**Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work nr. 5

Course: Formal languages and finite automata

Topic: Chomsky Normal Form

Elaborated:

st. gr. FAF-221 Ana Calugareanu

Verified:

asist. univ. Cretu Dumitru

Chişinău - 2024

**Theory**

Chomsky Normal Form (CNF) is a way of representing context-free grammars (CFGs) in a specific form, named after the renowned linguist and cognitive scientist Noam Chomsky. It has several important properties that make it useful for theoretical analysis and practical applications in areas such as natural language processing and parsing algorithms. Here's a breakdown of the key aspects of CNF:

1. Formal Definition: In Chomsky Normal Form, every production rule of the grammar is in one of two forms:

- A → BC

- A → a

where A, B, and C are non-terminal symbols, and a is a terminal symbol. The production A → ε (where ε represents the empty string) is allowed only if the start symbol appears on the right-hand side of a production.

2. Non-terminal Usage: In CNF, each non-terminal symbol (except for the start symbol, which can produce ε) must derive at least one string of terminal symbols.

3. Simplification: The removal of useless symbols and unproductive rules is often done before converting a grammar to CNF. Useless symbols are those that cannot be reached from the start symbol, and unproductive rules are those that cannot derive any terminal string.

4. Advantages:

- CNF simplifies the structure of the grammar, making it easier to analyze and process.

- It facilitates certain parsing algorithms, such as the CYK (Cocke-Younger-Kasami) algorithm, which operates efficiently on grammars in CNF.

- CNF helps in proving various properties of context-free languages, such as closure properties.

5. Conversion: Any context-free grammar can be converted to an equivalent grammar in Chomsky Normal Form. The process typically involves several steps, including:

- Eliminating ε-productions

- Eliminating unit productions (productions of the form A → B)

- Eliminating productions with more than two non-terminals on the right-hand side

- Introducing new non-terminals as necessary

**Objectives:**

1. Learn about Chomsky Normal Form (CNF)
2. Get familiar with the approaches of normalizing a grammar.
3. Implement a method for normalizing an input grammar by the rules of CNF.
4. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
5. The implemented functionality needs executed and tested.
6. A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
7. 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**

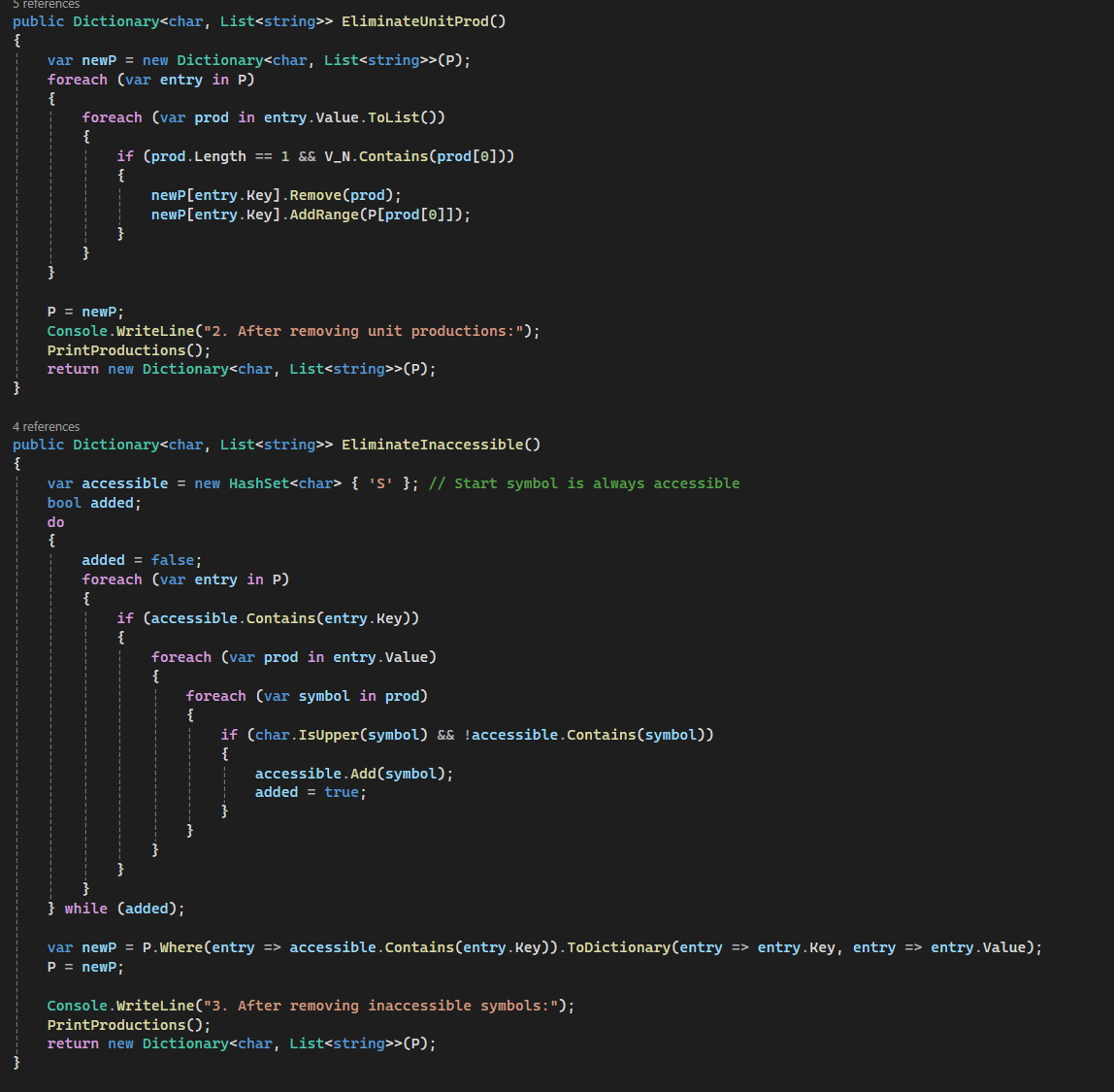
For implementation I chose to use C#, because it is a familiar language.

First of all, I define a class with constructor for grammar. Also, it will have methods for step-by-step obtaining the CNF, like RemoveEpsilon, EliminateUnit, EliminateInaccesible, EliminateUnprod and TransformToCNF

****

**Figure 1. Method to remove epsilon prod**

The provided code appears to be a C# function named `RemoveEpsilon` that is designed to modify a dictionary containing grammar productions by removing epsilon (empty string) productions. The function identifies non-terminals that produce an epsilon and then iteratively removes the epsilon occurrences from all productions. It handles the modifications without directly altering the original dictionary during the iteration by converting the list of productions to a new list. After the removal process, it prints a message to the console indicating the completion of the epsilon removal task and returns the modified dictionary. This function could be part of a compiler or parser that processes context-free grammars, especially when transforming grammars to remove unnecessary complexity or to prepare for further processing like parsing.

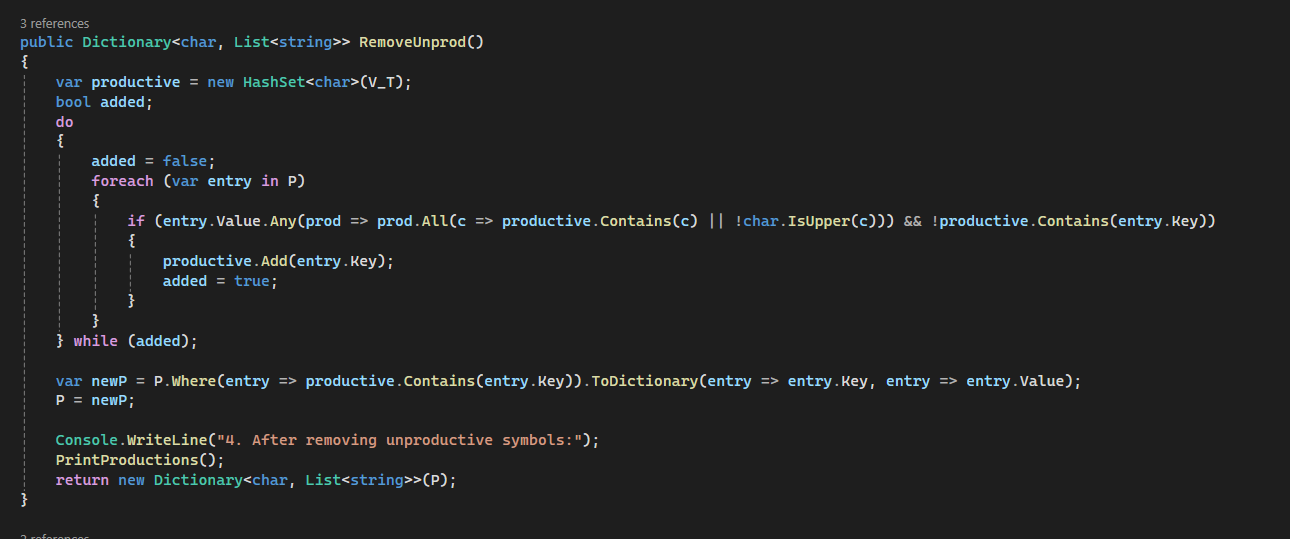


**Figure 2. Methods to eliminate unit prod. and inaccessible symbols**

The image contains two C# methods: `EliminateUnitProd` and `EliminateInaccessible`, which are likely part of a compiler or parser for context-free grammars. The `EliminateUnitProd` method seems to remove unit productions (productions that consist of a single non-terminal symbol) from a given grammar, represented by a dictionary of characters to lists of strings, where each character is a non-terminal symbol. It creates a new dictionary, iterates over the productions, and replaces each unit production with the productions of the non-terminal it points to.

The `EliminateInaccessible` method identifies and removes inaccessible symbols from the grammar. It uses a `HashSet<char>` to keep track of accessible symbols, initially containing just the start symbol ('S'). Through iteration, it adds new accessible symbols by checking the current productions until no new symbols are added. Afterwards, it creates a new dictionary including only the entries that have accessible non-terminal symbols and assigns this dictionary back to `P`.

Both methods finish their execution by printing a message to the console that indicates which step of the grammar simplification has been completed, and they return the new, modified dictionary representing the potentially simplified grammar. These operations are part of the grammar simplification process, which is commonly used in compilers and parsers to optimize and prepare grammars for more efficient parsing.



**Figure 3. Method to remove unproductive symbols**

The code snippet is a C# method named `RemoveUnprod()` which is designed to remove unproductive symbols from a context-free grammar. The method utilizes a `HashSet` to keep track of productive non-terminal symbols, ones that can eventually produce terminal strings. It iteratively marks non-terminals as productive if they lead to terminals or if they lead to combinations of terminals and already known productive non-terminals. This process repeats until no new productive symbols are found. After identifying all productive symbols, the method creates a new dictionary containing only the productive symbols and their associated productions. Finally, it prints out a message and returns the new dictionary without the unproductive symbols. This is a common step in optimizing grammars for parsing, ensuring that the grammar contains only the symbols that can actually be used to derive strings in the language.



**Figure 4. Method to construct CNF**

The code in the image is a C# function called `TransformToCNF()` that converts a context-free grammar into Chomsky Normal Form (CNF). It begins by creating a copy of the original grammar (`P`) and initializes a new dictionary to keep track of new variables introduced during the transformation. The function iterates over each production in the grammar, adding terminal productions directly to the new production list, while longer productions are broken down using newly created variables until all productions conform to CNF (i.e., right-hand sides are either two variables or a single terminal). When a pair of terminals or a terminal followed by a variable is encountered in a production, it substitutes this with a new variable and adds a corresponding production to ensure that the original structure of the grammar is preserved. Lastly, the function updates the original grammar with the CNF version, prints the final CNF to the console, and then returns it. This transformation is an important step in various algorithms in computational linguistics and formal language theory, such as parsing algorithms.

**Screenshots**



**Figure 6. Output for converted grammar**

**Figure 7. Output of Unit Test**

**Conclusions**

Context-free grammars (CFGs) and Chomsky Normal Form (CNF) are crucial concepts in formal language theory, providing a structured way to describe the syntax of languages through production rules. The process of transforming a CFG into CNF involves several steps, including removing epsilon productions, eliminating unit productions, and eliminating inaccessible and unproductive symbols. These steps aim to simplify the grammar's structure and make it easier to analyze and process.

Chomsky Normal Form offers several advantages, including facilitating parsing algorithms like the CYK algorithm and simplifying proofs of language properties. The conversion process ensures that the resulting grammar adheres to a standardized form, which aids in theoretical analysis and practical applications in areas such as natural language processing and parsing algorithms.

The provided Python implementations demonstrate the step-by-step transformation of a CFG into CNF, showcasing each transformation method's functionality through unit tests. These tests ensure that each transformation method produces the expected output, validating the correctness of the implementation.

In conclusion, understanding and implementing the conversion of CFGs to CNF is essential for formal language theory and computational linguistics. It provides a foundation for further exploration into language properties, parsing algorithms, and language processing applications. Additionally, the practical implementations serve as valuable tools for educational purposes and real-world applications in various computational fields.