1.What do you understand by grammars in toc?

In **Theory of Computation (TOC)**, a **grammar** is a formal system that defines the rules for

2.Differentiate Deterministic Finite Automata (DFA) & Non-Deterministic Finite Automata (NFA)?

3.Construction of NFA and DFA for the usage over the alphabet (fa,b]that contain all string that start with ab.



generating strings in a language. It consists of **non-terminals, terminals, production rules,**

and a **start symbol**.

# Types of Grammars (Chomsky Hierarchy):

Feature DFA (Deterministic Finite Automata)

A finite automaton where each

NFA (Non-Deterministic Finite Automata)

A finite automaton where a state

1⃣ NFA Construction

States & Transitions:

Definition

state has exactly **one transition** for can have **multiple transitions** (or

* **q0 (Start State):** Initially, no input is processed.

1. **Type 0 (Unrestricted Grammar)** – Generates **recursively enumerable languages**.
2. **Type 1 (Context-Sensitive Grammar - CSG)** – Generates **context-sensitive languages**, recognized by a **Linear Bounded Automaton (LBA)**.

each input symbol. none) for the same input symbol.

**Determinism** Completely predictable; only **one** Multiple paths may be followed path is followed for an input string. for an input string.

* + **q1:** After reading **'a'** from the input.
  + **q2 (Final State):** After reading **"ab"**, meaning the string starts correctly.
  + **q2 (Loop on 'a' and 'b'):** Once we reach this state, we can accept any combination of 'a' and 'b'.

1. **Type 2 (Context-Free Grammar - CFG)** – Generates **context-free languages**, recognized by a **Pushdown Automaton (PDA)**.
2. **Type 3 (Regular Grammar)** – Generates **regular languages**, recognized by a **Finite Automaton (FA)**.

Epsilon (ε) Moves

**No** ε-transitions (i.e., moves without consuming input).

Allows **ε-transitions** (state changes without input consumption).

State Transition Table (NFA)

State Input 'a' Input 'b'

**Transition Function** Maps **one** state per input symbol. Maps **one or more** states per

input symbol.

**q0** q1 ∅

**q1** ∅ q2

May require **more states** than an equivalent NFA.

State Explosion

Typically requires **fewer states**

than DFA.

**q2** q2 q2

Complexity (Implementation)

Easier to implement in hardware/software.

Harder to implement due to multiple possible transitions.

NFA Diagram

q0 --a--> q1 --b--> q2 (Final)

Equivalence

DFA and NFA are **equally powerful**, NFA can be converted into DFA meaning any NFA can be converted using the **subset construction**

q2 --a--> q2

q2 --b--> q2



into an equivalent DFA.

Transition table has a single entry for each symbol.

Example

algorithm.

Transition table may have multiple entries or ε-transitions.

2⃣ DFA Construction

To construct a DFA, we ensure that at **each step, only one transition** exists per input symbol.

States:

* **q0 (Start State):** Before reading any input.
* **q1:** After reading **'a'**.
* **q2 (Final State):** After reading **"ab"** (valid start).
* **q\_dead:** A dead state for invalid cases (e.g., starting with 'b').

State Transition Table (DFA)

**State Input 'a' Input 'b' q0** q1 q\_dead

**q1** q\_dead q2

**q2** q2 q2

**q\_dead** q\_dead q\_dead

DFA Diagram

q0 --a--> q1 --b--> q2 (Final) q2 --a--> q2

q2 --b--> q2

q0 --b--> q\_dead q1 --a--> q\_dead

q\_dead --a--> q\_dead q\_dead --b--> q\_dead

4.Explain in detail about the Myhill-Nerode theorem using suitable example.

## Introduction

The **Myhill-Nerode theorem** is a fundamental result in the **Theory of Computation**, used to:

1. **Characterize Regular Languages**
2. **Minimize Deterministic Finite Automata (DFA)**
3. **Prove that a Language is Not Regular**

The theorem provides a necessary and sufficient condition for a language to be **regular**, based on the concept of **right-invariant equivalence relations**.



## Formal Definition

A language LL over an alphabet Σ\Sigma is **regular** if and only if the number of **equivalence classes** induced by the **Nerode equivalence relation** is **finite**.

# Nerode Equivalence Relation (≡ₗ)

For a language LL, we define the relation **≡ₗ** as follows:

Two strings **x** and **y** are equivalent with respect to LL (**x ≡ₗ y**) if and only if:

∀z∈Σ∗,xz∈L ⟺ yz∈L\forall z \in \Sigma^\*, \quad xz \in L \iff yz \in L

This means that **x and y lead to the same behavior** for all possible extensions **z**.

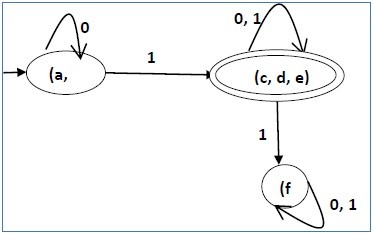


## Key Ideas Behind the Theorem

1. **Each equivalence class represents a unique behavior in the language.**
2. **The number of equivalence classes corresponds to the number of states in the minimal DFA.**
3. **If the number of equivalence classes is finite, the language is regular. Otherwise, it is non-regular.**

5.Minimize the given DFA shown below

Minimized DFA is



6.Prove that the following Language L fa"b" n>= ) is not a regular language.



## Method 1: Using the Pumping Lemma

The **Pumping Lemma** states that for any regular language LL, there exists a pumping length pp such that any string w∈Lw \in L with ∣w∣≥p|w| \geq p can be split into three parts:

w=xyzw = xyz

such that:

1. ∣xy∣≤p|xy| \leq p (first **p** characters)
2. ∣y∣>0|y| > 0 (non-empty, at least one character)
3. xykz∈Lxy^kz \in L for all k≥0k \geq 0 (repeating yy keeps it in LL)



# Step 1: Choose a String in L

Let’s choose w=apbpw = a^p b^p, where pp is the pumping length.

# Step 2: Split ww into xyzxyz

* + Since ∣xy∣≤p|xy| \leq p, **xy must contain only 'a's**.
  + Let’s assume **y = aⁿ**, where n>0n > 0 (since yy must be non-empty).

# Step 3: Pumping y (i.e., repeating y k times)

* + If we pump **y** (repeat it), we get:

wʹ=ap+nbpw' = a^{p+n} b^p

* + The new string **does not** belong to LL because it has **more 'a's than 'b's** (p+np+n 'a's and pp 'b's).

# Step 4: Contradiction

* + Since wʹ∉Lw' \notin L, LL does **not** satisfy the pumping lemma conditions.
  + Hence, LL is **not a regular language**.

7.Explain Chomsky Normal Form and Griebach Normal Form with example?

## 1⃣ Chomsky Normal Form (CNF)

A **CFG** is in **Chomsky Normal Form** if all production rules are of the form:

1. A→BCA \rightarrow BC (where **A, B, and C** are **non-terminal symbols** and **B, C ≠ start symbol**)
2. A→aA \rightarrow a (where **a** is a **terminal symbol**)
3. S→ϵS \rightarrow \epsilon (Only if SS is the **start symbol** and doesn't appear on the RHS)

# Example:

Given the CFG:

S→AB∣aS \rightarrow AB | a A→BC∣bA \rightarrow BC | b B→b,C→cB \rightarrow b, \quad C

\rightarrow c

This is **already in CNF** because:

* + Each production is either **A → BC** or **A → terminal**.

# Steps to Convert Any CFG to CNF:

1. **Eliminate Null Productions (A → ε)**
2. **Eliminate Unit Productions (A → B)**
3. **Convert Long Productions (A → BCD) into Binary Form (A → BC, C → D)**



2⃣ Greibach Normal Form (GNF)

A **CFG** is in **Greibach Normal Form** if all production rules are of the form:

A→aαA \rightarrow a\alpha

where:

* + **A** is a **non-terminal**.
  + **a** is a **terminal**.
  + **α** is a **sequence of non-terminals (possibly empty).**
  + The start symbol **must not appear on the right-hand side**.

**Example:**

Given the CFG:

S→AB∣aS \rightarrow AB | a A→BC∣bA \rightarrow BC | b B→b,C→cB \rightarrow b, \quad C \rightarrow c We convert it to **GNF**:

S→a∣bCS \rightarrow a | bC A→bCA \rightarrow bC B→bB \rightarrow b C→cC \rightarrow c

Now, each rule starts with a **terminal** followed by non-terminals

8.Define Pushdown Automata and also explain Acceptance by Final State and Acceptance by Empty Stack?

Pushdown Automata (PDA)

A **Pushdown Automaton (PDA)** is a type of **finite state machine** that includes an additional **stack** for memory. This allows it to recognize **Context-Free Languages (CFLs)**, which are more powerful than regular languages.

Formal Definition of PDA

A **PDA** is defined as a 7-tuple:

(Q,Σ,Γ,δ,q0,Z0,F)(Q, \Sigma, \Gamma, \delta, q\_0, Z\_0, F)

where:

* + **Q** = Finite set of states
  + **Σ** = Input alphabet
  + **Γ** = Stack alphabet
  + **δ** = Transition function: Q×Σϵ×Γϵ→Q×Γϵ∗Q \times Σ\_{\epsilon} \times Γ\_{\epsilon} \rightarrow Q

\times Γ\_{\epsilon}^\*

* + **q₀** = Initial state
  + **Z₀** = Initial stack symbol
  + **F** = Set of final states

The **stack** enables PDAs to **store and retrieve** data, allowing them to process nested structures (e.g., balanced parentheses).



Two Types of PDA Acceptance

There are **two ways** in which a PDA can accept a string:

1⃣ Acceptance by Final State

* + The PDA **accepts** the input string **if it reaches a final state after reading the entire input**.
  + It does **not matter** what is left in the stack.
  + This is similar to how a **Deterministic Finite Automaton (DFA)** works.

*Example*

For the language L={anbn∣n≥0}L = \{ a^n b^n | n \geq 0 \}:

1. Push 'a' onto the stack for each input 'a'.
2. For each 'b', pop an 'a' from the stack.
3. If the input ends and we reach a **final state**, we accept.

2⃣ Acceptance by Empty Stack

* + The PDA **accepts** the input string if, after reading the entire input, the **stack is empty**.
  + It does **not depend** on reaching a final state.
  + This method is useful when using the stack to track **matching symbols** (e.g., checking for balanced parentheses).

9.What do you understand DY Turing Machine& Explain multiple tapes turing machine? Explain Halting problem.

Deterministic Turing Machine (DTM)

A **Deterministic Turing Machine (DTM)** is a **Turing machine** where every configuration has at most **one possible move**. This means that for each state and tape symbol, there is at most one transition rule that determines:

1. The **next state**.
2. The **symbol** to be written on the tape.
3. The **direction** of head movement (**left or right**).

**Formal Definition of a DTM**

A DTM is a **7-tuple**:

(Q,Σ,Γ,δ,q0,qaccept,qreject)(Q, \Sigma, \Gamma, \delta, q\_0, q\_{accept}, q\_{reject})

where:

* + **Q** = Finite set of states
  + **Σ** = Input alphabet (excluding the blank symbol)
  + **Γ** = Tape alphabet (including the blank symbol ⊔)
  + **δ** = Transition function:

δ:Q×Γ→Q×Γ×{L,R}\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}

* + **q₀** = Initial state
  + **q\_{accept}** = Accept state
  + **q\_{reject}** = Reject state

A **DTM is deterministic** because for every combination of **state** and **tape symbol**, there is **only one possible move**.

Multiple Tape Turing Machine

A **Multiple Tape Turing Machine (MTM)** is an extension of a standard **Turing Machine (TM)** that has **more than one tape** and **multiple read/write heads**. Each tape operates independently but can be accessed simultaneously.

**Formal Definition**

A **Multiple Tape Turing Machine** is a **7-tuple**:

(Q,Σ,Γ,δ,q0,qaccept,qreject)(Q, \Sigma, \Gamma, \delta, q\_0, q\_{accept}, q\_{reject})

where:

* + **Q, Σ, Γ, q₀, q\_{accept}, q\_{reject}** have the same meaning as in a **single-tape TM**.
  + **δ** (transition function) now takes multiple tape symbols as input and modifies multiple tapes simultaneously: δ:Q×Γk→Q×Γk×{L,R,S}k\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k where **k** is the number of tapes.

**Working of a Multi-Tape TM**

* + Each **tape** has its own **head**.
  + The transition function **reads symbols from all tapes simultaneously**.
  + It **writes symbols** and **moves heads** on all tapes in one step.

10.Explain Recursive and Recursively Enumerable language. Explain P and NP type of Problem? write any three example of P and NP type problem?

## Recursive and Recursively Enumerable Languages

**1⃣ Recursive Languages (R)**

A language **L** is **Recursive** if there exists a **Turing Machine (TM)** that:

**Always halts** (either in an accept or reject state).

**Decides membership** (i.e., given an input string ww, it always determines whether w∈Lw \in L or w∉Lw \notin L).

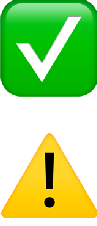
*Key Properties:*

* + Every **recursive language is decidable**.
  + If a problem is recursive, there exists an **algorithm** that solves it **in finite time**.

*Example:*

* + **Checking whether a given number is prime.**
  + **Palindrome checking.**
  + **String length comparison.**

**2⃣ Recursively Enumerable (RE) Languages**

A language **L** is **Recursively Enumerable (RE)** if there exists a **Turing Machine (TM)** that: Accepts strings in **L** (halts in an accept state).

**May not halt** for strings **not in L**.

*Key Properties:*

* + If w∈Lw \in L, the **TM eventually accepts**.
  + If w∉Lw \notin L, the **TM may loop indefinitely**.
  + Every **Recursive Language is RE, but not all RE languages are Recursive**.

*Example:*

* + **Halting Problem:** Determining whether an arbitrary Turing Machine halts on a given input.
  + **Post Correspondence Problem (PCP):** Checking whether a given set of string pairs forms a valid sequence.
  + **Diophantine Equations:** Determining whether integer solutions exist for polynomial equations.

P and NP Problems

**1⃣ P-Class (Polynomial Time)**

* + **Definition:** Problems that can be solved by a **deterministic Turing Machine (DTM)** in **polynomial time**

(O(nk)O(n^k), where kk is a constant).

* + **Characteristics:**
    - **Efficiently solvable** problems.
    - **Every problem in P is also in NP**.

*Examples of P Problems:*

1. **Sorting a list of numbers** (Merge Sort, Quick Sort, etc.).
2. **Finding the shortest path in a graph** (Dijkstra’s Algorithm).
3. **Matrix multiplication**.

**2⃣ NP-Class (Non-Deterministic Polynomial Time)**

* + **Definition:** Problems that can be **verified** in **polynomial time** by a **non-deterministic Turing Machine (NDTM)**.
  + **Characteristics:**
    - **Solutions can be checked quickly, but finding them may take exponential time**.
    - **Contains all problems in P**.
    - **If a problem is NP-complete, then solving it efficiently would solve all NP problems efficiently**.

*Examples of NP Problems:*

1. **Travelling Salesman Problem (TSP)** (Finding the shortest route visiting all cities exactly once).
2. **Knapsack Problem** (Maximizing value in a weight-limited bag).
3. **Boolean Satisfiability Problem (SAT)** (Finding satisfying assignments for a boolean formula).

**Key Difference:**

**Class Solution Finding Solution Verification P** Polynomial time Polynomial time

**NP** Exponential time (worst case) Polynomial time