MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

CEBAN VASILE FAF-223

Report

Laboratory work n.5

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

DISA, FCIM, UTM

Chișinău - 2024

Topic: Chomsky Normal Form

Objectives:

1. Learn about Chomsky Normal Form (CNF) [1].
2. Get familiar with the approaches of normalizing a grammar.
3. Implement a method for normalizing an input grammar by the rules of CNF.
    a. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
    b. The implemented functionality needs executed and tested.
    c. A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
    d. Also, another BONUS point would be given if the student will make the aforementioned function to accept any grammar, not only the one from the student's variant.

Converting Context Free Grammar to Chomsky Normal Form

A context free grammar (CFG) is in Chomsky Normal Form (CNF) if all production rules satisfy one of the following conditions:

- A non-terminal generating a terminal (e.g.; X->x)
- A non-terminal generating two non-terminals (e.g.; X->YZ)
- Start symbol generating ε. (e.g.; S-> ε)

Consider the following grammars,

G1 = {S->a, S->AZ, A->a, Z->z}

G2 = {S->a, S->aZ, Z->a}

The grammar G1 is in CNF as production rules satisfy the rules specified for CNF. However, the grammar G2 is not in CNF as the production rule S->aZ contains terminal followed by non-terminal which does not satisfy the rules specified for CNF.

Note –

- For a given grammar, there can be more than one CNF.
- CNF produces the same language as generated by CFG.

- CNF is used as a preprocessing step for many algorithms for CFG like CYK(membership algo), bottom-up parsers etc.
- For generating string w of length ‘n’ requires ‘2n-1’ production or steps in CNF.
- Any Context free Grammar that do not have ε in it’s language has an equivalent CNF.

Source Code:

```
public void convertToChomskyNormalForm() {
   Map<String, String> terminalNonTerminalsMap = new HashMap<>();
   Map<String, String> newNonTerminalsMap = new HashMap<>();
   Map<String, String> nonTerminalMapping = new HashMap<>();
   int newNonTerminalCounter = 0;

   Map<String, List<String>> updatedRules = new HashMap<>(getRules());

   for (Map.Entry<String, List<String>> entry : updatedRules.entrySet()) {
       String fromState = entry.getKey();
       List<String> toStates = entry.getValue();
       List<String> newToStates = new ArrayList<>();
       for (String production : toStates) {
           if (production.length() > 2) {
               String newNonTerminal = nonTerminalMapping.get(production);
               if (newNonTerminal == null) {
                   newNonTerminal =
generateNewNonTerminal(newNonTerminalsMap, newNonTerminalCounter++,
production);
                   newNonTerminalsMap.put(newNonTerminal,
production.substring(1));
                   nonTerminalMapping.put(production, newNonTerminal);
               }
               newToStates.add(production.charAt(0) + newNonTerminal);
           } else {
               newToStates.add(production);
           }
       }
       updatedRules.put(fromState, newToStates);
   }

   for (Map.Entry<String, String> entry : newNonTerminalsMap.entrySet()) {
       updatedRules.put(entry.getKey(), List.of(entry.getValue()));
   }

   for (Map.Entry<String, List<String>> entry : updatedRules.entrySet()) {
       String fromState = entry.getKey();
       List<String> toStates = entry.getValue();
       List<String> newToStates = new ArrayList<>();
```

```
        for (String production : toStates) {
            if (production.length() == 2 &&
Character.isLowerCase(production.charAt(0)) &&
Character.isUpperCase(production.charAt(1))) {
                String terminal = production.substring(0, 1);
                String nonTerminal =
terminalNonTerminalsMap.getOrDefault(terminal, null);
                if (nonTerminal == null) {
                    nonTerminal = generateNewNonTerminal(newNonTerminalsMap,
newNonTerminalCounter++, terminal);
                    terminalNonTerminalsMap.put(terminal, nonTerminal);
                }
                newToStates.add(nonTerminal + production.charAt(1));
            } else {
                newToStates.add(production);
            }
        }
        updatedRules.put(fromState, newToStates);
    }

    for (Map.Entry<String, String> entry : newNonTerminalsMap.entrySet()) {
        String nonTerminal = entry.getKey();
        String terminal = entry.getValue();
        updatedRules.put(nonTerminal, List.of(terminal));
    }

    setRules(updatedRules);
}
```

Description:

This method converts the grammar to Chomsky Normal Form. It replaces long productions (> 2 symbols) with new non-terminals, replaces single terminal productions with new non-terminals, and adds all new non-terminals to the set of rules.

```
public void removeUnitProductions() {
    Map<String, List<String>> updatedRules = new HashMap<>(getRules());
    boolean changed;
    do {
        changed = false;
         for (Map.Entry<String, List<String>> entry : updatedRules.entrySet())
{
            String fromState = entry.getKey();
            List<String> toStates = entry.getValue();
            List<String> newToStates = new ArrayList<>();
            for (String production : toStates) {
                                        if   (production.length()   ==   1   &&
Character.isUpperCase(production.charAt(0))) {
                                            List<String>   unitProductions   =
updatedRules.get(production);
                    if (unitProductions != null) {
                        newToStates.addAll(unitProductions);
                        changed = true;
                    }
```

```java
                } else {
                    newToStates.add(production);
                }
            }
            updatedRules.put(fromState, newToStates);
        }

        for (Map.Entry<String, List<String>> entry : updatedRules.entrySet())
{
            String fromState = entry.getKey();
            List<String> toStates = entry.getValue();
            List<String> newToStates = new ArrayList<>();
            for (String production : toStates) {
                                            if   (!(production.length()   ==   1   &&
Character.isUpperCase(production.charAt(0)))) {
                    newToStates.add(production);
                }
            }
            updatedRules.put(fromState, newToStates);
        }
    } while (changed);
    setRules(updatedRules);
}
```

Description:
This method removes unit productions from the grammar. It iteratively removes unit productions by replacing them with their corresponding productions until no more unit productions remain.

```java
private String generateNewNonTerminal(Map<String, String> newNonTerminals, int
counter, String symbols) {
 return newNonTerminal;
}
```

Description:
This method generates a new non-terminal symbol for use in the conversion process. It ensures that each new non-terminal has a unique identifier and returns the generated non-terminal.

```java
public void removeEpsilonProductions() {
   Map<String, List<String>> updatedRules = new HashMap<>(getRules());

   Set<String> epsilonStates = new HashSet<>();
   for (Map.Entry<String, List<String>> entry : updatedRules.entrySet()) {
       if (entry.getValue().contains("")) {
           epsilonStates.add(entry.getKey());
       }
   }

   for (String state : epsilonStates) {
       List<String> productions = updatedRules.get(state);
```

```
        List<String> updatedProductions = new ArrayList<>(productions);
        updatedProductions.remove("");
        updatedRules.put(state, updatedProductions);
    }

    for (String stateWithEpsilon : epsilonStates) {
         for (Map.Entry<String, List<String>> entry : updatedRules.entrySet())
{
            String fromState = entry.getKey();
            List<String> toStates = entry.getValue();
            List<String> newToStates = new ArrayList<>();
            for (String production : toStates) {
                if (production.contains(stateWithEpsilon)) {
                                         List<String>  replicatedProductions  =
replicateProduction(production, stateWithEpsilon);
                    newToStates.addAll(replicatedProductions);
                } else {
                    newToStates.add(production);
                }
            }
            updatedRules.put(fromState, newToStates);
        }
    }

    for (Map.Entry<String, List<String>> entry : updatedRules.entrySet()) {
        String fromState = entry.getKey();
        List<String> toStates = entry.getValue();
        Set<String> uniqueToStates = new HashSet<>();
        for (String production : toStates) {
            generateCombinations("", production, 0, uniqueToStates);
        }
        updatedRules.put(fromState, new ArrayList<>(uniqueToStates));
    }

    setRules(updatedRules);
}
```

Description:
This method removes epsilon (empty string) productions from the grammar. It identifies states with epsilon productions and removes them from the grammar rules. Then, it replicates productions containing states with epsilon productions to account for their absence and generates all possible combinations of productions.

```
private    List<String>    replicateProduction(String    production,    String
stateWithEpsilon) {
    List<String> replicatedProductions = new ArrayList<>();
        int   numInstances   =   (int)   production.chars().filter(ch   ->   ch   ==
stateWithEpsilon.charAt(0)).count();
    for (int i = 0; i < Math.pow(2, numInstances); i++) {
        StringBuilder sb = new StringBuilder(production);
        for (int j = 0; j < numInstances; j++) {
```

```
            if ((i & (1 << j)) != 0) {
                int index = sb.indexOf(stateWithEpsilon);
                sb.replace(index, index + 1, "");
            }
        }
        replicatedProductions.add(sb.toString());
    }
    return replicatedProductions;
}
```

Description:
This method replicates productions containing states with epsilon productions to account for their absence. It generates all possible combinations of productions by replacing epsilon symbols with empty strings and returns the replicated productions.

Conclusion

In this laboratory work, we aimed to delve into the concept of Chomsky Normal Form (CNF) and explore the methods for normalizing a grammar according to CNF rules. The primary objectives included gaining an understanding of CNF, implementing a method to normalize an input grammar, and testing the functionality of the implementation.

Throughout the process, we learned about the importance of CNF in formal language theory and its significance in various computational tasks, such as parsing and manipulation of context-free grammars. By dissecting the steps required to convert a grammar into CNF, we gained insights into grammar manipulation techniques crucial for language processing tasks.

The implementation involved encapsulating the normalization functionality within a method and structuring it appropriately within a class. We ensured the versatility of the method by designing it to accept any grammar, not limited to a specific variant. This adaptability enhances the utility of the implementation, making it applicable to a wide range of grammatical structures.

Furthermore, we emphasized the significance of testing the implemented functionality. Unit tests were employed to validate the correctness and robustness of the normalization process, ensuring that the converted grammars adhere to the CNF rules consistently across different inputs.

In conclusion, this laboratory work provided a hands-on experience in understanding and implementing CNF normalization techniques, reinforcing fundamental concepts in formal language theory and computational linguistics. The acquired knowledge and practical skills lay a solid foundation for tackling more advanced language processing tasks and exploring further areas in theoretical computer science.