Problem

Convert a given Context-Free Grammar (CFG) to a Chomsky Normal Form (CNF).

Problem Breakdown

The input file is given in a specific format which first needs to be read. The file contains the rules for CFG which the program shall convert into a CNF.

Implementation

The program is implemented using Java programming language. There is only one file which contains two java classes, one for reading the input file and storing the information into relevant data types and the other class which also utilizes the main method for the program to run, implements the program logic. The program logic follows the steps required to convert a given CFG to CNF.

Steps for converting a given CFG to CNF are as follows:

1. Introducing a new start.
    a. Check the given CFG rules, if a new start state needs to be introduced, introduce the new start. This can be done by searching the rules if the start leads to the start in any of the rules.
2. Removing Null Production.
    a. Any rule leading to an epsilon is considered as null production. Therefore, such rules are required to be removed. For removing these rules, remaining possible non-terminals/terminals are written in place of the epsilon.
3. Removing Unit Rules.
    a. According to the CNF, we cannot have any rule of the type A->A. Such rules are called unit rules. Therefore, in the third step for converting a given CFG to CNF, such rules are removed. For removing the unit rules, first the unit rules are identified and then for those rules, a terminal rule is found. Then the unit rule is replaced with the rule leading to a terminal.
    b. At the end of if there are any unreachable terminal rules they are removed.
4. Correct rules with non-terminals more than two.
    a. For a grammar to be in CNF it should follow the CNF rule which states that a rule cannot be of the form A -> BCD. A rule should only have two non-terminals or one terminal on the right side.
    b. In this step rules are searched and then if there isn't a rule that exists a new rule is generated. For the above given example, the new rules will be A -> BE where E -> CD. This way both rules will stand CNF.
5. Correct rules for terminals.
    a. For this step, if there is any rule of the form A -> aB which violates the CNF, we introduce a new rule if a rule that leads to terminal a does not exist already. So that the new rules will be of the form A -> CB where C -> a.

Code Explanation

The first class in the file is FileReader class which reads the file upon passing the File object which contains the file directory URL. It then reads the file into appropriate data types mainly *int, String* and *ArrayList<String>.*

The second class, which is also the main class for the program, is CFGtoCNF class. Starting with the constructor, the class invokes the constructor of the FileReader class for reading the given input file. Then the constructor invokes the functions in the order of the CGF to CNF conversion with an additional output function which outputs the resultant CNF to the console in the format stated in the project brief.

As mentioned previously, the first step for converting the CFG to CNF, a check is needed for adding a new start state. So, by utilizing the *checkStartStateNeeded()* function in the *addNewStartState()* function it is verified if a new start state is needed. The *checkStartStateNeeded()* returns a Boolean by going through the rules and checking if the original start state is found in any rules on the right hand side. If it is needed later function adds a new start state named as "R" and saves it to the class private variable as newStart and adds to the newNonTermianls list. Here the new start state is always "R" is hardcoded.

After adding a new start state, the next step is removing the null productions. For this step, *removeNullProduction()* method first invokes the *listOfNonTerminalsLeadingToEpsilon()* function which returns and ArrayList<String> of rules. The method makes this list by tracking the epsilon to the non-terminals. All the rules which lead to epsilon are added to this list. Since these rules are to be eliminated, the *removeNullProduction()* method then figures the rules that are to be replaced, by comparing the right hand side equals to "e" then it loops until the size of list with non-terminals leading to epsilons and checks if the right hand side of the rule matches the right hand side of the non-terminals leading to epsilon, another method to get all possible combinations is invoked which is passed the current rule iterating in the first loop and also the non-terminal leading to epsilon found in the sub-loop (this needs to be eliminated for). The result of this method call is added to an ArrayList<Sting> of productionToBeAdded. All the values in the productionToBeAdded are then added to the newRulesToBeReplaced. Here productionToBeAdded certainly can be omitted but for the ease of my own understanding I have used an extra ArrayList. The newRulesToBeRepalced might contains redundant rules, which are then removed and added to the private newRules list of the class. At this stage all the null productions are removed from the given CFG.

The third step is to remove unit rules. The main method for removing unit rules is *removeUnitProductions()*. This method first calls the *isRuleUnitProduction()* with parameter as the current rule in the loop and gets a Boolean value in return. If the value is true, the current rule in the iteration is stored in a list for the rules to be removed and a list of rules to add is also maintained. The later list gets all the relevant terminal rules by invoking the method *findTerminalForUnitProduction()*. As the name of the method suggests, it finds a non-terminal for the terminal on the right-hand side of the rule. After this, the function updates the newRules with the rules that needs to be added and remove ones that needs to be removed as found earlier. It then removes the redundant rules and in the end removes unreachable rules by calling *removeUnreachable().*

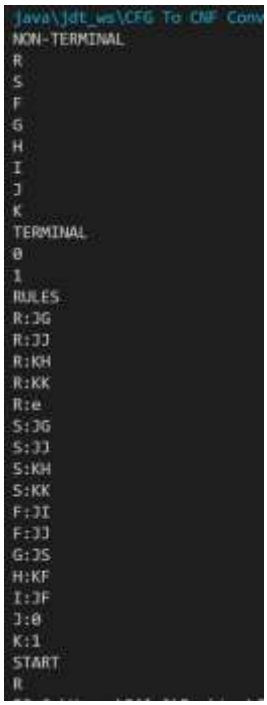
In the forth step, the main function is *removeNonTerminalsGreatThan2()* which utilizes *modifyRuleForNonTerminal().* The first function finds the rules that need to be removed by checking if there exist non-terminals of length greater than 2 on the right-hand side of the rule and maintains a list

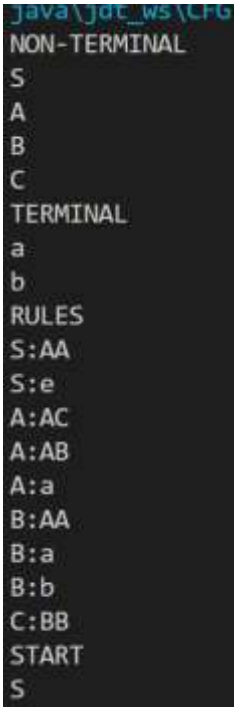
of those rules. Then it for the size of the rules to be removed, runs a for loop, which finds rules to be added. It searches the rules if there exists a rule which contains two non-terminals that need replacing it, gets that rule and adds to the list of the rules that need to be added. If there isn't a rule that already exists, it generates a new non-terminal and assigns the 2 non-terminals to the newly generated non-terminal. At the end it calls *modifyRuleForNonTerminal()* and passes both lists as parameters. The *modifyRuleForNonTerminal()* returns a list of rules after removing and adding new ones without redundancy.

The last step of converting CFG to CNF is correcting rules for the terminals. The main function of this step is *modifyRuleForTerminal()* which runs a loop for the size of the *newRules* then for each rule if the rule has both terminal and non-terminal on the right-hand side, if the rule has both terminal and non-terminal then it checks if the first character on the right is lower case or the second sets the flag true for that character. Then the function checks if a non-terminal leading to terminal exists, if it doesn't a new non-terminal symbol is created, and the terminal is added to the rule otherwise the existing rule at the i-th position is removed and corrected rule is added to the rule list.

modifyRulesIfHasTerminalsGreaterThanOne() is an additional method. If there is any rule containing a rule of type A -> aa, it corrects such rules by first checking if a rule exists such as B -> a then it changes the rule A -> BB otherwise it creates a new non-terminal symbol and corrects the rule.

Lastly, in case if introduced new start state the rule for this state will be like the ones to the old start state. So, this function copies those rules to the new start symbol. With this the conversion from CFG to CNF is completed and the CNF is now stored in newRules list. The output method then displays the information in the required format on console.





The outputs of the given G1.txt and G2.txt respectively.