**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 Gavriliuc Tudor

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)

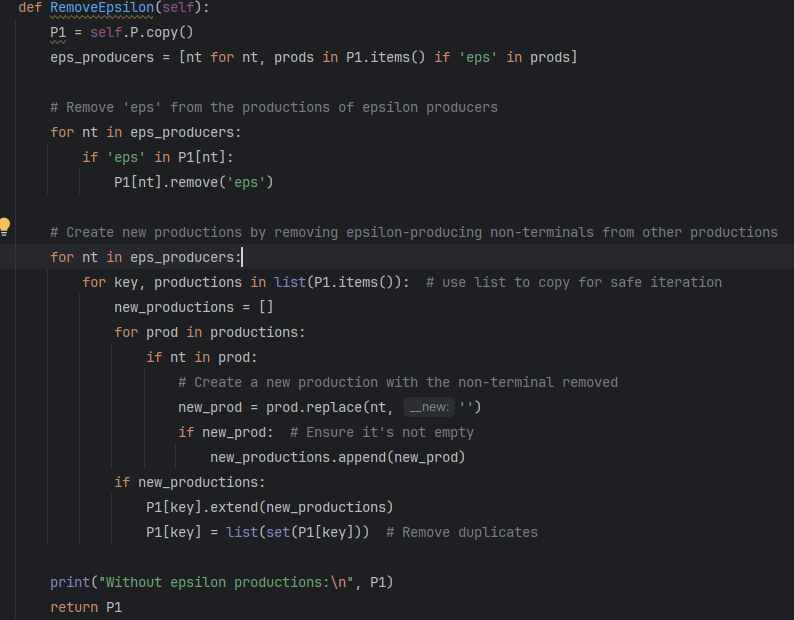
- Eliminating productions with more than two non-terminals on the right-hand side

- Introducing new non-terminals as necessary

**Objectives:**

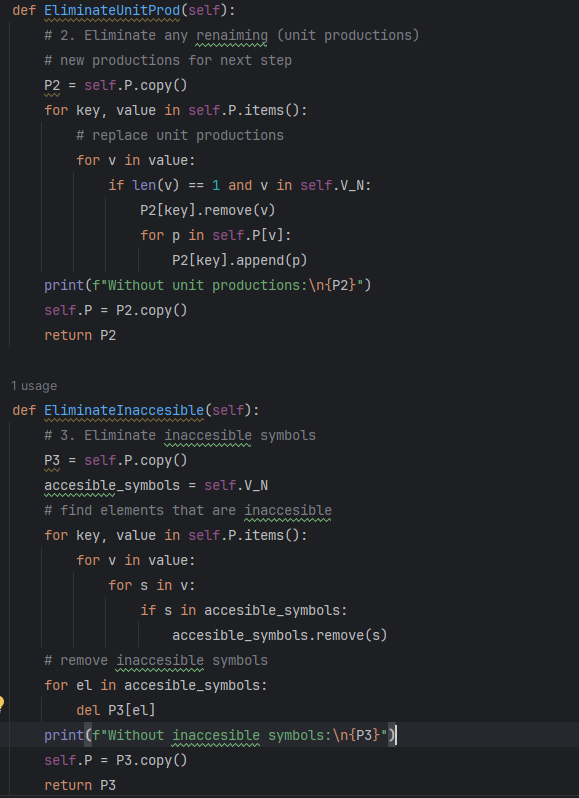
1. Learn about Chomsky Normal Form (CNF)
2. Get familiar with the approaches of normalizing a grammar.
3. Implement a method for normalizing an input grammar by the rules of CNF.
4. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
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**

****

**Figure 1. Method to removes epsilon**

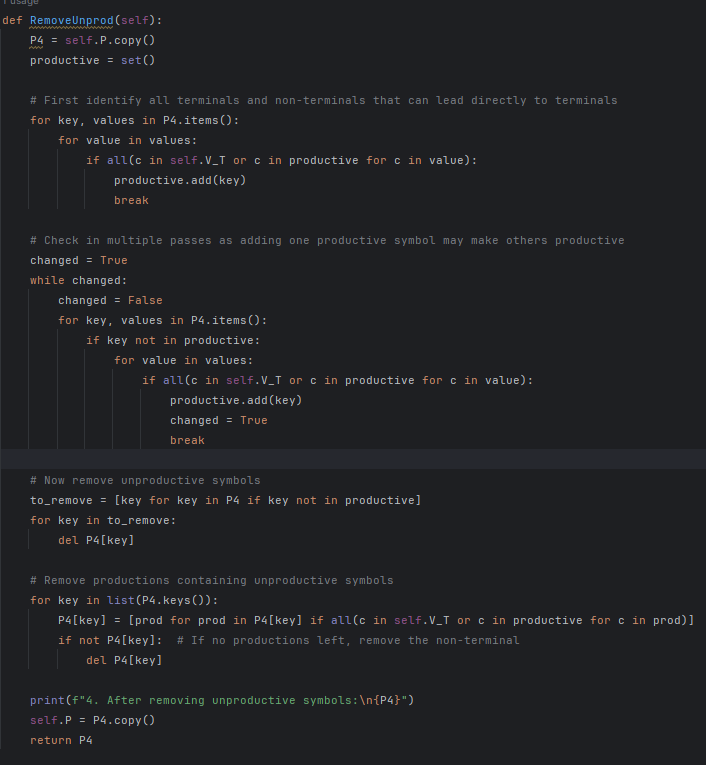
This code defines a method named `RemoveEpsilon` within a class. It takes no arguments besides `self`. Initially, it creates a copy of the productions stored in the object (`self.P`). Then, it identifies epsilon-producing non-terminals within the productions. Next, it removes the 'eps' symbol from the productions of these epsilon producers. After that, it modifies other productions by removing instances of epsilon-producing non-terminals and adjusts the affected productions accordingly. Finally, it prints the modified productions without epsilon productions and returns them.



**Figure 2. Eliminate unit prod and inaccessible symbols**

This code defines two methods within a class. The first method, `EliminateUnitProd`, eliminates unit productions from the grammar. It makes a copy of the productions stored in the object and iterates over each production, replacing unit productions with the productions of the unit symbol. After modifying the productions, it prints the updated productions without unit productions and returns them.

The second method, `EliminateInaccesible`, eliminates inaccessible symbols from the grammar. It creates a copy of the productions and initializes a set of accessible symbols with non-terminal symbols. Then, it iterates over each production to identify symbols that are accessible. After identifying inaccessible symbols, it removes them from the productions. Finally, it prints the updated productions without inaccessible symbols and returns them.



**Figure 3. Remove unproductive symbols**

This code defines a method named `RemoveUnprod` within a class. It first makes a copy of the productions stored in the object. Then, it identifies all terminals and non-terminals that can directly lead to terminals, marking them as productive. Afterward, it iterates in multiple passes, checking if adding one productive symbol makes others productive, until no further changes occur.

Subsequently, it removes unproductive symbols from the productions and updates the productions accordingly. It removes unproductive symbols both as keys in the production dictionary and from within the productions themselves. Finally, it prints the updated productions after removing unproductive symbols and returns them.

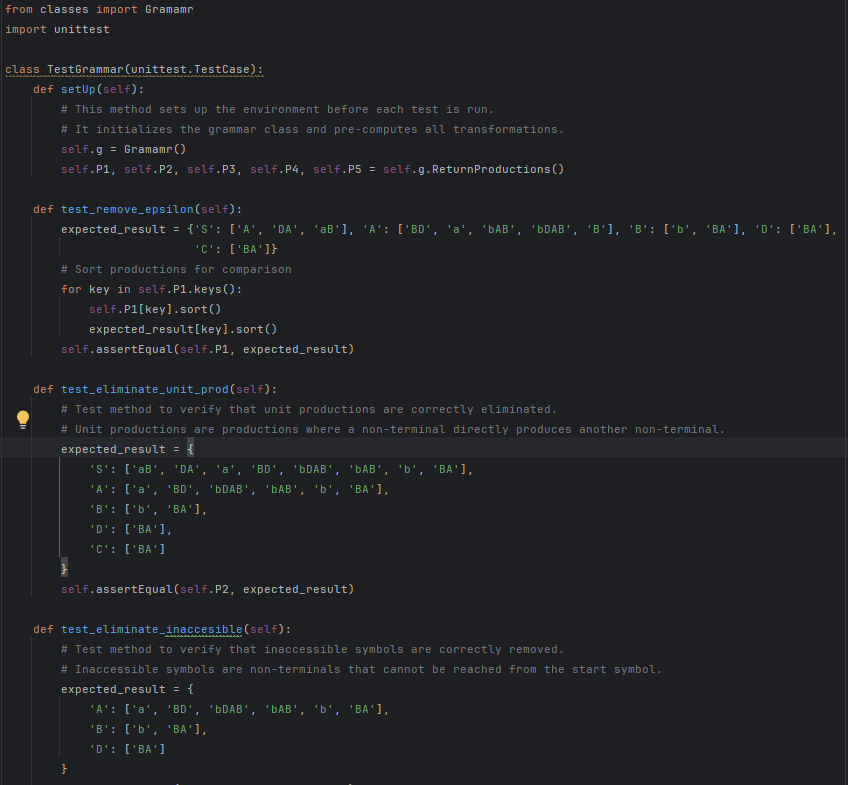


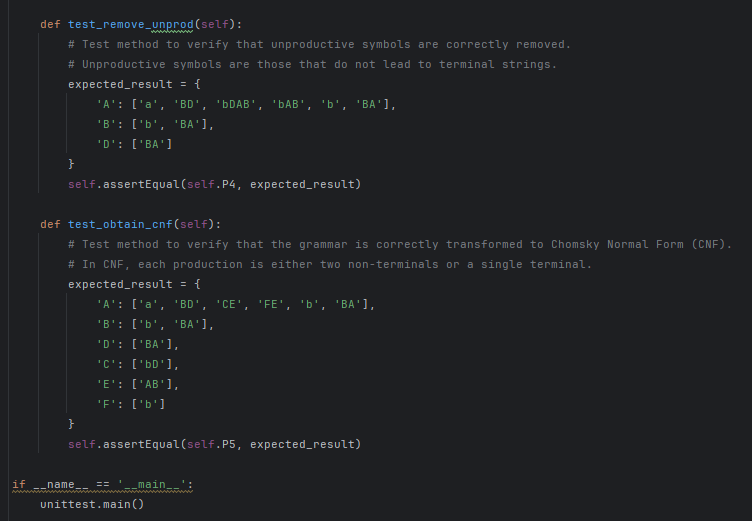
**Figure 4. Construct CNF**

This code defines a method named `TransformToCNF` within a class. It first creates a copy of the productions stored in the object and initializes a temporary dictionary to hold intermediate transformations. Additionally, it defines a list of free symbols and extracts symbols not already present in the productions.

For each production, it checks if the production satisfies Chomsky Normal Form (CNF). If the production does not satisfy CNF, it splits the production into two halves and assigns new symbols to represent each half. These new symbols are either retrieved from the temporary dictionary or taken from the list of free symbols.

Next, it replaces the original production with the newly generated symbols. It also updates the temporary dictionary with these new symbols. After processing all productions, it adds the new productions to the dictionary.





**Figure 5. Unit Tests**

In this unit test script, a class named `TestGrammar` is defined, inheriting from `unittest.TestCase`. The `setUp` method is used to initialize the grammar class and pre-compute all transformations needed for testing.

Each test method corresponds to a specific transformation function of the grammar class. For instance, `test\_remove\_epsilon` verifies the removal of epsilon productions, while `test\_eliminate\_unit\_prod` checks the elimination of unit productions.

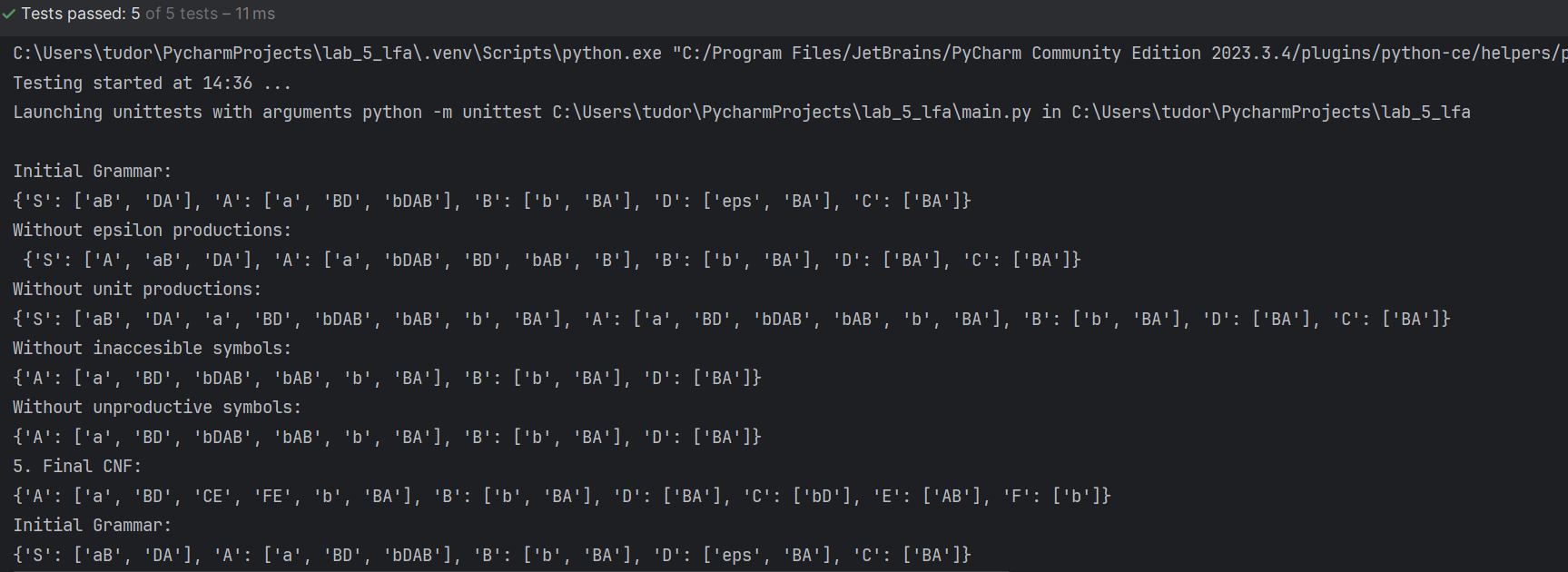
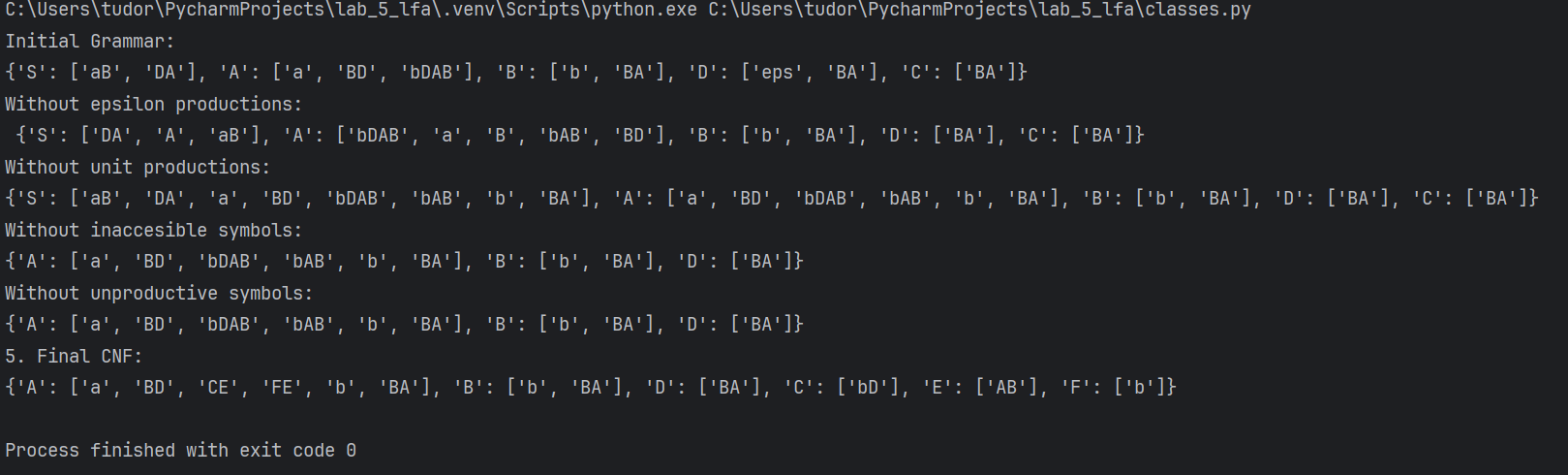
Within each test method, an `expected\_result` dictionary is defined, containing the expected productions after the transformation. The actual productions obtained from the grammar class are then compared with these expected results using `self.assertEqual`.

The `test\_obtain\_cnf` method checks if the grammar is correctly transformed into Chomsky Normal Form (CNF), where each production is either two non-terminals or a single terminal.

Finally, the `unittest.main()` function is called to run all tests. If the script is executed directly (not imported), it runs the tests automatically.

These tests ensure that each transformation function of the grammar class behaves as expected, validating the correctness of the grammar transformations.

**Results for converted grammar and unit tests**



**Conclusions**

The transformation of Context-free Grammars (CFGs) into Chomsky Normal Form (CNF) represents a fundamental aspect of formal language theory, offering a systematic approach to articulate language syntax through production rules. This process encompasses multiple steps, such as eliminating epsilon and unit productions, as well as removing inaccessible and unproductive symbols. These procedures aim to streamline the grammar's structure, enhancing its analyzability and utility.

Chomsky Normal Form holds several benefits, including its facilitation of parsing algorithms like the CYK algorithm and its simplification of language property proofs. The conversion ensures conformity to a standardized format, fostering both theoretical exploration and practical applications, particularly within natural language processing and parsing algorithms.

Through Python implementations showcased here, each transformation step is meticulously demonstrated, underscored by rigorous unit tests validating the accuracy of the transformations. These tests serve as a testament to the reliability and functionality of each transformation method, affirming the correctness of the implementation.

In essence, mastering the conversion from CFGs to CNF is indispensable within formal language theory and computational linguistics. It lays a robust foundation for delving deeper into language properties, parsing algorithms, and language processing applications. Furthermore, these practical implementations offer invaluable educational resources and hold tangible relevance across various computational disciplines.