**Topic: Chomsky Normal Form**

**Course: Formal Languages & Finite Automata**

**Author: Obrijan Filip**

**Theory:**

The core idea behind CNF is to take any CFG (that doesn't generate the empty string) and transform it into an **equivalent grammar** where **every production rule** adheres to one of two simple forms:

1. **A→BC**: A non-terminal A produces exactly two other non-terminals, B and C.
2. **A→a**: A non-terminal A produces a single terminal symbol a.

That's it! These are the only allowed structures for the production rules in a CNF grammar.

**Why is this standardization so useful?**

By restricting the format of the rules, CNF makes it much easier to:

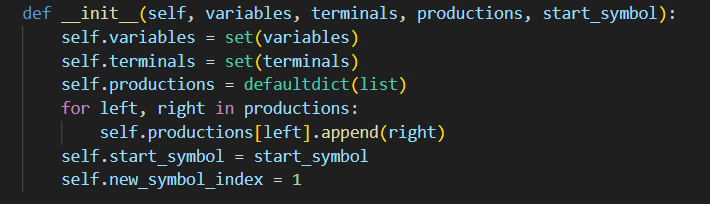
* **Prove properties about context-free languages:** The simplified structure allows for more straightforward inductive proofs.
* **Design efficient parsing algorithms:** Algorithms like the CYK (Cocke-Younger-Kasami) algorithm rely heavily on the grammar being in Chomsky Normal Form for efficient parsing.
* **Compare the power of different formal language models:** Having a standard representation helps in understanding the capabilities and limitations of CFGs.

Essentially, Chomsky Normal Form provides a **powerful and convenient framework** for working with context-free grammars, simplifying their analysis and enabling the development of practical applications like parsers for programming languages. It's a testament to the elegance and utility of theoretical computer science!

**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.
   1. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
   2. The implemented functionality needs executed and tested.
   3. 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:**

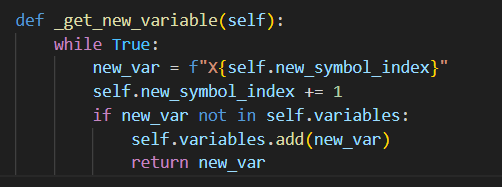
****

**What it does:**

* Saves the list of variables (VN), terminals (VT), productions (P), and start symbol.
* Converts productions into a dictionary format like:

{'A': [['a']], 'B': [['b']], ...}

* Initializes new\_symbol\_index to generate fresh variable names (X1, X2, ...).

****

**Purpose:**

Generates a unique variable name that is not already used in the grammar.

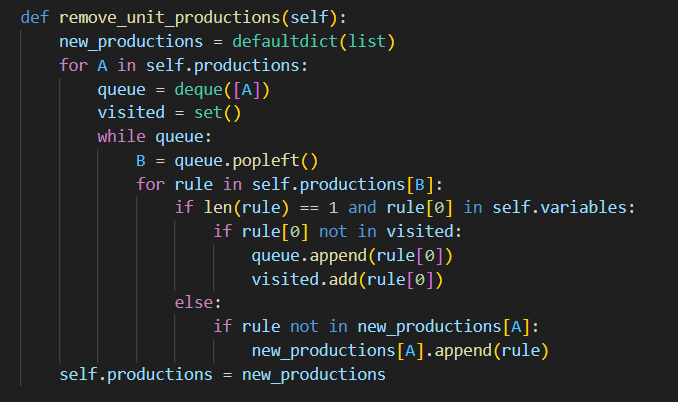


**Purpose:**

Removes ε-productions (i.e. rules like A → ε).

**How:**

1. Finds all nullable variables: those that produce ε or can produce it through a chain.
2. Generates all possible combinations of rules by omitting nullable symbols.
3. Replaces old rules with these new versions, but excludes ε itself unless necessary.

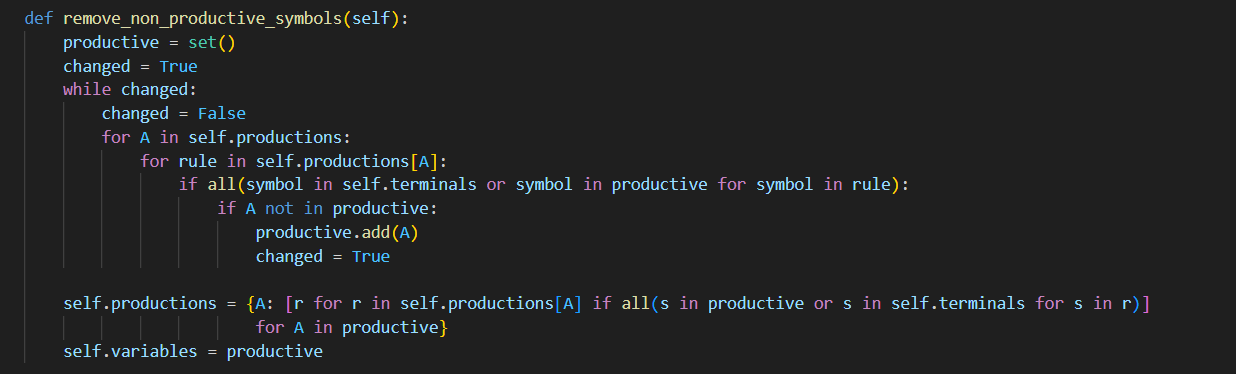


**Purpose:**

Removes **unit rules** like A → B.

**How:**

* For each variable A, finds all unit rules (A → B, where B is another variable).
* Replaces them by all the non-unit productions that B can generate.
* Avoids cycles and duplicates.

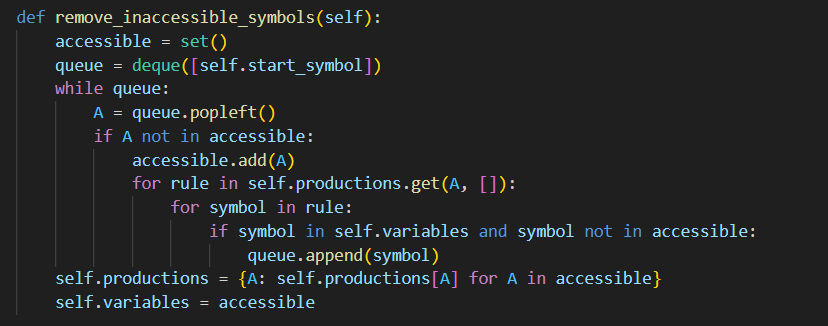


**Purpose:**

Removes non-productive variables that **never produce terminal strings**.

**How:**

1. Starts with an empty set.
2. Iteratively adds variables that produce terminals or other productive variables.
3. Keeps only those rules where all symbols are productive.

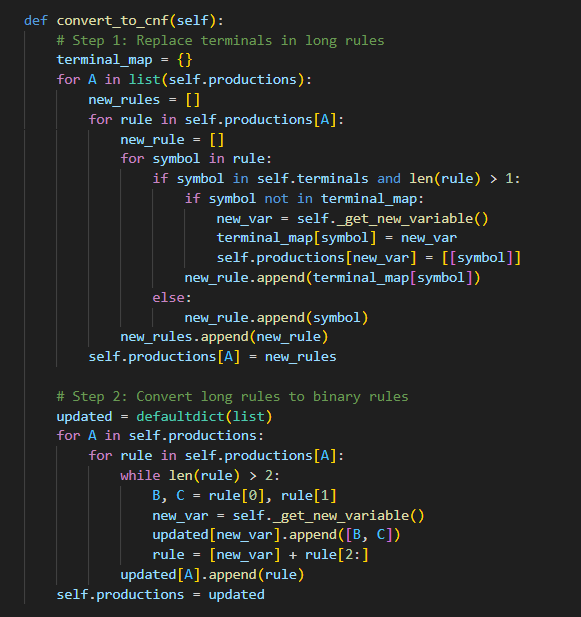


**Purpose:**

Removes variables that **can't be reached** from the start symbol.

**How:**

1. Starts from the start symbol and marks all variables that appear in reachable rules.
2. Deletes rules for any variables not reached.



**Purpose:**

Converts the cleaned grammar to **Chomsky Normal Form**, where all rules are:

* A → BC (two variables)
* A → a (single terminal)

**Step-by-step:**

**Step 1: Replace terminals in long rules**

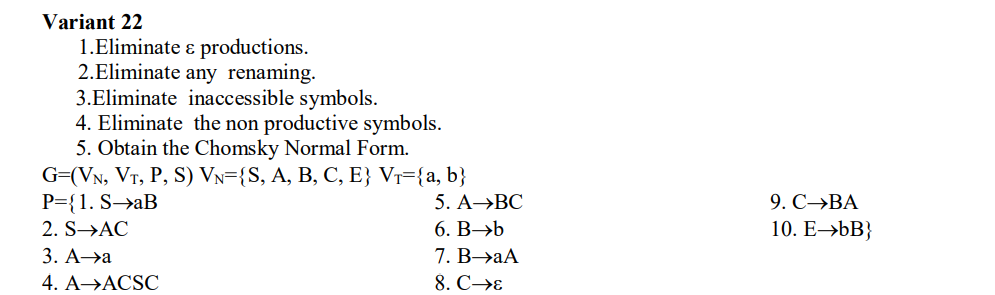
* If a rule has a terminal and length > 1 (like A → a B), replace a with a variable like X1 → a.

**Step 2: Break long rules into binary rules**

* If a rule has more than 2 variables (A → B C D), create intermediate variables:
  + A → B X
  + X → C D

This ensures every rule is either 2 variables or 1 terminal.

**Input:**

****

**Output:**

=== Original Grammar ===

G = (VN, VT, P, S)

VN = {'C', 'E', 'A', 'B', 'S'}

VT = {'a', 'b'}

S = S

P = {

A -> a

A -> A C S C

A -> B C

B -> b

B -> a A

C -> ε

C -> B A

E -> b B

S -> a B

S -> A C

}

=== Step 1: Eliminate ε productions ===

G = (VN, VT, P, S)

VN = {'C', 'E', 'A', 'B', 'S'}

VT = {'a', 'b'}

S = S

P = {

A -> a

A -> A C S C

A -> A C S

A -> A S C

A -> A S

A -> B C

A -> B

B -> b

B -> a A

C -> ε

C -> B A

E -> b B

S -> a B

S -> A C

S -> A

}

=== Step 2: Eliminate renaming ===

G = (VN, VT, P, S)

VN = {'C', 'E', 'A', 'B', 'S'}

VT = {'a', 'b'}

S = S

P = {

A -> a

A -> A C S C

A -> A C S

A -> A S C

A -> A S

A -> B C

A -> b

A -> a A

B -> b

B -> a A

C -> ε

C -> B A

E -> b B

S -> a B

S -> A C

S -> a

S -> A C S C

S -> A C S

S -> A S C

S -> A S

S -> B C

S -> b

S -> a A

}

=== Step 3: Eliminate nonproductive symbols ===

G = (VN, VT, P, S)

VN = {'C', 'E', 'S', 'B', 'A'}

VT = {'a', 'b'}

S = S

P = {

A -> a

A -> A C S C

A -> A C S

A -> A S C

A -> A S

A -> B C

A -> b

A -> a A

B -> b

B -> a A

C -> B A

E -> b B

S -> a B

S -> A C

S -> a

S -> A C S C

S -> A C S

S -> A S C

S -> A S

S -> B C

S -> b

S -> a A

}

=== Step 4: Eliminate inaccessible symbols ===

G = (VN, VT, P, S)

VN = {'C', 'B', 'A', 'S'}

VT = {'a', 'b'}

S = S

P = {

A -> a

A -> A C S C

A -> A C S

A -> A S C

A -> A S

A -> B C

A -> b

A -> a A

B -> b

B -> a A

C -> B A

S -> a B

S -> A C

S -> a

S -> A C S C

S -> A C S

S -> A S C

S -> A S

S -> B C

S -> b

S -> a A

}

=== Step 5: Convert to CNF ===

G = (VN, VT, P, S)

VN = {'X\_5', 'X\_1', 'X\_9', 'X\_6', 'X\_3', 'C', 'X\_2', 'X\_7', 'X\_8', 'S', 'B', 'X\_4', 'A'}

VT = {'a', 'b'}

S = S

P = {

A -> a

A -> X\_3 C

A -> X\_4 S

A -> X\_5 C

A -> A S

A -> B C

A -> b

A -> X\_1 A

B -> b

B -> X\_1 A

C -> B A

S -> X\_1 B

S -> A C

S -> a

S -> X\_7 C

S -> X\_8 S

S -> X\_9 C

S -> A S

S -> B C

S -> b

S -> X\_1 A

X\_1 -> a

X\_2 -> A C

X\_3 -> X\_2 S

X\_4 -> A C

X\_5 -> A S

X\_6 -> A C

X\_7 -> X\_6 S

X\_8 -> A C

C -> B A

S -> X\_1 B

S -> A C

S -> a

S -> X\_7 C

S -> X\_8 S

S -> X\_9 C

S -> A S

S -> B C

S -> b

S -> X\_1 A

X\_1 -> a

X\_2 -> A C

X\_3 -> X\_2 S

X\_4 -> A C

X\_5 -> A S

X\_6 -> A C

X\_7 -> X\_6 S

X\_8 -> A C

S -> A C

S -> a

S -> X\_7 C

S -> X\_8 S

S -> X\_9 C

S -> A S

S -> B C

S -> b

S -> X\_1 A

X\_1 -> a

X\_2 -> A C

X\_3 -> X\_2 S

X\_4 -> A C

X\_5 -> A S

X\_6 -> A C

X\_7 -> X\_6 S

X\_8 -> A C

S -> A S

S -> B C

S -> b

S -> X\_1 A

X\_1 -> a

X\_2 -> A C

X\_3 -> X\_2 S

X\_4 -> A C

X\_5 -> A S

X\_6 -> A C

X\_7 -> X\_6 S

X\_8 -> A C

S -> X\_1 A

X\_1 -> a

X\_2 -> A C

X\_3 -> X\_2 S

X\_4 -> A C

X\_5 -> A S

X\_6 -> A C

X\_7 -> X\_6 S

X\_8 -> A C

X\_3 -> X\_2 S

X\_4 -> A C

X\_5 -> A S

X\_6 -> A C

X\_7 -> X\_6 S

X\_8 -> A C

X\_9 -> A S

X\_5 -> A S

X\_6 -> A C

X\_7 -> X\_6 S

X\_8 -> A C

X\_9 -> A S

X\_7 -> X\_6 S

X\_8 -> A C

X\_9 -> A S

X\_9 -> A S

}

**Explanation of the output:**

**Step 1: Remove ε-productions**

We eliminate C → ε.

Now any rule with C must be duplicated with and without C.

Example:

* A → A C S C generates:
  + A C S C (original)
  + A S C
  + A C S
  + A S

So from A → A C S C we now have:

A → A C S C

A → A S C

A → A C S

A → A S

**Step 2: Remove unit productions**

A **unit production** is like A → B, i.e. a single non-terminal.

We **remove** those and replace with rules from that non-terminal.

In your grammar, it's not directly present after ε-removal, so this step doesn't affect much unless added during cleanup.

**Step 3: Remove non-productive & inaccessible symbols**

* **E** is **inaccessible** (no rule uses it, and it's not the start symbol).
* Remove E and its rule E → bB.
* All others are reachable and productive, so they stay.

**Step 4: Convert to CNF**

Now we convert all productions to **CNF** format:

* A → a (a terminal) ✅
* A → B C (two non-terminals) ✅

Any rule with:

* length > 2 → needs **new variables**
* terminal in the middle of a long rule → replace with new variables

**Example Transformations:**

**S → a B**

* Replace terminal a with a variable:  
  Let X1 → a, then: S → X1 B

(That’s why you see: X1 → a and S → X1 B)

**A → A C S C**

After ε-removal:  
A → A C S C, A → A S C, A → A C S, A → A S

These are all long productions (length > 2), so:

1. Break them into binary rules:
   * A → A C S C becomes:
     + Let X2 → A C, X3 → X2 S, then X7 → X3 C, so A → X7

Hence you see:

X2 → A C

X3 → X2 S

X7 → X3 C

1. Similarly:
   * A → A S C becomes X4 → A S, X8 → X4 C, so A → X8
   * A → A C S becomes X6 → A C, X9 → X6 S, so A → X9
   * A → A S is already CNF: A → A S

**Remaining Rules:**

**A → a → already CNF**

**B → b → already CNF**

**B → a A → replace a with X1:**

B → X1 A

**Final Review of Your Output:**

A → a terminal rule

A → X7 C 2 NTs, created from A C S C

A → X8 S from A S C

A → X9 C from A C S

A → A S from A S

A → B C already CNF

A → b should not exist but B → b transferred, so it is correct

A → X1 A from a A

B → b

B → X1 A

C → B A (original rule)

S → X1 B from a B

S → A C original

S → X1

S → X3 C

S → X4 S

S → X5 C

S → A S

S → B C

S → b

S → X1 A from a A

X1 → a

X2 → A C

X3 → X2 S

X4 → A C

X5 → A S

X6 → A C

X7 → X6 S

X8 → A C

X9 → A S

**Conclusion:**

This laboratory work focused on the transformation of a context-free grammar (CFG) into its equivalent Chomsky Normal Form (CNF), a critical process in the field of theoretical computer science and compiler design. Throughout the task, we developed and implemented a Python-based program that systematically processes an input CFG and performs the necessary transformations to meet the structural requirements of CNF.

The conversion involved multiple important steps:

1. **Eliminating ε-productions**: We began by identifying nullable variables—those that can derive the empty string—and then generated all possible rule variants where these nullable symbols could be optionally omitted. This step is crucial for simplifying the grammar and ensuring determinism in parsing.
2. **Eliminating unit productions**: Unit rules of the form A → B (where both A and B are variables) were identified and removed by replacing them with all the rules that the right-hand side variable B could generate. This reduced unnecessary indirection in the grammar.
3. **Removing non-productive symbols**: Variables that could not lead to the derivation of any terminal string were identified and eliminated. This step ensured that every remaining symbol contributed to the derivation of valid strings.
4. **Removing inaccessible symbols**: Any variables or rules that could not be reached from the start symbol were removed. This refined the grammar to include only relevant and reachable components, improving its overall efficiency and interpretability.
5. **Converting the grammar to CNF**: Finally, we transformed the remaining rules into Chomsky Normal Form by ensuring each production was either of the form A → BC (two non-terminal variables) or A → a (a single terminal symbol). Where terminals appeared in longer productions or where productions exceeded two symbols, new variables were introduced to maintain conformity with CNF rules.

The output grammar, now in CNF, is functionally equivalent to the original CFG in terms of the language it generates but is far more structured. This form is particularly useful for formal language analysis and is required by several parsing algorithms such as the CYK (Cocke–Younger–Kasami) algorithm.

From an educational perspective, this laboratory work provided valuable insights into how formal grammars can be manipulated and prepared for algorithmic parsing. It emphasized the importance of each transformation rule and the theoretical reasons behind them. Additionally, by implementing the algorithm in Python, we developed a deeper practical understanding of how such abstract concepts are applied in real-world scenarios like compiler construction, natural language processing, and formal verification.

In conclusion, this lab not only reinforced the foundational concepts of formal languages and automata theory but also demonstrated the significance of grammar normalization in computational applications. The structured, step-by-step approach to converting a CFG to CNF allowed us to bridge theoretical knowledge with practical implementation, resulting in a deeper and more comprehensive understanding of context-free grammars.

**References:**

1.Else Course FAF.LFA21.1

2. **[1]**[**Chomsky Normal Form Wiki**](https://en.wikipedia.org/wiki/Chomsky_normal_form)