2016-17(4th batch)

1. a) Analyze the terms: Theorems, Lemmas and Corollaries. 4
   b) What is Automata Theory? Describe the applications of Finite Automata. 4
   c) A containment hierarchy of classes of formal languages has been classified as The Chomsky Hierarchy. Describe with suitable figure(s). 4

2. a) Define push down automata with example. 3
   b) Give pushdown automata that recognize the following languages: 5
      (a) $A = \{ w \in \{0, 1\}^* \mid w \text{ contains at least three 1s} \}$
      (b) $B = \{ w \in \{0, 1\}^* \mid w = w^R \text{ and the length of w is odd} \}$
   c) Use the pumping lemma to prove that the language $A = \{ 0^{2n} 1^{3n} 0^n \mid n \geq 0 \}$ is not context free. 4

3. a) What is Turing Machine? Give advantages of it. 3
   b) Construct a Turing Machine that accepts the language of palindromes over {a,b} . Also specify the moves to trace the strings abaa,abba,aabaa. 6
   c) What is meant by a halting problem in a Turing Machine? Explain with an example. 3

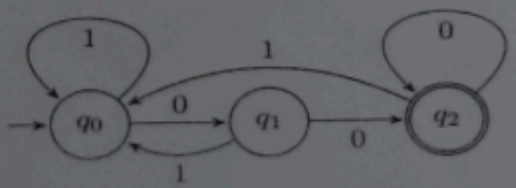
4. a) Let, $\Sigma = \{0, 1\}$ 6
      Construct a DFA for the following language:
      L = {w | w is a binary string that has even number of 1s and even number of 0s}
   b) Construct a NFA for the following: Strings where the first symbol is present somewhere later on at least once. 6

5. a) Build an NFA for the following language: L = { w | w ends in 01} 4
   b) Describe the Advantages & Caveats for NFA. 3
   c) Convert the NFA from Question 5(a), to DFA. 5

6. a) Why explicit ε-transitions in finite automata is important? 2
   b) Build an ε-NFA for the following language: 4
      L = {w | w is empty, or if non-empty will end in 01}
   c) Convert ε-NFA to DFA based on Question 6(b) 6

7. a) Describe the relation between Regular Expressions (RE) and Finite Automata. Show with figure that they are interchangeable. 6
   b) Convert the following RE to ε-NFA: (0+1)*01(0+1)* 6

8. a) Construct a context-free grammar for the following DFA: 6



   b) Show that the grammar ({S}, {a, b}, R, S) with rules R = S → aS | aSbS | ϵ is ambiguous. 4
   c) Does a push down automata have memory? Justify. 2

a) Analyze the terms: **Theorems, Lemmas and Corollaries.** (4)

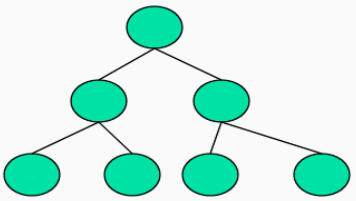
Well, they are basically just **facts**: some result that has been arrived at.

- A Theorem is a **major** result
- A Corollary is a theorem that **follows on** from another theorem
- A Lemma is a **small** results (less important than a theorem)

We typically refer to:

- A major result as a "**theorem**"
- An intermediate result that we show to prove a larger result as a "**lemma**"
- A result that follows from an already proven result as a "**corollary**"

An example:
Theorem: *The height of an n-node binary tree is at least floor(lg n)*
Lemma: *Level i of a perfect binary tree has 2^i nodes.*
Corollary: *A perfect binary tree of height h has 2^{h+1}-1 nodes.*



Here's a clear analysis of the terms **Theorems**, **Lemmas**, and **Corollaries**:

---

1. Theorem

- A **theorem** is a significant, proven statement in mathematics.
- It is usually an important result that forms the backbone of a theory or provides insight into a major concept.
- Theorems are typically proven using axioms, definitions, and previously proven theorems or lemmas.

Example:
The Pythagorean Theorem states that in a right-angled triangle:
[a^2 + b^2 = c^2]

---

2. Lemma

- A **lemma** is a “helping theorem” — a result that is proved primarily to assist in proving a larger or more significant theorem.

- Lemmas are not usually of independent importance but are crucial for breaking down complex proofs.

Example:
A technical lemma might be used to establish properties of functions that are later used in proving the Fundamental Theorem of Calculus.

---

3. Corollary

- A **corollary** is a statement that follows directly from a theorem with little or no additional proof.

- It is an immediate consequence of a theorem, often noted for clarity or completeness.

Example:
From the theorem that “the sum of the angles in a triangle is 180°,” a corollary is that “each angle of an equilateral triangle is 60°.”

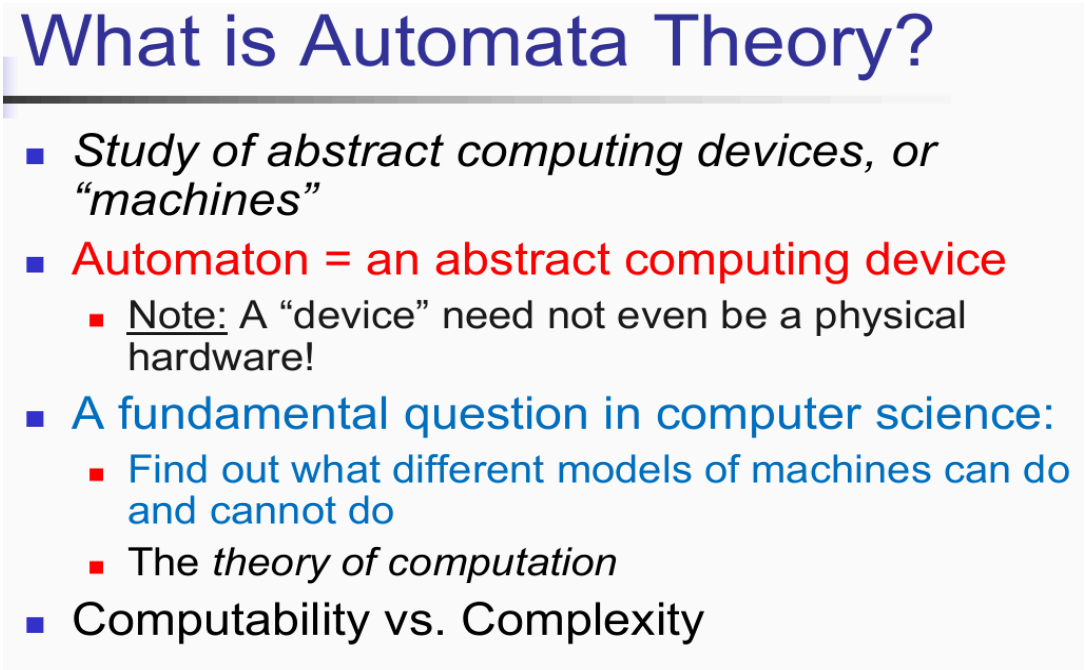
Summary Table:

Term	Purpose	Importance
Theorem	Major, central result	High
Lemma	Tool to assist in proving a theorem	Medium (supportive role)
Corollary	Immediate consequence of a theorem	Low to Medium

---

b) What is **Automata Theory**? Describe the applications of Finite Automata. (4)

Automata Theory is a branch of theoretical computer science that deals with the study of abstract machines or mathematical models of computation (called automata) and the computational problems that can be solved using these machines. It provides the foundation for understanding how languages are defined and recognized by computational systems.



Central Concepts of Automata Theory:

1. Automata

Automata are abstract, mathematical models of computation that perform operations on input strings. Types of automata include:

- **Finite Automata (FA)** – Used for recognizing regular languages.
- **Pushdown Automata (PDA)** – Used for recognizing context-free languages.
- **Turing Machines (TM)** – A model of general-purpose computation.

2. Alphabets (Σ) and Strings

- **Alphabet (Σ):** A finite set of symbols.
  Example: Σ = {0, 1}

- **String:** A sequence of symbols from the alphabet.
  Example: "0101" is a string over the alphabet {0, 1}.

---

3. Languages

A **language** is a set of strings formed using the alphabet. Automata classify these languages based on the type of grammar and machine used to recognize them:

- **Regular Languages** – Recognized by finite automata.

- **Context-Free Languages** – Recognized by pushdown automata.

- **Recursively Enumerable Languages** – Recognized by Turing machines.

---

4. Grammar

A **grammar** defines the structure of a language using production rules. According to the Chomsky hierarchy, grammars are classified into:

- Type-3: Regular Grammar

- Type-2: Context-Free Grammar

- Type-1: Context-Sensitive Grammar

- Type-0: Unrestricted Grammar

---

5. Transition Functions

Defines how the automaton moves from one state to another based on the input symbol. For example:

- In **Deterministic Finite Automata (DFA)**, a state and input symbol lead to exactly one state.

- In **Non-deterministic Finite Automata (NFA)**, they may lead to multiple possible states.

---

6. Determinism vs Non-determinism

- **Deterministic Automata**: One transition per input.
- **Non-deterministic Automata**: Multiple possible transitions.
- Both DFA and NFA recognize **regular languages** and are equivalent in terms of computational power.

---

7. Acceptance of a Language

An automaton accepts a string if, after processing all input symbols, it reaches a valid accepting (or final) state.

Applications of Finite Automata (One-line each):

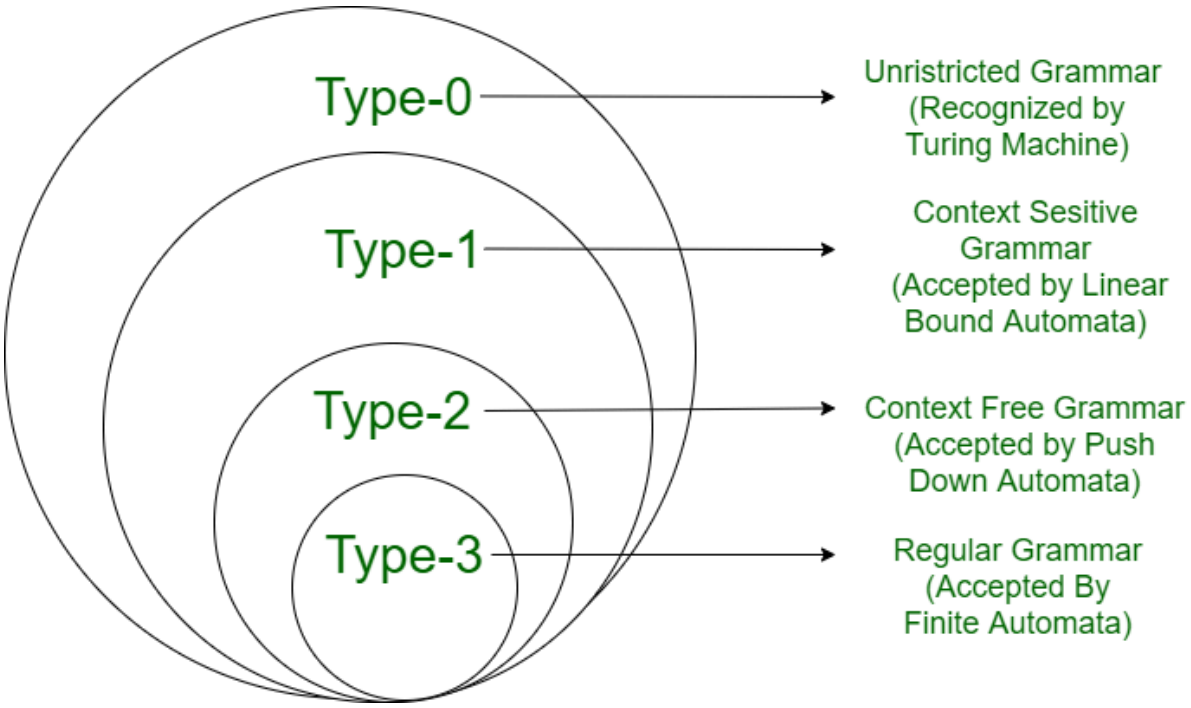
1. **Lexical Analysis in Compilers:** Used to identify tokens in source code like keywords and operators.
2. **String Processing:** Helps find and replace patterns in text.FA is used to search for patterns in text, such as identifying keywords in a search engine or searching for substrings in text editors.
3. **Spell Checkers:** Checks if typed words match valid dictionary patterns.
4. **Artificial Intelligence (Simple AI):** Models simple decision-based systems or behavior.
5. **System Verification:** Tests if a system performs correctly in all its states.
6. **Compiler Design:** Used in various phases of compiler construction.
7. **Digital Circuit Design:** Designs sequential logic circuits like Mealy and Moore machines.

8. **Simple Device Control:** Controls basic machines like elevators or washing machines with defined states.

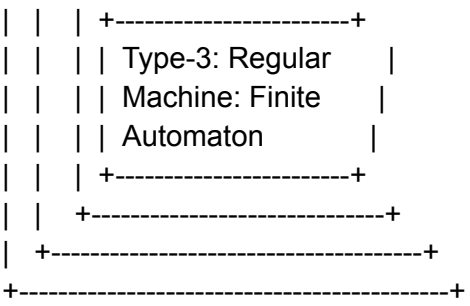
9. **Natural Language Processing:** Recognizes patterns in text or speech for language tools.

---

c) A **containment hierarchy** of classes of formal languages has been classified as The **Chomsky Hierarchy**. Describe with suitable figure(s). (4)




```
+-----------------------------------------+
|          Type-0: Unrestricted           |
|  (Recursively Enumerable Languages)     |
|  Machine: Turing Machine                |
|                                         |
|  +-----------------------------------+  |
|  |    Type-1: Context-Sensitive      |  |
|  |  Machine: Linear Bounded Automaton|  |
|  |                                   |  |
|  |   +---------------------------+   |  |
|  |   | Type-2: Context-Free      |   |  |
|  |   | Machine: Pushdown Automaton|  |  |
|  |   |                           |   |  |
```



Each class is included inside the next, representing containment.

Type-3 ⊆ Type-2 ⊆ Type-1 ⊆ Type-0

The **Chomsky Hierarchy**, developed by Noam Chomsky, organizes formal languages into a hierarchy based on the **type of grammars** and **machines** needed to recognize them. It consists of **four levels**, each more expressive than the one above it.

The following table summarizes each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognizes it, and the form its rules must have:

Grammar	Language	Automaton	Production rules(constraints)
Type-0	Recursively Enumerable	Turing machine	α –>ß (no constraints)
Type-1	Context Sensitive	Linear-bounded non-deterministic Turing machine	α –>ß | α | <= | ß |
Type-2	Context Free	Nondeterministic pushdown automaton	α –>ß | α | <= | ß | | α | = 1.
Type-3	Regular	Finite state automaton	V –> VT / T (or) V –> TV /T

This is a hierarchy. Therefore every language of type 3 is also of type 2, 1 and 0. Similarly, every language of type 2 is also of type 1 and type 0, etc.

Levels of the Chomsky Hierarchy:

1. **Type-0: Unrestricted Grammars**
    - **Languages:** Recursively Enumerable Languages
    - **Machine:** Turing Machine
    - **Most powerful** – can express any computable language.
2. **Type-1: Context-Sensitive Grammars**
    - **Languages:** Context-Sensitive Languages
    - **Machine:** Linear Bounded Automaton
    - Rules depend on context; more restricted than Type-0.
3. **Type-2: Context-Free Grammars**
    - **Languages:** Context-Free Languages
    - **Machine:** Pushdown Automaton
    - Mostly used in programming language syntax, such as in compilers.
4. **Type-3: Regular Grammars**
    - **Languages:** Regular Languages
    - **Machine:** Finite Automaton
    - Most restricted; often used in text processing and lexical analysis.

2. Pushdown Automata (PDA) and Pumping Lemma

a) Define **push down automata** with an example. (2)

a) Define Pushdown Automata (PDA) with Example

A **Pushdown Automaton (PDA)** is a type of automaton that uses a stack as an additional memory structure. It is an extension of Finite Automata (FA) that allows it to recognize a broader class of languages — specifically, **context-free languages (CFLs)**.

A **Pushdown Automaton (PDA)** is a type of **automaton** that uses a **stack** in addition to its finite control.
It is more powerful than a **Finite Automaton (FA)** but less powerful than a **Turing Machine**.

PDA is mainly used to recognize **Context-Free Languages (CFLs)**.

Formal Definition:

A **PDA** is defined as a 7-tuple:$M=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$

Where:

Symbol	Meaning
Q	Finite set of states
Σ	Input alphabet
Γ	Stack alphabet
δ	Transition function: $\delta(q, a, X) \rightarrow (p, \gamma)$ where: – q = current state – a = current input symbol (or ε) – X = top of stack symbol – γ = string to replace X on stack
q_0	Start state
Z_0	Initial stack symbol
F	Set of accepting (final) states

Working Principle:

- PDA reads input from left to right.
- It can **push** or **pop** symbols on the stack.
- The **stack** provides **memory**, allowing PDA to recognize patterns like matching parentheses.
- PDA can accept input by:
    1. **Final state**, or
    2. **Empty stack.**

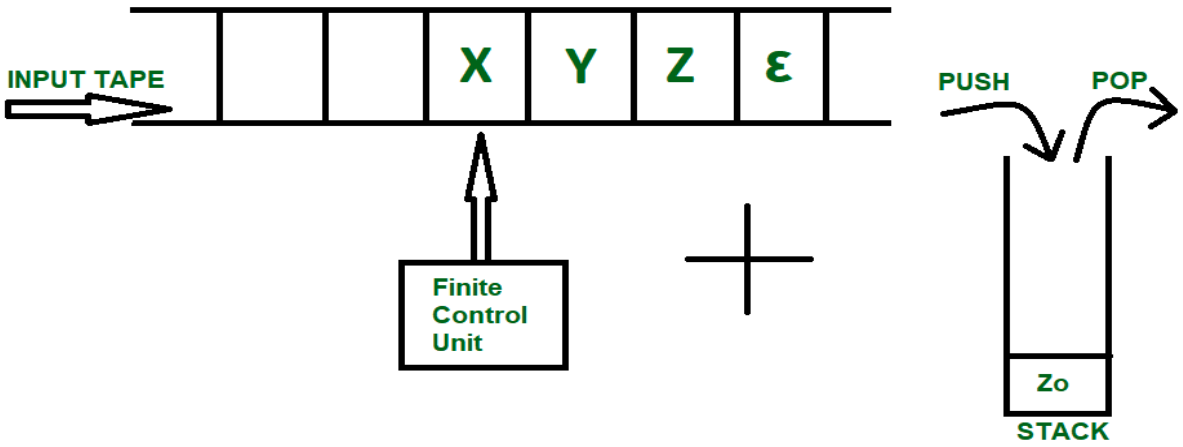


Diagram (conceptually):
Input Tape → a b b a
Stack → Z0 ↓
States → q0, q1, qf

Transition example:

δ(q0, a, Z0) = (q1, AZ0)
δ(q1, b, A) = (q1, ε)

Meaning:

- When reading a, push A on stack.
- When reading b, pop A from stack.

Example:

PDA for language:
[
L = { a^n b^n \ | \ n ≥ 0 }
]
Steps:

- For each a, push symbol (say X) on stack.
- For each b, pop one X.
- Accept if stack becomes empty at end.

The stack allows the PDA to remember an unlimited amount of information, making it suited for languages with nested structures like parentheses.

---

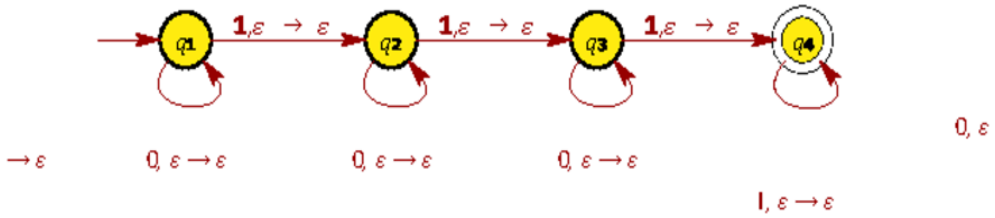
b) Give pushdown automata that recognize the following languages: (5)

- (A) $A = \{w \in \{0,1\}^* \mid w \text{ contains at least three } \mathbf{1s}\}$
- (B) $B = \{w \in \{0,1\}^* \mid w = w^R \text{ and the length of } w \text{ is } \mathbf{odd}\}$

Question with Solutions Part 5

1. Give pushdown automata that recognize the following languages. Give both a drawing and 6-tuple specification for each PDA.

(a) $A = \{w \in \{0,1\}^* \mid w$ contains at least three 1s$\}$ **Answer:**



We formally express the PDA as a 6-tuple $(Q,\Sigma,\Gamma,\delta,q_1,F)$, where

- $Q = \{q_1,q_2,q_3,q_4\}$
- $\Sigma = \{0,1\}$
- $\Gamma = \{0,1\}$
- transition function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$ is defined by

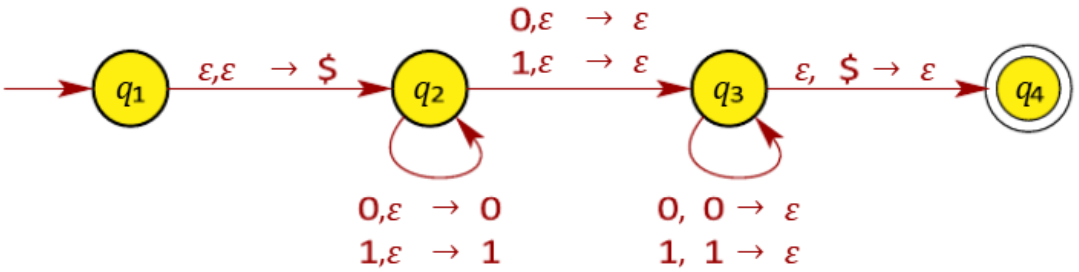
Input:	0			1			ε		
Stack:	0	1	ε	0	1	ε	0	1	ε
q_1			$\{(q_1,\varepsilon)\}$			$\{(q_2,\varepsilon)\}$			
q_2			$\{(q_2,\varepsilon)\}$			$\{(q_3,\varepsilon)\}$			
q_3			$\{(q_3,\varepsilon)\}$			$\{(q_4,\varepsilon)\}$			
q_4			$\{(q_4,\varepsilon)\}$			$\{(q_4,\varepsilon)\}$			

Blank entries are $\emptyset$.

- q_1 is the start state
- $F = \{q_4\}$

Note that A is a regular language, so the language has a DFA. We can easily convert the DFA into a PDA by using the same states and transitions and never push nor pop anything to/from the stack.

(b) $B = \{w \in \{0,1\}^* \mid w = w^R$ and the length of w is odd$\}$ **Answer:**



For any string www in BBB:

- Length is **odd** → $|w|=2n+1$ for some $n\geq 0$
- The string is a **palindrome** → the first nnn symbols must match the last nnn in reverse order
- There is **one middle symbol** that doesn't need to match anything

---

Simplified PDA Operation

1. **q2 (Push Phase):**
    - Read the first half of the string (first nnn symbols)
    - Push each symbol onto the stack
2. **Transition q2 → q3 (Middle):**
    - Non-deterministically guess the **middle symbol**
    - Read it and **don't touch the stack**
3. **q3 (Pop Phase):**
    - Read the last half (last nnn symbols)

- For each input symbol, **pop** the stack and ensure they match (reverse order)

4. Accept if:

   - The stack is back to the **start symbol** (empty apart from initial marker)

   - Input is fully consumed

reverse order) the symbols before the middle.

We formally express the PDA as a 6-tuple $(Q,\Sigma,\Gamma,\delta,q_1,F)$, where

- $Q = \{q_1,q_2,q_3,q_4\}$
- $\Sigma = \{0,1\}$
- $\Gamma = \{0,1,\$\}$ (use \$ to mark bottom of stack)
- transition function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$ is defined by

Input:	0				1				ε			
Stack:	0	1	$	ε	0	1	$	ε	0	1	$	ε
q_1												$\{(q_2,\$)\}$
q_2				$\{(q_2,0),(q_3,\varepsilon)\}$				$\{(q_2,1),(q_3,\varepsilon)\}$				
q_3	$\{(q_3,\varepsilon)\}$					$\{(q_3,\varepsilon)\}$					$\{(q_4,\varepsilon)\}$	
q_4												

Blank entries are $\emptyset$.

- q_1 is the start state
- $F = \{q_4\}$

c) Use the **pumping lemma** to prove that the language

$A = \{0^{2n}1^{3n}0^n \mid n \geq 0\}$ is not context free.

To prove that the language

$$A = \{0^{2n}1^{3n}0^n \mid n \geq 0\}$$

is **not context-free**, we will use the **Pumping Lemma for Context-Free Languages**.

Pumping Lemma for CFLs (Informal Statement)

If A is a context-free language, then there exists a constant p (the **pumping length**) such that any string $s \in A$ with $|s| \geq p$ can be split into five parts:

$$s = uvwxy$$

such that:

1. $|vwx| \leq p$
2. $|vx| \geq 1$ (i.e., v or x is not empty)
3. For all $i \geq 0$, the string $uv^iwx^iy \in A$

Proof by Contradiction

Assume:

The language A is context-free.

Then by the pumping lemma, there exists a pumping length p.

Step 1: Choose a String in A

Let's pick:

$$s = 0^{2p}1^{3p}0^p$$

Clearly, $s \in A$, with $n = p$. Also, $|s| = 2p + 3p + p = 6p \geq p$, so the pumping lemma applies.

Step 2: Split the String

The pumping lemma says:

$$s = uvwxy$$

with:

- $|vwx| \leq p$
- $|vx| \geq 1$
- $uv^iwx^iy \in A$ for all $i \geq 0$

Step 3: Analyze vwx

Since $|vwx| \leq p$, the substring vwx can only span **one** of the following blocks (can't span all of them as each block is ≥ p long):

1. The **first block of 0s** (i.e., the 0^{2p})
2. The **block of 1s** (i.e., the 1^{3p})
3. The **last block of 0s** (i.e., the 0^{p})

We handle each case to show a contradiction.

Case 1: vwx is within the first block of 0s

- Pumping v and x changes the number of 0s in the first block.
- New string after pumping: uv^2wx^2y will have **more than $2n$ 0s** in the first block, but the other blocks won't change proportionally.
- So the string won't be of the form $0^{2n}1^{3n}0^n$.
- **Contradiction**.

Case 2: vwx is within the 1s

- Pumping v and x changes the number of 1s.
- New string: number of 1s is no longer $3n$, but the first and last blocks remain unchanged.
- Not in A.
- **Contradiction**.

Case 3: vwx is within the last block of 0s

- Pumping adds or removes 0s from the last block only.
- The number of 0s in the last block will not match the required n, making the structure invalid.
- **Contradiction**.

Conclusion

In all cases, pumping v and x produces a string not in A, which contradicts the pumping lemma. Therefore:

A is not a context-free language.

a) What is **Turing Machine**? Give advantages of it. (3)

1. Turing Machine (TM): Definition

A **Turing Machine (TM)** is a **mathematical model of computation** that defines an abstract machine capable of simulating any computer algorithm.
It was introduced by **Alan Turing (1936)** and is used to describe **what can be computed** and **how efficiently**.

Formal Definition:

A **Turing Machine** can be represented as a 7-tuple:M=(Q,Σ,Γ,δ,q0,qaccept,qreject)

Where:

- Q: Set of states
- Σ\Sigma: Input alphabet
- Γ\Gamma: Tape alphabet (Σ⊆Γ)
- δ\delta: Transition function δ(q, X) → (p, Y, D) where D ∈ {L, R} (Left or Right movement)
- q0: Start state
- q_accept: Accepting state
- q_rejectq: Rejecting state

Working Principle:

- TM has an **infinite tape** divided into cells.
- Each cell contains one symbol (from Γ).
- A **tape head** reads/writes symbols and moves **left or right**.
- Based on the **current state** and **tape symbol**, the **transition function δ** determines:
    - The **next state**,
    - The **symbol to write**, and
    - The **direction to move**.
- If TM reaches **q_accept**, the input is **accepted**; if **q_reject**, it's **rejected**.

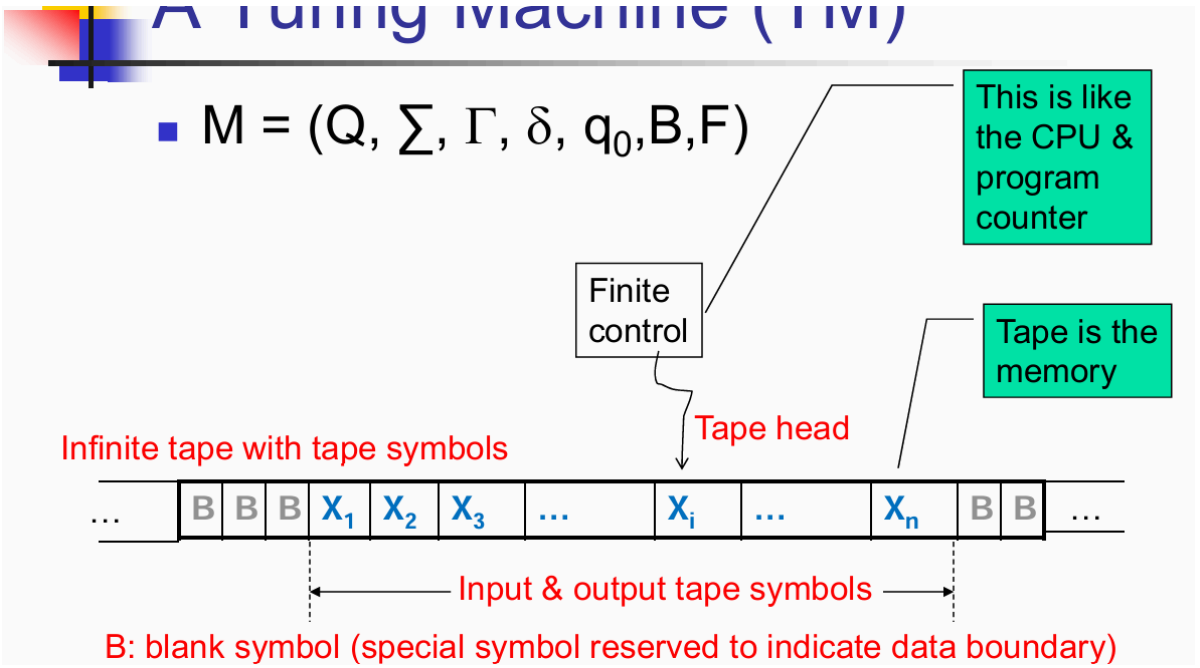


Diagram (conceptually):

... | 1 | 0 | 1 | 1 | _ | _ | ...

↑

Tape Head

At each step:

δ(q, X) = (p, Y, D)

means: In state q reading X → write Y, move D (L or R), and go to state p.

Example:

A TM that accepts strings with equal number of 0's and 1's is a **non-trivial** example.

Here are the **advantages of Turing Machine**, with each explained in **one line** and including **more points**:

Advantages of Turing Machine (TM)

1. **Universal Model of Computation:**
   Turing Machines can simulate any algorithm or computational process.

2. **Simple Yet Powerful Design:**
   Despite its basic components, a TM can model complex systems and processors.

3. **Foundation for Computer Science:**
   It provides the theoretical basis for understanding computation and algorithmic limits.

4. **Supports Infinite Memory:**
   The infinite tape allows handling arbitrarily large input sizes, unlike finite automata.

5. **Recognizes Recursively Enumerable Languages:**
   It can recognize languages beyond the power of context-free and regular languages.

6. **Flexible Variants:**
   Supports extensions like multi-tape or non-deterministic models for advanced computation.

7. **Basis for Decidability and Undecidability:**
   Helps classify problems into decidable and undecidable categories.

8. **Can Model Real Computers:**
   Though abstract, a TM can simulate any modern computer's behavior and logic.

9. **Useful in Algorithm Design:**
   It provides a standard for expressing and analyzing algorithms formally.

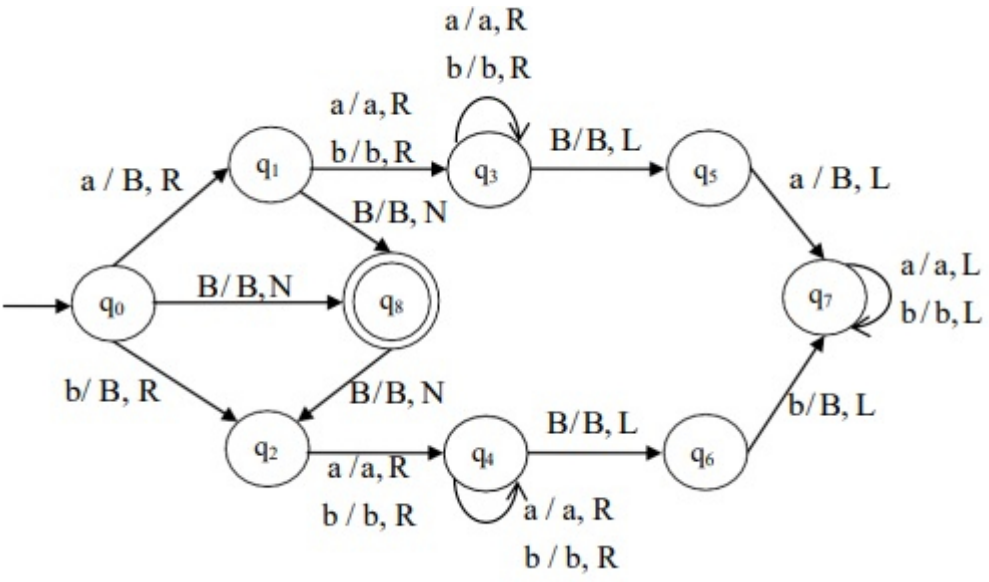
10. **Enables Proofs in Complexity Theory:**
    Used in proving computational complexity, NP-hardness, and other theoretical frameworks.

---

b) Construct a Turing Machine that accepts the language of **palindromes** over{a, b}. Also specify the moves to trace the strings {abba},{abbaa},{aabba} (6)

Turing Machine Design (Idea):

1. **Mark the first character** (replace it with X) and remember its value.
2. **Move to the end of the tape** and find the matching last character.
3. **If matching**, replace it with X.
4. Repeat for the next inner characters.
5. If all characters are correctly matched → **accept**.
6. If a mismatch occurs → **reject**.



The Turing machine, M is given by M = (Q, Σ, Γ, δ, q0, B, F)

Where,

Q = {q0, q1, q2, q3, q4, q5, q6, q7, q8}

Σ = {a, b}

Γ = {a, b, B}

δ ⇒ Given by the above mentioned transition diagram,

q0 = {q0}

B = {B}

F = {q8}

Current State	Current Symbol	New Symbol	Move	Next State
q0	a	a	R	q0
q0	b	b	R	q0
q0	_	_	L	q1
q1	a	a	L	q1
q1	b	b	L	q1
q1	_	_	R	q2
q2	a	_	R	q3
q2	b	_	R	q4
q3	a	a	R	q3
q3	b	b	R	q3
q3	_	_	R	accept
q4	a	a	R	q4
q4	b	b	R	q4
q4	_	_	R	accept

In this table:

---

1. Trace for abba (Palindrome)

Initial: abba
Step 1: Xbba ← Mark first 'a'
Step 2: XbbX ← Match last 'a'
Step 3: XXbX ← Mark first 'b'

Step 4: XXXX ← Match last 'b'
Final: All Xs → ACCEPT

---

2. Trace for `abbaa` (NOT a palindrome)

Initial: abbaa
Step 1: Xbbaa ← Mark first 'a'
Step 2: XbbaX ← Match last 'a'
Step 3: XXbaX ← Mark first 'b'
Step 4: XXbaX ← Last char is 'a', but expected 'b' → REJECT

---

3. Trace for `aabba` (NOT a palindrome)

Initial: aabba
Step 1: Xabba ← Mark first 'a'
Step 2: XabbX ← Match last 'a'
Step 3: XXbbX ← Mark next 'a'
Step 4: XXbbX ← Last is 'b' but expected 'a' → REJECT

---

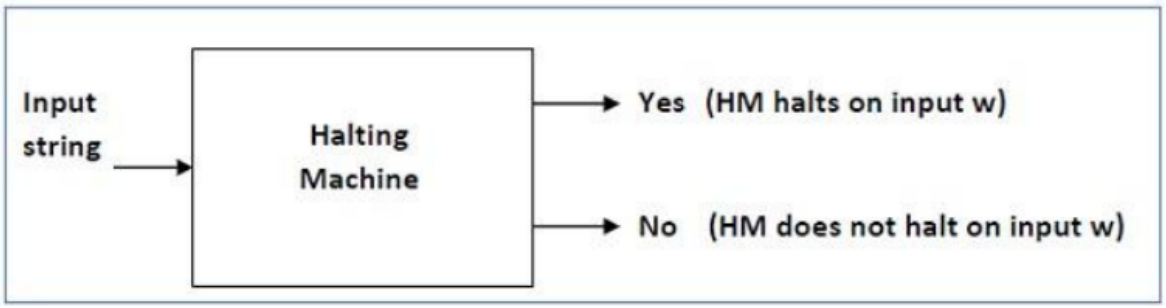
c) What is meant by a **halting problem** in a Turing Machine? Explain with an example. (3)

The **halting problem** is the question of determining whether a given Turing Machine **will eventually halt** (stop) or **run forever** on a given input.

Alan Turing proved that a general solution to this problem **does not exist** — it is **undecidable**.

Input − A Turing machine and an input string w.

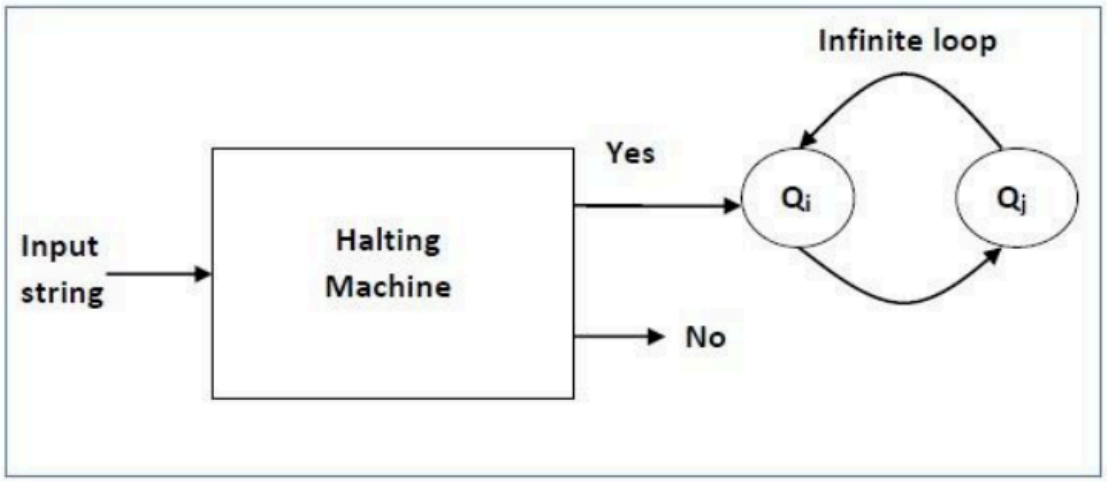
Problem − Does the Turing machine finish computing of the string w in a finite number of steps? The answer must be either yes or no.



Now we will design an inverted halting machine (HM)' as −

If **H** returns YES, then loop forever.
If **H** returns NO, then halt.

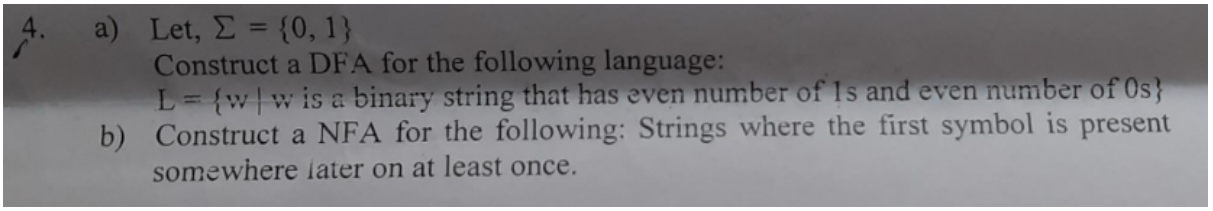
The following is the block diagram of an 'Inverted halting machine' –



Further, a machine $\mathbf{(HM)_2}$ which input itself is constructed as follows –

If $(HM)_2$ halts on input, loop forever.
Else, halt.

Here, we have got a contradiction. Hence, the halting problem is **undecidable**.


4. a) Let, $\Sigma = \{0, 1\}$
Construct a DFA for the following language:
$L = \{w \mid w$ is a binary string that has even number of 1s and even number of 0s$\}$
b) Construct a NFA for the following: Strings where the first symbol is present somewhere later on at least once.

Khatai Ans valo kor dewa ache pls visit this page

a) DFA for the Language:

L = { w has an even number of 1s and even number of 0s} }

Alphabet: sigma = {0, 1})

Idea

To track both:

- **Even or odd number of 0s**, and
- **Even or odd number of 1s**

We need to *remember* the state of both — so we need **4 states**:

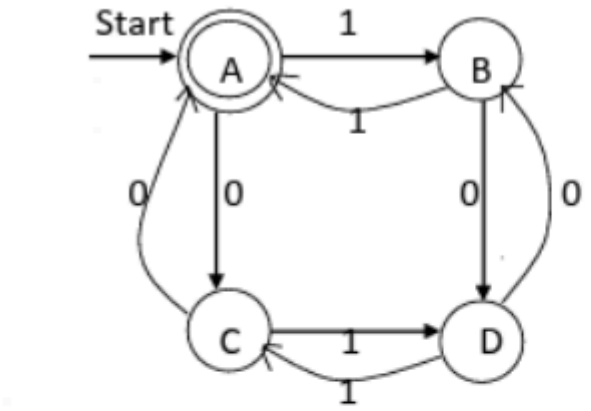
State	Meaning
q_0	Even 0s, Even 1s
q_1	Odd 0s, Even 1s
q_2	Even 0s, Odd 1s
q_3	Odd 0s, Odd 1s

State Transitions

Current State	Input = 0	Input = 1
q_0	$\rightarrow q_1$	$\rightarrow q_2$
q_1	$\rightarrow q_0$	$\rightarrow q_3$
q_2	$\rightarrow q_3$	$\rightarrow q_0$
q_3	$\rightarrow q_2$	$\rightarrow q_1$

DFA Diagram

Transition diagram:



Transition table:

δ	0	1
→*A	C	B
B	D	A
C	A	D
D	C	B

- (q_0) is the **start** and also the **accepting** state (since both counts start at even)
- Only (q_0) is accepting, since it represents **even 0s and even 1s**

Final Answer: Summary

- **States:** (Q = {q_0, q_1, q_2, q_3})
- **Start State:** (q_0)

- **Accepting State:** ({q_0})
- **Alphabet:** ({0, 1})
- **Transition Function:** As shown in the table above

This DFA recognizes all binary strings that contain **an even number of 0s and an even number of 1s**.

b) Construct an NFA for the following: Strings where the first symbol is present somewhere later on at least once.[6]

To construct an **NFA** for the language:

> **Strings where the first symbol is present somewhere later on at least once**

This means:

- The **first character** of the input (either `0` or `1`) must **reappear later** in the string.
- Examples:
  - Accepted: `00`, `0110`, `1001`, `010110`
  - Rejected: `01`, `10` (because the first symbol does not repeat later)

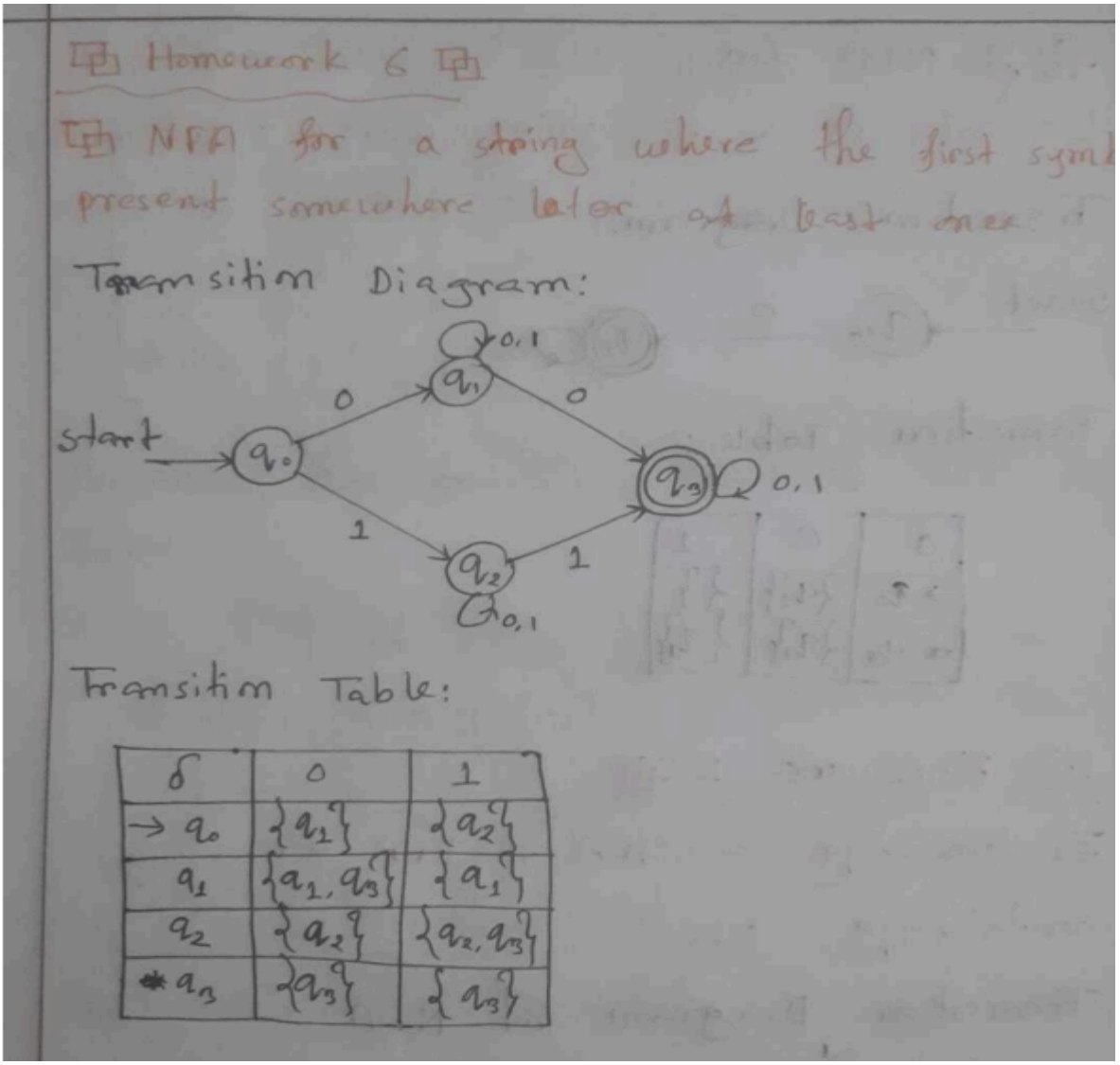
💡 Idea:

We can design the NFA using nondeterminism:

1. Read the first symbol (`0` or `1`) and remember it using states.
2. Then move through the rest of the string.
3. If we find the same symbol again, accept.

States:

- `q0`: Start state (before reading first character)
- `q1`: Remember first symbol was `0`
- `q2`: Remember first symbol was `1`
- `qf`: Accepting state (once the first symbol is seen again)
- `qd`: Dead state (optional — not always necessary in NFA)



a) Build an **NFA** for the following language: L = { w ends in **01**}

Idea

- We need strings that **end with "01"**.
- The NFA can read any sequence of `0`s and `1`s first (`Σ*`) and then check the **last two symbols**.
- Nondeterminism allows us to **guess** when we are at the last two symbols.

NFA Construction

States:

- `q0`: Start state, before seeing the ending `01`
- `q1`: Saw `0` as the second last symbol
- `q2`: Saw `01` → Accepting state

Explanation:

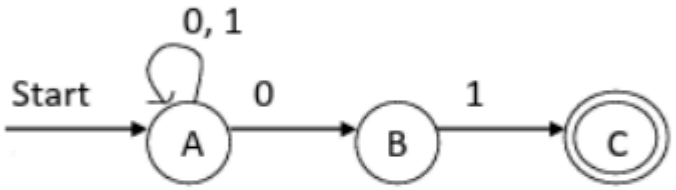
- `q0` reads any prefix (loops on 0 or 1)
- Nondeterministically jumps to `q1` when it sees a `0` that **might be the second last symbol**
- If next symbol is `1`, move to `q2` → accept
- Accept state is `q2`

Let, ∑ = {0, 1}

A NFA for the following language:

L = {w | w ends in 01}

Transition diagram:



Transition table:

δ	0	1
→ A	{A,B}	{A}
B	{ }	{C}
* C	{ }	{ }

b) Advantages and Caveats of NFA (Nondeterministic Finite Automaton)

Advantages of NFA

1. **Simplicity of Design:**
   Easier to design than DFA for some complex languages because nondeterminism allows multiple choices.

2. **Fewer States:**
   Often requires **fewer states** than a DFA for the same language.

3. **Flexible Transitions:**
   Can use **ε-transitions** (moves without input), which makes modeling certain patterns easier.

4. **Good for Theoretical Analysis:**
   Useful in proofs and conversions (e.g., from regex to automata).

5. **Expressiveness:**
   Can represent the same languages as DFA (all regular languages) but more succinctly.

Caveats / Disadvantages of NFA

1. **Nondeterminism Not Directly Implementable:**
   Computers cannot directly execute nondeterministic choices; must simulate with DFA or backtracking.

2. **Conversion to DFA Can Cause State Explosion:**
   Converting NFA to DFA may result in exponentially more states (subset construction).

3. **Complexity in Simulation:**
   Checking acceptance requires **tracking multiple paths**, which can be less efficient.

4. **Ambiguity in Transitions:**
   Multiple transitions for the same input can complicate analysis or implementation.

5. Limited Power: Even though NFA is more flexible, it still has some limitations and cannot handle very complex patterns or languages.

✅ **Summary:**

- **Advantages:** Easier design, fewer states, uses ε-moves, concise representation.

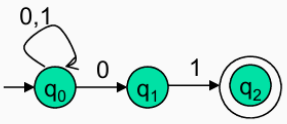
- **Caveats:** Harder to implement directly, possible exponential growth when converted to DFA, multiple paths to track.

c) Convert the NFA from Question 5(a), to **DFA**. L={w | w ends in 01},Σ={0,1}.

NFA to DFA construction: Example

- *L = {w | w ends in 01}*

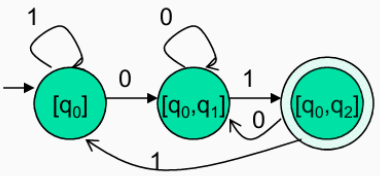
NFA:



δ_N	0	1
q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
$*q_2$	Ø	Ø

DFA:

δ_D
Ø
$[q_0]$
$[q_1]$
$*[q_2]$
$[q_0,q_1]$
$*[q_0,q_2]$
$*[q_1,q_2]$
$*[q_0,q_1,q_2]$



δ_D	0	1
$[q_0]$	$[q_0,q_1]$	$[q_0]$
$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_2]$
$*[q_0,q_2]$	$[q_0,q_1]$	$[q_0]$

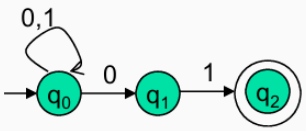
0. Enumerate all possible subsets
1. Determine transitions
2. Retain only those states reachable from $\{q_0\}$

27

NFA to DFA: Repeating the example using *LAZY CREATION*

- *L = {w | w ends in 01}*

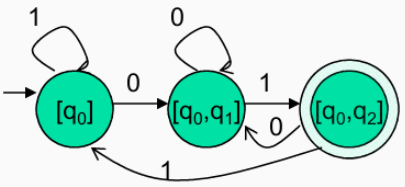
NFA:



δ_N	0	1
q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
$*q_2$	Ø	Ø

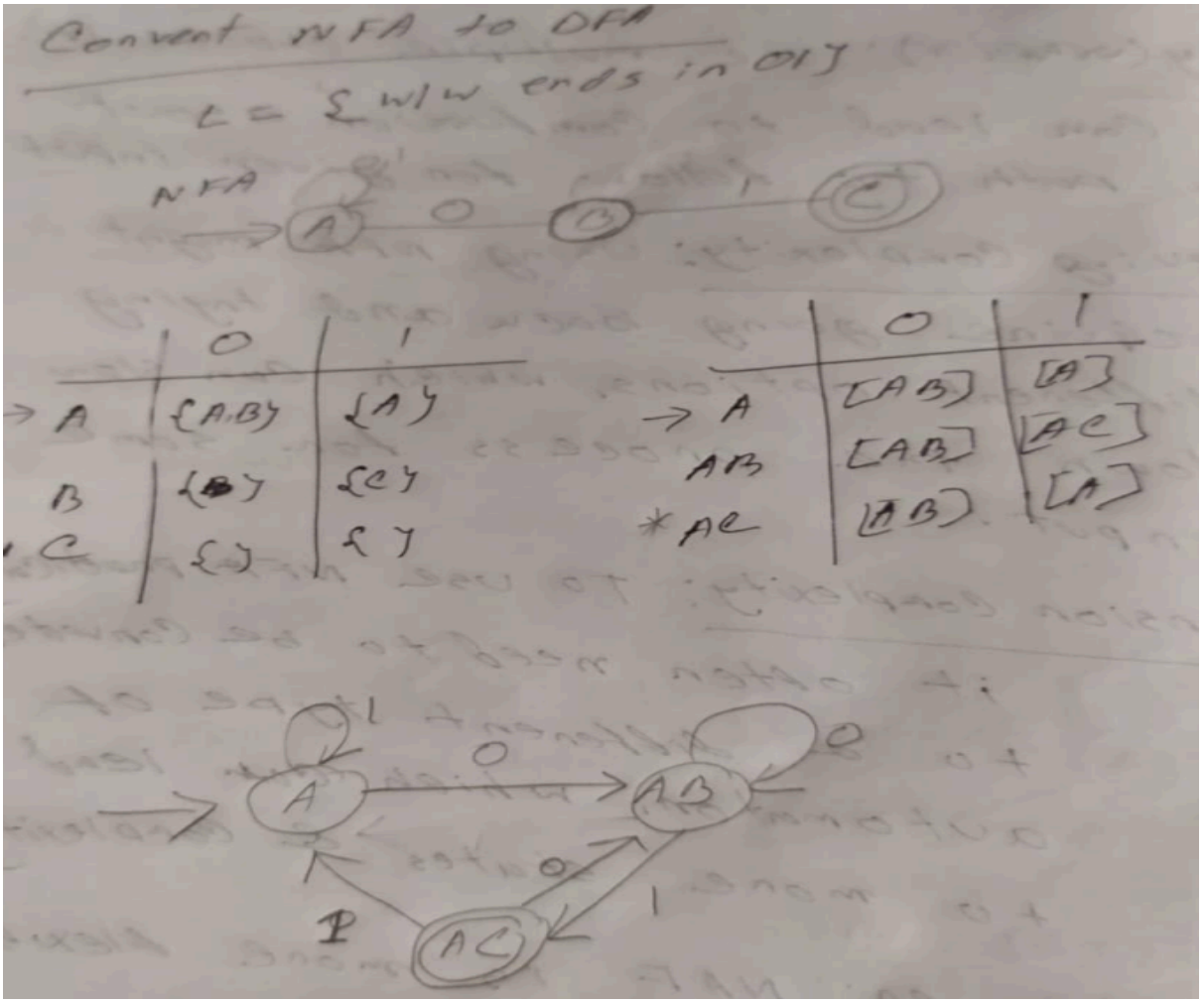
DFA:

δ_D	0	1
$[q_0]$	$[q_0,q_1]$	$[q_0]$



Main Idea:

Introduce states as you go (on a need basis)



6. Epsilon-NFA and Conversion

a) Why explicit **epsilon-transitions** in finite automata is important? (2)

FA with ε-Transitions

- We can allow explicit ε-transitions in finite automata
  - i.e., a transition from one state to another state without consuming any additional input symbol
  - Explicit ε-transitions between different states introduce non-determinism.
  - Makes it easier sometimes to construct NFAs

***Definition:* ε-NFAs are those NFAs with at least one explicit ε-transition defined.**

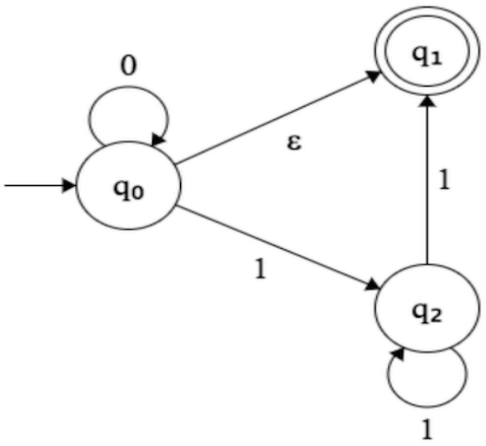
- ε -NFAs have one more column in their transition table

33

An **ε-transition** in a finite automaton is a move from one state to another **without consuming any input symbol**.

Importance of ε-Transitions (Short Version)

1. **Simplifies NFA construction** from regular expressions.
2. **Allows branching** without consuming input.
3. **Supports nondeterminism** efficiently.
4. **Combines smaller automata** into larger ones.
5. **Helps in NFA → DFA conversion** using ε-closure.



b) Build an **epsilon-NFA** for the following language: L = \{ w is **empty**, or if non-empty will end in **01**}

Idea

- The language includes the **empty string** → use an **ε-transition** from start to accepting state.
- Non-empty strings must **end with 01** → similar to the DFA/NFA for `ends in 01`.
- Use ε-transitions to handle **empty string** or branching.

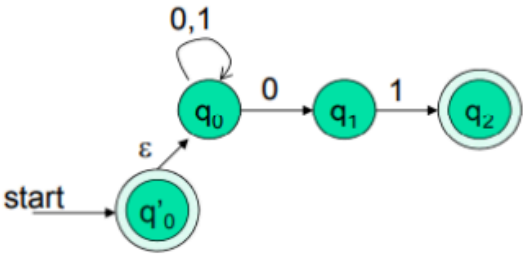
States

- `q0`: Start state
- `q1`: Saw a `0` that **might be second-last symbol**
- `q2`: Saw `01` → Accepting state
- Accepting state: `q0` (for ε / empty) and `q2`

Explanation:

1. ε-transition from `q0` → `q2` allows **accepting empty string**.
2. For non-empty strings:
    - Track last two symbols using `q1` → `q2`.
    - Accept if string ends with `01`.

L = {w | w is empty, or if non-empty will end in 01}



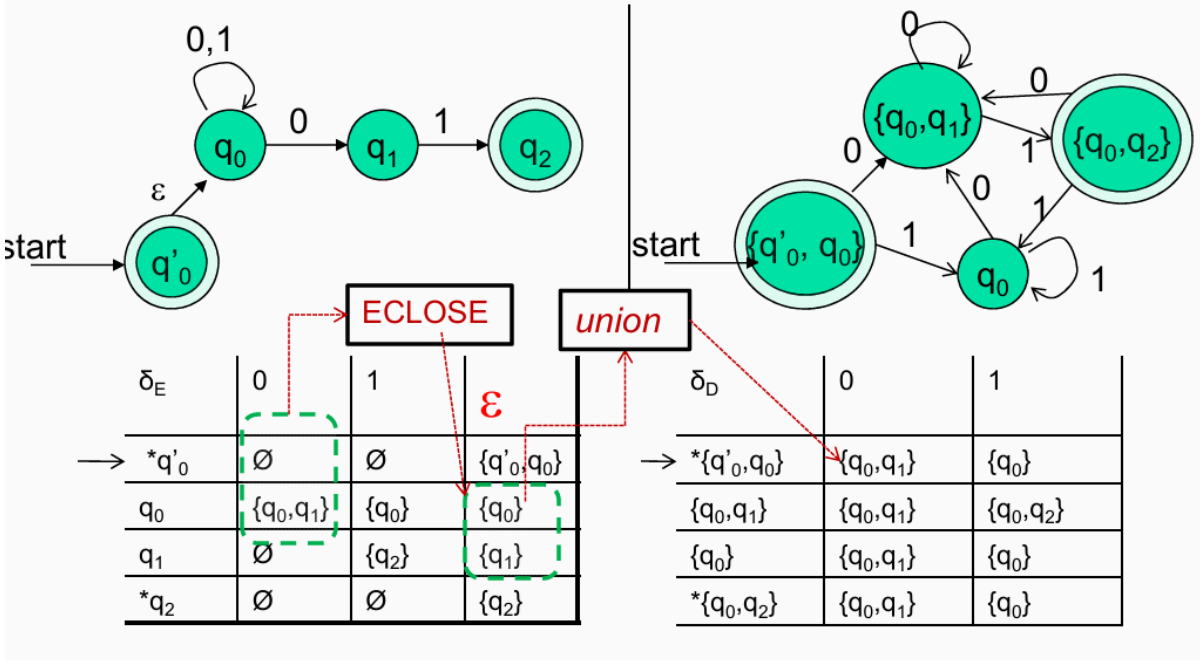
δ_E	0	1	ε
→ *q'_0	Ø	Ø	$\{q'_0,q_0\}$
q_0	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
*q_2	Ø	Ø	$\{q_2\}$

δ_D	0	1
→ *$\{q'_0,q_0\}$		
...		

c) Convert **epsilon-NFA** to **DFA** based on Question 6(b).

Example: ε-NFA ➔ DFA

L = {w | w is empty, or if non-empty will end in 01}



δ_E	0	1	ε
→ *q'_0	Ø	Ø	$\{q'_0,q_0\}$
q_0	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
*q_2	Ø	Ø	$\{q_2\}$

δ_D	0	1
→ *$\{q'_0,q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
*$\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$

7. Regular Expressions (RE) and Finite Automata

a) Describe the relation between **Regular Expressions (RE) and Finite Automata**. Show with figure that they are **interchangeable**. (6)

Definition

- **Regular Expression (RE):**
  A formula that describes a set of strings over an alphabet. Example: R=(a|b)∗ab enerates all strings ending with `ab`.

- **Finite Automaton (FA):**
  A state machine (DFA or NFA) that **accepts exactly the same language** as described by a regular expression.

Relation Between RE and FA

1. **Every Regular Expression → NFA/DFA**
   - Given a regular expression, we can **construct an NFA** (possibly with ε-transitions) that accepts the same language.
   - Example: `R = a(b|c)` → NFA with branching using ε-transitions.
2. **Every FA → Regular Expression**
   - For any DFA/NFA, there exists a **regular expression** that represents the set of strings accepted by the FA.
   - This is usually done using **state elimination method**.

Finite Automata

Finite automata is an abstract computing device. It is a mathematical model of a system with discrete inputs, outputs, states, and a set of transitions from state to state that occurs on input symbols from the alphabet.

Regular Expression

Regular Expressions are the expressions that describe the language accepted by Finite Automata. It is the representation certain set of strings in an algebraic fashion

The languages accepted by some regular expressions are referred to as Regular languages.

The various operations in the regular language are

1) Union

If R and S are two regular languages, their union R U S is also a Regular Language.

R U S = {a | a is in R or a is in S}

2) Intersection

If R and S are two regular languages, their intersection is also a Regular Language.

L ∩ M = {ab | a is in R and b is in S}

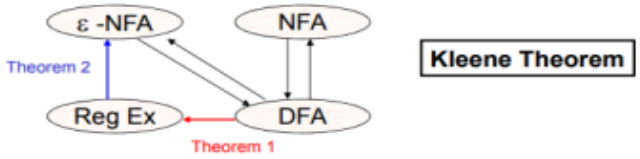
3) Kleene closure

If R is a regular language, its Kleene closure R1* will also be a Regular Language.

R* = Zero or more occurrences of language R

- To show that they are interchangeable, consider the following theorems:
  - *Theorem 1: For every DFA A there exists a regular expression R such that L(R)=L(A)*
  - *Theorem 2: For every regular expression R there exists an ε-NFA E such that L(E)=L(R)*

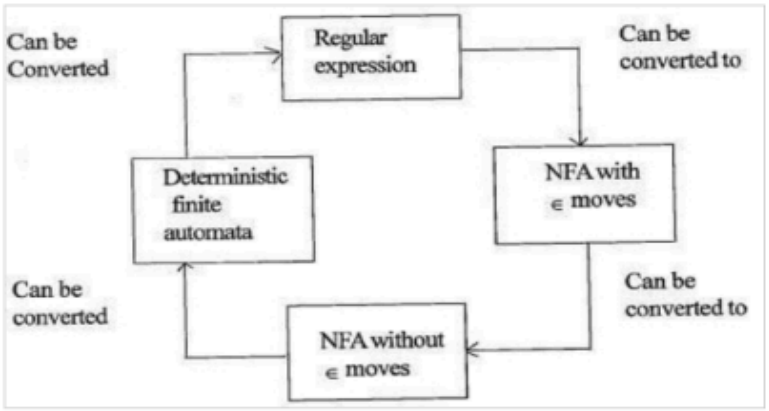
ook

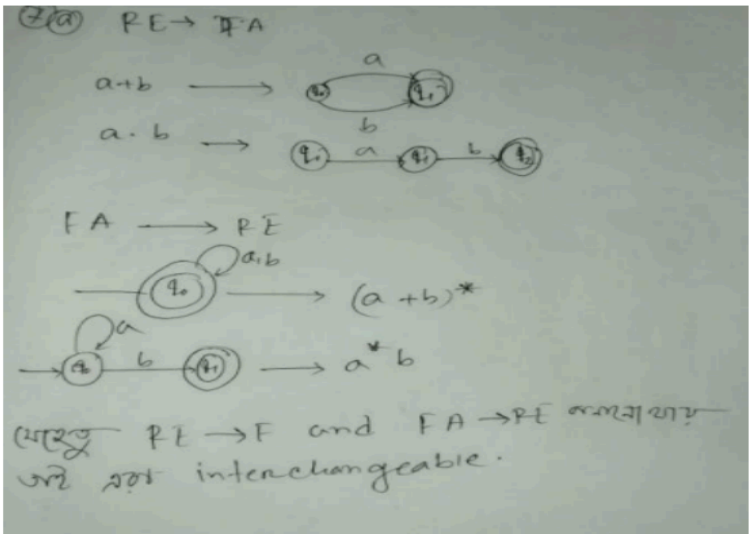


Or, .

Relationship

The relationship between FA and RE is as follows –

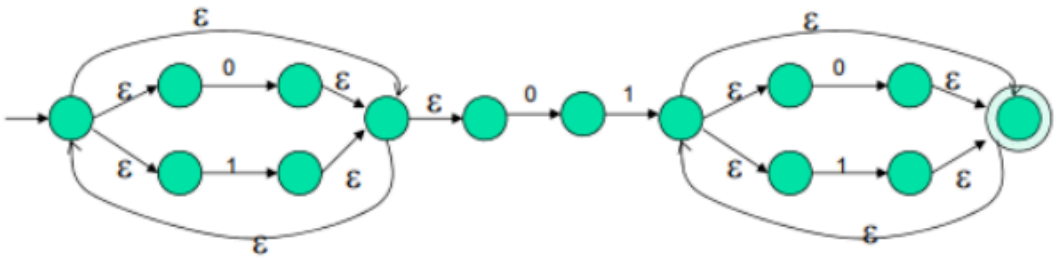




b) Convert the following RE to ε-NFA: (0+1)*01(0+1)* [6]

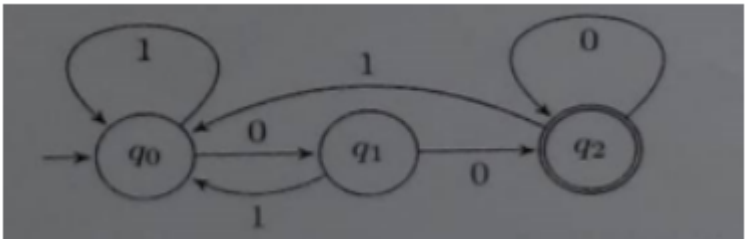
Example: (0+1)*01(0+1)*

(0+1)* 01 (0+1)*



8. Context-Free Grammar (CFG) and PDA Memory

a) Construct a **context-free grammar** for the following **DFA**: (6)



CONVERSION OF DFA TO CONTEXT FREE GRAMMAR

solution

step1: Assign a variable for Each state

$q_0 \rightarrow R$

solution

step1: Given states are q_0, q_1 and q_2.

Transactions are 0 and 1.

step2: Add a Rule $q_i \rightarrow a q_j$, if there is a Transaction from q_i to q_j

$q_0 \rightarrow 0q_1 \mid 1q_0$

$q_1 \rightarrow 0q_2 \mid 1q_0$

$q_2 \rightarrow 0q_2 \mid 1q_0$

step3: If q_i is an ~~accept~~ final state, then add a Rule $q_i \rightarrow \epsilon$

Here q_2 is the ~~acc~~ Final state

$\therefore q_2 \rightarrow \epsilon$

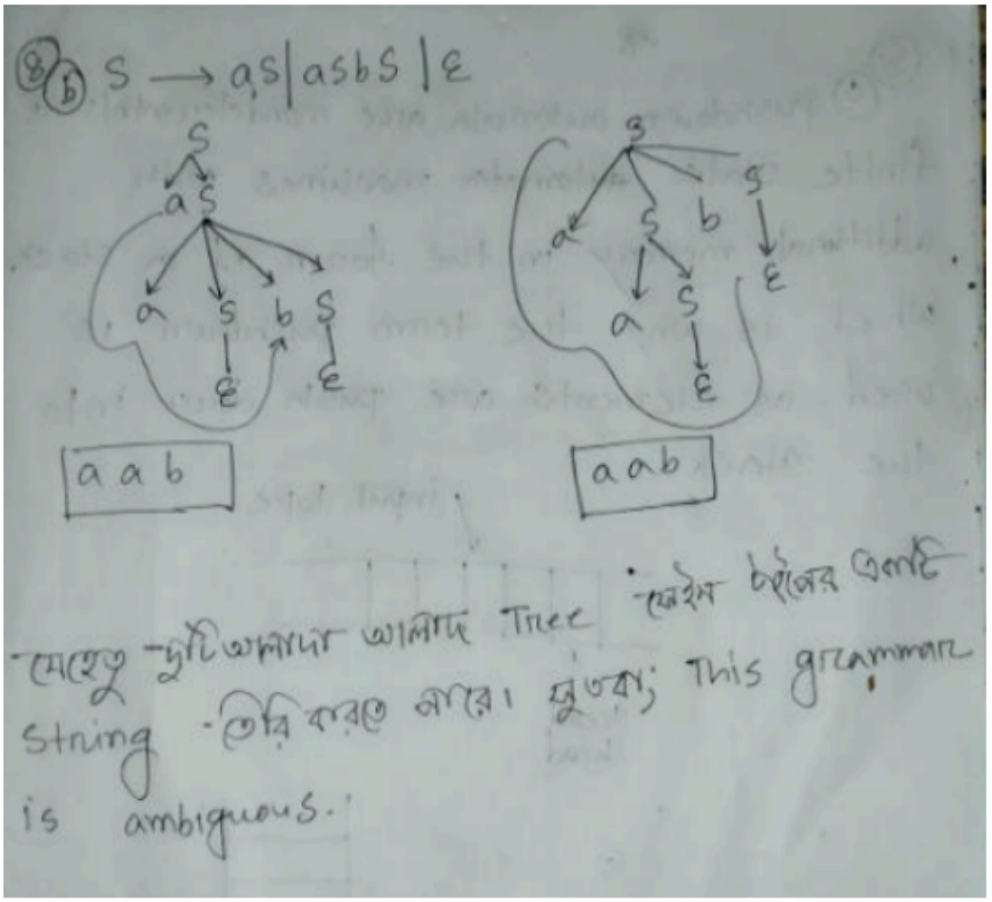
step4:

$q_0 \rightarrow 0q_1 \mid 1q_0$

$q_1 \rightarrow 0q_2 \mid 1q_0$

$q_2 \rightarrow 0q_2 \mid 1q_0 \mid \epsilon$

b) Show that the grammar ({S}, {a, b), R, S) with rules R = S→aS | aSbS | € is ambiguous. [4]



Step 1: Understand Ambiguity

A grammar is **ambiguous** if there exists **at least one string** in the language that has **two or more distinct parse trees** (or derivations).

Step 2: Find a Candidate String

Consider the string:w=aab

First derivation:

1. $S \Rightarrow aS$
2. $S \Rightarrow aSbS$ for the second S in `aS`? Let's check carefully.

- Start: S
- Option 1: $S \Rightarrow aS \rightarrow$ `aS`
- S in `aS`: $S \Rightarrow aSbS \rightarrow$ `aaSbS b`? Wait, let's make it simple.

Better candidate: `aab`:

Derivation 1:

1. $S \Rightarrow aS \rightarrow$ `aS`
2. $S \Rightarrow aSbS \rightarrow$ `a(aS)bS`
3. First S → ε → `a(a)bS` → `aabS`
4. Last S → ε → `aab` ✅

Derivation 2 (Different parse tree):

1. $S \Rightarrow aSbS \rightarrow$ `aS b S`
2. First S → `aS` → `a a S`
3. Second S → ε → `a a b S`
4. Last S → ε → `aab` ✅

Step 4: Conclusion

- The string aab has **two distinct parse trees**, so the grammar is **ambiguous**

Ans. For grammar to be ambiguous, there should be more than one parse tree for same string.

Above grammar can be written as

S → aSbS

S → bSaS

S → ∈

Lets generate a string ***'abab'.***

So, now parse tree for 'abab'.

Left most derivative parse tree 01

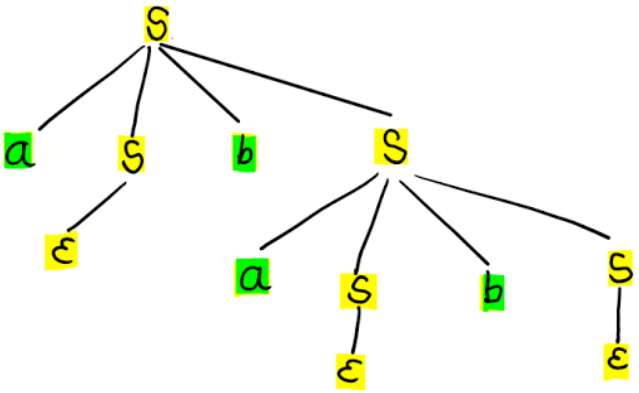
S → aSbS

S → a∈bS

S → a∈baSbS

S → a∈ba∈b∈

S → abab



Parse Tree 01

Left most derivative parse tree 02

S → aSbS

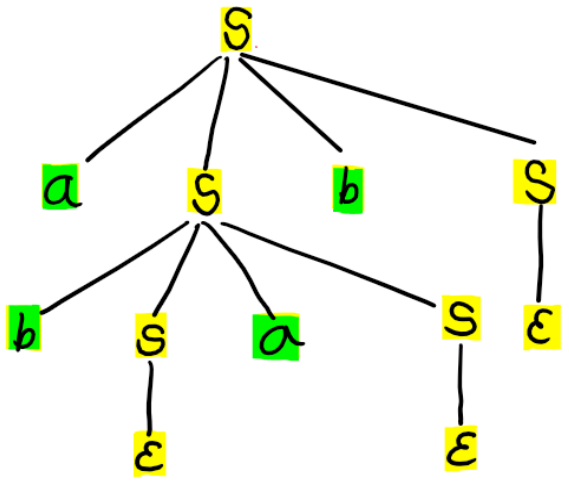
S → abSaSbS

S → ab∈aSbS

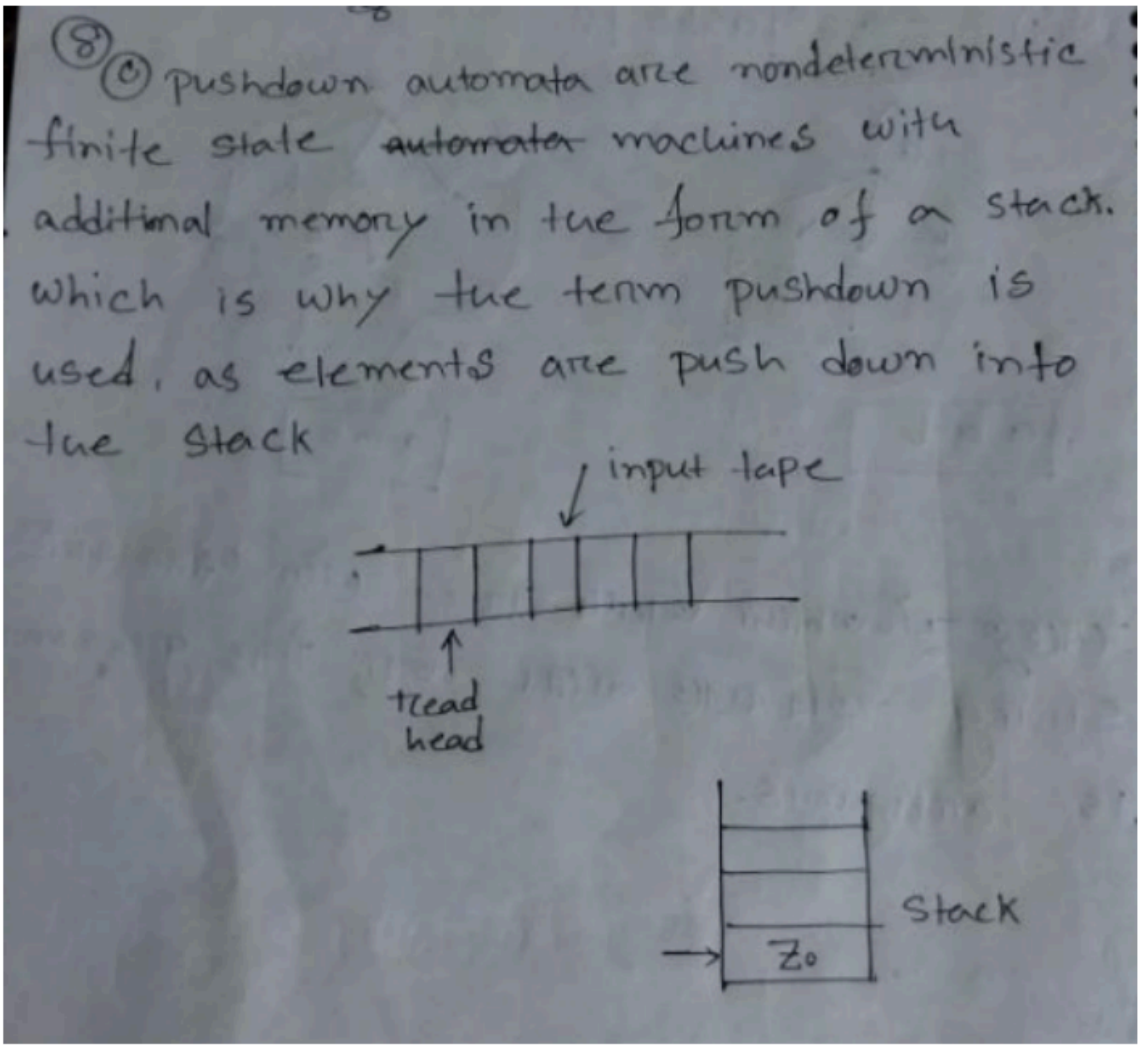
S → ab∈a∈bS

S → ab∈a∈b∈

S → abab



c) Does a pushdown automata have memory? Justify.[2]



Does a Pushdown Automaton (PDA) have memory?

Answer: Yes.

1. **PDAs have a stack.**
    - A stack is like a vertical pile of boxes where you can **put things on top (push)** or **take things off (pop)**.
2. **The stack remembers information.**
    - Unlike a simple finite automaton that “forgets” everything except its current state, a PDA can **remember many symbols** in the stack.

3. **Example:** Language (L = { a^n b^n \mid n \ge 0 })

- Step 1: Read each a → **push a onto the stack**
- Step 2: Read each b → **pop one a from the stack**
- Step 3: If the stack is empty at the end → accept

Here, the **stack "remembers" how many as were read**, which is why PDA has memory.

✅ **Simple takeaway:**

- **Finite automata** → no memory except state
- **PDA** → memory via stack