# Topic: ChomskyNormalForm

# **Course: Formal Languages & Finite Automata**

**Author: Doschinescu Dan**

**Group: FAF-222**

**Variant: 8**

**Chisinau 2024**

**1.Theory**

Chomsky Normal Form (CNF) is a significant concept in formal language theory and computational linguistics. It represents a way to standardize the structure of context-free grammars, making them easier to analyze and process by algorithms. In CNF, each production rule is restricted to have either two non-terminals or one terminal on the right-hand side. This simplification allows for efficient parsing algorithms, such as the CYK algorithm, to be applied. Additionally, CNF eliminates ambiguity, ensuring that each sentence has only one possible parse tree. This is crucial for many natural language processing tasks. Transforming a grammar into CNF involves multiple steps, such as eliminating ε-productions, unit productions, and converting long productions into binary productions. While CNF imposes some limitations on grammars, its benefits in terms of computational efficiency and unambiguous parsing are substantial. Understanding and applying CNF is fundamental in the development of parsing algorithms and syntactic analysis in computational linguistics. It serves as a cornerstone in the study of formal languages and their application in natural language processing systems.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**2.Objectives**

Learn about Chomsky Normal Form (CNF) [1].

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.



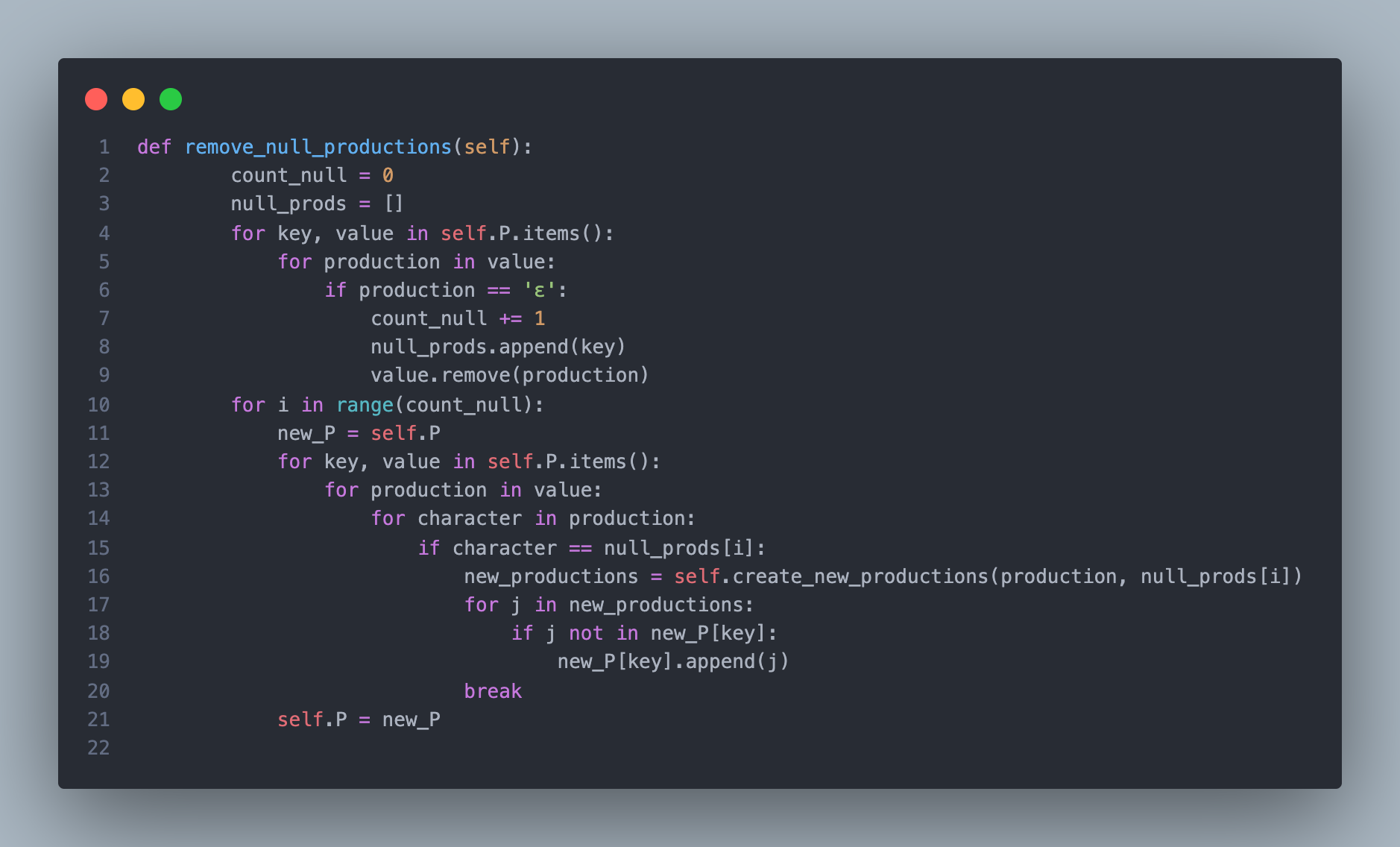
code defines a class named GenericGrammar, which represents a generic context-free grammar. It has an initializer method \_\_init\_\_ that takes four parameters: vn (non-terminals), vt (terminals), p (production rules), and s (start symbol). These parameters initialize the non-terminals, terminals, production rules, and start symbol of the grammar, respectively. The class also contains a method named print\_grammar() which prints out the details of the grammar in a structured format. It prints the non-terminals, terminals, and production rules one by one, along with their respective values. Finally, it prints the start symbol of the grammar. This class provides a basic framework for working with context-free grammars in Python, allowing users to initialize, store, and print grammars with ease.



This block of code contains a class named Grammar that inherits from GenericGrammar and implements methods for converting a context-free grammar to Chomsky Normal Form (CNF). cfg\_to\_cnf() is the main method responsible for converting the grammar to CNF. It calls several helper methods in sequence to perform the necessary transformations. start\_symbol\_rhs\_removal() is a method that removes the start symbol from the right-hand side of any production rule. It does so by iterating through the production rules and characters in each rule, and if it finds the start symbol (self.S) on the right-hand side, it raises a custom exception BreakFromLoops. This method is useful to ensure that the start symbol doesn't appear on the right side of any production, which is a requirement in CNF. The custom exception BreakFromLoops is defined at the beginning of the code. It's used to break out of multiple nested loops when the start symbol is found on the right-hand side of a production rule. If the start symbol is found on the right-hand side of any production, the exception is caught, and the method replaces the original production rules with a new rule where 'X' is the start symbol. This is to satisfy the CNF requirement that the start symbol only appears on the left-hand side of production rules. The method then updates the grammar's production rules (self.P), start symbol (self.S), and list of non-terminals (self.Vn).



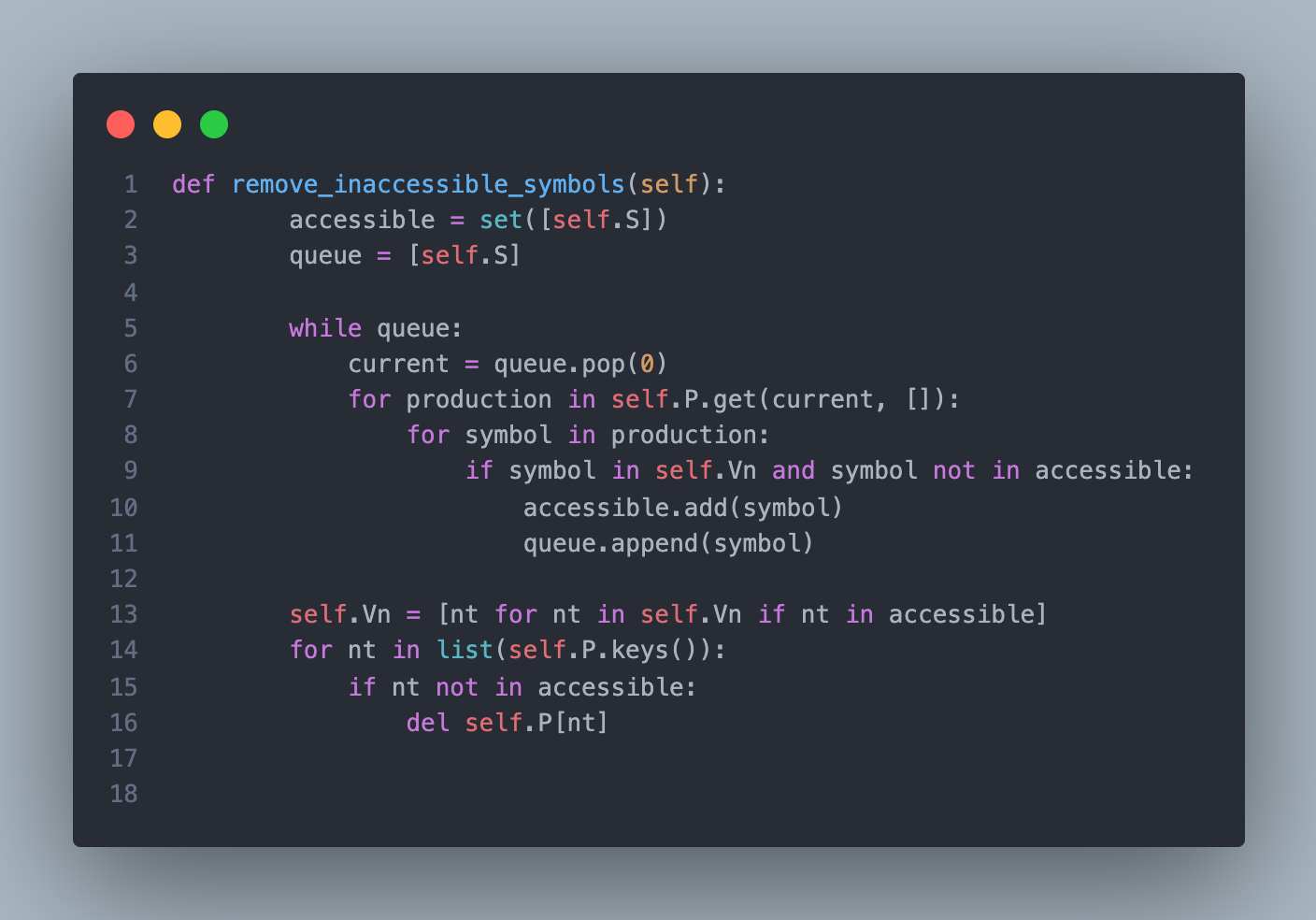
This function, create\_new\_productions, takes a production rule and a character as input. It iterates through the characters of the production, and if it finds a match with the specified character, it creates a new production by removing that character. These new productions are then returned as a list of strings. Essentially, this function generates all possible productions by removing occurrences of a specified character from the original production rule.



This block of code is a method named remove\_null\_productions, which removes null (ε) productions from the grammar. It starts by initializing variables to count null productions (count\_null) and to store non-terminals with null productions (null\_prods). It then iterates through each production in the grammar, and if it finds a null production, it increments the null count, adds the non-terminal to null\_prods, and removes the null production from the list of productions. After identifying all null productions, it iterates through each null-producing non-terminal. For each null-producing non-terminal, it generates new productions by removing occurrences of that non-terminal from existing productions using the create\_new\_productions method. These new productions are then added to the grammar's production rules, effectively eliminating null productions. Finally, it updates the grammar's production rules accordingly. This method ensures the grammar is in a form suitable for conversion to Chomsky Normal Form (CNF), where null productions are not allowed.



This block of code defines a method named remove\_unit\_productions within a grammar class, responsible for eliminating unit productions (A → B) from the grammar. Initially, it removes any direct unit productions where a non-terminal directly produces itself (e.g., A → A). It iterates through each production rule (value) of each non-terminal (key) and removes the production if it matches the non-terminal itself. It then enters a loop (while changes) to iteratively remove indirect unit productions. This loop continues until no more changes are made. Within the loop, it iterates through each production rule of each non-terminal. If a production consists of a non-terminal (indicating a possible unit production), it removes that non-terminal from the production rule. It then adds all productions of the removed non-terminal to the current non-terminal's production rules, effectively replacing the unit production with the productions of the removed non-terminal. This process continues until no further changes are made, ensuring all indirect unit productions are eliminated from the grammar. Overall, this method ensures that the grammar contains no unit productions, a step often taken in preparing a grammar for further transformations or analysis.



This block of code defines a method named remove\_inaccessible\_symbols within a grammar class, responsible for eliminating non-terminals that are not reachable from the start symbol. It starts by initializing a set accessible containing the start symbol self.S and a queue queue with the start symbol. Using a breadth-first search approach, it iteratively explores productions reachable from the start symbol. It removes symbols that are not accessible from the start symbol by traversing through the productions of each symbol in the queue, adding any new accessible symbols to the accessible set and queue. Once all accessible symbols are identified, it updates the list of non-terminals self.Vn to contain only those reachable from the start symbol. Additionally, it removes any production rules associated with non-terminals that are not accessible from the start symbol. This process ensures that the grammar only contains symbols that are reachable from the start symbol, which is essential for further transformations and analyses.

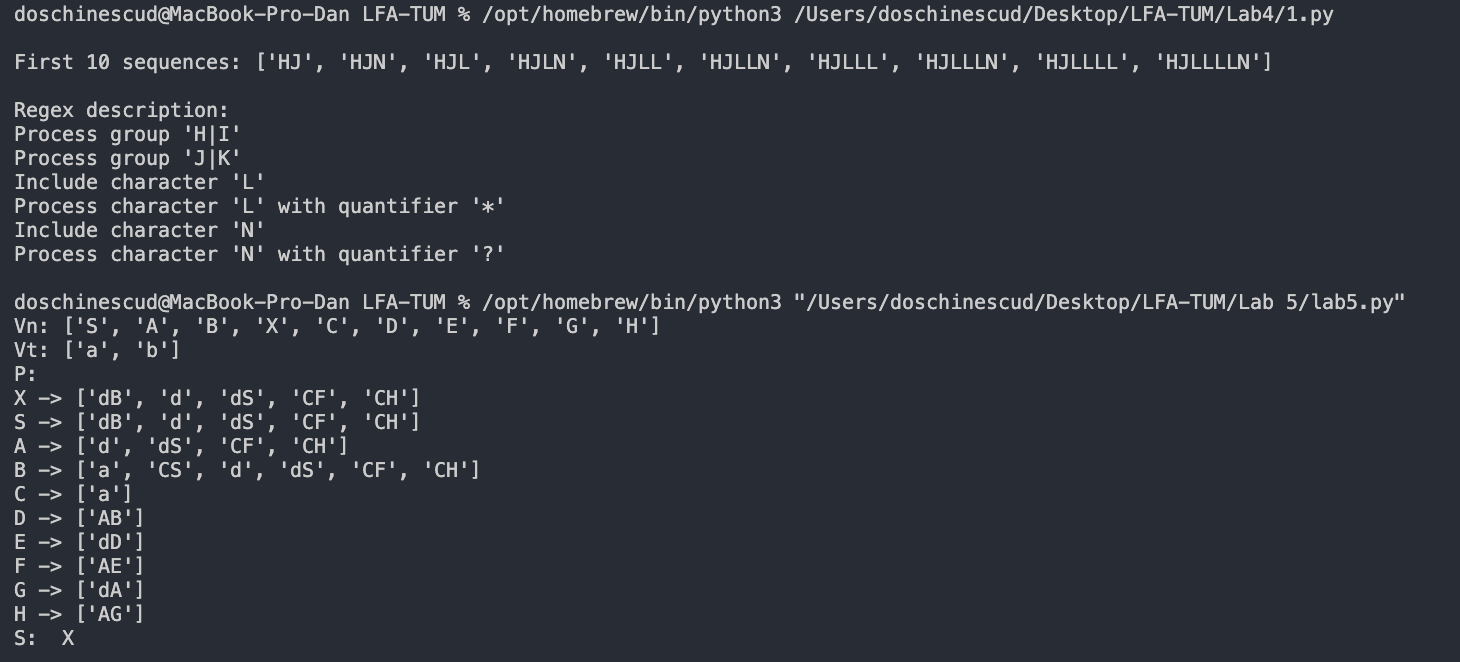


This block of code defines a method named replace\_terminals\_with\_nonterminals within a grammar class, responsible for replacing terminal symbols with corresponding non-terminal symbols. It starts by defining a nested function new\_nonterminal(existing) which generates a new non-terminal symbol that hasn't been used yet. It initializes an empty dictionary terminal\_to\_nonterminal to keep track of the mapping between terminal symbols and non-terminal symbols. The method then iterates through each production rule of the grammar and replaces terminal symbols with non-terminal symbols. For each terminal symbol found in a production rule, it generates a new non-terminal symbol if it hasn't been created yet and updates the mapping. It constructs new production rules with terminal symbols replaced by corresponding non-terminal symbols, storing them in a dictionary new\_P. Finally, it updates the grammar's production rules with the new rules containing non-terminal symbols, effectively replacing terminals with non-terminals throughout the grammar. This process ensures that the grammar only contains non-terminal symbols, which is a requirement for further transformations to Chomsky Normal Form (CNF).



This block of code defines a method named reduce\_production\_length within a grammar class, aimed at reducing the length of production rules to at most two symbols. It starts by defining a nested function new\_nonterminal(existing) which generates a new non-terminal symbol that hasn't been used yet. It initializes dictionaries existing\_binaries and new\_productions\_dict to keep track of existing binary productions and new productions created during the process. The method iterates through each production rule of the grammar. If a production rule has more than two symbols, it repeatedly breaks down the rule into binary productions until only two symbols remain. For each pair of symbols removed from the end of the production, if it hasn't been converted to a binary production before, it generates a new non-terminal symbol and updates dictionaries to keep track of the new binary productions. Finally, it updates the grammar's production rules with the new binary productions, effectively reducing the length of production rules to at most two symbols, as required for Chomsky Normal Form (CNF).

**Output:**



**5.Conclusions:**

In this series of code blocks, we've explored various methods to transform a context-free grammar into Chomsky Normal Form (CNF), an essential step in many natural language processing tasks. We began by addressing null productions, unit productions, and inaccessible symbols, ensuring the grammar meets the strict requirements of CNF. Then, we replaced terminals with non-terminals to adhere to CNF's restriction of binary productions. Additionally, we reduced production lengths to two symbols, further aligning with CNF. These transformations are crucial as CNF simplifies parsing algorithms and eliminates ambiguity in parsing trees. The methods demonstrated here provide a systematic approach to ensure grammars are suitable for efficient parsing and syntactic analysis. By understanding and implementing these transformations, developers gain a deeper understanding of formal language theory and its practical applications in computational linguistics. This process underscores the importance of meticulous attention to detail and thorough testing in language processing tool development. Overall, these transformations pave the way for robust and efficient natural language processing systems capable of handling a wide range of linguistic phenomena.

**References:**

1. <https://www.geeksforgeeks.org/write-regular-expressions/>
2. <https://coderpad.io/blog/development/the-complete-guide-to-regular-expressions-regex/>
3. <https://docs.python.org/3/library/re.html>
4. <https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html>