1.Define alphabet, string, and language with examples.

1. Alphabet: - An **alphabet** is a finite, non-empty set of symbols.
Example: {a, b} is an alphabet containing two symbols.

2. String: - A **string** is a finite sequence of symbols taken from an alphabet.
Example: "abba" is a string over the alphabet {a, b}.

3. Language: - A **language** is a set of strings formed using symbols from a given alphabet.
Example: {“a”, “ab”, “bba”} is a language over the alphabet {a, b}.

2. Explain Chomsky's hierarchy with an example for each type of language.

Type 0 – Recursively Enumerable Languages (Unrestricted Grammars):

- Most general type; accepted by a **Turing Machine**.
- **Example:** Language of valid programs in a general-purpose programming language.

Type 1 – Context-Sensitive Languages:

- Rules depend on the context; accepted by a **Linear Bounded Automaton**.
- **Example:** $\{ a^n b^n c^n \mid n \geq 1 \}$ — equal numbers of a, b, and c.

Type 2 – Context-Free Languages:

- Rules have a single non-terminal on the left; accepted by a **Pushdown Automaton**.
- **Example:** Balanced parentheses like "(()())".

Type 3 – Regular Languages:

- Simplest; accepted by a **Finite Automaton**.
- **Example:** Strings with only a and b, like { a*, ab*, (ab) * }.

3. Determine how many states are required to build an equivalent Finite Automata for the given regular expression (0+1) *(00+11) (0+1) *

✅ **Step-by-step Explanation:**

The expression **(0+1) *(00+11) (0+1) *** accepts **all strings that contain either "00" or "11" as a substring**, with any combination of 0s and 1s before or after.

We are asked to find the **minimum number of states** needed to build a **Deterministic Finite Automaton (DFA)** that accepts this language.

🔍 **Key Insight:**

To construct a DFA that accepts strings containing either **"00" or "11"**, we can:

- Track the **recent symbols** to detect either **"00"** or **"11"**.
- Once we detect "00" or "11", we go to an **accepting state** that accepts anything afterward.

📊 **DFA Construction Overview:**

We need states to track the following:

1. **q0** – Start state

2. **q1** – Last input was 0
3. **q2** – Last input was 1
4. **q3** – Accepting state after detecting "00" or "11"

Transitions:

- From **q0**:
  - 0 → q1
  - 1 → q2
- From **q1**:
  - 0 → q3 (detected "00")
  - 1 → q2
- From **q2**:
  - 1 → q3 (detected "11")
  - 0 → q1
- From **q3**:
  - 0 or 1 → stay in q3 (accept anything)

✅ **Final Answer:**

4 states are required to build the equivalent DFA for the given regular expression.

3. Interpret the reason why the length of output is more than input in a Moore Machine?

In a **Moore Machine**, the **output is associated with states**, not transitions. This means:

- An **output is produced for every state**, including the **initial state**, **before** any input is processed.
- So, for an input string of length **n**, the machine visits **(n + 1)** states (starting + n transitions).
- Therefore, the **output length = input length + 1**.

Example:

If input = "101" (length 3), the Moore machine produces 4 outputs — one for the initial state, and one for each transition.

5.Illustrate the way of designing a DFA.

1. **Identify the Alphabet (Σ):**
   - Define the set of input symbols.
   - *Example:* Σ = {0, 1}
2. **Define the States (Q):**
   - List all necessary states based on the problem.
   - Include a **start state** and one or more **accepting (final) states**.

Construct the Transition Function (δ):

- For each state and input symbol, define exactly **one next state**.
- $\delta: Q \times \Sigma \rightarrow Q$

Mark the Start State (q_0): - The state where the DFA begins.

Define the Accepting States (F): - Set of states where input strings are accepted.

Example: DFA to accept strings ending with "01" over Σ = {0, 1}:

- **States (Q):** {q0, q1, q2}
- **Start state:** q0
- **Accepting state:** q2
- **Transitions:**
  - q0 → 0 → q1
  - q1 → 1 → q2

5.Illustrate an example to explain the process used to convert a non-deterministic finite automaton to deterministic finite automata.

Step-by-Step: Converting NFA to DFA (Subset Construction Method)

Example NFA: Let the NFA accept strings ending in "01" over Σ = {0, 1}.

NFA Description:

- States: {q0, q1, q2}
- Start state: q0
- Accepting state: q2
- Transitions:
  - q0 → 0 → q0, q1 *(non-deterministic)*
  - q0 → 1 → q0
  - q1 → 1 → q2

🔄 **Conversion Process (Subset Construction):**

Step 1: Create DFA States as Sets of NFA States

Start with ε-closure of start state: DFA Start state = {q0}

Now compute transitions for each input:

Current DFA State	Input	Resulting NFA States	New DFA State
{q0}	0	{q0, q1}	A = {q0, q1}
{q0}	1	{q0}	B = {q0}
{q0, q1} (A)	1	{q0, q2}	C = {q0, q2}

Current DFA State	Input	Resulting NFA States	New DFA State
{q0, q1} (A)	0	{q0, q1}	A
{q0, q2} (C)	0	{q0, q1}	A
{q0, q2} (C)	1	{q0}	B

✅ **Final DFA Components:**

- **States:** { {q0}, {q0, q1}, {q0, q2} }
- **Start State:** {q0}
- **Accepting State:** Any set that includes q2 → i.e., **{q0, q2}**
- **Deterministic transitions** (1 for each symbol from each state)

6. Establish a DFA to accept strings of a's and b's starting with the string ab

DFA Components:

- **Alphabet (Σ):** {a, b}
- **States (Q):** {q0, q1, q2, q3}
- **Start state:** q0
- **Accepting state:** q2
- **Dead state:** q3 (optional, for invalid paths)

🔁 **Transition Table:**

Current State	Input	Next State	Description
q0	a	q1	Read first 'a'
q0	b	q3	Invalid start
q1	b	q2	Valid "ab" prefix
q1	a	q3	Invalid: starts with "aa"
q2 (accepting)	a	q2	Accept rest
q2 (accepting)	b	q2	Accept rest
q3 (dead)	a	q3	Trap state
q3 (dead)	b	q3	Trap state

7. Construct the transition diagram of a FA which accepts all strings of 1's and 0's in which both the number of 0's and 1's are even.

States: Let each state represent (parity of 0s, parity of 1s):

State	Meaning
q0	Even 0s, Even 1s (**Start & Accepting**)
q1	Even 0s, Odd 1s
q2	Odd 0s, Even 1s
q3	Odd 0s, Odd 1s

Transition Table:

Current State	Input	Next State
q0	0	q2
q0	1	q1
q1	0	q3
q1	1	q0
q2	0	q0
q2	1	q3
q3	0	q1
q3	1	q2

Accepting State: Only **q0** is accepting (when both counts are even).

8. Briefly describe the different types of grammar on the basis of Productions.

Type 0 – Unrestricted Grammar:

- **Production form:** $\alpha \rightarrow \beta$
- α and β can be any combination of terminals and non-terminals, with $\alpha \neq \varepsilon$.
- **Most powerful**, used in Turing Machines.
- **Example***:* ABa → aBb

Type 1 – Context-Sensitive Grammar:

- **Production form:** $\alpha A \beta \rightarrow \alpha \gamma \beta$
- Length of $\gamma \geq 1$ (i.e., RHS is at least as long as LHS).
- Used in **Linear Bounded Automata**.
- **Example:** aAb → abb

Type 2 – Context-Free Grammar:

- **Production form:** A → γ
- A single non-terminal on the left, γ can be any string.
- Used in **Pushdown Automata**.
- **Example:** S → aSb | ε

Type 3 – Regular Grammar:

- **Production form:** A → aB or A → a
- Right-linear (or left-linear) productions.
- Used in **Finite Automata**.
- **Example***:* A → aB, B → b

9. Explain the difference between two types of FA with an example.

Aspect	Deterministic Finite Automaton (DFA)	Non-deterministic Finite Automaton (NFA)
Definition	For each state and input symbol, there is **exactly one** next state.	For a state and input symbol, there can be **zero, one, or multiple** next states.
Transitions	Transitions are **deterministic** and unique.	Transitions can be **non-deterministic** (multiple or no transitions).
ε-moves	No ε (epsilon) transitions allowed.	Can have ε-transitions (move without consuming input).
Acceptance	Accepts if the unique path ends in an accepting state.	Accepts if **any** possible path ends in an accepting state.

10. List out the identities of Regular expression.

1. **Identity for Union**:
   $R + \emptyset = R$
   (Union with the empty set yields the original regular expression.)
2. **Identity for Concatenation**:
   $R \cdot \varepsilon = \varepsilon \cdot R = R$
   (Concatenation with epsilon does not change the regular expression.)
3. **Identity for Kleene Star**:
   $R^* = \varepsilon + R \cdot R^*$
   (Defines that Kleene star includes epsilon and repeated applications of R.)

11. State and Prove Arden's Theorem

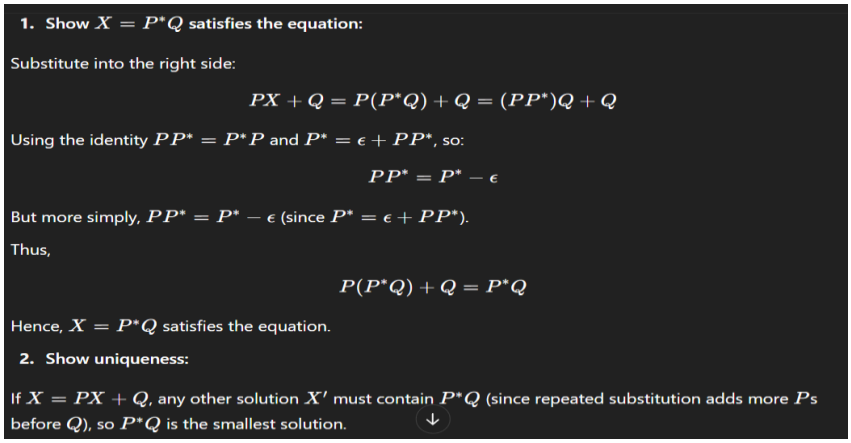
If P and Q are regular expressions over an alphabet, and P does **not** contain the empty string ϵ epsilonϵ, then the equation

X = PX + Q

has the **unique** solution

X = P ∗ Q

Proof : -



12. Explain the applications of Context Free Grammar?

Programming Language Syntax:

- CFGs are used to **define the syntax** of programming languages.
- Example: C, Java, and Python grammar rules (like loops, if-statements) are described using CFG.

Compiler Design:

- CFG is used in **parsing** to build syntax trees during the compilation process.
- Helps detect **syntax errors** in source code.

Natural Language Processing (NLP):

- Used to model the **structure of natural languages** (English, etc.).
- CFGs help in **parsing sentences** to understand grammatical structure in AI applications.

13. Explain Leftmost and Rightmost derivation tree with an example.

Leftmost Derivation:

In leftmost derivation, at each step, the leftmost non-terminal is replaced first.

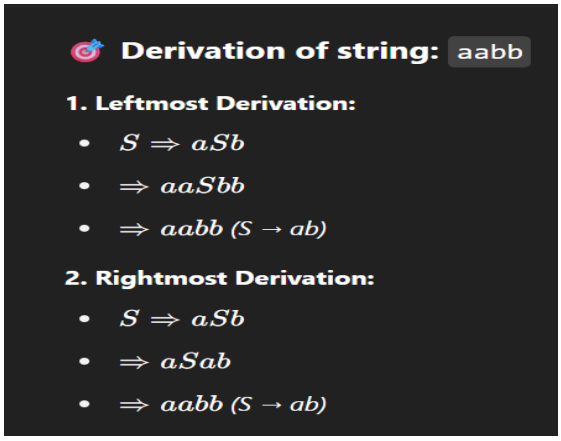
Rightmost Derivation:

In **rightmost derivation**, at each step, the **rightmost non-terminal** is replaced first.

Example Grammar (CFG):

Let

- S → aSb | ab



14.Define Moore's Machine with an example.

A **Moore Machine** is a type of Finite Automaton where the **output is associated with states**, not transitions.

- Output depends **only on the current state**, not the input symbol.
- It is defined as a 6-tuple:
  M= (Q, Σ, Δ, δ, λ, q0) M where:
  - Q: set of states
  - Σ: input alphabet
  - Δ: output alphabet
  - δ: transition function (Q × Σ → Q)
  - λ: output function (Q → Δ)
  - q0: initial state

🎯 **Example:**

Let's design a Moore Machine that **outputs 1 if the number of 1's seen so far is even**, else **0**.

States:

- **q0**: Even number of 1's → Output = 1
- **q1**: Odd number of 1's → Output = 0

Transitions:

Current State	Input	Next State
q0	0	q0
q0	1	q1
q1	0	q1
q1	1	q0

Output Function: λ(q0) = 1

λ(q1) = 0

15. Explain the working and formal defination of Push Down Automata utilizing a stack.

A Pushdown Automaton (PDA) is a 7-tuple:

M= (Q, Σ, Γ, δ, q0, Z0, F)

Where:

- Q: Finite set of states
- Σ: Input alphabet
- Γ: Stack alphabet
- δ: Transition function

δ: Q× (Σ ∪ {ϵn}) ×Γ→P(Q×Γ∗)

- q0 ∈ q: Initial state
- Z0 ∈ Γ: Initial stack symbol
- F ⊆ Q: Set of accepting states

⚙️ **Working of PDA (Using a Stack):**

- PDA reads an input symbol and top of the stack.
- Based on this, it:
    - Moves to a new state
    - Pushes or pops symbols on the stack
    - Optionally reads ε (empty) instead of input or stack
- It accepts strings by either:
    - Final state
    - Or empty stack (depending on the design)

📘 **Example:**

PDA for language L={a^nb^n|n≥0}

Idea:

- Push a symbol for each a onto the stack.
- Pop one for each b.
- Accept if stack is empty and input is finished.

16. Express the DFA to accept strings of a's and b's having a sub string aa with a diagram ?

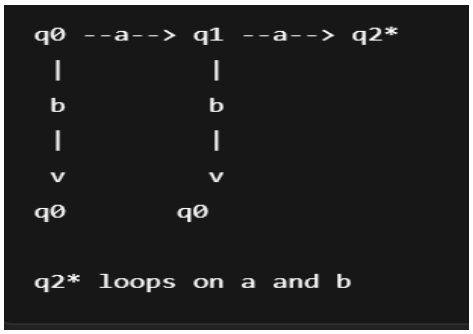
Let's define **3 states**:

State	Meaning
q0	Start state; haven't seen "a" yet

State	Meaning
q1	Seen one "a"
q2	Seen "aa" → **Accepting State**

Transition Table:

Current State	Input	Next State	Explanation
q0	a	q1	Seen first "a"
q0	b	q0	Ignore b
q1	a	q2	Found "aa" → Accept
q1	b	q0	Restart search
q2 (accepting)	a	q2	Stay accepting
q2 (accepting)	b	q2	Stay accepting



17. Apply the rules to Convert the grammar into CNF by an example ?

- All production rules must be in the form:
    - A→BC (where A, B, C are non-terminals and B, C are not the start symbol)
    - A→ a (where a is a terminal)
    - (Optionally, S→ ε S→ ε, only if the original grammar allows the empty string)

Example S → ASA | aB

A → B | S

B → b | ε

Step 2: Remove unit productions: S → ASA | aB | a

A → ASA | aB | a | b

B → b

Step 3: Convert to CNF format: a → A_T

b → B_T

So we define: A_T → a

B_T →b

Final CNF Grammar:

S → A1 A | A_T B | A_T

A1 → A S

A → A1 A | A_T B | A_T | B_T

B → B_T

A_T → a

B_T → b

18. Explain the key differences between Mealy and Moore machines. In what scenarios would you prefer to use each type of finite state machine? Compare and contrast their advantages and disadvantages.

Feature	**Mealy Machine**	**Moore Machine**
Output Depends On	Current **state and input**	Only on the **current state**
Output Timing	Output can **change instantly** with input	Output **changes only on state transition**
State Complexity	Typically requires **fewer states**	Often requires **more states**
Reaction Speed	**Faster** reaction to inputs	May react **slower**, as output lags a transition

When to Use Each :-

- **Mealy Machine:**
  - When **immediate output reaction** is needed (e.g., real-time systems).
  - When aiming for a **compact design** (fewer states).
- **Moore Machine:**
  - When **predictable and stable outputs** are important (e.g., digital circuit design).
  - Easier to **design and debug** due to state-based output.

Advantages & Disadvantages

- **Mealy:**
  - Faster response
  - Fewer states
  - More complex logic due to input-dependent output
- **Moore:**
  - Simpler design
  - More reliable output timing
  - May require more states

19. Determine the Instantaneous description (ID) in PDA with an example.

An **Instantaneous Description (ID)** of a **Pushdown Automaton (PDA)** represents its **current configuration** at any point during computation.
It is written as: (q, w, γ)

Where:

- Q = current **state**
- W = **remaining input**
- Γ = **stack contents** (top at left)

✅ **Example**

Let's consider a PDA that accepts the language: L={a^nb^n|n≥1}

PDA Transitions:

- Δ (q0, a, Z) = (q0, AZ)
- Δ (q0, a, A) = (q0, AA)
- Δ (q0, b, A) = (q1, ε)
- Δ (q1, b, A) = (q1, ε)
- Δ (q1, ε, Z) = (q accept, Z)

20. Explain the equivalence of CFL and PDA.

Definition

- A **Context-Free Language (CFL)** is a language that can be generated by a **Context-Free Grammar (CFG)**.
- A **Pushdown Automaton (PDA)** is a machine that accepts languages using **a stack** for memory.

✅ **Equivalence Statement**

A language is **context-free if and only if** there exists a **Pushdown Automaton (PDA)** that accepts it.

This means:

- **Every CFL** can be accepted by **some PDA**.
- **Every PDA** corresponds to a **CFG** that generates the same language.

✅ **Two-Way Conversion**

1. **CFG → PDA:**
    - Construct a PDA that simulates the **leftmost derivation** of the CFG using a stack.
2. **PDA → CFG:**
    - For every PDA, a CFG can be constructed by **tracking transitions** from state to state using grammar rules.

21. Discuss the ambiguity in CFG with an example.

Definition:- A Context-Free Grammar (CFG) is said to be ambiguous if at least one string in the language can have more than one leftmost derivation or more than one parse tree.

Why It Matters:- Ambiguity creates confusion in parsing, especially in programming languages where the meaning (semantics) must be clear.

✅ **Example :-** Consider the CFG:

E → E + E

E → E * E

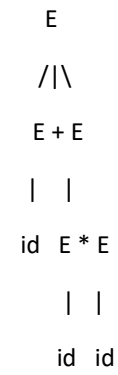
E → (E)

E → id

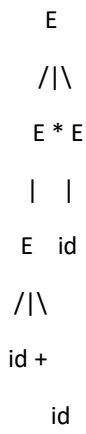
String: id + id * id

This string has **two different parse trees**:

1. <u>First Parse Tree (Addition first)</u>



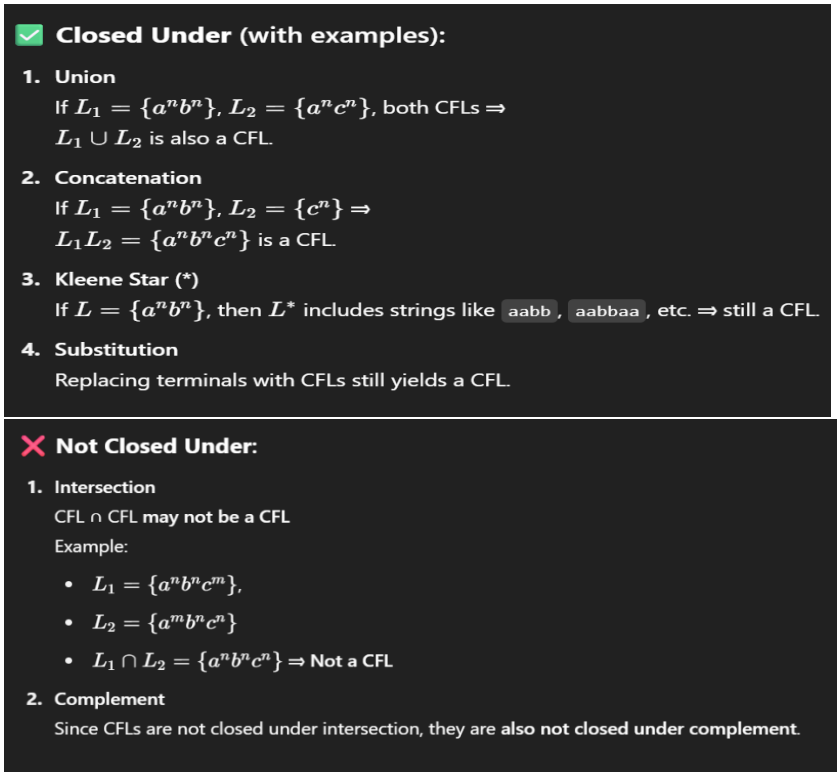
2. <u>Second Parse Tree (Multiplication first)</u>



22. Explain the Closure properties of Context Free Languages with example.

Closure properties describe the ability of a language class (like CFLs) to remain **within the class** when certain operations are applied.

Context-Free Languages (CFLs) are **closed under some operations**, but **not all**.



23. Describe Chomsky Normal Form (CNF) and Greibach Normal Form (GNF)

1. Chomsky Normal Form (CNF)

A **Context-Free Grammar (CFG)** is in **CNF** if all production rules are of the form:

- A→BC (where A, B, C are non-terminal symbols, and B, C ≠ start symbol)
- A→ a (where a is a terminal)
- Optionally: S→ ε (if the language includes the empty string)

🔸 **Purpose**: Simplifies parsing and is used in algorithms like the **CYK algorithm**.

Example:

S → AB

A → a

B → b

✅ **2. Greibach Normal Form (GNF)**

A CFG is in **GNF** if all production rules are of the form:

- A→ aα (where a is a terminal and α\alphaα is a string of non-terminals, possibly empty)

🔸 **Purpose**: Useful for building **top-down parsers**, especially **LL(1)** parsers.

Example:

S → aA

A → bS | c

24. Explain the process for converting PDA into an equivalent CFG.

Step-by-Step Conversion Process :- Assume the PDA is of the form:

- Accepts by **empty stack** (can be adjusted if it accepts by final state)
- Has states: QQQ, input alphabet: ΣΣΣ, stack alphabet: ΓΓΓ

Steps:

1. **Create variables of the form A_{pq}**- Each variable A_{pq} represents the set of strings that take the PDA from state p to state q, **emptying the stack** in the process.
2. **Add production rules**- For each transition of the PDA:
    - Simulate how a string transitions the PDA from one state to another while manipulating the stack.
    - Add corresponding grammar rules for combinations of stack operations and state transitions.
3. **Define the start variable** - The start symbol is typically A_{q0qf}, where:
    - q0 = start state of the PDA
    - qf = final state (or state where the stack becomes empty)
4. **Add ε-productions if needed** - To represent transitions that consume no input or manipulate the stack without changing the state.

25. Explain linear grammar with example.

A **linear grammar** is a type of **context-free grammar** where **each production rule has at most one non-terminal symbol** on the right-hand side.

In other words, the production rules are of the form:

- A→ xBy , or
- A→ w

where A,B are non-terminals and x, y, w are strings of terminals (possibly empty), but **only one non-terminal is allowed per production**.

✅ **Types of Linear Grammar**

- **Right-linear grammar:** The non-terminal appears at the **right end** (e.g., A→ aB or A→ a)
- **Left-linear grammar:** The non-terminal appears at the **left end** (e.g., A→ Ba or A →a)

✅ **Example -** Consider the grammar:

S → aS | bA | c

A → a | aS

26. Define computations of a TM.

A **computation** of a **Turing Machine (TM)** is the **sequence of steps** the machine performs on an input string, moving from one configuration (instantaneous description) to another until it **halts** (accepts or rejects).

Each step is determined by:

- The **current state**
- The **symbol under the tape head**
- The **transition function**

✅ **Turing Machine Configuration**

A **configuration** (or Instantaneous Description, ID) is written as: αqβ

Where:

- α: tape content to the **left** of the head
- q: current **state**
- β: tape content under and **right** of the head

✅ **Computation Steps**

1. **Start configuration**: TM begins in the start state with the input on the tape and head at the first symbol.
2. **Transition**: Based on the current state and symbol, it:
    - Writes a symbol
    - Moves Left or Right
    - Changes state
3. **Halting**: TM stops when it reaches an **accepting**, **rejecting**, or undefined configuration.

27. Describe linear bounded automaton.

A **Linear Bounded Automaton (LBA)** is a **restricted type of Turing Machine (TM)** where the **tape is bounded** to the **length of the input**.

- The tape head **cannot move** beyond the portion of the tape that contains the input.

✅ **Key Characteristics**

1. **Deterministic or Non-deterministic**: Can be either.
2. **Tape Usage**: The TM's tape is limited to **a linear function of the input size**.
3. **Acceptance**: It accepts or rejects by **halting** in an accepting or rejecting state.

✅ **Formal Definition**

An LBA is a **non-deterministic Turing Machine** with:

- A **tape** restricted to cells **containing the input**
- **No access** to infinite tape space beyond input length

✅ **Example**

For input string abba, the LBA tape is limited to **just 4 cells**.
The LBA might check if the string is a **palindrome** using **in-place replacement** and matching characters without moving outside the 4 cells.

28. Enumerate in detail about the different variations of the Turing Machine?

Multi-Tape Turing Machine

- Has **multiple tapes**, each with its own read/write head.
- Transition function reads and writes on all tapes simultaneously.
- **Faster** in practice, but **equivalent in power** to standard TM.

✅ **Example:** One tape for input, another for intermediate calculations.

🔷 **2. Non-deterministic Turing Machine (NTM)**

- Can make **multiple moves** from a given configuration.
- Accepts if **any sequence of transitions** leads to an accept state.
- **Equivalent** in power to deterministic TMs (DTMs), but may solve problems **faster** theoretically.

🔷 **3. Multi-track Turing Machine**

- A **single tape** is divided into **tracks**, each cell holds a tuple of symbols.
- One head reads/writes to all tracks at once.
- Useful for storing **multiple types of information** in parallel.

🔷 **4. Two-way Infinite Tape Turing Machine**

- The tape is **infinite in both directions** (left and right).
- Equivalent to standard TM (which is infinite in one direction only).
- Can be simulated using a standard TM with **proper encoding**.

29. State Church Turing Thesis and how is it important in context of Turing Machines.

The **Church-Turing Thesis** states:

"Any function that can be effectively computed by an algorithm can also be computed by a Turing Machine."

It implies that **Turing Machines** capture the full power of **mechanical computation**.

✅ **Importance in Context of Turing Machines**

1. **Foundation of Computability Theory**
   It defines what is **computable** in principle and sets the **limits** of what machines can do.
2. **Standard Model of Algorithms**
   Turing Machines are used as the **universal model** for all algorithms and programming languages.
3. **Helps Classify Problems**
   Using Turing Machines, we can distinguish between:
   - **Decidable** and **Undecidable** problems

- **Computable** and **Non-computable** functions

4. **Universality Concept**
   Led to the concept of a **Universal Turing Machine**, which is the basis for modern general-purpose computers.

30. Describe the Halting Problem of Turing Machine

Definition- The **Halting Problem** is the decision problem of determining whether a given **Turing Machine M** will **halt** (i.e., stop) on a given **input w** or run **forever**.

✅ **Proof Idea (Sketch)**

Assume a machine **H** exists that solves the halting problem.
We can design a paradoxical machine that **uses H** to construct an input that leads to **a contradiction**, similar to the "liar paradox."

This contradiction proves **H cannot exist** → the Halting Problem is **undecidable**.

✅ **Importance**

- Demonstrates **limits of computation**.
- Helps identify **undecidable problems** in computer science.
- Foundation for results in **computability theory** and **complexity theory**.

31. Define Post-Correspondence Problem in Turing Machine

Definition - The **Post Correspondence Problem (PCP)** is a **decision problem** introduced by **Emil Post**.

Given two lists of strings over some alphabet:

A= [a1, a2 ,..., an]

B= [b1, b2,..., bn]

The task is to determine whether there exists a **sequence of indices** i1, i2,..., ik (with repetitions allowed), such that:

$a_{i1}a_{i2}...a_{ik} = b_{i1}b_{i2}...b_{ik}$

✅ **Example**

Let:

- A= [ab, a]
- B= [a, ba]
- Check if any combination of indices gives the same string on both sides:

Try sequence [1,2]:

- A: ab a = aba
- B:a ba = aba ✅ Match found

32. Give an example to examine the concept of Undecidable Problem?

Example to Examine the Concept of an Undecidable Problem:

One of the most well-known examples of an **undecidable problem** is the **Halting Problem**.

The Halting Problem:

- **Problem Statement**: Given a program P and an input I, determine whether P will eventually stop (halt) when run with input I, or if it will run forever.
- **Why It's Undecidable**: In 1936, **Alan Turing** proved that there is **no general algorithm** that can solve the halting problem for all possible program-input pairs. This means we cannot write a program that correctly determines for *every* possible case whether another program halts or not.

Significance:

This problem illustrates the **limits of computation** — some problems simply **cannot be solved** by any algorithm, no matter how powerful or complex.

33. State RICE Theorem and how is it important.

Statement: Rice's Theorem states that any non-trivial property of the language recognized by a Turing machine is undecidable.

- A **property** refers to a characteristic of the language (e.g., "is the language empty?", "does it contain a specific string?", etc.).
- A **non-trivial property** means that the property is true for **some** Turing machines and false for **others**.

Importance of Rice's Theorem:

- It tells us that we **cannot decide any non-trivial semantic property** about programs in general.
- It **generalizes the undecidability** of many problems (like the Halting Problem).
- It is a **powerful tool** for proving undecidability — if you can show a property is non-trivial and relates to the language accepted by a Turing machine, then it's undecidable.

34. Enumerate the ways to convert a multi-tape to a single tape Turing machine

1. **Encoding Multiple Tapes on One tape**:- Represent all tape contents of the multi-tape machine on a single tape by dividing the tape into sections using special symbols (e.g., #) to separate each virtual tape.
2. **Track Head Positions with Markers**:- Use special symbols or markers to indicate the current position of the read/write head on each virtual tape (e.g., using an overline or extra symbol to mark the head position on each tape).
3. **Simulate One Step at a Time**:
   The single-tape machine simulates **one move** of the multi-tape machine by:
   - Scanning the entire tape to find the head positions and read symbols.
   - Determining the next move using the transition function.
   - Scanning again to update symbols and move virtual heads.
4. **Use Extra States to Manage Simulation**:
   Add extra states to the single-tape machine to control scanning, symbol replacement, and head movement, ensuring that the simulation reflects the behavior of the original multi-tape machine.

35. A PDA is more powerful than a finite automaton. Criticize this statement.

1. **Power in Terms of Language Recognition***)*:
    - A **PDA (Pushdown Automaton)** can recognize **context-free languages**, such as balanced parentheses or palindromes.
    - A **finite automaton (FA)** can only recognize **regular languages**, which are simpler and do not require memory of nested structures.
2. **Use of Memory (Stack)**:
    - A PDA has access to a **stack**, providing **limited memory**.
    - A finite automaton has **no memory** apart from its current state.
    - This stack gives PDA the power to handle **recursive patterns**, unlike FA.
3. **Proper Limitation of the Claim**:
    - While a PDA is more powerful in terms of language recognition, it is **not more powerful computationally than a Turing machine**.
    - The statement should not be generalized beyond FA vs PDA comparison.
4. **Conclusion and Contextual Clariy**:
    - The statement is **valid in the context of language classes**, but it must specify that the comparison is within the **Chomsky hierarchy**, between **regular** and **context-free** languages.

36. Formulate a multi head Turing Machine for checking whether a binary string is a palindrome or not.

- Input: Binary string (e.g., w) on the tape.
- Heads: Two heads (H1 and H2) starting at the **leftmost** and **rightmost** positions of the input string, respectively.

Steps:

1. **Initialization:**
    - H1 at the first symbol (left end of the string).
    - H2 at the last symbol (right end of the string).
2. **Compare Symbols:**
    - Compare the symbols under H1 and H2.
    - If they are **not equal**, **reject** (string is not a palindrome).
    - If equal, continue.
3. **Move Heads:**
    - Move H1 one step to the **right**.
    - Move H2 one step to the **left**.
4. **Repeat Steps 2 and 3** until:
    - The heads **cross** (i.e., H1 passes H2), or

- They meet at the same symbol (for odd length strings).

5. **Accept:**
    - If all compared symbols matched, **accept** (string is a palindrome).

37. Justify how turing machine can work as Enumerator

1. **Definition of Enumerator:**
   An **enumerator** is a machine that **outputs all strings** of a language, possibly in some order, by printing them one by one.

2. **Turing Machine with Output Capability:**
   A Turing machine can be modified to **generate and print strings** on a separate output tape or portion of its tape.

3. **Enumerating a Language:**
   The Turing machine systematically **generates all possible strings** over the alphabet (e.g., by length or lex order), and for each string, it **runs a decider or recognizer** to check if it belongs to the language. If yes, it outputs (prints) the string.

4. **Result:**
   This process produces **all and only the strings in the language**, hence the Turing machine acts as an enumerator, demonstrating the equivalence between **Turing-recognizable languages** and **enumerable languages**.

38. Express the use of recursive and recursive enumerable languages.

1. **Recursive Languages (Decidable Languages):**
    - These languages have **algorithms that always halt** and decide membership for any input string.
    - **Use:** They are important in applications where **guaranteed decision-making** is required, such as **compilers**, **syntax checking**, and **program verification**.

2. **Recursively Enumerable (RE) Languages:**
    - These languages have **algorithms that may halt and accept** if the string is in the language but may run forever otherwise.
    - **Use:** Useful in situations like **semi-decision problems**, where recognizing positive instances is possible, e.g., **proof verification**, **search problems**, and **automated theorem proving**.

3. **Practical Implication:**
    - Recursive languages enable **reliable and complete algorithms**, while RE languages allow **partial solutions** where full decision is impossible or unknown.

4. **Theoretical Importance:**- The distinction helps classify problems in **computability theory** and understand the **limits of algorithmic solvability**.

39. Explain NFA with epsilon with an example

1. **Definition:**- An **NFA with epsilon transitions (ε-NFA)** is a nondeterministic finite automaton that allows the machine to move from one state to another **without consuming any input symbol**, via ε-transitions.

2. **How it works:**
    - The machine can jump between states spontaneously using ε-transitions.
    - This increases the flexibility and simplifies the construction of NFAs for certain languages.
3. **Example:** Consider an ε-NFA with states {q0, q1, q2}, alphabet {a, b}, and transitions:
    - From q0 to q1 via ε (no input consumed).
    - From q1 to q2 on input a.
    - q2 is the accepting state.

This ε-NFA accepts any string starting with a because it can move from q0 to q1 freely, then read a to reach q2.

4. **Significance:**
    - ε-NFAs make it easier to build automata for complex languages.
    - They can be converted to equivalent NFAs without ε-transitions and then to DFAs.

40. Explain with an example of an ambiguous CFG.

1. **Definition:**- A CFG is **ambiguous** if there exists at least one string in the language that can have **more than one distinct parse tree** (or derivation).
2. **Example CFG:** Consider the grammar for arithmetic expressions:

$S \rightarrow S + S \mid S * S \mid (S) \mid a$

Here, a represents an operand (like a number).

3. **Ambiguity Example:** The string: a + a * a

can have **two different parse trees**:

- One where + is evaluated first: (a + a) * a
- Another where * is evaluated first: a + (a * a)

4. **Significance:**
   Ambiguity causes **parsing difficulties** and **uncertainty** in interpretation, which is undesirable in programming languages and compilers. Resolving ambiguity often requires rewriting the grammar or using precedence rules.

41. Explain the context-sensitive languages, and how do they differ from context-free languages?

1. **Definition:** Context-sensitive languages are languages that can be generated by **context-sensitive grammars (CSGs)**, where production rules are of the form:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

Here, A is a non-terminal and γ is a non-empty string of terminals and/or non-terminals. The key is that the non-terminal A can be replaced by γ **only in a specific context** (surrounded by α and β).

2. **Characteristics:**

- The length of the string on the left side of the production is **less than or equal to** the length on the right side (no shrinking rules except possibly for the start symbol).
- CSLs can be recognized by a **linear bounded automaton (LBA)**, which is a Turing machine with limited tape.

3. **Difference from Context-Free Languages (CFLs):**

- CFLs have production rules of the form A→ γ, where a single non-terminal is replaced **regardless of context**.
- CSLs are **more powerful** because their rules depend on context; they can describe languages that CFLs cannot, such as {a^n b^n c^n |n≥1}.

42. Generalize are context-sensitive grammars defined, and what distinguishes them from context-free grammars?

1. **General Definition of CSGs** : A **context-sensitive grammar** is a formal grammar where **all production rules** are of the form:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where:

   - A is a non-terminal,
   - αand β are strings of terminals and/or non-terminals (context),
   - γ is a **non-empty** string,
   - and the length of the right-hand side is **greater than or equal** to the left-hand side (i.e., no shrinking rules).

2. **Length Non-Decreasing Property** : Productions in CSGs must be **non-contracting**, meaning the output string is **not shorter** than the input. This ensures computations do not reduce the size of the string arbitrarily.

3. **Distinction from CFGs** : In **context-free grammars**, rules are of the form:

$$A \rightarrow \gamma$$

where A is a single non-terminal and γ is any string of terminals/non-terminals — **no context involved**.

   - In **CSGs**, rules are **context-dependent**, meaning a non-terminal is rewritten **only in a specific surrounding (context)**.

4. **Implication and Example** :

   - **CSGs are more powerful** than CFGs; they can generate languages like {a^n b^n c^n |n≥1}, which CFGs **cannot** generate.
   - Recognized by **linear bounded automata**, while CFGs are recognized by **pushdown automata.**

43. Can you explain the concept of linear bounded automata (LBA) and their connection to context-sensitive languages?

1. **Definition of LBA** :
   A **Linear Bounded Automaton (LBA)** is a **non-deterministic Turing machine** where the tape size is **linearly bounded** by the length of the input. That means it cannot use more tape cells than a constant multiple of the input length.

2. **Structure and Constraints** :

   - Unlike standard Turing machines, an LBA's tape head cannot move beyond the portion of the tape containing the input.

- It uses **bounded memory** which ensures computations stay within a fixed space.

3. **Connection to Context-Sensitive Languages (CSLs)** :
    - **LBAs recognize exactly the set of context-sensitive languages.**
    - This means:
        - If a language is accepted by an LBA, it is context-sensitive.
        - If a language is context-sensitive, there exists an LBA that accepts it.
4. **Significance** :
    - LBAs are important because they represent a **realistic model of computation** with limited memory.
    - They provide a bridge between **practical computation (space-bounded)** and **formal language theory (CSLs)**.

44. Describe some examples of languages that are context-sensitive but not context-free, and explain the process using context-sensitive grammars.

Example Language:

$$L = \{ a^n b^n c^n \mid n \geq 1 \}$$

This language contains strings with **equal numbers of a's, b's, and c's**, in that order.

- **Not Context-Free:**
  This language **cannot** be generated by any context-free grammar, as context-free grammars can't enforce **multiple dependencies** (e.g., matching counts of three different symbols).
- **Context-Sensitive:**
  It **can** be generated by a context-sensitive grammar (CSG), where the productions depend on the surrounding context.

2. Context-Sensitive Grammar for L = { a^n b^n c^n }

Here's a simplified version of a CSG that generates this language:

Non-terminals: S, A, B, C
Terminals: a, b, c
Start symbol: S

Productions:

1. S→ aSBC | abc
2. CB→BC
3. aB → ab
4. bB → bb
5. bC → bc
6. cC → cc

Explanation:

- Rule 1 generates matched triples of a, B, and C, building the structure.

- Rules 2-6 **reorder** and **replace** non-terminals in context to eventually produce a^n b^n c^n.
- Context is used (e.g., aB → ab) to ensure that transformation happens in a valid position.

3. Another CSL Example:

$$L = \{ww \mid w \in \{a, b\}*\}$$

The language of all **duplicated strings** is also **context-sensitive but not context-free** because it requires the machine to **remember and match an arbitrary-length prefix**.

45. Classify the computational power of linear bounded automata compare to other types of automata, such as Turing machines?

1. **LBA vs. Finite Automata (FA) and Pushdown Automata (PDA)**
    - **LBAs are more powerful** than both finite automata and pushdown automata.
    - FA can recognize **regular languages**, PDA can recognize **context-free languages**, but **LBAs can recognize context-sensitive languages**, which are strictly more complex.
2. **LBA vs. Turing Machines (TM)**
    - A **Turing machine is more powerful** than an LBA.
    - TM can use **unbounded memory**, while LBA is restricted to tape space **proportional to input size**.
    - Hence, TM can recognize all **recursively enumerable languages**, not just context-sensitive ones.
3. **Language Classes Recognized**
    - **LBA**: Recognizes **context-sensitive languages (CSLs)**.
    - **TM**: Recognizes **recursively enumerable (RE)** and **recursive languages**.
    - **PDA**: Recognizes **context-free languages (CFLs)**.
    - **FA**: Recognizes **regular languages**.
      (Hierarchy: Regular ⊂ CFL ⊂ CSL ⊂ RE)
4. **Summary of Power Hierarchy**
    - **FA < PDA < LBA < TM**
    - Each model strictly increases in computational power and memory capacity.
    - LBAs are important because they offer a **space-bounded model** of computation.

46. Discover the relationship between context-sensitive grammars and linear bounded automata in terms of language recognition?

1. **Definition of Context-Sensitive Grammars (CSGs)**
    - CSGs are grammars where production rules are **context-dependent** and do **not decrease** the length of strings (except possibly the start symbol rule).
    - They generate **context-sensitive languages (CSLs)**.

2.Definition of Linear Bounded Automata (LBAs)

- LBAs are **restricted Turing machines** that operate within **space linearly bounded** by the size of the input.
- They recognize **exactly** the class of **context-sensitive languages**.

3. **Equivalence in Language Recognition** *(1 mark)*
   - There is a **one-to-one correspondence** between context-sensitive grammars and LBAs:
     - Every language generated by a **CSG** can be **recognized by an LBA**.
     - Every language accepted by an **LBA** can be **generated by a CSG**.
4. **Conclusion**
   - **CSGs and LBAs are equivalent in power** with respect to language recognition.
   - This relationship forms the foundation of the **context-sensitive** level in the **Chomsky hierarchy**, illustrating how grammar-based and machine-based models align.

47. Describe the concept of ε-closure in the context of converting a DFA to an NFA. Provide an example.

1. **Definition of ε-Closure**:
   - The **ε-closure** of a state in an NFA (with ε-transitions) is the **set of states reachable** from that state by **any number of ε-transitions**, including the state itself.
   - It is used to simulate the effect of **ε-moves without consuming input symbols**.
2. **Use in Conversion (NFA to DFA, not DFA to NFA)** *:*
   - ε-closure is primarily used when converting an **ε-NFA to a DFA**, not from DFA to NFA.
   - It helps determine the **new set of reachable states** for each input by considering all ε-transitions first.
3. **Example** *:*
   Consider an ε-NFA with states {q0, q1, q2} and transitions:
   - q0 —ε→ q1
   - q1 —a→ q2

Then:

- **ε-closure(q0) = {q0, q1}**
- **ε-closure(q1) = {q1}**
- **ε-closure(q2) = {q2}**

If we apply input 'a' from state q0:

- We first go to ε-closure(q0) = {q0, q1}
- From q1, 'a' goes to q2 → so new state becomes **ε-closure({q2}) = {q2}**

48. Describe DFA with an example.

A **DFA** is a finite state machine where **each state has exactly one transition** for each input symbol. It accepts or rejects strings based on whether they reach an **accepting (final) state** after processing all input symbols.

Formal Definition:

A DFA is a 5-tuple:

$$(Q, \Sigma, \delta, q0, F)$$

Where:

- Q = finite set of states
- Σ = input alphabet
- Δ = transition function
- q0 = start state
- F = set of accepting states

Example:

DFA to accept strings over {0, 1} that end with **'01'**:

- States: Q={q0,q1,q2}
- Start state: q0
- Accepting state: q2
- Transitions:
    - $q_0 \rightarrow 0 q_1$
    - $q_1 \rightarrow 1 q_2$
    - Other inputs return to earlier states accordingly

String "1101" leads to q2 , so it's **accepted**.

49. Describe NFA with an example.

An **NFA** is a finite state machine where **multiple transitions** for the same input symbol (or even ε-transitions) **are allowed** from a single state. It accepts a string if **at least one path** leads to an accepting state.

Formal Definition:

An NFA is a 5-tuple:

$$(Q, \Sigma, \delta, q0, F)$$

Where:

- Q = set of states
- Σ = input alphabet
- Δ = transition function $Q \times \Sigma \rightarrow 2^Q$

- q0 = start state
- F = set of accepting states

Example:

NFA to accept strings over {0,1} that **end with '1'**:

- States: Q= {q0, q1} Q
- Start state: q0
- Accepting state: q1
- Transitions:
    - q0→0q0
    - q0→1q0,

String "101" has a path ending at q1q_1q1 → **Accepted**

50 . Correlate the Pumping Lemma for Regular Languages and explain its significance in the context of proving that a language is not regular.

Definition of Pumping Lemma - The **Pumping Lemma** states that for any **regular language** L, there exists a pumping length p such that **any string** s ∈L with |s|≥p can be **split into three parts**:

$$s=xyz$$

such that:

- |xy|≤p
- |y|≥1
- For all I ≥ 0, the string xy^iz ∈ L

2. Purpose of the Lemma - The lemma provides a **property all regular languages must satisfy**.
If a language **violates** this property, it **cannot be regular**.

3. Using the Lemma to Prove Non-Regularity *(1 mark)*- To **prove a language is not regular** using the pumping lemma:

- Assume the language is regular.
- Let p be the pumping length.
- Choose a string s∈L with |s|≥p.
- Show that **no matter how you split** s=xyz , pumping y (i.e., repeating it) causes the string to **leave the language**, violating the lemma.

4. Example & Significance - Language L={a^nb^n| n≥0} is **not regular**.
Using the pumping lemma:

- Let s = a^p b^p
- No matter how s is split as xyz , pumping y adds more **a's**, breaking the balance between a's and b's.
- Thus, xy^iz ∉ L for some i, so L is **not regular**.

51. Briefly explain three properties of regular languages

Closure Properties:-
Regular languages are closed under operations like union, intersection, concatenation, and Kleene star. This means that applying these operations to regular languages always produces another regular language.

Finite Automaton Representation:
Every regular language can be represented by a finite automaton (either deterministic or non-deterministic). This means a machine with a finite number of states can recognize any regular language.

Regular Expressions:
Regular languages can be described using regular expressions. If a language can be defined by a regular expression, it is regular, and vice versa.

52. Illustrate the role of ε-closures in the conversion process from an NFA to a DFA. Provide examples to illustrate your explanation.

When converting an **NFA with ε-transitions (ε-NFA)** to a **DFA**, ε-closures are used to ensure that all possible states reachable through ε-transitions are considered without consuming input symbols. This ensures the resulting DFA accurately simulates the behavior of the ε-NFA.

Definition: The **ε-closure** of a state *q* is the set of all states that can be reached from *q* using only ε-transitions (including *q* itself).

Example: Consider an ε-NFA with states:

- **States**: {q0, q1, q2}
- **Alphabet**: {a}
- **Transitions**:
    - q0 —ε→ q1
    - q1 —a→ q2

Step 1: Find ε-closure(q0):

- From q0, via ε, we can go to q1.
- So, **ε-closure(q0) = {q0, q1}**

Step 2: For input 'a':

- From ε-closure(q0) = {q0, q1}, we look at 'a' transitions:
    - q0 has no 'a' transition
    - q1 —a→ q2

Step 3: Find ε-closure of q2:

- q2 has no ε-transitions, so **ε-closure(q2) = {q2}**

So, in the DFA:

- From state {q0, q1}, on input 'a', go to state {q2}

53. Explain Reduced Form of a CFG and how to obtain them?

Reduced Form of a CFG (Context-Free Grammar) :- The **Reduced Form** of a CFG is a simplified version of the grammar where:

1. **All useless symbols are removed**.
2. **Only the necessary (productive and reachable) variables and rules remain**.

This makes the grammar more efficient and easier to analyze or use in parsers.

Steps to Obtain the Reduced Form:

1. **Remove Non-Productive Symbols**
    - A symbol is **productive** if it can derive a string of terminal symbols.
    - Eliminate variables and rules that do not lead to terminal strings.
2. **Remove Unreachable Symbols**
    - A symbol is **reachable** if it can be reached from the start symbol.
    - Eliminate variables and rules that are not reachable from the start symbol.

Example:

Given CFG:
S → AB | a
A → aA | ε
B → bB
C → c

- **C** is **unreachable** (not used in any derivation from S) → remove it.
- **B → bB** never leads to a terminal string (no base case) → **B is non-productive** → remove it.

Reduced CFG:
S → a
A → aA | ε

54. Define Instantaneous Description in Turing Machine.

Instantaneous Description (ID) in Turing Machine – 4 Marks

An **Instantaneous Description (ID)** of a Turing Machine is a formal representation of the machine's current configuration at any step of computation. It provides a snapshot of the machine's:

1. **Current state**
2. **Tape contents**
3. **Position of the tape head**

Format: An ID is usually written as:
u q v
Where:

- **u**: string to the left of the tape head
- **q**: current state of the machine

- **v**: string starting from the symbol under the tape head and to its right

This indicates the tape contains **uv**, the machine is in state **q**, and the head is scanning the **first symbol of v**.

Example:

If the tape has a b c, the head is on b, and the machine is in state q1, the ID is:
a q1 bc

55. Define Halt state in Turing Machine:

A **halt state** in a Turing Machine is a special state where the machine **stops computation**. Once it enters a halt state, **no further moves are made**, and the machine terminates its operation.

Types of Halt States:

1. **Accepting (Final) Halt State**:
    - Indicates the input string is **accepted**.
    - Often denoted as **q_accept**.
2. **Rejecting Halt State** (optional):
    - Indicates the input is **rejected**.
    - Often denoted as **q_reject**.

Some Turing Machines may only have one halt state (accepting), and reject by looping forever.

Characteristics:

- No transitions are defined **from** a halt state.
- The machine **halts** when it reaches this state during computation.

Example: If a Turing Machine reaches **q_accept** after processing the input, the machine halts, and the input is **accepted**.

56. Describe the concept of a Turing Machine's tape head movement and its significance in computation.

The **tape head** in a Turing Machine is the component that **reads and writes** symbols on the tape and **moves left or right** after each operation. Its movement is directed by the machine's **transition function** based on the current state and the symbol being read.

Types of Movements:

- **Left (L)**: The head moves one cell to the left.
- **Right (R)**: The head moves one cell to the right.
- **Stay (optional, S)**: In some models, the head may stay in place (not standard in basic Turing Machines).

Significance in Computation:

1. **Sequential Access**: The head moves along the tape to **access and modify symbols**, allowing step-by-step computation.
2. **Control Flow**: Movement decisions help control the **logic and flow** of algorithms (e.g., scanning for a pattern or matching symbols).

3. **Simulating Memory**: The tape with the movable head acts like **infinite memory**, enabling complex tasks like looping, recursion, or string manipulation.

Example: If the head reads 1, the machine might write 0, move to the right, and enter a new state. This simple step enables broader computation over time.

57. Discuss the concept of non-deterministic Turing Machines (NTMs) and explain how they differ from deterministic Turing Machines (DTMs).

A Non-deterministic Turing Machine (NTM) is a theoretical model of computation where, for a given state and tape symbol, the machine can choose from multiple possible transitions. It can be thought of as "trying all possibilities at once" through parallel computational paths.

Key Features of NTMs:

- At any step, the machine can branch into multiple copies, each following a different transition.
- The input is accepted if at least one branch leads to an accepting (halt) state.

Difference Between NTM and DTM:

Feature	Deterministic TM (DTM)	Non-deterministic TM (NTM)
Transition Function	Exactly one action for each state-symbol pair	Multiple possible actions per pair
Computation Path	Only one path per input	Many possible paths per input
Acceptance	Accepts if the single path reaches halt	Accepts if any path reaches halt
Implementation	Practical (models real computers)	Theoretical (used in complexity theory)

58. Explain the significance of the halting problem in the theory of computation, and discuss its implications for Turing Machines.

1. **Undecidability:**
   Alan Turing proved that the Halting Problem is **undecidable**—there is **no general algorithm** (or Turing Machine) that can correctly determine, for all possible inputs and machines, whether the machine will halt.
2. **Limits of Computation:**
   It establishes that **not all problems are solvable by algorithms**, setting boundaries on what computers can and cannot do.

Implications for Turing Machines:

- **Existence of Undecidable Problems:**
  It shows that some languages cannot be recognized by any Turing Machine, no matter how powerful.
- **No Universal Debugger:**
  You cannot write a program that checks every other program for infinite loops in all cases.
- **Impact on Software and AI:**
  It informs us that perfect prediction or verification of program behavior is **impossible** in general.

59 . Define the concept of Turing Machine decidability and provide an example of a decidable language.

A language is said to be **decidable** if there exists a **Turing Machine** that will:

- **Always halt** (either accept or reject)
- For **every input string**

This means the machine **decides** whether a string belongs to the language or not, without running forever.

Definition: A language **L** is **decidable** if there is a Turing Machine **M** such that:

- For every string **w**:
    - If **w ∈ L**, then **M accepts w**
    - If **w ∉ L**, then **M rejects w**

Example of a Decidable Language:

Language:
L = { w ∈ {a, b}* | w has an equal number of a's and b's }

Explanation: A Turing Machine can be constructed to **count** the number of a's and b's and **compare** them. Since this can be done in a finite number of steps and always halts, the language is **decidable**.

60 . Discuss the Church-Turing thesis and its significance in the theory of computation.

1. **Equivalence of Models:** It proposes that all reasonable models of computation (like lambda calculus, recursive functions, and Turing Machines) have the same computational power.
2. **Foundation of Computability:** It provides a formal basis for what it means for a function or problem to be computable.
3. **Limits of Computation:** The thesis implies that if a problem cannot be solved by a Turing Machine, it cannot be solved by **any** algorithmic means.

Note: The Church-Turing thesis is **not a proven theorem** but an accepted principle based on extensive evidence and consensus.

Conclusion: The Church-Turing thesis shapes the entire field of theoretical computer science by defining the boundary of what machines can compute and guiding the development of algorithms and computation theory

61. Predict how left and right recursion can be obtained from a parse tree with example

Recursion in grammar rules can be identified by analyzing the parse tree structure of a language construct.

Left Recursion:

- Occurs when a non-terminal **calls itself as the leftmost child** in its production.
- The parse tree shows the same non-terminal appearing at the **left edge** of its subtree.

Example:
Grammar rule:
$A \rightarrow A\ \alpha \mid \beta$

Parse tree for input derived from A looks like:

A

/ \

A α

Here, A is the left child of A → **left recursion**.

Right Recursion:

- Occurs when a non-terminal **calls itself as the rightmost child**.
- The parse tree shows the same non-terminal appearing at the **right edge** of its subtree.

Example:
Grammar rule:
A → β | α A

Parse tree for input derived from A looks like:

A

/ \

α A

Here, A is the right child of A → **right recursion**.

62. Prove that the union of two regular languages is always regular.

Let $\mathbf{L_1}$ and $\mathbf{L_2}$ be two regular languages.

1. **Definition (1 mark):** Since L_1 and L_2 are regular, there exist finite automata $\mathbf{M_1}$ and $\mathbf{M_2}$ that recognize L_1 and L_2, respectively.
2. **Construction (1 mark):** Construct a new finite automaton **M** that recognizes the language $\mathbf{L = L_1 \cup L_2}$. This can be done using the **product construction** (for DFAs) or a **new start state with ε-transitions** (for NFAs).
3. **Closure Property (1 mark):** The class of regular languages is **closed under union**, meaning the union of two regular languages is also regular.
4. **Conclusion (1 mark):** Therefore, $\mathbf{L = L_1 \cup L_2}$ is regular.

63. Analyze and classify Chomsky's Hierarchy of formal languages.

1. **Type 0 – Recursively Enumerable Languages**
    - **Generated by**: Unrestricted grammars
    - **Recognized by**: Turing Machines
    - **Most powerful** class; includes all computable languages
    - **No restrictions** on grammar rules
2. **Type 1 – Context-Sensitive Languages**
    - **Generated by**: Context-sensitive grammars
    - **Recognized by**: Linear Bounded Automata

- **Grammar rules**: $\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$
- **More powerful** than context-free but less than Type 0

3. **Type 2 – Context-Free Languages**
   - **Generated by**: Context-free grammars
   - **Recognized by**: Pushdown Automata
   - **Grammar rules**: $A \rightarrow \gamma$, where A is a non-terminal and γ is a string of terminals and/or non-terminals
   - Common in **programming languages and parsers**
4. **Type 3 – Regular Languages**
   - **Generated by**: Regular grammars
   - **Recognized by**: Finite Automata (DFA/NFA)
   - **Least powerful**, but simplest and fastest to process
   - Used in **lexical analysis, pattern matching**

64. Determine distinguishable, undistinguishable states, dead and unreachable state for any given DFA.

Distinguishable States (1 mark):

Two states $\mathbf{q_1}$ and $\mathbf{q_2}$ are **distinguishable** if there exists **at least one input string** that leads to:

- acceptance from one state, and
- rejection from the other.

✅ **Method:** Use the **state minimization table-filling algorithm** to find such state pairs.

2. Indistinguishable States (1 mark):

States are **indistinguishable** if **no input string** can differentiate them — i.e., they behave identically for all inputs.

✅ These states can be **merged** during DFA minimization.

3. Dead State (1 mark):

A **dead state** (also called a **trap state**) is a state from which:

- **no accepting state can be reached** on any input string.

✅ Once entered, the DFA remains in the dead state and **never accepts**.

4. Unreachable State (1 mark):

An **unreachable state** is:

- **not reachable** from the **initial/start state** on any sequence of inputs.

✅ These states are **redundant** and can be **removed** from the DFA.

65. With example explain the extended transition function for a DFA

Definition : For a DFA M= (Q, Σ, δ, q0, F) the extended transition function **δ*(q, w)** maps a state **q** and a string **w ∈ Σ*** to the **state reached** after processing the **entire string w**.

It is defined **recursively**:

- **Base case:** $\delta^*(q, \varepsilon) = q$
- **Recursive step:** $\delta^*(q, xa) = \delta(\delta^*(q, x), a)$, where $x \in \Sigma^*$, $a \in \Sigma$

2. Example DFA (1 mark):

Let DFA M= (Q, Σ, δ, q0, F) be defined as:

- $Q = \{q_o, q_1\}$, $\Sigma = \{0, 1\}$, q_o = start state, $F = \{q_1\}$
- δ is defined as:
    - $\delta(q_o, 0) = q_o$
    - $\delta(q_o, 1) = q_1$
    - $\delta(q_1, 0) = q_1$
    - $\delta(q_1, 1) = q_1$

This DFA **accepts strings that contain at least one '1'**.

3. Apply Extended Transition :

Let's evaluate **$\delta^*(q_o, 0010)$**:

- $\delta^*(q_o, \varepsilon) = q_o$
- $\delta^*(q_o, 0) = \delta(q_o, 0) = q_o$
- $\delta^*(q_o, 00) = \delta(q_o, 0) = q_o$
- $\delta^*(q_o, 001) = \delta(q_o, 1) = q_1$
- $\delta^*(q_o, 0010) = \delta(q_1, 0) = q_1$

➡️ So, $\delta^*(q_o, 0010) = q_1$ (which is accepting)

66. Define epsilon closure of a state with example and describe its significance in Automata.

Definition: The **epsilon closure** (ε-closure) of a state **q** in a **Nondeterministic Finite Automaton with ε-transitions (ε-NFA)** is the **set of all states** that can be reached **from q by taking zero or more ε-transitions** (i.e., transitions without consuming any input symbol).

2. Notation and Meaning (1 mark):

- Denoted as **ε-closure(q)**
- Always includes the state **q itself**
- Helps in identifying **all possible current states** after free (ε) moves

3. Example: Consider an ε-NFA with states {q0, q1, q2}, where:

- ε-transitions:
    - q0 → q1

- q1 → q2

- Then:
  - ε-closure(q0) = {q0, q1, q2}
  - ε-closure(q1) = {q1, q2}
  - ε-closure(q2) = {q2}

4. Significance :

- **Critical for converting ε-NFA to DFA** using **subset construction**
- Helps track all possible states the automaton could be in without consuming input

67. Distinguish between three types of Finite State Machines.

Type	Description	Output Depends On	Example Use Case
DFA (Deterministic Finite Automaton)	**Each state has exactly one transition for each input symbol.**	**No output (accept/reject only)**	**Token recognition in compilers**
NFA (Nondeterministic Finite Automaton)	**States may have multiple or no transitions for an input symbol; may include ε-moves.**	**No output (accept/reject only)**	**Pattern matching, regex engines**
Mealy Machine	**FSM with outputs on transitions.**	**Output depends on state & input**	**Serial data encoding**
Moore Machine	**FSM with outputs on states.**	**Output depends on current state only**	**Traffic light control systems**

68. Discuss different properties of Regular Sets with proper examples.

Closure Properties :- Regular sets are closed under several operations. That means performing these operations on regular sets produces another regular set.

✅ **Examples:**

- Union: If $L_1 = \{a, b\}$, $L_2 = \{c\}$, then $L_1 \cup L_2 = \{a, b, c\}$ is regular
- Concatenation: If $L_1 = \{a\}$, $L_2 = \{b\}$, then $L_1 \cdot L_2 = \{ab\}$ is regular
- Kleene star: If $L = \{a\}$, then $L^* = \{\varepsilon, a, aa, aaa, ...\}$ is regular
- Intersection, Complement, Difference — all preserve regularity

2. **Representability by Finite Automata :-**

Every regular set can be accepted by a Deterministic or Nondeterministic Finite Automaton (DFA or NFA).

✅ Example:

- Regular expression a* is accepted by a DFA with one state looping on 'a'.

3. Regular Sets are Decidable :- There are algorithms to determine whether a string belongs to a regular set or not — hence, membership testing is decidable.

✅ **Example**: Given a DFA for L = strings ending with "01", we can always decide whether "1001" ∈ L (Yes).

4. Limitations of Regular Sets :- Regular sets cannot describe languages that require memory, such as matching parentheses or palindromes.

✅ **Example:** The language L = {$a^n b^n$ | n ≥ 0} is not regular, because it requires counting — which finite automata cannot do.

69. Explain the formal description of Linear Bounded Automata. Also state the closure properties of the language supported by it.

Formal Description of Linear Bounded Automaton (LBA) - A **Linear Bounded Automaton (LBA)** is a **restricted Turing Machine** where the tape size is **linearly bounded** by the length of the input. It recognizes **context-sensitive languages**.

Definition:

An LBA is a 7-tuple:

$$M=(Q, \Sigma, \Gamma, \delta, q0, q_{accept}, q_{reject})$$

Where:

- **Q**: Finite set of states
- **Σ**: Input alphabet (does not include the blank symbol)
- **Γ**: Tape alphabet (includes Σ and special symbols, including a blank symbol ␣)
- **δ**: Transition function

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

- q_0: Start state
- q_a: Accept state
- q_r: Reject state

Key Restrictions:

- The tape is **bounded** — LBA **cannot move beyond** the portion of the tape containing the input.
- It cannot write or access beyond this **input length boundary**.

✅ **Example Language Recognized by LBA**:

L = { a^n b^n c^n n ≥ 1 }

This is **not context-free**, but it **is context-sensitive** — hence accepted by an LBA.

2. Closure Properties of Context-Sensitive Languages - The languages accepted by LBAs (context-sensitive languages) are **closed under** the following operations:

Operation	Closure Status
Union	✅ Closed

Operation	Closure Status
Intersection	✅ Closed
Concatenation	✅ Closed
Kleene Star	❌ *Not closed*
Complement	✅ Closed (proved by Immerman–Szelepcsényi theorem)
Reversal	✅ Closed
Substitution	✅ Closed

70. Define the moves made by the Turing Machine with an example

A **Turing Machine (TM)** moves by reading a symbol from the tape, **changing state**, **writing a new symbol**, and then **moving the tape head** either **Left (L)** or **Right (R)**.

Each move is defined by the **transition function**:

$$\Delta (q, a) = (q', b, D)$$

Where:

- q: current state
- a: symbol read
- q': next state
- b: symbol to write
- D: direction to move (L or R)

2. Example Transition (1 mark)

Let:

$$\Delta (q0, 1) = (q1, X, R)$$

This means:

- If the TM is in state $\mathbf{q_0}$ and reads **1**, it:
    - **writes** X in place of 1
    - **transitions** to state q_1
    - **moves right (R)** on the tape

3. Full Example (1 mark)

Suppose the tape contains:

... ␣ ␣ 1 1 0 ␣ ␣ ...

↑

(Head at 1)

Initial state: q_0
Transition:

$$\Delta\ (q0, 1)=(q1, X, R)$$

After the move:

- **1 is replaced by X**
- **Head moves right to next 1**
- **TM enters state q_1**

New tape:

... ␣ ␣ X 1 0 ␣ ␣ ...

↑

(Head here)

71. Recognize operations that are possible on the tape of a turing machine.

The machine can **read the symbol** currently under the tape head.

✅ **Example:** If the head is on symbol '1', it reads '1' to decide the next move.

2. Write Operation - The machine can **write a new symbol** (or overwrite an existing one) on the tape at the current position.

✅ **Example:** It can change a '1' to 'X' to mark that the symbol has been processed.

3. Move Left or Right - The tape head can **move one cell to the left (L)** or **right (R)** after each operation.

✅ This allows scanning the tape in both directions, unlike simpler machines.

4. No Move / Stay (Optional, in some models) - In extended Turing Machines, the head may also be allowed to **stay in place (S)** without moving.

✅ Useful for performing multiple operations at the same cell.

72. Compare between the different representations of Finite Automata to recognize a string.

Representation	Description	Advantages	Disadvantages
Deterministic Finite Automaton (DFA)	Each state has exactly one transition for every input symbol.	Simple to implement and simulate; fast recognition (one path).	Can have many states for some languages; no ε-moves.
Nondeterministic Finite Automaton (NFA)	States can have multiple transitions for the same input or ε-transitions.	Often smaller and easier to construct than DFA; more expressive in design.	Requires backtracking or simulation of multiple paths; slower directly.
NFA with ε-transitions (ε-NFA)	NFA extended with transitions on ε (no input consumption).	Simplifies construction of automata from regular expressions; easier modular design.	Requires ε-closure computation; less straightforward simulation.

73. Assess the utility of different parts of a machine while processing any input.

1. **Input Tape / Input String :-**

- Holds the input to be processed.
- Provides the symbols one-by-one for the machine to read and process.

2. **Control Unit / Finite Set of States :-**

- Keeps track of the machine's current status (state).
- Determines the next action based on current state and input symbol.

3. **Transition Function :-**

- Defines how the machine moves between states or modifies the tape.
- Maps current state and input symbol to the next state (and tape operations if applicable).

4. **Output Mechanism / Accept States :**

- Determines if the input is accepted or rejected.
- Final states indicate successful recognition or computation completion.

74. Analyze the steps necessary to minimize the CFG.

1. **Remove Useless Symbols -**

- **Eliminate non-generating symbols:** Remove variables that do not derive any terminal string.
- **Eliminate unreachable symbols:** Remove variables that cannot be reached from the start symbol.

2. **Eliminate ε-Productions -** Remove productions of the form A→ϵA (except possibly for the start symbol) by rewriting rules to account for nullable variables.

3. **Eliminate Unit Productions -** Remove productions where a variable produces another single variable, e.g., A→B , by substituting B's productions directly.

4. **Simplify Productions -** Replace long or complex right-hand sides with shorter ones (e.g., convert to Chomsky Normal Form) to reduce the grammar size and complexity.

75. Criticize the Decision problems for CFLs with example?

1. **Emptiness Problem (Decidable) (1 mark)**

- **Question:** Is the language generated by a CFL empty?
- **Status:** Decidable; algorithms exist to check if the start symbol can generate any terminal string.
- **Example:** For CFG G with no derivation to terminals, the answer is "Yes, empty."

2. **Membership Problem (Decidable) (1 mark)**

- **Question:** Does a string www belong to a CFL L?
- **Status:** Decidable via parsing algorithms like CYK or Earley's algorithm in polynomial time.
- **Example:** Check if "abba" ∈ L={a^nb^n}L → Yes, accepted.

3. **Equivalence Problem (Undecidable)**

- **Question:** Are two CFLs equal?
- **Status:** Undecidable; no algorithm can always determine if L(G1)=L(G2)
- **Example:** Given two CFGs generating complex languages, it's impossible to decide equivalence generally.
- **Inclusion Problem (Undecidable)**
- **Question:** Is L(G1) ⊆ L(G2) ?
- **Status:** Undecidable; no general method to check if one CFL is a subset of another.
- **Example:** Checking if L1 ⊆ L2 for arbitrary CFGs is not computable.

76. Create a Turing machine which accepts even palindrome

The TM accepts strings of the form: ww^R

where www is any string over {0,1} and w^R is the reverse of www, with **even length**.

2. High-Level Idea:

- **Step 1:** Check if input length is even; if odd, reject immediately.
- **Step 2:** Repeatedly compare the **leftmost** and **rightmost** symbols:
    - Move right to find the last non-blank symbol.
    - Compare it with the first symbol.
    - If they match, replace both with a special marker (say 'X') and continue inward.
    - If mismatch, reject.
- **Step 3:** If all pairs match and the tape is all marked (or empty), accept.

3. Components of the TM: States:

- q0 : start
- q_{check} : move right to find the rightmost symbol
- $q_{compare0}$, $q_{compare1}$: compare symbols
- q_{move_left}: return to the leftmost unmarked symbol
- q_{accept} , q_{reject}

- **Tape Alphabet:** {0, 1, X, ⊔} where XXX marks matched symbols and ⊔ is blank.

4. Example Run (String: 0110):

- Compare leftmost '0' with rightmost '0' → match → mark both as 'X'.
- Move inward, compare next left '1' with next right '1' → match → mark both.
- Tape becomes: XXXX → all matched → accept.

77. Compare and contrast the advantages and disadvantages of Mealy and Moore machines in various applications.

Feature	Mealy Machine	Moore Machine
Output Depends On	Current **state and input**	Only on the **current state**
Output Timing	**Immediate** response to input changes	Output changes **after state transition**
Design Complexity	Typically **fewer states** required	Might require **more states** for same behavior
Reaction Speed	**Faster**, as output can change in the middle of a state	**Slower**, must wait for a full transition
Simplicity & Predictability	Slightly more complex due to input-driven output	**Simpler** and easier to debug

Applications:

- **Mealy Machine:** Useful in **real-time systems**, where quick reaction is required (e.g., protocol converters, serial data processors).
- **Moore Machine:**- Preferred in **hardware circuits** and synchronous designs where stable output is needed (e.g., counters, traffic lights).

78. Define Mealy Machine and it's output characteristics.

Definition of Mealy Machine :- A **Mealy Machine** is a type of **Finite State Machine (FSM)** where the **output is determined by both the current state and the current input**.

Formal Definition: - A Mealy Machine is a 6-tuple:

$$M = (Q, \Sigma, \Delta, \delta, \lambda, q0)$$

Where:

- Q: Finite set of states
- Σ: Input alphabet
- Δ: Output alphabet
- δ: $Q \times \Sigma \rightarrow Q$: State transition function
- λ: $Q \times \Sigma \rightarrow \Delta$: Output function
- $q0 \in Q$: Initial state

2. Output Characteristics (2 marks)

- **Input-Dependent Output:**
  Output is generated **on transitions**, based on both **state and input**.
- **Faster Response:**
  Since the output is produced **immediately when input is received**, Mealy machines often react **faster** than Moore machines.
- **Compact Design:**
  They generally require **fewer states** than equivalent Moore machines for the same behavior.

Example: If λ(q1,0)=1, then in state q1, receiving input 0 produces output 1.

79. Design a Turing machine to accept a Palindrome

A **palindrome** is a string that reads the same forward and backward.
We design a Turing Machine (TM) to accept strings over {0, 1} that are **palindromes of any length** (including odd and even).

2. High-Level Approach - The Turing Machine works as follows:

1. **Compare first and last symbols**:
    - Replace the **leftmost symbol** with a marker (e.g., 'X').
    - Move right to the **rightmost unmarked symbol**, compare it.
    - If matched, mark it (e.g., with 'X') and return to the start.
    - If mismatch, **reject**.
2. **Repeat** until all symbols are marked or a single center symbol remains (for odd-length strings).
3. If successfully marked all, **accept**.

3. Main Transitions (Example, 1 mark)

- Δ (q0, 0) = (q1, X, R) → mark leftmost 0 and go right
- Δ (q1 ,0) = (q1, 0, R) ... until last unmarked symbol
- Δ (q2,0) = (q3, X, L) → matched last 0, mark and return
- Repeat for symbol '1' similarly
- If no mismatch and tape is empty or only Xs → δ ($q$final, _)=accept

4. Example Run - Input: 0110

- Compare '0' with '0' → match, mark both
- Compare '1' with '1' → match, mark both
- All matched → **ACCEPT**

80. Estimate the minimized DFA.

1.Remove Unreachable States (1 mark)

- Start from the initial state.
- Traverse using all possible input symbols.
- **Remove any state** that is **not reachable** from the initial state.

2. Identify and Merge Equivalent States (1 mark)

- Use **partitioning (table-filling method)**:
    - Initially, mark all distinguishable state pairs (final vs. non-final).
    - Iteratively mark more pairs based on transition behavior.
    - **Unmarked pairs** are **equivalent** and can be **merged**.

3. Build Minimized DFA (1 mark)

- Replace each group of equivalent states with a **single state**.
- Update transitions accordingly.
- Resulting DFA has **fewer states** but recognizes the **same language**.

Example: Original DFA has 6 states.
After removing unreachable states and merging equivalent ones:
Minimized DFA has 3 states (estimated).

81. Write the difference between Turing-Recognizable and Turing-Decidable Languages

Feature	Turing-Recognizable Language (Recursively Enumerable)	Turing-Decidable Language (Recursive)
Definition	**A language for which a Turing Machine halts on accepted inputs, but may loop forever on non-members.**	**A language for which a Turing Machine halts on all inputs (accept or reject).**
Halting Requirement	**May not halt on some inputs (non-members).**	**Always halts for every input.**
Acceptance Behavior	**Accepts valid strings, but might run forever on invalid ones.**	**Accepts or rejects every string in finite time.**
Closure Properties	**Closed under union, intersection. Not closed under complement.**	**Closed under union, intersection, and complement.**
Example Language	**The Halting Problem (TM accepts input) – Recognizable, not decidable.**	**Set of valid arithmetic expressions – Decidable.**