Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 5:

Chomsky Normal Form

Elaborated:

st. gr. FAF-223 Cravcenco Dmitrii

Verified:

asist. univ. Dumitru Cretu

Chişinău – 2024

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.
    1. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
    2. The implemented functionality needs executed and tested.
    3. A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
    4. 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.

Implementation description

In order to convert a grammar to Chomsky Normal Form, I implemented a class that extends the Grammar class.

```
public class ChomskyNormalForm extends Grammar {
```

In this class I receive in constructor a Grammar object and I initialize the grammar object with the grammar object received in the constructor.

```
    public ChomskyNormalForm(Grammar grammar) {

        this.setNonTerminals(myGrammar.getNonTerminals());
        this.setTerminals(myGrammar.getTerminals());
        this.setStartSymbol(myGrammar.getStartSymbol());
        this.setRules(myGrammar.getRules());
    }
```

The `convertToCNF()` method is responsible for transforming a given context-free grammar into Chomsky Normal Form (CNF). CNF is a standardized form of grammar where every production rule is either of the form A → BC (where A, B, and C are non-terminal symbols) or A → a (where A is a non-terminal symbol and a is a terminal symbol). This normalization simplifies parsing algorithms and facilitates further grammar analysis.

Step-by-Step Explanation

1. **Remove ε (epsilon) productions**:
    - Identify and remove any ε-productions (productions that derive the empty string).
    - Handle cases where non-terminals can derive ε by generating additional productions that exclude ε.
2. **Remove unit productions**:
    - Remove unit productions of the form A → B, where A and B are non-terminals.

   - Replace unit productions with their corresponding productions, effectively expanding the grammar.
3. **Eliminate inaccessible states**:
   - Identify and remove any non-terminals that are not reachable from the start symbol.
   - Traverse the grammar, starting from the start symbol, and mark reachable states.
   - Remove unreachable states and their associated productions.
4. **Eliminate non-productive rules**:
   - Identify and remove any non-terminals that cannot derive terminal strings.
   - Traverse the grammar, starting from the start symbol, and mark productive states.
   - Remove non-productive states and their associated productions.
5. **Convert to Chomsky Normal Form**:
   - Modify the remaining grammar rules to adhere to Chomsky Normal Form requirements:
     - All productions are either of the form A → BC or A → a, where A, B, and C are non-terminals, and a is a terminal.
     - Handle long productions (> 2 symbols) by introducing new non-terminals and splitting the production into smaller units.
     - Replace single terminal productions with new non-terminals.
     - Ensure that all non-terminals in the grammar have at most two symbols.
     - Clean up and optimize the grammar rules.

```
public void convertToCNF() {
        removeEpsilonProductions();
        removeUnitProductions();
        eliminateInaccessibleStates();
        eliminateNonProductiveRules();
        convertToChomskyNormalForm();
    }
```

The `removeEpsilonProductions()` method is responsible for removing epsilon productions from the grammar. Epsilon productions are productions that derive the empty string (ε). The method iterates through the grammar rules and identifies non-terminals that can derive ε. It then generates additional productions to exclude ε from those non-terminals.

```
    private void removeEpsilonProductions() {
        ...
        for (String nonTerminal : epsilonNonTerminals) {
            ...
            for (String production : rules.get(nonTerminal)) {
                if (production.equals("")) continue;
                ...

                    if (epsilonNonTerminals.contains(String.valueOf(symbol)))
continue;
                    ...
                newProductions.add(newProduction.toString());
                ...
            rules.get(nonTerminal).addAll(newProductions);
        }
        rules.values().forEach(productions -> productions.remove(""));
    }
```

The ‘removeUnitProductions()’ method is responsible for removing unit productions from the grammar. Unit productions are productions of the form A → B, where A and B are non-terminals. The method iterates through the grammar rules and replaces unit productions with their corresponding productions, effectively expanding the grammar.

```
    private void removeUnitProductions() {
        ...
            ...
                if (production.length() == 1 &&
nonTerminals.contains(production)) {
                    ...
                    rules.get(nonTerminal).addAll(rules.get(production));
                }
            rules.get(nonTerminal).removeIf(production -> production.length() ==
1 && nonTerminals.contains(production));
    }
```

The `eliminateInaccessibleStates()` method is responsible for eliminating inaccessible states from the grammar. Inaccessible states are non-terminals that are not reachable from the start symbol. The method traverses the grammar, starting from the start symbol, and marks reachable states. It then removes unreachable states and their associated productions.

```
    public void eliminateInaccessibleStates() {
        Set<String> accessibleStates = new HashSet<>();
        ...
        do {
                    ...
                            if (Character.isUpperCase(state) &&
!accessibleStates.contains(state + "")) {
                                accessibleStates.add(state + "");
                            }
         ...
        } while (changed);
        Map<String, List<String>> updatedRules = new HashMap<>();
        for (Map.Entry<String, List<String>> entry : getRules().entrySet()) {
            ...
            if (accessibleStates.contains(fromState)) {
                ...
                        if (Character.isUpperCase(state) &&
!accessibleStates.contains(state + "")) {
                            valid = false;
                            break;
                        }
                    }
                    if (valid) {
                        updatedToStates.add(production);
                }
            }
        }
        // Update the rules with the modified ones
        setRules(updatedRules);
    }
```

The eliminateNonProductiveRules() method is responsible for eliminating non-productive rules from the grammar. Non-productive rules are non-terminals that cannot derive terminal strings.

```
    private void eliminateNonProductiveRules() {
        ...
        do {
            ...
            for (String nonTerminal : nonTerminals) {
                ...
                if (productions.stream().allMatch(production ->
production.chars().allMatch(Character::isLowerCase))) {
                    productiveStates.add(nonTerminal);
                }
            }
            ...
        } while (changed);
        ...
    }
```

The convertToChomskyNormalForm() method is responsible for converting the remaining grammar rules to Chomsky Normal Form. It modifies the grammar rules to adhere to CNF requirements, such as having productions of the form A → BC or A → a (where A, B, and C are non-terminals, and a is a terminal). In it, I also generate new non-terminals for long productions (> 2 symbols) and replace single terminal productions with new non-terminals.

```
    private void convertToChomskyNormalForm() {
        ...
        for (String nonTerminal : nonTerminals) {
            ...
            for (String production : rules.get(nonTerminal)) {
                ...
                if (production.length() == 1 && terminals.contains(production))
{
                    ...
                    rules.get(nonTerminal).add(newProduction);
                } else if (production.length() > 2) {
                    ...
                    rules.get(nonTerminal).add(newProduction);
                }
            }
        }
        ...
    }
```

Also I implemented unit testing for the implemented methods. The tests cover various scenarios and edge cases to ensure the correctness and reliability of the CNF transformation method.

```
public class ChomskyNormalFormTest {
    @Test
    public void testRemoveEpsilonProductions() {
    }
    ...
```

Conclusions and Results

The CNF transformation project provided valuable insights into Chomsky Normal Form principles and grammar normalization techniques. Through diligent research and implementation efforts, we successfully developed a method to convert grammars into CNF, meeting project objectives. Thorough testing ensured the reliability and correctness of the transformation method. Additionally, the optional extension enhanced the method's flexibility, allowing it to handle a broader range of grammars. Overall, the project contributed to a deeper understanding of CNF and its practical applications in grammar processing.

Results

- **Successful Implementation:** The CNF transformation method effectively transformed input grammars into CNF.
- **Testing:** Rigorous testing ensured the correctness and reliability of the implemented method.
- **Optional Extension:** Optionally extended the method to accept any context-free grammar, showcasing versatility and adaptability.