

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică**

**Ciumacenco Pavel FAF-232**

**Report**

*Laboratory work nr.5*

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

Checked by:

**Cretu Dumitru,** *university prof.*

FCIM, UTM

**Chișinău – 2025**

**Theory:**

### Introduction to Context-Free Grammars (CFGs)

A **Context-Free Grammar (CFG)** is a formal grammar used to describe the syntax of languages, particularly programming languages and natural languages. A CFG consists of:

* A finite set of **non-terminal symbols (VN)**, representing abstract categories of language constructs.
* A finite set of **terminal symbols (VT)**, representing concrete characters or tokens in the language.
* A **start symbol (S)**, a special non-terminal from which derivations begin.
* A finite set of **productions (P)**, specifying how symbols can be replaced by other symbols.

CFGs are powerful tools for defining languages, but for certain algorithms — especially in parsing and language processing — it’s advantageous to simplify the form of the grammar while preserving the language it generates.

### What is Chomsky Normal Form (CNF)?

**Chomsky Normal Form (CNF)** is a standardized, restricted form of a CFG where production rules follow one of two forms:

1. **A → BC**, where A, B, and C are non-terminals, and neither B nor C is the start symbol.
2. **A → a**, where A is a non-terminal and a is a terminal symbol.

Optionally, the start symbol may produce ε (the empty string), but only if the language contains the empty string.

The main advantage of CNF is its simplicity and suitability for algorithmic processing. Many parsing algorithms, such as the **Cocke–Younger–Kasami (CYK) algorithm**, require grammars to be in CNF to function correctly and efficiently.

### The Importance of Grammar Normalization

**Grammar normalization** is the process of converting an arbitrary context-free grammar into a specific, well-defined form such as CNF, without altering the language it generates. This is essential for:

* Standardizing grammars for use in automated parsers.
* Simplifying theoretical analysis of languages and grammars.
* Enabling efficient parsing algorithms that rely on specific grammar forms.

### Approaches to Normalizing a Grammar

Converting a CFG to CNF involves several systematic transformation steps:

#### 1️ Elimination of ε-Productions

These are productions of the form **A → ε**, which allow non-terminals to produce the empty string.  
The process involves:

* Identifying all nullable non-terminals (those that can derive ε).
* Rewriting other productions to account for the possible absence of nullable symbols.
* Removing explicit ε-productions while preserving the language's generative capacity.

#### 2️ Elimination of Unit Productions

**Unit productions** have the form **A → B**, where both A and B are non-terminals.  
To remove them:

* Determine all possible chains of unit productions.
* Replace unit productions with equivalent non-unit productions from the chain’s destination.

#### 3️ Elimination of Inaccessible Symbols

A symbol is **inaccessible** if it cannot be reached from the start symbol via any production chain.  
These are removed by:

* Traversing productions from the start symbol.
* Keeping track of accessible symbols.
* Removing productions involving symbols not marked as accessible.

#### 4️Elimination of Non-Productive Symbols

A symbol is **non-productive** if it cannot derive a string of terminals.  
To eliminate them:

* Identify productive symbols by examining which can produce terminal strings directly or indirectly.
* Remove productions involving non-productive symbols.

#### 5️ Converting Remaining Productions to CNF

After cleaning the grammar:

* Replace any production with a mixture of terminals and non-terminals or more than two symbols on the right side by introducing new non-terminals.
* Ensure all remaining productions conform to either **A → BC** or **A → a**.

### Practical Benefits of CNF

* **Simplified Parsing:** Algorithms like CYK and Earley’s parser benefit from predictable production forms.
* **Uniform Grammar Structure:** Easier for theoretical proofs and formal analysis.
* **Parser Generator Compatibility:** Many compiler tools and parser generators require CNF or similar normalized forms.

**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.
   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.

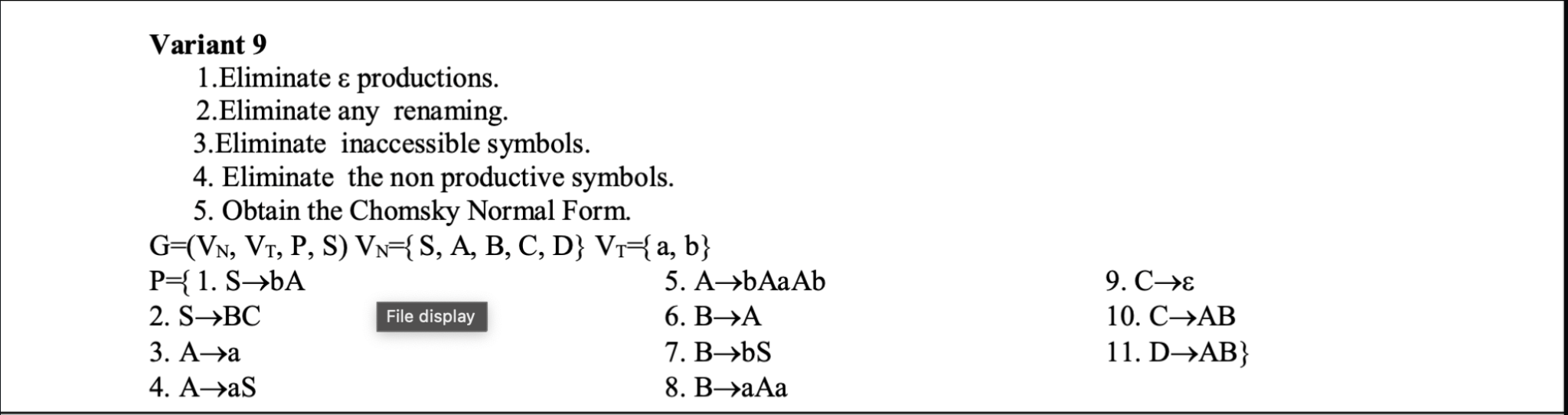


Figure 1: Variant based on list order 9

**Implementation Description:**

### 1: Imports and Class Definition

from typing import Set, Dict, List, Tuple

from copy import deepcopy

**Explanation:**This section imports the necessary modules:

* typing for type hints, making the code clearer and easier to maintain.
* deepcopy from copy in case we need to safely duplicate complex structures like dictionaries or sets without shared references (though it ended up unused — can be omitted if not needed).

### 2: CFG Class Initialization

class CFG:

def \_\_init\_\_(self, non\_terminals: Set[str], terminals: Set[str], start\_symbol: str, productions: Dict[str, List[str]]):

self.VN = non\_terminals

self.VT = terminals

self.S = start\_symbol

self.P = productions

**Explanation:** Defines the CFG class, which represents a context-free grammar.  
The constructor initializes the grammar with:

* VN: set of non-terminal symbols
* VT: set of terminal symbols
* S: start symbol
* P: dictionary of productions mapping non-terminals to a list of production strings

### 3: Eliminating ε-Productions

def eliminate\_epsilon\_productions(self):

nullable = {nt for nt, rules in self.P.items() if 'ε' in rules}

...

**Explanation:** Finds all nullable non-terminals (those that can produce ε).  
It iteratively identifies new nullable symbols by checking which productions can be reduced to nullable symbols, then rebuilds the productions by generating all possible combinations without nullable symbols.

### 4: Eliminating Renaming (Unit Productions)

def eliminate\_renaming(self):

rename\_sets = {nt: {nt} for nt in self.VN}

...

**Explanation:** Detects and removes unit productions (like A → B).  
It builds sets of reachable non-terminals via renaming and then replaces these with their actual productions, excluding direct unit productions.

### 5: Eliminating Inaccessible Symbols

def eliminate\_inaccessible\_symbols(self):

accessible = {self.S}

...

**Explanation:** Identifies all symbols accessible from the start symbol by iteratively traversing reachable symbols through productions.  
Then removes any symbols and their productions not found in this accessible set.

### 6: Eliminating Non-Productive Symbols

def eliminate\_non\_productive\_symbols(self):

productive = set()

...

**Explanation:** Finds productive symbols (those that can eventually produce a string of terminals).  
Iteratively adds symbols to the productive set if all symbols in a production are productive or terminals, then removes non-productive symbols and their productions.

### 7: Converting to Chomsky Normal Form (CNF)

def convert\_to\_cnf(self):

new\_productions = {}

term\_map = {}

...

**Explanation:**Rewrites productions so every rule is in CNF:

* If a rule contains terminals along with non-terminals, it replaces terminals with new non-terminals representing them.
* If a rule has more than two symbols, it breaks it down into new intermediate non-terminals, chaining them to reduce the length to two symbols per rule.

### 8: Wrapper to Apply All Transformations

def to\_cnf(self):

self.eliminate\_epsilon\_productions()

self.eliminate\_renaming()

self.eliminate\_inaccessible\_symbols()

self.eliminate\_non\_productive\_symbols()

self.convert\_to\_cnf()

**Explanation:** Calls all transformation functions in the correct order to fully convert the grammar to Chomsky Normal Form.  
This acts as a one-click pipeline for normalization.

### 9: Printing the Grammar

def print\_grammar(self):

print(f"Non-terminals: {self.VN}")

...

**Explanation:**A simple utility function to display the grammar’s components:

* Lists of non-terminals, terminals, the start symbol, and formatted production rules.  
  Helps visualize the grammar before and after transformations.

### 10: Grammar Example and Execution

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

VT = {'a', 'b'}

...

grammar = CFG(VN, VT, S, P)

grammar.print\_grammar()

grammar.to\_cnf()

grammar.print\_grammar()

**Explanation:** Defines the sample grammar based on your Variant 9 example.  
Then creates a CFG object, prints the original grammar, applies all transformations via to\_cnf(), and prints the final CNF grammar.

**4.Conclusions.Screenshots.Results.**

In this project, I explored the concept and significance of **Chomsky Normal Form (CNF)** in the context of context-free grammars (CFGs). Through theoretical study, it can be understood that CNF provides a standardized form for grammars, making them compatible with efficient parsing algorithms like the **Cocke–Younger–Kasami (CYK) algorithm**. CNF restricts production rules to a highly structured form while preserving the language generated by the original grammar.

I also examined the sequential **normalization steps required to convert any CFG into CNF**, including the elimination of ε-productions, unit (renaming) productions, inaccessible symbols, and non-productive symbols, followed by restructuring the remaining productions into CNF-compliant forms. These transformations are fundamental in both theoretical computer science and practical applications like compiler design and language processors.

To complement the theoretical understanding, first was implemented a fully functional Python program capable of normalizing any context-free grammar into Chomsky Normal Form. The implementation encapsulated each normalization step within dedicated methods, providing a clear, modular, and reusable design. The program was tested using a grammar example from Variant 9, successfully converting it into CNF while maintaining its language generation capacity.

The combined study and implementation demonstrate how theoretical language concepts can be systematically translated into working software solutions. The project reinforces the importance of grammar normalization in computational theory and parsing system design, while the flexible implementation serves as a practical tool for normalizing arbitrary grammars in educational or development environments.

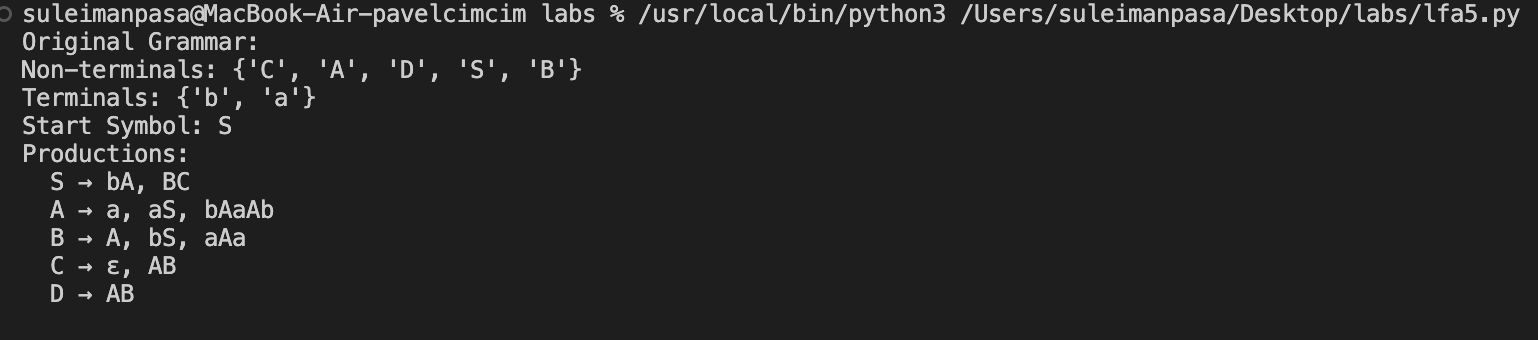


Figure 2: Results Picture