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 Bujilov Dmitrii

Verified:

asist. univ. Dumitru Cretu

Chişinău - 2024

Objectives:

Learn about Chomsky Normal Form (CNF).

Get familiar with the approaches of normalizing a grammar.

Implement a method for normalizing an input grammar by the rules of CNF.

The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).

The implemented functionality needs executed and tested.

A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.

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

Grammar Class Overview:

The Grammar class is designed to represent a formal grammar, which is a set of rules that define the syntax of a language. In formal language theory, a grammar consists of:

Non-terminal symbols: Symbols representing variables that can be replaced by other symbols according to the grammar's rules.

Terminal symbols: Symbols that represent the actual content of the language (e.g., characters in a string).

Productions: Rules that specify how non-terminal symbols can be replaced with other symbols (non-terminal or terminal).

Start symbol: A special non-terminal symbol that represents the entry point of the grammar.

The class uses a Map<String, List<String>> to store productions, where the key is a non-terminal symbol, and the value is a list of possible productions (rules) for that symbol.

Methods and Corresponding Unit Tests:

eliminateEpsilonProductions():

Method Description:

This method removes epsilon productions from the grammar. Epsilon productions are rules that allow a non-terminal symbol to produce an empty string (""). While epsilon productions can be useful in some contexts, they are not allowed in certain grammar normal forms like Chomsky Normal Form (CNF).

The method likely identifies epsilon productions and removes them, adjusting other rules accordingly to maintain consistency.

Unit Test:

The test first sets up a grammar with various production rules, including epsilon productions.

After invoking the method, the test verifies that the grammar no longer contains epsilon productions for the non-terminal symbol "C".

This is checked using an assertion to ensure that the list of productions for "C" does not contain the empty string.

eliminateRenamingProductions():

Method Description:

This method removes renaming productions from the grammar. Renaming productions are rules that directly map one non-terminal symbol to another, such as A -> B.

The method eliminates such rules and replaces them with the productions of the renamed non-terminal, simplifying the grammar.

Unit Test:

The test sets up a grammar with various production rules, including renaming productions.

After applying the method, the test checks that the grammar's productions remain consistent with the original ones, as renaming productions should not change the underlying grammar's structure.

An assertion is used to confirm the productions remain unchanged.

eliminateInaccessibleSymbols():

Method Description:

This method removes symbols from the grammar that cannot be reached from the start symbol.

The method iteratively finds accessible symbols by following production rules from the start symbol, then removes any symbols not reachable from the start symbol.

Unit Test:

The test sets up a grammar with various production rules, including inaccessible symbols.

After invoking the method, the test checks that the grammar no longer contains the inaccessible symbol "E".

An assertion confirms the production map does not include the inaccessible symbol.

eliminateNonProductiveSymbols():

Method Description:

This method removes non-productive symbols from the grammar. Non-productive symbols are those that cannot ultimately produce terminal symbols.

The method finds productive symbols (those leading to terminal productions) and eliminates any symbols not in the productive set.

Unit Test:

The test sets up a grammar with various production rules, including non-productive symbols.

After invoking the method, the test checks that the grammar no longer contains the non-productive symbol "E".

An assertion ensures the production map matches the expected outcome.

normalizeToChomskyForm():

Method Description:

This method transforms the grammar into Chomsky Normal Form (CNF). In CNF, all production rules must be either in the form A -> BC (two non-terminals) or A -> a (a terminal symbol).

The method includes converting longer productions into binary rules and possibly introducing new non-terminal symbols to achieve CNF.

Unit Test:

The test sets up a grammar with various production rules.

After applying the method, the test checks that the grammar adheres to CNF.

The helper function countSymbolsWithVariablesAsOneSymbol is used to ensure all productions have either one or two symbols (in the form of variables or terminals).

Additional assertions verify that the grammar does not contain epsilon productions or inaccessible symbols.

The provided unit tests are important for validating the correctness of the methods in the Grammar class. Each test checks the behavior of a specific method and ensures the expected outcome is achieved. Main parts of the code are:

```
Set<String> epsilonSymbols = findSymbolsWithEpsilonProductions();
        Map<String, List<String>> newProductions = new HashMap<>();
        for(Map.Entry<String, List<String>> entry : productions.entrySet()) {
            String nonTerminal = entry.getKey();
            List<String> productionsList = entry.getValue();
            List<String> newProductionsList = new ArrayList<>();
            for(String production : productionsList) {
                if(production.isEmpty()) {
                    continue;
                }
                boolean added = false;
                for(String nullableSymbol : epsilonSymbols) {
                    int count = production.length() - production.replace(nullableSymbol,
"").length();
                    if(count == 0 && !added) {
                        newProductionsList.add(production);
                        added = true;
                    } else {
                        String newProduction = production;
                        for(int i = 0; i < count; i++) {
                            newProduction = newProduction.replaceFirst(nullableSymbol, "");
                            if(!added) newProductionsList.add(production);
                            newProductionsList.add(newProduction);
                            added = true;
                        }
                    }
                }
            }
            newProductions.put(nonTerminal, newProductionsList);
        }
        this.productions = newProductions;
        determineTerminalsAndNonTerminals();
    }
public void eliminateRenamingProductions() {
```

```
        Map<String, List<String>> newProductions = new HashMap<>(productions);
        while (countRenamingProductions(newProductions) > 0) {
            for (Map.Entry<String, List<String>> entry : newProductions.entrySet()) {
                String nonTerminal = entry.getKey();
                List<String> productionsList = entry.getValue();
                List<String> newProductionsList = new ArrayList<>();
                for (String production : productionsList) {
                    if (production.length() == 1 && Character.isUpperCase(production.charAt(0)))
{
                        newProductionsList.addAll(productions.getOrDefault(production,
Collections.emptyList()));
                    } else {
                        newProductionsList.add(production);
                    }
                }
                newProductions.put(nonTerminal, newProductionsList);
            }
        }
        this.productions = newProductions;
        determineTerminalsAndNonTerminals();
    }
public void eliminateInaccessibleSymbols() {
        Set<String> reachableSymbols = findReachableSymbols();
        productions.keySet().retainAll(reachableSymbols);
        determineTerminalsAndNonTerminals();
    }
public void eliminateNonProductiveSymbols() {
        Set<String> productiveSymbols = findProductiveSymbols();
        productions.keySet().retainAll(productiveSymbols);
        determineTerminalsAndNonTerminals();
    }
public void normalizeToChomskyForm() {
        eliminateEpsilonProductions();
        eliminateRenamingProductions();
```

```
    eliminateInaccessibleSymbols();
    eliminateNonProductiveSymbols();
    convertToChomskyForm();
}
```

Conclusions and Results

In conclusion, the Grammar class and its various methods play a crucial role in the manipulation and transformation of formal grammars, which are integral to the field of computational linguistics and language processing. By offering a comprehensive set of functions to eliminate epsilon and renaming productions, remove inaccessible and non-productive symbols, and normalize the grammar to Chomsky Normal Form (CNF), the class provides a versatile toolkit for working with a wide range of formal grammars and languages.

These grammar transformations serve multiple purposes, including streamlining and optimizing language parsing and processing tasks. The ability to effectively handle epsilon productions helps prevent ambiguity and unnecessary complexity in the grammar, while eliminating renaming productions reduces redundancy and simplifies the structure of the grammar. Furthermore, the removal of inaccessible and non-productive symbols ensures that the grammar remains focused and efficient, avoiding superfluous or unreachable rules.

The process of normalizing the grammar to CNF is particularly valuable, as it standardizes the form of the grammar and facilitates various algorithms for parsing and analyzing languages. This transformation makes it easier to work with the grammar in a structured and consistent manner, improving the efficiency and accuracy of language processing tasks.

The comprehensive unit tests provided for each method offer thorough validation of the implementations, ensuring that the transformations are carried out correctly and produce the expected outcomes. By systematically testing each aspect of the grammar transformations, the class is able to maintain a high level of integrity and consistency, instilling confidence in its usage across various applications.

Overall, the Grammar class serves as a valuable and reliable foundation for applications in language processing, such as compilers and interpreters. Its ability to perform a wide range of transformations on formal grammars ensures that it can be used effectively in different contexts and for various purposes.