```
import java.util.*;
public class CNFT {
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
// Input Non-Terminal and Terminal Symbols
System.out.print("Enter the NonTerminal Symbols (separated by spaces): ");
String[] NT_symbols = scanner.nextLine().split(" ");
System.out.print("Enter the Terminal Symbols (separated by spaces): ");
String[] T_symbols = scanner.nextLine().split(" ");
Set<String> terminals = new HashSet<>(Arrays.asList(T_symbols));
// Input Productions
Map<String, List<String>> productions = new HashMap<>();
for (String nt : NT_symbols) {
System.out.print(nt + " -> ");
String[] prodArray = scanner.nextLine().split("/");
productions.put(nt, new ArrayList<>(Arrays.asList(prodArray)));
}
// Eliminate Null Productions
eliminateNullProductions(productions);
// Eliminate Unit Productions
eliminateUnitProductions(productions, terminals);
// Convert to CNF
convertToCNF(productions, terminals);
// Display the final CFG after conversion to CNF
System.out.println("CFG in CNF:");
printCFG(productions);
scanner.close();
}
// Function to Eliminate Null Productions
static void eliminateNullProductions(Map<String, List<String>> productions) {
Set<String> nullable = new HashSet<>();
boolean changed;
// Identify Nullable Symbols
do {
changed = false;
for (Map.Entry<String, List<String>> entry : productions.entrySet()) {
String nt = entry.getKey();
List<String> prods = entry.getValue();
for (String prod : prods) {
if (prod.equals("^")) { // Nullable production
if (nullable.add(nt)) changed = true;
break;
}
boolean allNullable = true;
for (char symbol : prod.toCharArray()) {
```

```
if (!nullable.contains(String.valueOf(symbol))) {
allNullable = false;
break;
}
}
if (allNullable && nullable.add(nt)) changed = true;
}
}
} while (changed);
// Generate New Productions without Nullables
for (Map.Entry<String, List<String>> entry : productions.entrySet()) {
String nt = entry.getKey();
List<String> prods = entry.getValue();
Set<String> newProds = new HashSet<>();
for (String prod : prods) {
if (!prod.equals("^")) {
newProds.addAll(generateCombinations(prod, nullable));
}
}
productions.put(nt, new ArrayList<>(newProds));
}
}
// Function to Eliminate Unit Productions
static void eliminateUnitProductions(Map<String, List<String>> productions, Set<String>
terminals) {
Map<String, Set<String>> unitProds = new HashMap<>();
for (String nt : productions.keySet()) {
unitProds.put(nt, new HashSet<>());
}
// Find Unit Productions
for (String nt : productions.keySet()) {
for (String prod : productions.get(nt)) {
if (productions.containsKey(prod) && !terminals.contains(prod)) {
unitProds.get(nt).add(prod);
}
}
}
// Replace Unit Productions with Actual Productions
for (String nt : unitProds.keySet()) {
Set<String> newProds = new HashSet<>();
for (String unit : unitProds.get(nt)) {
newProds.addAll(productions.get(unit));
}
productions.get(nt).removeIf(unitProds.get(nt)::contains);
productions.get(nt).addAll(newProds);
```

```
}
}
// Convert to CNF
static void convertToCNF(Map<String, List<String>> productions, Set<String> terminals) {
Map<String, String> nonTerminalMap = new HashMap<>();
int counter = 1;
// Replace long productions with non-terminals
for (Map.Entry<String, List<String>> entry : productions.entrySet()) {
String nt = entry.getKey();
List<String> prods = entry.getValue();
List<String> newProds = new ArrayList<>();
for (String prod : prods) {
if (prod.length() > 2) {
StringBuilder newProd = new StringBuilder();
for (int i = 0; i < prod.length(); i += 2) {
String part = (i + 1 < prod.length()) ? prod.substring(i, i + 2) : prod.substring(i, i +
1);
if (part.length() == 2) {
String newNonTerminal = "X" + counter++;
nonTerminalMap.put(newNonTerminal, part);
newProd.append(newNonTerminal);
} else {
newProd.append(part);
}
}
newProds.add(newProd.toString());
} else {
newProds.add(prod);
}
}
productions.put(nt, new ArrayList<>(new HashSet<>(newProds)));
}
// Replace terminals with non-terminals
for (Map.Entry<String, List<String>> entry : productions.entrySet()) {
String nt = entry.getKey();
List<String> prods = entry.getValue();
List<String> newProds = new ArrayList<>();
for (String prod : prods) {
StringBuilder newProd = new StringBuilder();
for (char c : prod.toCharArray()) {
if (terminals.contains(String.valueOf(c))) {
String newNonTerminal = "X" + counter++;
nonTerminalMap.put(newNonTerminal, String.valueOf(c));
newProd.append(newNonTerminal);
} else {
```

```
newProd.append(c);
}
}
newProds.add(newProd.toString());
}
productions.put(nt, new ArrayList<>(new HashSet<>(newProds)));
}
// Add new non-terminals to productions
for (Map.Entry<String, String> entry : nonTerminalMap.entrySet()) {
String nt = entry.getKey();
String prod = entry.getValue();
productions.putIfAbsent(nt, new ArrayList<>(Arrays.asList(prod)));
}
}
// Generate Combinations by Eliminating Nullable Symbols
private static Set<String> generateCombinations(String prod, Set<String> nullable) {
Set<String> combinations = new HashSet<>();
List<Integer> positions = new ArrayList<>();
for (int i = 0; i < prod.length(); i++) {
if (nullable.contains(String.valueOf(prod.charAt(i)))) {
positions.add(i);
}
}
int n = positions.size();
for (int i = 0; i < (1 << n); i++) {
StringBuilder sb = new StringBuilder(prod);
for (int j = 0; j < n; j++) {
if ((i & (1 << j)) != 0) {
sb.setCharAt(positions.get(j), '^');
}
}
combinations.add(sb.toString().replace("^", ""));
}
return combinations;
}
// Function to Print CFG
static void printCFG(Map<String, List<String>> productions) {
for (Map.Entry<String, List<String>> entry : productions.entrySet()) {
String nt = entry.getKey();
List<String> prods = entry.getValue();
System.out.print(nt + " -> ");
System.out.println(String.join(" / ", prods));
}
}
}
```

OUTPUT:

```
Enter the NonTerminal Symbols (separated by spaces): A B C
Enter the Terminal Symbols (separated by spaces): a b
A -> BCA/a/BC/^
B -> CB
C -> bC/ba
CFG in CNF:
A -> BC / X1A / X2
B -> CB
C -> X4X5 / X3C
X1 -> BC
X2 -> a
X3 -> b
X4 -> b
X5 -> a
```

```
Enter the NonTerminal Symbols (separated by spaces): S A B
Enter the Terminal Symbols (separated by spaces): a b
S -> a/aA/B
A -> aBB/^
B -> Aa/b
CFG in CNF:
X8 -> b
A -> X1B
B -> AX2 / X3 / X4
S -> X8 / X5A / AX6 / X7
X1 -> aB
X2 -> a
X3 -> a
X4 -> b
X5 -> a
X6 -> a
X7 -> a
```

OBSERVATION:

In this experiment, I learned about how to eliminate null and unit production stepwise by applying rules and following sequential steps, how we can then convert into Chomsky Normal Form(CNF). For unit production finding derivables and then replace with its own production .

CONCLUSION:

The conversion of CFG to CNF was studied and implemented successfully.