THEORY OF COMPUTATION HOMEWORK

Description:

1. Summary

The aim of the project is to convert Context Free Grammar to Chomsky Normal Form. The input of the program is text file (CFG.txt). The program includes four main steps.

1. Eliminate empty character (remove epsilon symbol)
2. Eliminate unit production (and remove useless productions)
3. Eliminate terminals
4. Break variable strings longer than 2

We have completed these steps required in our project and in this section we will give information about what the project does and the special working principles. After the reading from the text file was completed, we moved on to the first step, the empty character elimination step, in this section, we removed the empty string(€) parts in the non-terminal and added the values that can be taken into the entire non-terminal value containing the empty string, and we removed the empty string part from the existing structure with "removeEpsilon" function, this was the first step we were expected to show on the output when converting from "CFG" to "CNF".

Then we started to do the eliminate unit production part of step 2, in this section, we were asked to eliminate the stand-alone non-terminal values and assign them to their non-terminal values, in the code, we did this in the "removeSingleVariable" function, in this function, we created two different ArrayList, "keySet" and "productionList", and we added the value we received with the "next" function over "Iterator" and we applied "keySet" to "mapVariableProduction" that we created in HashMap type and assign it to "set" variable. Then we checked the "size", "temp" and "productionList.get(i)" values, removing the non-terminal value and adding a new value instead.

The third step we were expected to show on the output was the eliminate terminals section, in this section, we were asked to make the whole structure non-terminal by eliminating terminal values. We did the elimination of non-terminal values by using the twoTerminalandOneVariable function, in this section we added it to the variable "newProduction" with the "substring", taking it into if according to the length of the temp value. While performing the elimination process, we first created the variables corresponding to the terminal values as new variables in our code, then in the next section we converted them to non-terminal variables with all the variables.

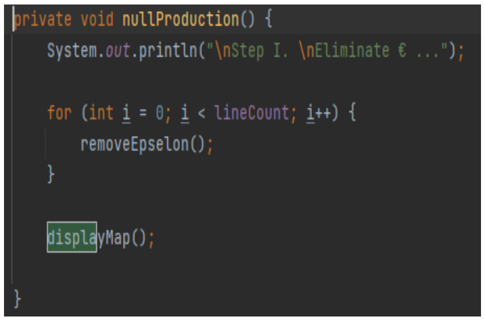
When we came to the fourth and final step, we had already made the necessary arrangements for this step in the eliminate terminals section. We have created each string value according to the elements it contains. We checked the values in these values with "if (productionList.size() > 1)" and we created and checked a separate if for each "temp.length()" state, we checked this part in "removeThreeTerminal()" function. As a result of this part, we have applied the break variable strings longer than 2 condition in our project. As a result of we did Terminal values in string are converted to non-terminal values and string values longer than 2 are converted, In this way, the final version of our output section for the project was formed. The conversion from "CFG" to "CNF" is successful.

2. Explanation of Steps of Program

We focused on the first part requested from us in order to be able to implement these steps before coming to the steps requested from us in the project, generally, in the assignment we` were asked to take a CFG type structure and convert it to CNF and we needed to get these CFG structures from a "CFG.txt" file then we had to convert these structures we received to CNF according to the specified steps. Therefore, we started creating the structure by reading the values in the given text file. To return how many lines are in the entered file, we entered the file name into the “countLineNumberReader” function and kept the return value in a variable.

We then create an array that will have the same length as this variable and with the file read command, we put each line we read in the file into this array, respectively, so that the values in the file were in the array line by line. Then, there was a part that should come to the output screen just before the step parts, so we printed this part first in the main part. In this section, we were asked for show the values in the text file under the CFG Form heading. That's why we printed the structures that we read from the text file and put them in the array and wrote them to the screen in a for loop. Then we started printing on the screen by following the steps.

a. Eliminate empty characters **(removeEpsilon Function):**

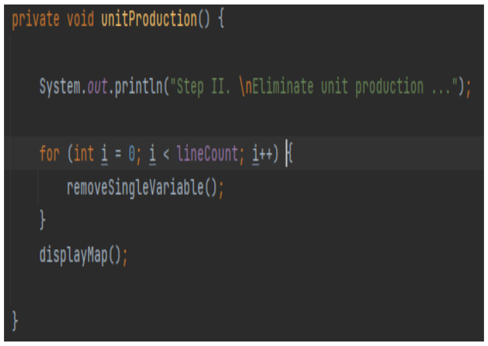


```
private void nullProduction() {
    System.out.println("\nStep I. \nEliminate € ...");

    for (int i = 0; i < lineCount; i++) {
        removeEpselon();
    }

    displayMap();

}
```

The first step includes another function that searches every single row and detects empty characters. The function has two main parts. Both of them start while loop. First loop does not determine length or size. However second while loop relevant size. The first loop detects empty symbol (€) . If the function detects empty symbol, remove that and create new string which is epsilonFound. epsilonFound function store main symbol. In other words, if any symbol contains empty symbol, epsilonFound store which symbol or symbols contain empty symbol. Second while loop finds row contains symbols that contains empty symbol. If the length of line is longer than one, the function removes only empty characters. Else, remove all lines. After removing empty character, edits rest of lines, if they contain symbol that returns empty character. After all procedure, program runs next function. The method first iterates over the entries in the map using an iterator itr, and checks if any of the production rules contain the symbol "€". If a production rule contains "€", and it has more than one element, the "€" symbol is removed from the rule. If the production rule has only one element, the entire entry is removed from the map. The method then iterates over the entries in the map again using an iterator itr2, and for each entry, it checks the production rules for the presence of the "€" symbol. If the "€" symbol is found in a production rule, the method modifies the production rule by either removing the "€" symbol or adding a new production rule that does not contain the "€" symbol.

The mapVariableProduction map contains the production rules for the variables in the form of key-value pairs, where the keys are the variables and the values are lists of production rules in the form of strings. The epsilonFound variable is a string that stores the variable for which a production rule containing "€" was found. The productionRow and productionList variables are lists of production rules in the form of strings. The itr and itr2 variables are iterators used to iterate over the entries in the mapVariableProduction map. The entry variable is a Map.Entry object that represents an entry in the map. The "i" and "j" variables are loop counters.

b. Eliminate unit production **(unitProduction Function):**



```
private void unitProduction() {

    System.out.println("Step II. \nEliminate unit production ...");

    for (int i = 0; i < lineCount; i++) {
        removeSingleVariable();
    }
    displayMap();

}
```

The second step includes another function too. That function removes single variable. If a symbol returns just one symbol, function replaces that symbol to other's line. So, unit productions unnecessarily increase the number of steps in a derivation. It returns the removeSingleVariable function inside the unitProduction function in the for loop, so we will explain the removeSingleVariable function. The method first iterates over the entries in the map using an iterator itr4, and then creates a list of the keys in the map called keySet. The method then iterates over the production rules for the current entry using a loop counter i, and for each production rule, it iterates over the characters in the rule using a loop counter j. the method iterates over the keys in the keySet list using a loop counter k, and checks if the character is equal to one of the keys. If the character is equal to a key, the method retrieves the production rules for that key using the mapVariableProduction map and store them in a list called productionValue.

The method then iterates over the production rules in the productionValue list using a loop counter l, and checks if any of the production rules contain the key. The mapVariableProduction map contains the production rules for the variables in the form of key-value pairs, where the keys are the variables and the values are lists of production rules in the form of strings. The key and key2 variables are strings that store the keys for the entries in the mapVariableProduction map.

c. Eliminate terminals **(twoTerminalandOneVariable Function):**
Any Context Free Grammar needs to a one terminal or two non-terminal symbols.
So, if there is more than one terminal symbol, the program must convert terminal values to non-terminal. Step 3 works like that. Function uses iterator to check every line. And the function search single line. If there is more than one terminal, the program creates a new non-terminal symbol. The function starts generating ASCII code 77 and every time that needs to new symbol, the function generates new symbol. The ASCII code of that new symbol is the next ASCII code of the last created one. And the function replaces all terminal values that every line of Context Free Grammar with the created non-terminal symbol.

The checkDuplicateInProductionList() method is used to check if a given production already exists in either the tempList map or the mapVariableProduction map. If it does, it means that the production has already been added to the CFG and there is no need to add it again. The asciiBegin variable is used to generate unique non-terminal symbols by converting ASCII values to characters. It starts at the ASCII value of (M=77) and is incremented by 1 for each new non-terminal symbol that is added to the CFG.

d. Break variable strings longer than 2 **(removeThreeTerminal Function):**
Other rule of Chomsky normal form is, there is just two non-terminal symbols. If there are more than two non-terminals in each line, the program must create another non-terminal that represents two non-terminals. The fourth step of this program considers that requirement. This function includes two iterators and checks all combinations of symbols. Store that values help with ArrayList. The method first iterates through the mapVariableProduction object to obtain a list of keys. The method then iterates through the mapVariableProduction object again and checks if a production has two or more

terminals If it does, the method will try to replace one or more of these terminals with a variable from keyList.

The replacement is done by checking if a substring of the production is equal to the value of a key in keyList. If it is, the method will replace the substring with the key and add the modified production back to the mapVariableProduction object. If the production has only two terminals, the method will check each terminal individually to see if it can be replaced with a key from keyList. The method will continue this process until it has processed all productions in mapVariableProduction.

Description of our data structure **(HashMap):**

In Java, a HashMap is a map implementation that uses a hash table to store key-value pairs. It allows you to store keys of any object type and retrieve the corresponding values. It uses a technique called hashing to store the key-value pairs in a way that allows for fast insertion, deletion, and retrieval of values based on their keys. When you create a HashMap, you specify the types of the keys and values that it will store. For example, you might create a HashMap<String, Integer> to store Strings as keys and Integers as values, or a HashMap<Person, String> to store Person objects as keys and Strings as values. To add a key-value pair to a HashMap, you can use the put method HashMap is an efficient data structure for storing and retrieving key-value pairs, but it does not maintain the order of the elements.

Pseudocode:

Input: CFG is written in “CFG.txt”

Output: Chomsky Normal Form that converted from input CFG

1. Set variables of CFG up. (grammar is kept as Map<String, List<String>>)
2. Read .txt file and implement Map.
3. ConvertToCNF

a. convertToMap()

```
# Split the input string on newline characters
splitedEnterInput = splitEnter(input)
# Iterate through all the elements in the splitedEnterInput list
for i in range(splitedEnterInput.length):
  # Split the current element on the "-" and "|" characters
  tempString = splitedEnterInput[i].split("-|\\|")
  # Get the first element (variable)
  variable = tempString[0].trim()
```

```
        # Get the rest of the elements (productions)
        production = Arrays.copyOfRange(tempString, 1,
    tempString.length)
        # Initialize an empty list to store the productions
        productionList = []
        # Trim the empty space from each production
        for k in range(production.length):
          production[k] = production[k].trim()
        # Add the productions to the productionList
        for j in range(production.length):
          productionList.add(production[j])
        # Add the variable and productionList to the
    mapVariableProduction map
        mapVariableProduction[variable] = productionList
```

b. removeEpsilon()

```
# Create an iterator for the mapVariableProduction map
itr = mapVariableProduction.entrySet().iterator()

# Iterate over the key-value pairs in the map
while itr.hasNext():
    # Get the current key-value pair
    entry = itr.next()
    # Get the value (a list of productions) for the current key
    productionRow = entry.getValue()

    # If the production row contains the character '€' (epsilon)
    if "€" in productionRow:
        # If the production row has more than one element
        if len(productionRow) > 1:
            # Remove the '€' character from the production row
            productionRow.remove('€')
            # Set epsilonFound to the current key
            epsilonFound = entry.getKey().toString()
        else:
            # Set epsilonFound to the current key
            epsilonFound = entry.getKey().toString()
            # Remove the current key-value pair from the map
            mapVariableProduction.remove(epsilonFound)

# Create an iterator for the mapVariableProduction map
itr2 = mapVariableProduction.entrySet().iterator()
```

```
   # Iterate over the key-value pairs in the map
while itr2.hasNext():
    # Get the current key-value pair
    entry = itr2.next()
    # Get the value (a list of productions) for the current key
    productionList = entry.getValue()

    # Iterate over the productions in the production list
    for i in range(0, len(productionList)):
        # Get the current production
        temp = productionList[i]

        # Iterate over the characters in the production
        for j in range(0, len(temp)):
            # If the current character is equal to epsilonFound
            if epsilonFound == temp[j]:
                # If the length of the production is 2
                if len(temp) == 2:
                    # Remove the epsilonFound character from the production
                    temp = temp.replace(epsilonFound, "")
                    # If the modified production is not in the production list for the
current key, add it
                    if temp not in
mapVariableProduction.get(entry.getKey().toString()):

mapVariableProduction.get(entry.getKey().toString()).add(temp)
```

c. removeSingleVariable()

```
# Create an iterator for the mapVariableProduction map
itr4 = mapVariableProduction.entrySet().iterator()
# Initialize the key, key2, and keyHolder variables to null
key = null
key2 = null
keyHolder = ""

# Iterate over the key-value pairs in the map
while itr4.hasNext():
    # Get the current key-value pair
    entry = itr4.next()
    # Get the set of keys in the map
    set = mapVariableProduction.keySet()
```

```
        # Convert the key set to an ArrayList called keySet
        keySet = ArrayList(set)
        # Get the value (a list of productions) for the current key
        productionList = entry.getValue()
              # Iterate over the productions in the production list
        for i in range(0, len(productionList)):
            # Get the current production
            temp = productionList[i]
            # Iterate over the characters in the production
            for j in range(0, len(temp)):
                # Iterate over the keys in the keySet
                for k in range(0, len(keySet)):
                    # If the current key is equal to the production
                    if keySet[k] == temp:
                        # Set key to the current key
                        key = entry.getKey().toString()
                        # Get the value (a list of productions) for the key temp
                        productionValue = mapVariableProduction.get(temp)

                        # Iterate over the productions in the productionValue list
                              for l in range(0, len(productionValue)):
                            # Set key2 to the current production
                            key2 = productionValue[l]
                            # If the production contains the key
                            if key in productionValue[l]:
                                # Replace the key with the temp in the key2 production
                                keyHolder = key2.replace(key, temp)
                          # Add the modified production to the value list for the key
temp
                                mapVariableProduction.get(temp).add(keyHolder)
                        # Remove the temp production from the productionList for the
current key
                        productionList.remove(temp)
```

d. twoTerminalandOneVariable()

```
        # Create an iterator for the mapVariableProduction map

        itr5 = mapVariableProduction.entrySet().iterator()

        # Initialize a key variable to null
```

```
key = None
# Set the ASCII value for the beginning character to 77 (C)
asciiBegin = 77
# Initialize an empty map to store new variables
tempList = {}

while itr5.hasNext():
  # Get the current entry
  entry = itr5.next()
  # Get the key set for the mapVariableProduction map
  keySet = set(mapVariableProduction.keys())
  # Get the value (production list) for the current entry
  productionList = entry.getValue()
  # Initialize found1, found2, and found flags to false
  found1 = False
  found2 = False
  found = False

  # Iterate through all the elements in the production list
  for i in range(productionList.size()):
    # Get the current element
    temp = productionList[i]
    # Iterate through all the characters in the element
    for j in range(temp.length()):
      # If the element has three characters
      if temp.length() == 3:
        # Get the middle two characters from the element
        newProduction = temp[1:3]
        # If the new production is not a duplicate in the tempList map or the mapVariableProduction map
        if checkDuplicateInProductionList(tempList, newProduction) and checkDuplicateInProductionList(mapVariableProduction, newProduction):
          # Set the found flag to true
          found = True
        else:
          # Set the found flag to false
          found = False
        # If the found flag is true
        if found:
          # Create a new variable with the new production as its value
          newVariable = [newProduction]
          # Set the key to the current ASCII character
          key = chr(asciiBegin)
          # Add the new variable to the tempList map
          tempList[key] = newVariable
          # Increment the ASCII value
          asciiBegin += 1
      # If the element has two characters
      elif temp.length() == 2:
        # Iterate through all the keys in the keySet
        for k in range(keySet.size()):
          # If the current character is not equal to the current key
          if keySet[k] != temp[j]:
            # Set the found flag to false
            found = False
          else:
                  # Set the found flag to true
                  found = True
                  # Break out of the loop
                  break
              # If the found flag is false
              if not found:
                # Get the current character
                newProduction = temp[j]
                # If the new production is not a duplicate in the tempList map or
      the mapVariableProduction map
                if checkDuplicateInProductionList(tempList, newProduction) and
      checkDuplicateInProductionList(mapVariableProduction,
      newProduction):
                  # Create a new variable with the new production as its value
                  newVariable = [newProduction]
                  # Set the key to the current ASCII character
                  key = chr(asciiBegin)
```

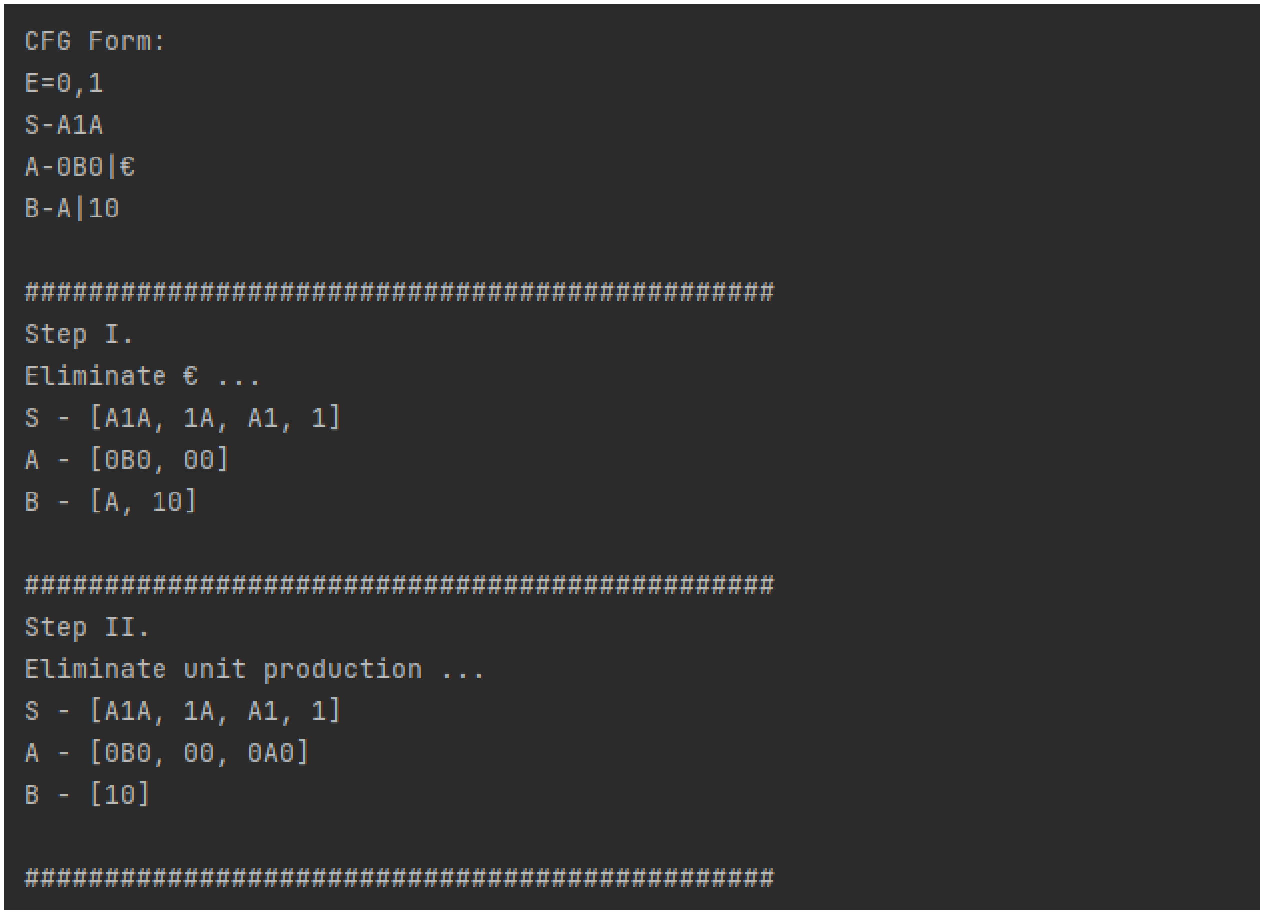
e. removeThreeTerminal()

```
    # Create an iterator for the mapVariableProduction map
itr = mapVariableProduction.entrySet().iterator()
# Initialize an empty list to store keys
keyList = []
# Create another iterator for the mapVariableProduction map
itr2 = mapVariableProduction.entrySet().iterator()

# Obtain keys that are used to eliminate two terminals and above
while itr.hasNext():
  # Get the current entry
  entry = itr.next()
  # Get the value (production row) for the current entry
  productionRow = entry.getValue()
  # If the production row has less than two elements
  if productionRow.size() < 2:
      # Add the key to the keyList
      keyList.add(entry.getKey())

  # Find more than three terminal or combination of variable and terminal to
eliminate them
  while itr2.hasNext():
    # Get the current entry
    entry = itr2.next()
    # Get the value (production list) for the current entry
    productionList = entry.getValue()
    # If the production list has more than one element
    if productionList.size() > 1:
      # Iterate through all the elements in the production list
      for i in range(productionList.size()):
        # Get the current element
        temp = productionList[i]
        # Iterate through all the characters in the element
        for j in range(temp.length()):
          # If the element has more than two characters
          if temp.length() > 2:
            # Split the element into two parts
            stringToBeReplaced1 = temp[:temp.length() - j]
            stringToBeReplaced2 = temp[j:]
            # Iterate through all the keys in the keyList
            for key in keyList:
              # Get the values for the current key
              keyValues = mapVariableProduction[key]
              # Get the first value in the keyValues list
              value = keyValues[0]
              # If the first part of the element matches the value
              if stringToBeReplaced1 == value:
                # Remove the element from the production list
                mapVariableProduction[entry.getKey()].remove(temp)
                # Replace the first part of the element with the key
                temp = temp.replace(stringToBeReplaced1, key)
                # If the modified element is not already in the production list
                if temp not in mapVariableProduction[entry.getKey()]:
                  # Add the modified element to the production list
                  mapVariableProduction[entry.getKey()].add(i, temp)
              # If the second part of the element matches the value
              elif stringToBeReplaced2 == value:
                # Remove the element from the production list
                mapVariableProduction[entry.getKey()].remove(temp)
```

```
# Replace the second part of the element with the key
temp = temp.replace(stringToBeReplaced2, key)
# If the modified element is not already in the production list
if temp not in mapVariableProduction[entry.getKey()]:
  # Add the modified element to the production list
  mapVariableProduction[entry.getKey()].add(i, temp)
```

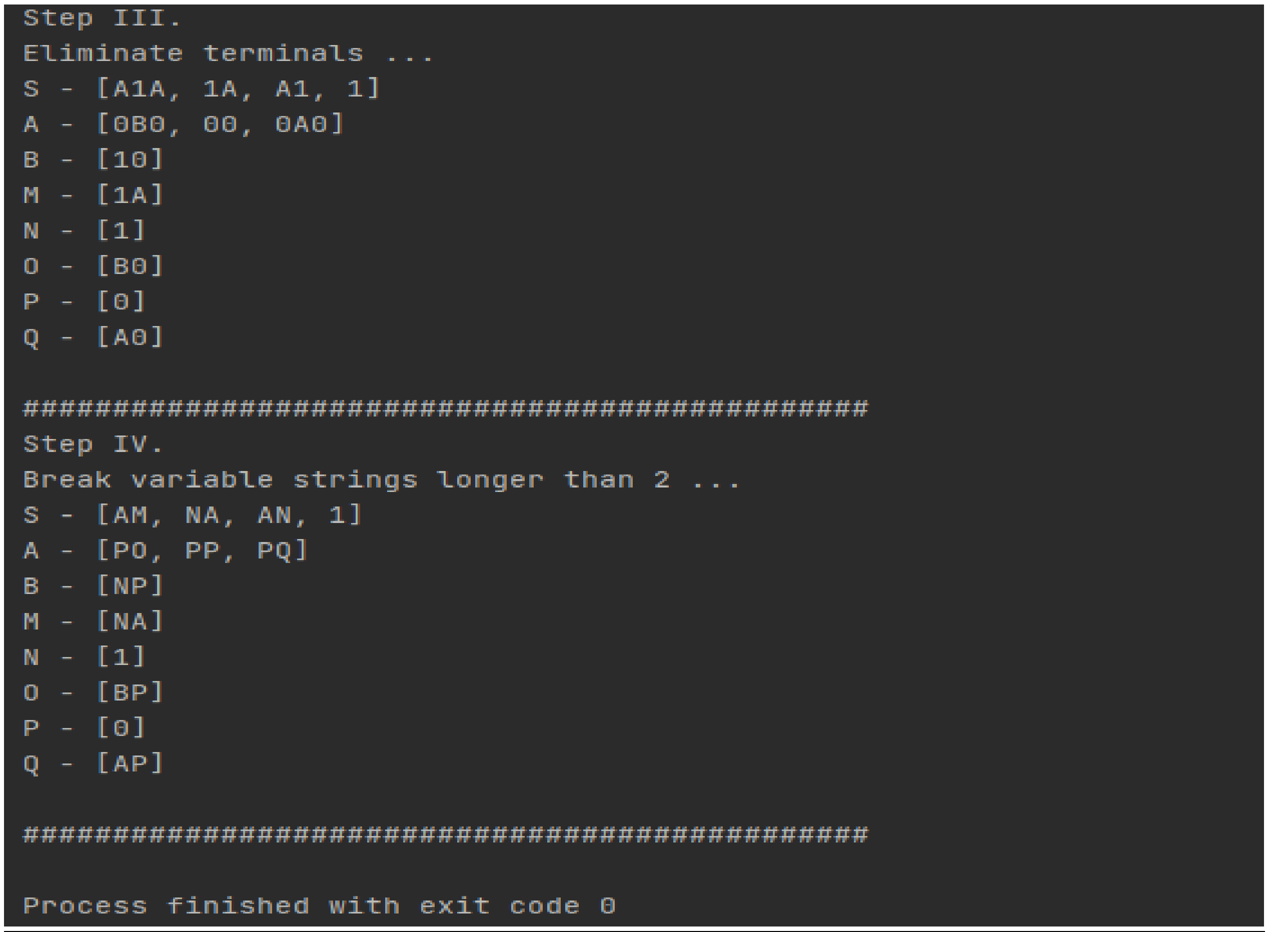
Screenshots:

Code-Part1:



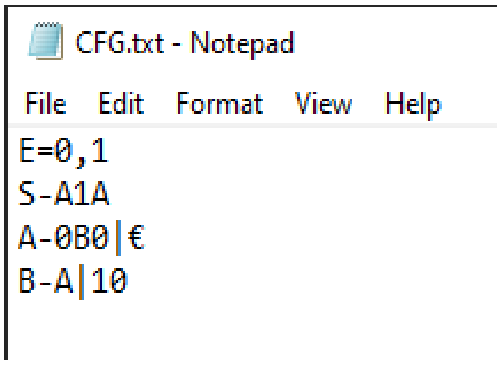
```
CFG Form:
E=0,1
S-A1A
A-0B0|€
B-A|10

################################################
Step I.
Eliminate € ...
S - [A1A, 1A, A1, 1]
A - [0B0, 00]
B - [A, 10]

################################################
Step II.
Eliminate unit production ...
S - [A1A, 1A, A1, 1]
A - [0B0, 00, 0A0]
B - [10]

################################################
```

Code-Part2:



```
Step III.
Eliminate terminals ...
S - [A1A, 1A, A1, 1]
A - [0B0, 00, 0A0]
B - [10]
M - [1A]
N - [1]
O - [B0]
P - [0]
Q - [A0]

###############################################
Step IV.
Break variable strings longer than 2 ...
S - [AM, NA, AN, 1]
A - [PO, PP, PQ]
B - [NP]
M - [NA]
N - [1]
O - [BP]
P - [0]
Q - [AP]

###############################################

Process finished with exit code 0
```

Text-File: (For our own example)



```
E=0,1
S-A1A
A-0B0|€
B-A|10
```

OKAN UÇAR 2019510079

BATUHAN DOĞAN 2019510027