Formal Languages & Finite Automata

Laboratory work 5:

Regular expressions

Student: Grigoraș Dumitru
Verified: prof., Cojuhari Irina
univ. assist., Crețu Dumitru

Chișinău, 2024

Theory

Chomsky Normal Form (CNF) is a specific form used in formal language theory and automata theory to simplify and analyze context-free grammars (CFGs). It is named after Noam Chomsky, who introduced the concept as part of his work in generative grammar and linguistic theory.

Definition and Characteristics:

1. **Production Rules**: In CNF, production rules are of two types:
    - Unit Productions: Rules of the form A -> B, where A and B are non-terminal symbols.
    - Binary Productions: Rules of the form A -> BC, where A, B, and C are non-terminal symbols.
2. **Non-terminal Symbols**: CNF grammars have a set of non-terminal symbols (variables) that represent syntactic categories or parts of speech in a language.
3. **Terminal Symbols**: Terminal symbols are the actual words or tokens in the language that derive from the non-terminal symbols through production rules.
4. **Start Symbol**: CNF grammars have a designated start symbol that serves as the entry point for deriving strings in the language.
5. **Derivation Process:** The derivation process in CNF involves applying production rules iteratively until terminal symbols are reached, forming valid strings in the language defined by the grammar.

Benefits and Applications:

1. **Simplification**: CNF simplifies the structure of context-free grammars by restricting production rules to a specific format, making it easier to analyze and manipulate language rules.
2. **Parsing Efficiency**: CNF grammars enable efficient parsing algorithms, such as the CYK (Cocke-Younger-Kasami) algorithm, which can parse strings in O(n^3) time complexity for CNF grammars.
3. **Language Recognition:** CNF plays a crucial role in language recognition tasks, including syntactic analysis, semantic parsing, and natural language processing.
4. **Syntax Analysis:** CNF grammars are used in syntax analysis tools and compilers to analyze the grammatical structure of programming languages and generate parse trees.
5. **Theoretical Foundations:** CNF is foundational in formal language theory and automata theory, providing insights into the computational properties of context-free languages and their relation to other language classes.

Overall, Chomsky Normal Form provides a standardized and structured representation of context-free grammars, facilitating efficient parsing and analysis of languages in various computational contexts.

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

Implementation

Grammar class

Grammar represents a context-free grammar (CFG) and contains essential components necessary to define and work with grammars in formal language theory and computational linguistics. Let's break down the key elements and functionality of this class:

1. Non-terminals (Vn):
    - The Vn attribute is a set representing the non-terminal symbols of the grammar. Non-terminals are symbols that can be replaced by other symbols during the derivation of strings in the language defined by the grammar.
2. Terminals (Vt):
    - The Vt attribute is a set representing the terminal symbols of the grammar. Terminals are symbols that appear in the final strings derived from the grammar and cannot be further expanded or replaced.
3. Production Rules (productions):
    - The productions attribute is a map that associates each non-terminal symbol with a list of strings representing possible productions or expansions. Each production rule specifies how a non-terminal can be replaced by a sequence of terminals and/or non-terminals.
4. Start Symbol (startSymbol):
    - The startSymbol attribute represents the start symbol of the grammar, which serves as the initial symbol from which derivation of strings begins.
5. Constructor:
    - The constructor of the Grammar class initializes the non-terminals (Vn), terminals (Vt), production rules (productions), and the start symbol (startSymbol) based on the provided arguments.
6. Getters and Setters:
    - Getter methods (getVn(), getVt(), getProductions(), getStartSymbol()) allow access to the attributes of the grammar.
    - Setter methods (setVn(), setVt(), setProductions(), setStartSymbol()) enable modification of the grammar components if needed.

CNFConverter class:

CNFConverter is designed to convert a context-free grammar (CFG) into Chomsky Normal Form (CNF), which is a specific form of context-free grammars known for its simplicity and ease of parsing. Let's explore the key functionalities and processes implemented in this class:

1. Class Attributes:
    - mapVariableProduction: A Map that stores the production rules of the grammar, where each non-terminal symbol maps to a list of strings representing its productions.
    - epselonFound: A string variable used during epsilon elimination to track the non-terminal containing epsilon productions.
    - lineCount: An integer variable to store the number of production rules in the input grammar.
    - input: A string variable to hold the input context-free grammar.
2. Conversion Methods:
    - convertCFGtoCNF(): Initiates the conversion process from CFG to CNF by calling multiple helper methods in sequence.
    - eliminateEpselon(): Eliminates epsilon productions from the grammar.
    - removeEpselon(): Helper method to remove epsilon productions from the production rules.
    - removeDuplicateKeyValue(): Removes duplicate key-value pairs from the production rules map.
    - eliminateSingleVariable(): Eliminates single variables in every production.
    - removeSingleVariable(): Helper method to remove single variables from the production rules.
    - onlyTwoTerminalandOneVariable(): Ensures that each production has only two non-terminals or one terminal.
    - removeThreeTerminal(): Eliminates productions with three terminals or a combination of variables and terminals.
3. Utility Methods:
    - printMap(): Prints the current production rules map.
    - splitEnter(String input): Splits the input grammar into separate production rules based on newline characters.
    - convertStringtoMap(): Converts the input grammar string into a map representation of production rules.
4. Main Method:
    - The main method demonstrates the usage of the CNFConverter class by providing an input context-free grammar and converting it to CNF.

Overall, the CNFConverter class encapsulates the logic and algorithms required to transform a CFG into CNF, facilitating the understanding and manipulation of context-free grammars in a standardized form suitable for various computational tasks.

CNFConverterTest class:

CNFConverterTest class is a JUnit test class designed to test the functionality and correctness of methods in the CNFConverter class, which is responsible for converting a context-free grammar (CFG) into Chomsky Normal Form (CNF). Let's break down the key aspects of this test class:

1. Imports:
    - import static org.junit.Assert.*;: Static imports from JUnit to use assertion methods without referencing the Assert class explicitly.
    - import org.junit.Before;: Import for the Before annotation to designate methods that should run before each test method.
    - import org.junit.Test;: Import for the Test annotation to designate test methods.
2. Class Declaration:
    - public class CNFConverterTest { ... }: Defines the CNFConverterTest class for testing the CNFConverter class.
3. Instance Variable:
    - private CNFConverter cnf;: Declares an instance variable cnf of type CNFConverter to be used for testing.
4. Setup Method:
    - @Before public void setUp() { ... }: Annotated method that runs before each test method to set up the test environment. It initializes the cnf object.
5. Test Methods:
    - @Test public void testRemoveEpselon() { ... }: Annotated test method to test the removeEpselon method of CNFConverter. It sets up a test map with epsilon productions and checks if epsilon is correctly removed.
    - @Test public void testRemoveSingleVariable() { ... }: Annotated test method to test the removeSingleVariable method of CNFConverter. It sets up a test map with single variables and checks if single variables are correctly removed.
    - @Test public void testEliminateThreeTerminal() { ... }: Annotated test method to test the eliminateThreeTerminal method of CNFConverter. It sets up a test map with productions containing three terminals and checks if they are eliminated as expected.
6. Test Cases:
    - Each test method contains test cases where specific scenarios are set up using test maps and input data to simulate different conditions and verify the behavior of the methods being tested.
    - Assertions like assertFalse are used to check expected outcomes based on the transformations applied by the methods being tested.

Overall, the CNFConverterTest class provides a structured approach to testing the functionality of the CNFConverter class by defining test cases and using JUnit's assertion methods to validate the correctness of the conversion algorithms implemented in CNFConverter.

Results

Test for variant 14:

S -> ASA | aB +
A -> B | S +
B -> b | ε

```
Converting CFG to CNF...
Original Grammar:
S -> [ASA, aB]
A -> [B, S]
B -> [b, ε]

Remove Epselon....
S -> [ASA, aB, a, SA, AS, S]
A -> [B, S]
B -> [b]

Remove Duplicate Key Value ...
S -> [ASA, aB, a, SA, AS]
A -> [B, S]
B -> [b]

Remove Single Variable in Every Production ...
S -> [ASA, aB, a, SA, AS]
A -> [b, ASA, aB, a, SA, AS]
B -> [b]

Assign new variable for two non-terminal or one terminal ...
S -> [ASA, aB, a, SA, AS]
A -> [b, ASA, aB, a, SA, AS]
B -> [b]

Replace two terminal variable with new variable ...
S -> [ASA, aB, a, SA, AS]
A -> [b, ASA, aB, a, SA, AS]
B -> [b]


CNF Grammar:
S -> [ASA, aB, a, SA, AS]
A -> [b, ASA, aB, a, SA, AS]
B -> [b]
```

Test for another variant:
S -> ASA | aB
A -> B | S
B -> b | ε

```
Converting CFG to CNF...
Original Grammar:
S -> [ASA, aB]
A -> [B, S]
B -> [b, ε]

Remove Epselon....
S -> [ASA, aB, a, SA, AS, S]
A -> [B, S]
B -> [b]

Remove Duplicate Key Value ...
S -> [ASA, aB, a, SA, AS]
A -> [B, S]
B -> [b]

Remove Single Variable in Every Production ...
S -> [ASA, aB, a, SA, AS]
A -> [b, ASA, aB, a, SA, AS]
B -> [b]

Assign new variable for two non-terminal or one terminal ...
S -> [ASA, aB, a, SA, AS]
A -> [b, ASA, aB, a, SA, AS]
B -> [b]

Replace two terminal variable with new variable ...
S -> [ASA, aB, a, SA, AS]
A -> [b, ASA, aB, a, SA, AS]
B -> [b]


CNF Grammar:
S -> [ASA, aB, a, SA, AS]
A -> [b, ASA, aB, a, SA, AS]
B -> [b]
```

Conclusion

The lab work focused on developing a CNFConverter class to transform context-free grammars (CFGs) into Chomsky Normal Form (CNF), complemented by the CNFConverterTest class for comprehensive testing. Key objectives included implementing CNF conversion algorithms and ensuring their accuracy through testing. The CNFConverter successfully handled epsilon removal, single variable elimination, and transformations involving multiple terminals. Rigorous testing scenarios in CNFConverterTest verified the correctness and robustness of the conversion process across various grammar structures. Insights gained encompassed formal language theory, testing methodologies, and the importance of code validation under diverse conditions. Future considerations may involve extending CNFConverter functionalities and enhancing test coverage for continued robustness and reliability.

Bibliography

[1] Formal Languages and Compiler Design Accessed February 14, 2024. https://else.fcim.utm.md/pluginfile.php/110457/mod_resource/content/0/Theme_1.pdf.

[2] Regular Language. Finite Automata. Accessed February 15, 2024. https://drive.google.com/file/d/1rBGyzDN5eWMXTNeUxLxmKsf7tyhHt9Jk/view