array called data to which will hold all the permutations of charReplacement.
Then, it sends the values to method named repeatingPermutationRecursion.

public void repeatingPermutationRecursion(String str, char[] data, int last, int index, List<Integer> charIndexList, Rule tempRule, HashSet<Rule> result) {}

repeatingPermutationRecursion calls itself for every character in charReplacement for every index in charIndexList. When the whole data array is constructed, the right side of the rule is split with the substring method and in between the chars from the data array is added in a for loop. This modifies the result HashSet.

The list of all HashSet is then collected. So, they could be used to add to rules of the cnf in a for loop with a method named addRuleToCfg.
public void addRuleToCfg(Rule newRule, ContextFreeGrammar cfg) {…}

The method checks if the rule exists with a java stream method.

The epsilon is removed with removeRuleFromCfg method.

public void removeRuleFromCfg(Rule ruleToBeRemoved, ContextFreeGrammar cfg) {…}

The method checks that if there are no other rules with the same left side it remove the non-terminal symbol from nonTerminalSymbols.

When there are no epsilon rules are left the while loop breaks.

Unit rules are dealt with in a while loop.
A random left side of a unit rule is chosen with the name randomUnitRuleLeftHandSide. It finds the first rule where the right side is equal to the left side of a rule with the length of one, then it is assigned as a char. When there is no char the loop is broken.
All the unit rules using the randomUnitRuleLeftHandSide is clooected as a HashSet with java stream method called unitRules.
In a for loop new rules are generated. The rules are filtered where the left side of the rule is equals to the right side of the unit rule, then a new rule is created. The left side of the rule is left side of the unit rule and the right side of the new rule is the right side of the matching rule. The result is then collected to a HashSet call newRuleSet with a java stream method. Then in a for loop each new rule is added. Then the unit rule is deleted.
Rules are then filtered where the left side of the rule is equal to the unit rule and if there are any rule other than the unit rule that contains the left side of the rule in their right side. This checks if there are any disconnected rules, which are then deleted.

If the createCnfFromCfg method is called to covert an empty set, where there is only one rule where the left side is equal to right side, the previous while loop deletes it as a unit rule, so new rule is used to fix the bug.

Lastly rest of the problems are fixed
All rules are filtered where the right side has only two symbols but contains a terminal variable or the right side has more than two symbols, the result is then collected as a HashSet called nonConformingRules.

Every right side of each non-conforming rule is checked for terminal variables, if they exist a new rules is created with the method createRuleFromRightHandSide. The left side of the new rule is used to replace the terminal symbol.

public Rule createRuleFromRightHandSide(String tempRightHandSide, HashSet<Rule> repeatableRules, ContextFreeGrammar cfg) {…}

createRuleFromRightHandSide checks that if the rule exists in the HashSet repeatebleRules it just return the repeatableRule. But if it doesn't exist, it adds the new rule to rules then return the rule.

The right side of the non-conforming rule is then sent to a method called groupCharactersByTwo as a string.

public String groupCharactersByTwo(char leftHandSide, String str, int length, HashSet<Rule> repeatableRules, ContextFreeGrammar cnf) {…}

groupCharactersByTwo is a recursive method. It takes the first to symbols of the string and call itself for the rest of the string and concatenates the return values. Then creates a rule with the createRuleFromRightHandSide, then concotanes the new left side to the return value. If the length of the string is just one, it concatenates the string to the return value. If the return value is bigger than two it calls itself with the return value. This method groups all non -terminals by two.

The result is then sorted using the stream method using the compare method in RuleSorter class. RuleSorter has a constructor which takes the start variable, rules, and non-terminal symbols. They are used to create a character list called variableOrder. VariableOrder is filled in a while loop fist by finding all the variables in the right side of the rules with the start variable on the left side. This repeats for all variables. Compare method uses variableOrder to sort the rules.

toString methods of these classes are overridden, so when printing the new cnf object, the result will be printed.

4. Results

G1.txt

G1 input from file:

NON-TERMINAL
S
F
TERMINAL
0
1
RULES
S:00S
S:11F
F:00F
F:e
START
S

G1 output to console:

NON-TERMINAL
B
S
D
U
F
Z
L
TERMINAL
0
1
RULES
B:DD
B:LF
B:ZS
D:1
L:DD
F:UU
F:ZF
Z:UU
S:DD
S:LF
S:ZS
U:0
START
B

G2.txt

G2 input from file:

NON-TERMINAL
S
A
B
TERMINAL
a
b
RULES
S:a
S:aA
S:B
A:aBB
A:e
B:Aa
B:b
START
S

G2 output to console:

NON-TERMINAL
A
B
J
L
V
TERMINAL
a
b
RULES
L:AV
L:VA
L:a
L:b
A:JB
V:a
J:VB
B:AV
B:a
B:b
START
L