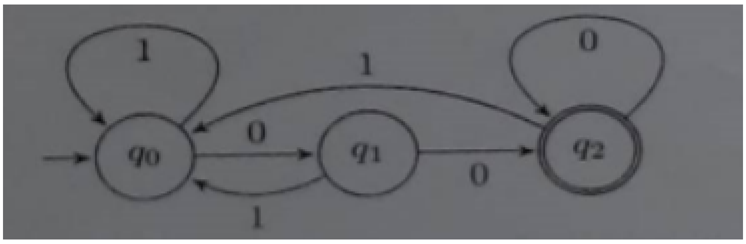
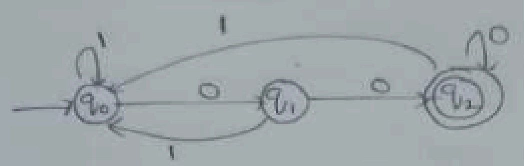
1.a) Construct a **context-free grammar** for the following **DFA**: (6)



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

$\therefore q_2 \rightarrow \epsilon$

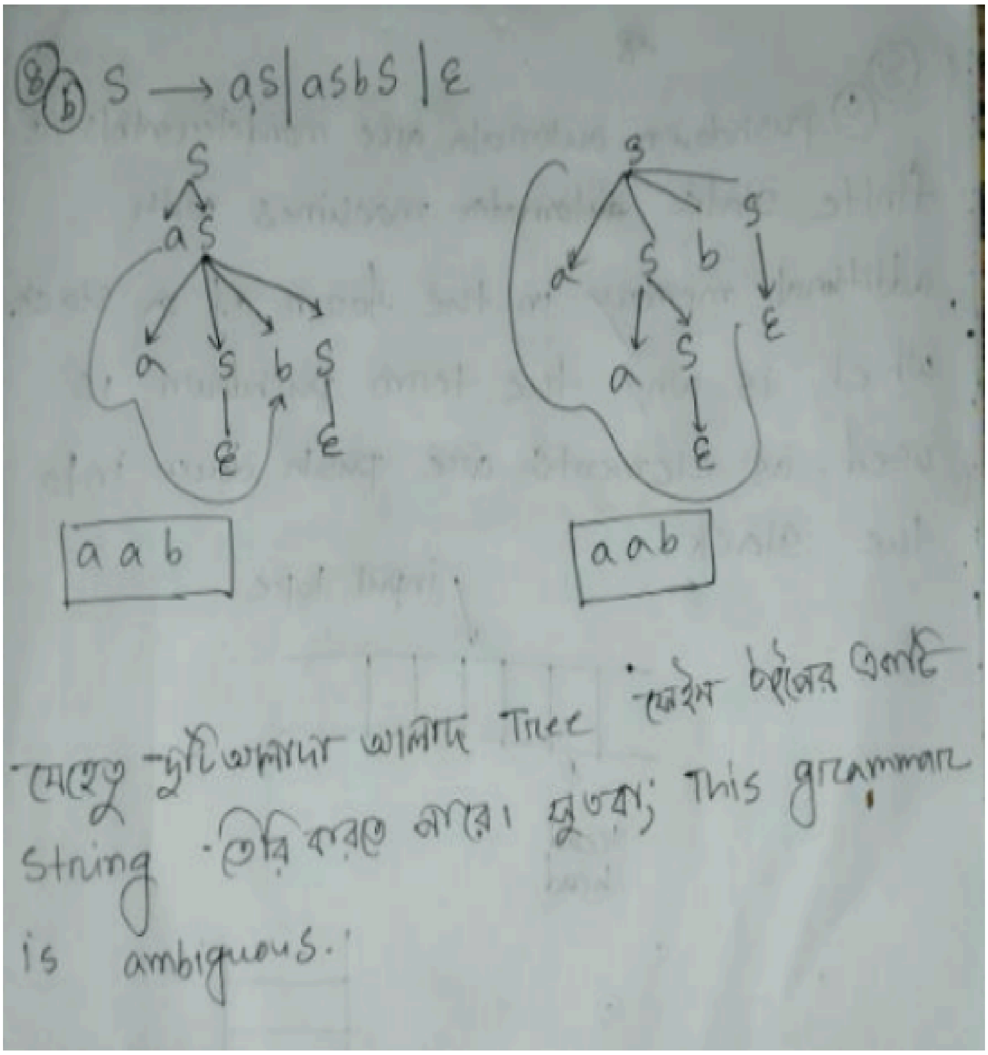
step4:

$q_0 \rightarrow 0q_1 \mid 1q_0$

$q_1 \rightarrow 0q_2 \mid 1q_0$

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

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

Better candidate: `aab`:

Derivation 1:

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

Derivation 2 (Different parse tree):

1. $S \Rightarrow aSbS$ → `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

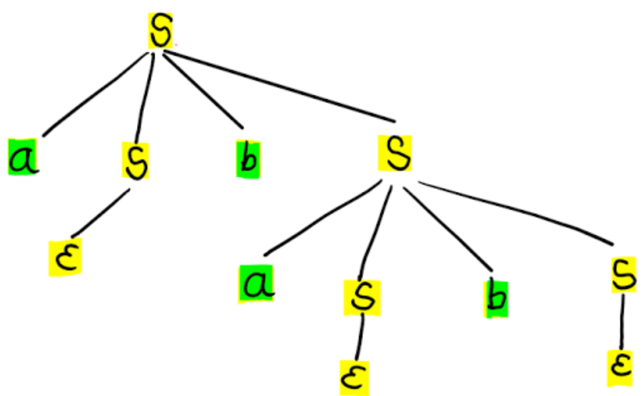
S → aSbS

S → a∈bS

S → a∈baSbS

S → a∈ba∈b∈

S → abab



Parse Tree 01

Left most derivative parse tree 02

S → aSbS

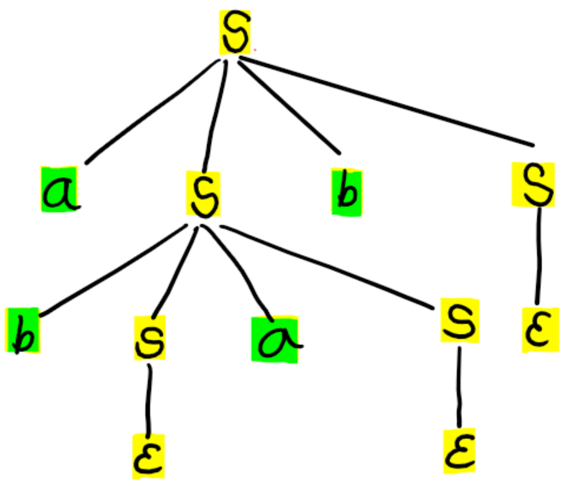
S → abSaSbS

S → ab∈aSbS

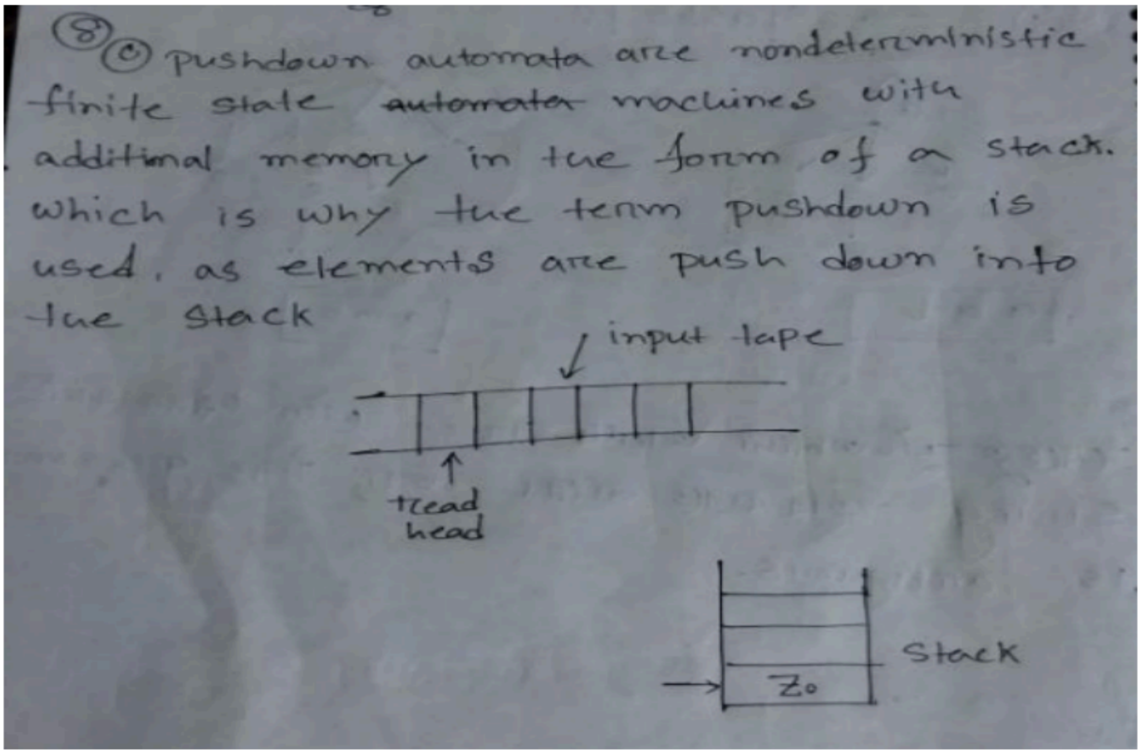
S → ab∈a∈bS

S → ab∈a∈b∈

S → abab



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


(8) (c) Pushdown automata are nondeterministic finite state ~~automata~~ machines with additional memory in the form of a stack. which is why the term pushdown is used, as elements are push down into the stack

Does a Pushdown Automaton (PDA) have memory?

Answer: Yes.

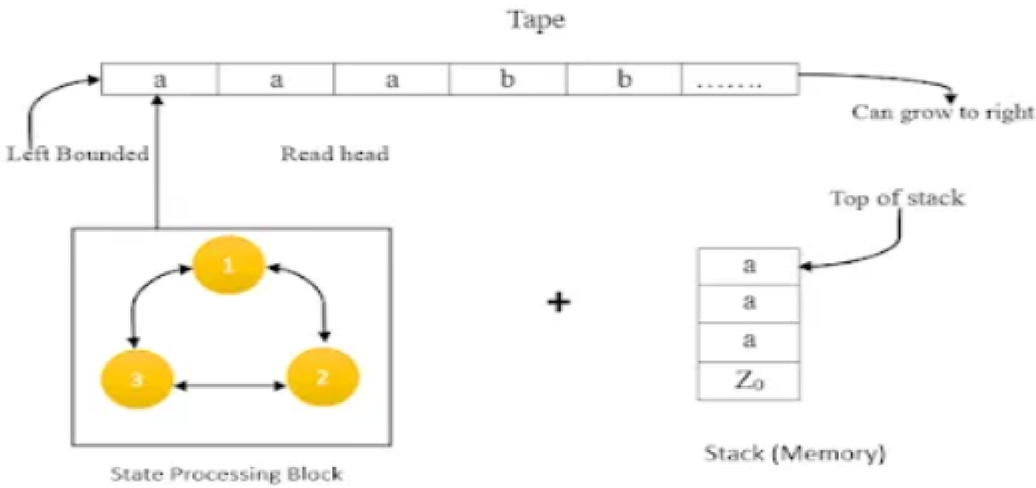
1. **PDAs have a stack.**
    - A stack is like a vertical pile of boxes where you can **put things on top (push)** or **take things off (pop)**.
2. **The stack remembers information.**
    - Unlike a simple finite automaton that “forgets” everything except its current state, a PDA can **remember many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

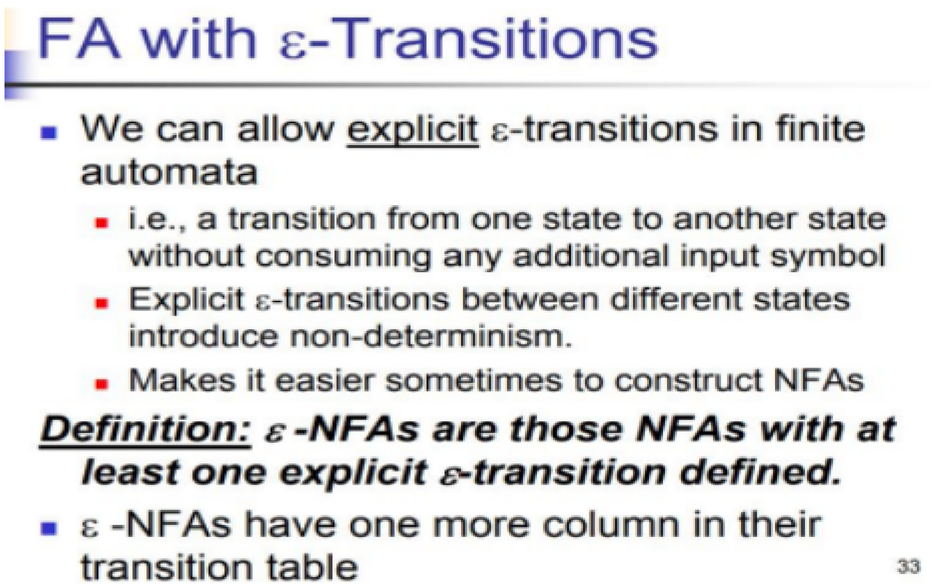
Here, the **stack "remembers" how many as were read**, which is why PDA has memory.

✅ **Simple takeaway:**

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



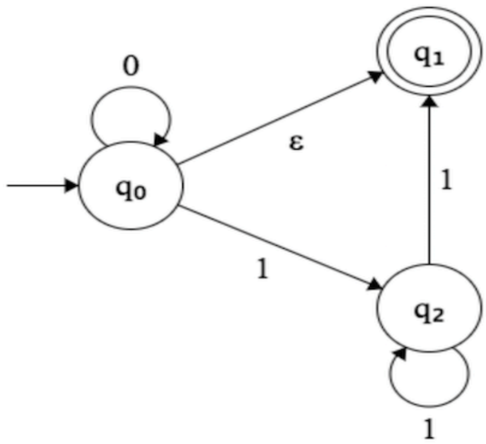
2.a) Why explicit **epsilon-transitions** in finite automata is important? (2)



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.

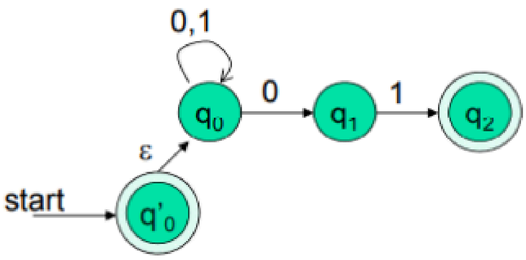
States

- q`0`: Start state
- q1: Saw a `0` that **might be second-last symbol**
- q2: Saw `01` → Accepting state
- Accepting state: q`0` (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`.

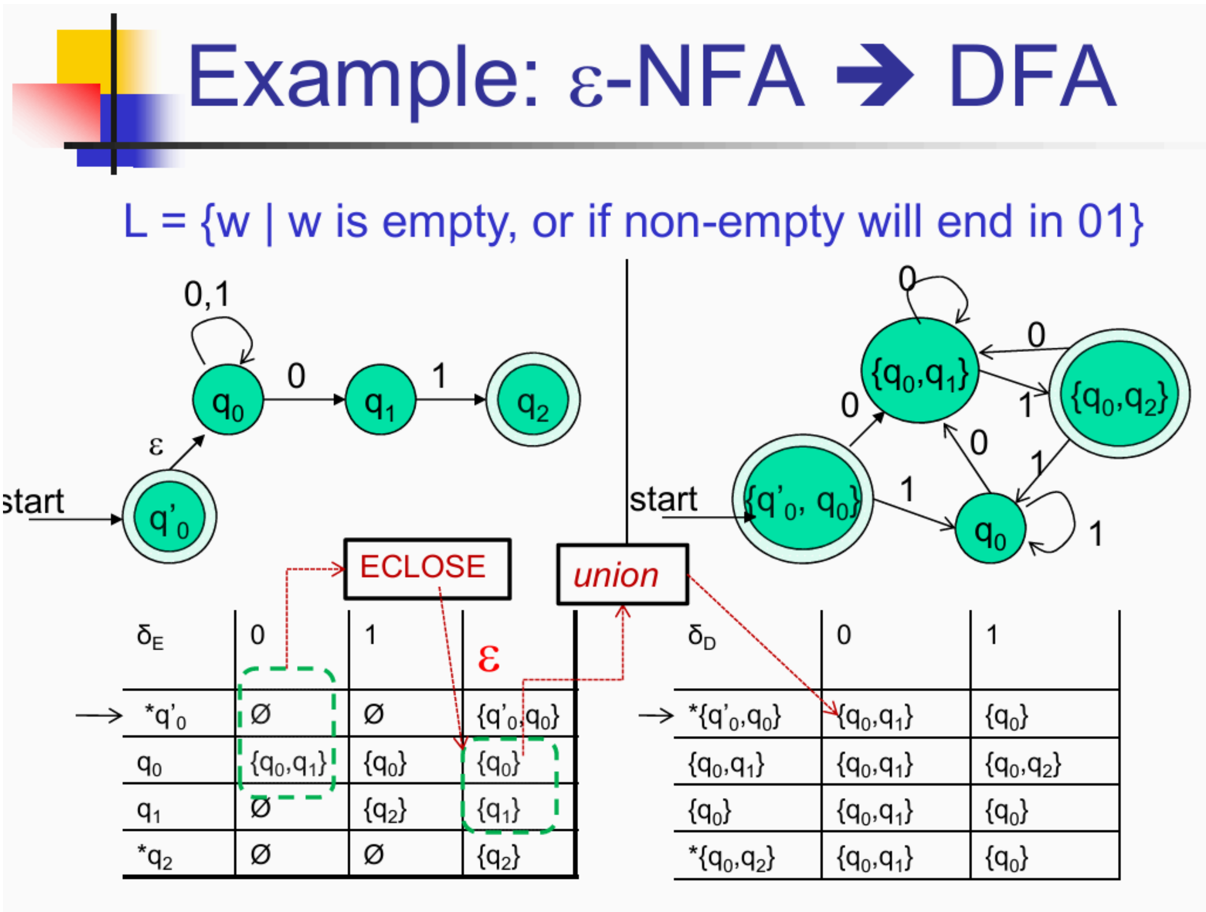
L = {w | w is empty, or if non-empty will end in 01}



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}		
…		

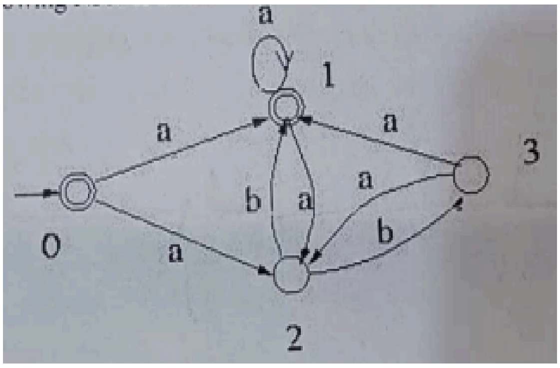
c) Convert **epsilon-NFA** to **DFA** based on Question 6(b).



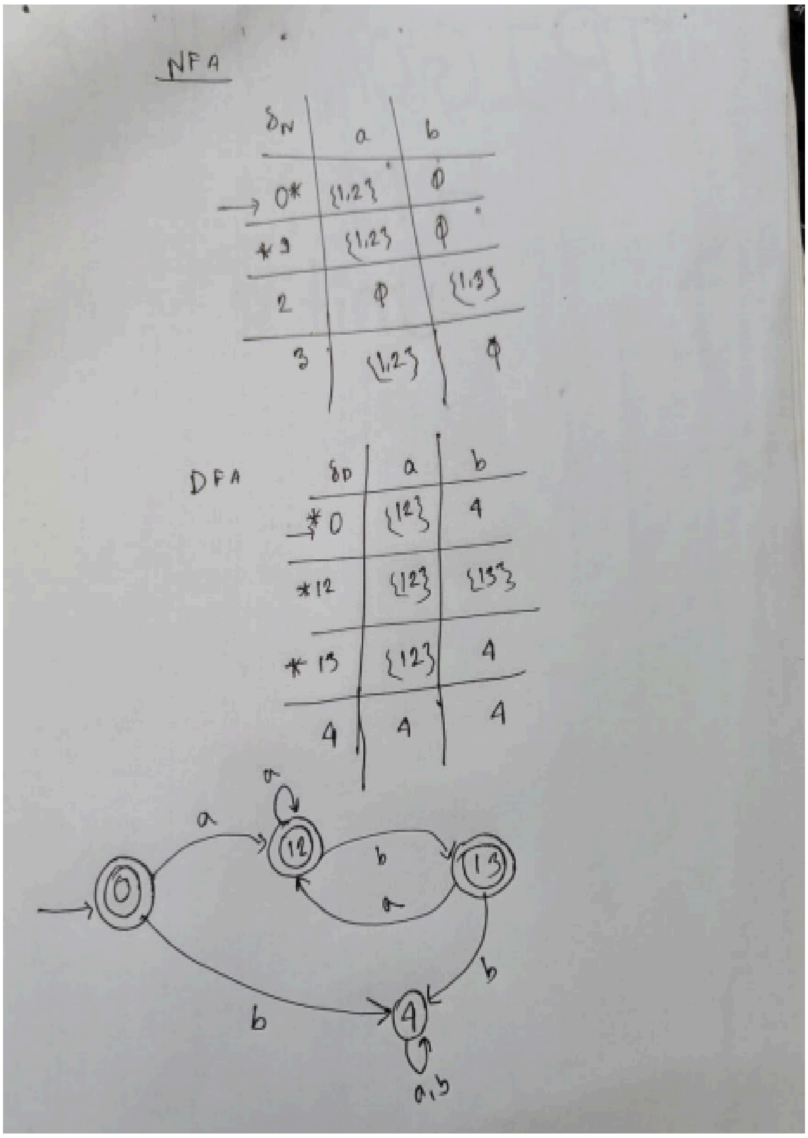
3.a) Differentiate between Finite State and Turing Machines.

Topic	Finite State Machine (FSM)	Turing Machine (TM)
1. Memory	Has *finite* memory (only states)	Has *infinite* memory (infinite tape)
2. Computing Power	Recognizes *Regular Languages* only	Recognizes *Recursively Enumerable Languages* (most powerful model)
3. Tape/Input Handling	Reads input once, cannot modify input	Can read, write, and move head left/right on tape
4. Structure	Only states and transitions	Control unit + infinite tape + read/write head
5. Acceptance Capability	Limited computation	Can perform any algorithmic computation
6. Determinism	Can be deterministic (DFA) or nondeterministic (NFA)	Usually deterministic (but NTM exists)
7. Usage	Lexical analysis, pattern matching, simple systems	General computation model, algorithm simulation, computability theory
8. Complexity	Simple to design	More complex to design

b)Convert the following NFA to DFA



Sol:



NFA

δ_N	a	b
→ 0*	{1,2}	φ
*1	{1,2}	φ
2	φ	{1,3}
3	{1,2}	φ

DFA

δ_D	a	b
→ *0	{12}	4
*12	{12}	{13}
*13	{12}	4
4	4	4

c) How a DFA processes strings?

⭐ Steps of DFA Processing

1. Start in the initial state

The DFA always begins in a special state called the **start state (q_0)**.

2. Read the input string from left to right

The DFA reads the string **one symbol at a time**.

3. Move to the next state using the transition function

For each symbol, the DFA applies its transition function:

$$\delta(\text{current state}, \text{input symbol}) = \text{next state}$$

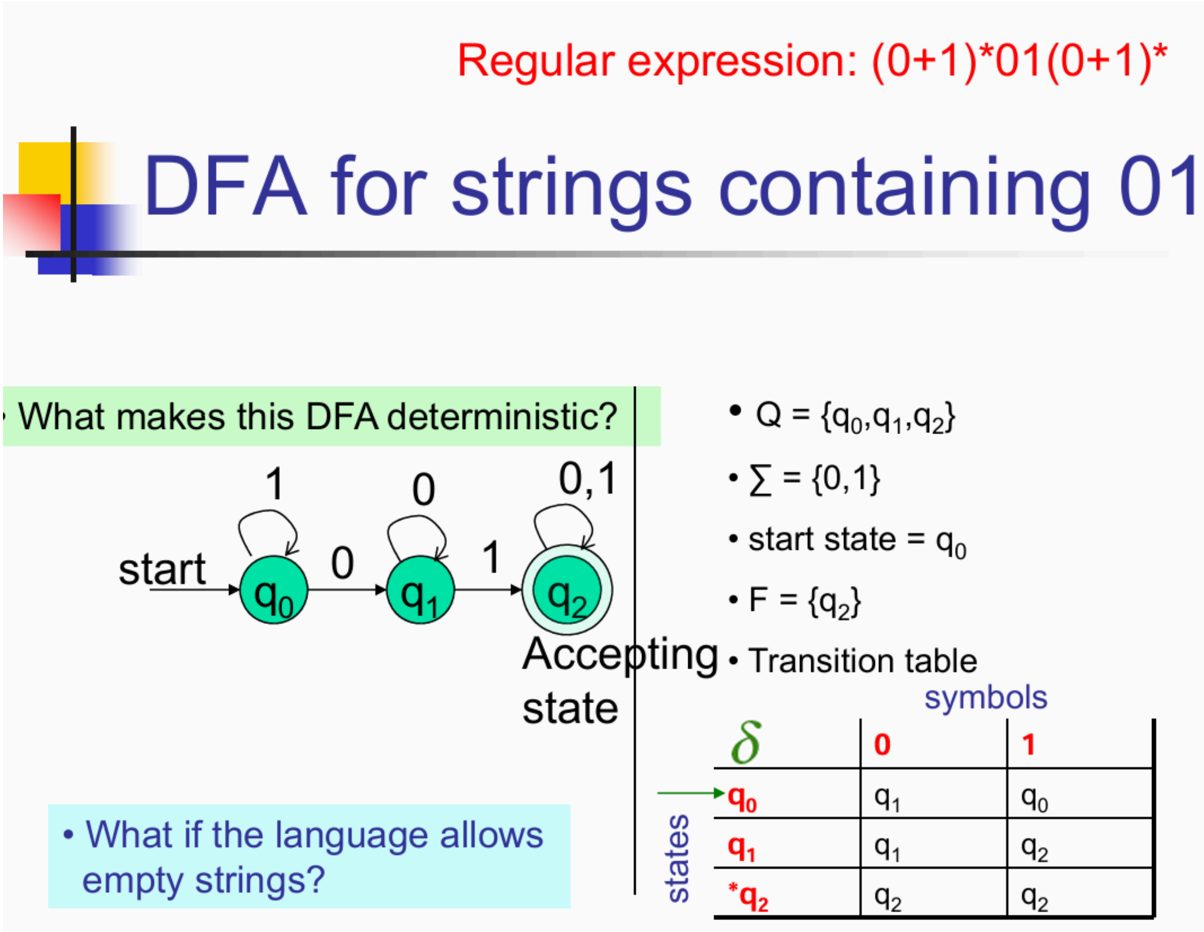
Because it is deterministic, **there is exactly one possible next state** for each (state, symbol) pair.

4. Continue until the entire string is read

The DFA keeps moving between states until **all characters** of the input have been processed.

5. Acceptance or Rejection

- If the DFA ends in an **accepting (final) state**, the string is **Accepted**.
- If it ends in a **non-final state**, the string is **Reject**.



4.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: $M=(Q,\Sigma,\Gamma,\delta,q0,Z0,F)$

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.**

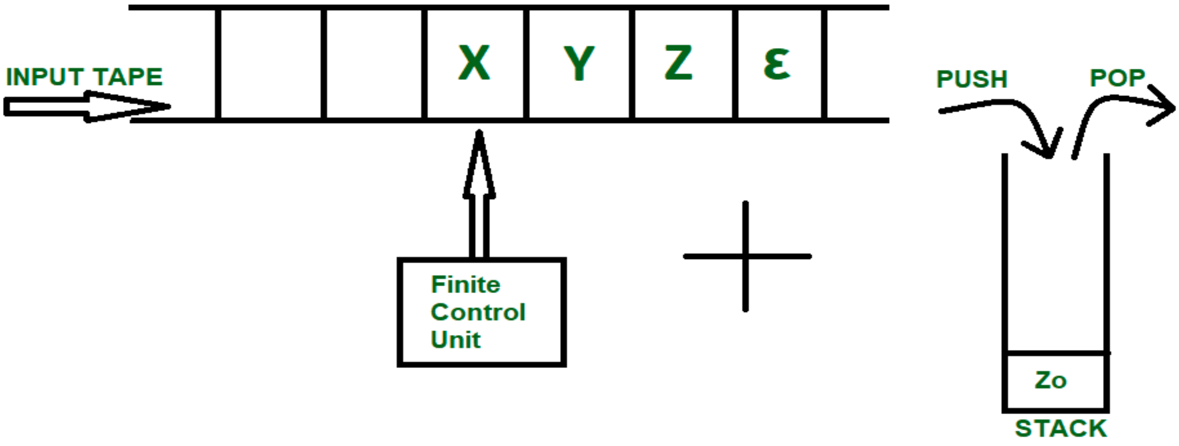


Diagram (conceptually):
Input Tape → a b b a
Stack → Z0 ↓
States → q0, q1, qf

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.

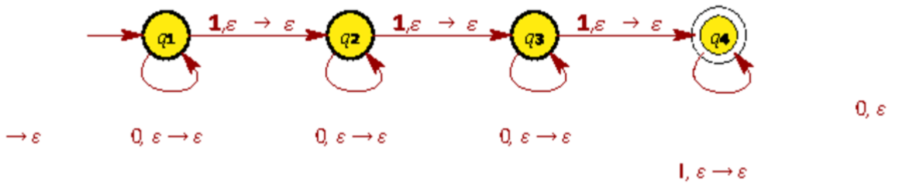
b) Give pushdown automata that recognize the following languages: (5)

- (A) $A = \{w \in \{0,1\}^* \mid w$ contains at least three 1s $\}$
- (B) $B = \{w \in \{0,1\}^* \mid w = w^R$ and the length of w is **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 $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$ is defined by

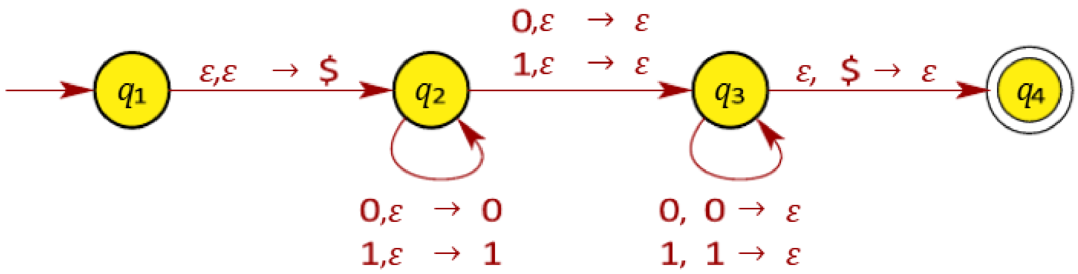
Input:	0			1			ε		
Stack:	0	1	ε	0	1	ε	0	1	ε
q_1			$\{(q_1,\varepsilon)\}$			$\{(q_2,\varepsilon)\}$			
q_2			$\{(q_2,\varepsilon)\}$			$\{(q_3,\varepsilon)\}$			
q_3			$\{(q_3,\varepsilon)\}$			$\{(q_4,\varepsilon)\}$			
q_4			$\{(q_4,\varepsilon)\}$			$\{(q_4,\varepsilon)\}$			

Blank entries are ∅.

- 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≥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

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 $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$ 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.

5.

4. a) Let, $\Sigma = \{0, 1\}$
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.

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**:

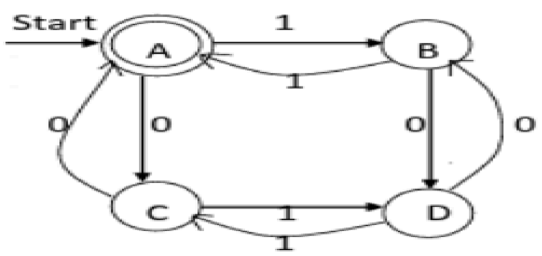
State	Meaning
q_0	Even 0s, Even 1s
q_1	Odd 0s, Even 1s
q_2	Even 0s, Odd 1s
q_3	Odd 0s, Odd 1s

State Transitions

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

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.

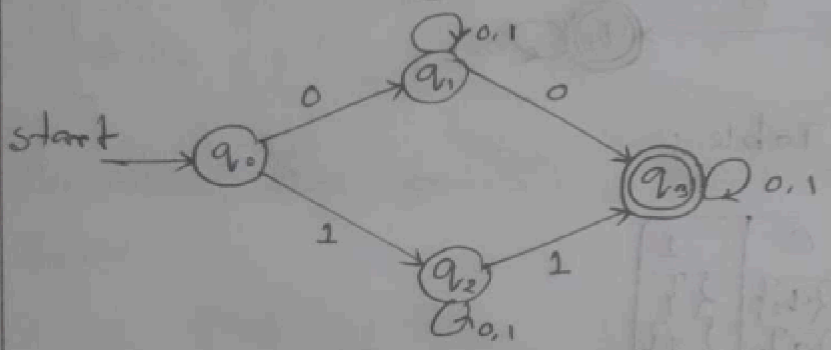
States:

- `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)

Homework 6

NFA for a string where the first symbol present somewhere later at least once:

Transition Diagram:



Transition Table:

δ	0	1
$\rightarrow q_0$	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_1, q_3\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2, q_3\}$
$* q_3$	$\{q_3\}$	$\{q_3\}$

a) Prove that the following language is either regular or not.

A= { w w w | w ∈ {a, b} * }

✅ Proof (Using Pumping Lemma)

Assume, for contradiction, that **A is regular**.

Then by the pumping lemma, there exists a pumping length **p** such that any string **s ∈ A** with length ≥ p can be pumped.

Step 1: Choose a suitable string

Let

$$w = a^p$$

Then pick a string from A:

$$s = www = a^p a^p a^p = a^{3p}$$

This string is clearly in A.

Step 2: Pumping lemma decomposition

According to the lemma,

$$s = xyz$$

with the conditions:

1. $|xy| \leq p$
2. $|y| \geq 1$
3. For all $i \geq 0$, the string

$$xy^i z$$

must still be in A (if the language is regular).

Step 3: Analyze the structure of x, y, z

Since $|xy| \leq p$, both **x and y are inside the first block of a's**:

$$xyz = a^p a^p a^p$$

Let:

- $x = a^k$
- $y = a^m$ (where $m \geq 1$)
- $z = a^{3p-k-m}$

Step 4: Pumping (take i = 0)

Pump down:

$$xy^0z = xz = a^{3p-m}$$

This string has **fewer than p a's in the first block**.

So the new string **cannot** be divided into three **equal** blocks.

Original structure:

$$a^p \ a^p \ a^p$$

Pumped-down structure:

$$a^{p-m} \quad a^p \quad a^p$$

Now the 3 parts are not equal.

Thus:

$$xy^0z \notin A$$

Step 5: Contradiction

The pumping lemma says the pumped string must stay in A, but it **does not**.

Therefore our assumption that **A is regular** is false.

b) Prove that if we add a finite set of strings to a regular language, the result is a regular language.

If **L** is a regular language and **F** is a finite set of strings, then

$$L \cup F$$

is also a **regular language**.

⭐ Proof 1 (Using Closure Properties)

This is the simplest and cleanest proof.

Step 1: Regular languages are closed under union

One of the fundamental properties of regular languages is:

$$\text{If } L_1 \text{ and } L_2 \text{ are regular, then } L_1 \cup L_2 \text{ is regular.}$$

Step 2: Every finite set of strings is regular

A finite set of strings is regular because:

- For each string s, the language $\{s\}$ can be recognized by a simple DFA.
- A finite union of these is also regular.

So a finite set

$$F = \{s_1, s_2, \ldots, s_k\}$$

is regular, because it can be described by the regular expression:

$$s_1 \mid s_2 \mid \cdots \mid s_k$$

Step 3: Use closure under union

Since

- L is regular, and
- F is regular,

their union

$$L \cup F$$

must also be regular.

⭐ Proof 2 (Constructive DFA-Based Proof)

(Use this if examiner expects a construction.)

Let:

- L be a regular language with DFA

$$M = (Q, \Sigma, \delta, q_0, F)$$

- $F = \{s_1, s_2, \ldots, s_k\}$ be a finite set.

Construction idea:

For each string s_i:

- Build a small DFA M_i that accepts **only** s_i.
- Take the union of **all DFAs**:

$$M \cup M_1 \cup M_2 \cup \cdots \cup M_k$$

Since finite automata are closed under union, the final automaton accepts

$$L \cup \{s_1, s_2, \ldots, s_k\}$$

Thus the union is regular.

c) Write the closure properties of regular languages.

A closure property is a characteristic of a class of languages (such as regular, context-free, etc.) where applying a specific operation (like union, intersection, concatenation, etc.) to languages within that class results in a language that is also within the same class.

Regular languages are **closed** under several important operations — meaning that if you apply these operations to regular languages, the result is **also a regular language**.

Here's a summary of the main closure properties 👇

Operation	Description	Result
Union	If L_1 and L_2 are regular, then $L_1 \cup L_2$ is also regular.	✅ Regular
Intersection	If L_1 and L_2 are regular, then $L_1 \cap L_2$ is regular.	✅ Regular
Complement	If L is regular, then its complement $\overline{L}$ is also regular.	✅ Regular
Difference	If L_1 and L_2 are regular, then $L_1 - L_2$ is regular.	✅ Regular
Concatenation	If L_1 and L_2 are regular, then $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$ is regular.	✅ Regular
Kleene Star	If L is regular, then $L^* = \{x_1x_2...x_n \mid n \geq 0, x_i \in L\}$ is regular.	✅ Regular
Reversal	If L is regular, then the set of all reversed strings L^R is regular.	✅ Regular
Homomorphism	If L is regular and h is a homomorphism, then $h(L)$ is regular.	✅ Regular
Inverse Homomorphism	If L is regular and h is a homomorphism, then $h^{-1}(L)$ is regular.	✅ Regular

Operation	Closed?	Description
Union ($L_1 \cup L_2$)	✅ Yes	Combines all strings from both languages. If L_1 and L_2 are regular, the union is regular.
Intersection ($L_1 \cap L_2$)	✅ Yes	Contains only strings common to both languages. Regular languages are closed under intersection.
Set Difference ($L_1 - L_2$)	✅ Yes	Contains strings in L_1 but not in L_2. Regular languages are closed under difference.

*Complement (¬L or $\Sigma - L$)**	✅ Yes	Contains all strings over the alphabet not in L. Complement of a regular language is regular.
Concatenation (L_1L_2)	✅ Yes	All strings formed by taking a string from L_1 followed by a string from L_2.
Kleene Star (L*)	✅ Yes	All strings formed by concatenating zero or more strings from L.
Kleene Plus (L^+)	✅ Yes	All strings formed by concatenating one or more strings from L ($L^+ = L \cdot L^*$).
Reversal (L^R)	✅ Yes	All strings of L reversed. Regular languages are closed under reversal.
Homomorphism (h(L))	✅ Yes	Replace symbols in strings of L according to a homomorphism h. The result is regular.
Inverse Homomorphism ($h^{-1}(L)$)	✅ Yes	The set of strings mapped into L under a homomorphism h. Still regular.
Substitution	✅ Yes	Replace symbols in L with strings from regular languages; result is regular.
Intersection with a Regular Language	✅ Yes	Intersecting any language with a regular language preserves regularity if the first language is regular.
Union with a Regular Language	✅ Yes	Union with a regular language preserves regularity.

Subset Operation	❌ No	Determining if a language is a subset of another does not necessarily yield a regular language.
Infinite Union	❌ No	Infinite union of regular languages may not be regular.

💡 **In short:**

> **The class of regular languages is closed under all standard language operations.**

This is one of the reasons **regular languages** are so powerful and useful in automata theory and compiler design.

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

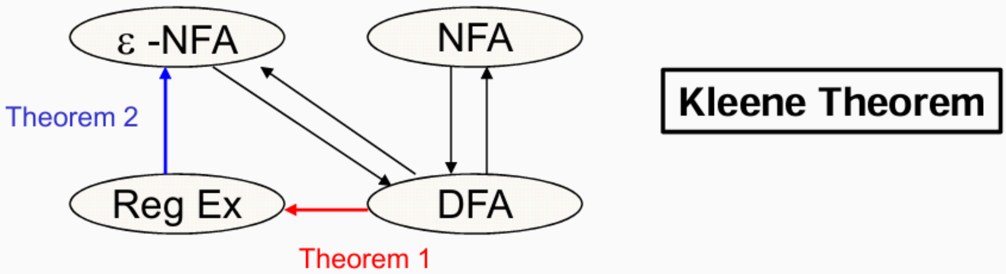
Relation between Regular Expressions and Finite Automata (Short & Clear)

- **Regular Expressions (RE)** and **Finite Automata (FA)** describe the **same class of languages**, called **Regular Languages**.
- For every RE, there exists an equivalent FA that accepts the same language.
- For every FA (DFA/NFA), there exists an equivalent RE describing its language.
- Thus, **RE and FA are equivalent and interchangeable** in expressive power.

Finite Automata (FA) & Regular Expressions (Reg Ex)

- 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)*

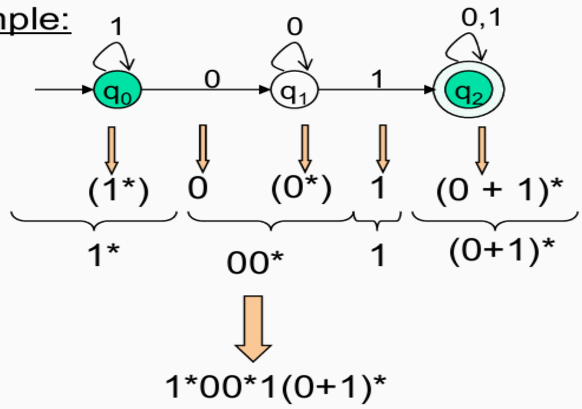
ɔok



DFA to RE construction

Informally, trace all distinct paths (traversing cycles only once) from the start state to *each of the* final states and enumerate all the expressions along the way

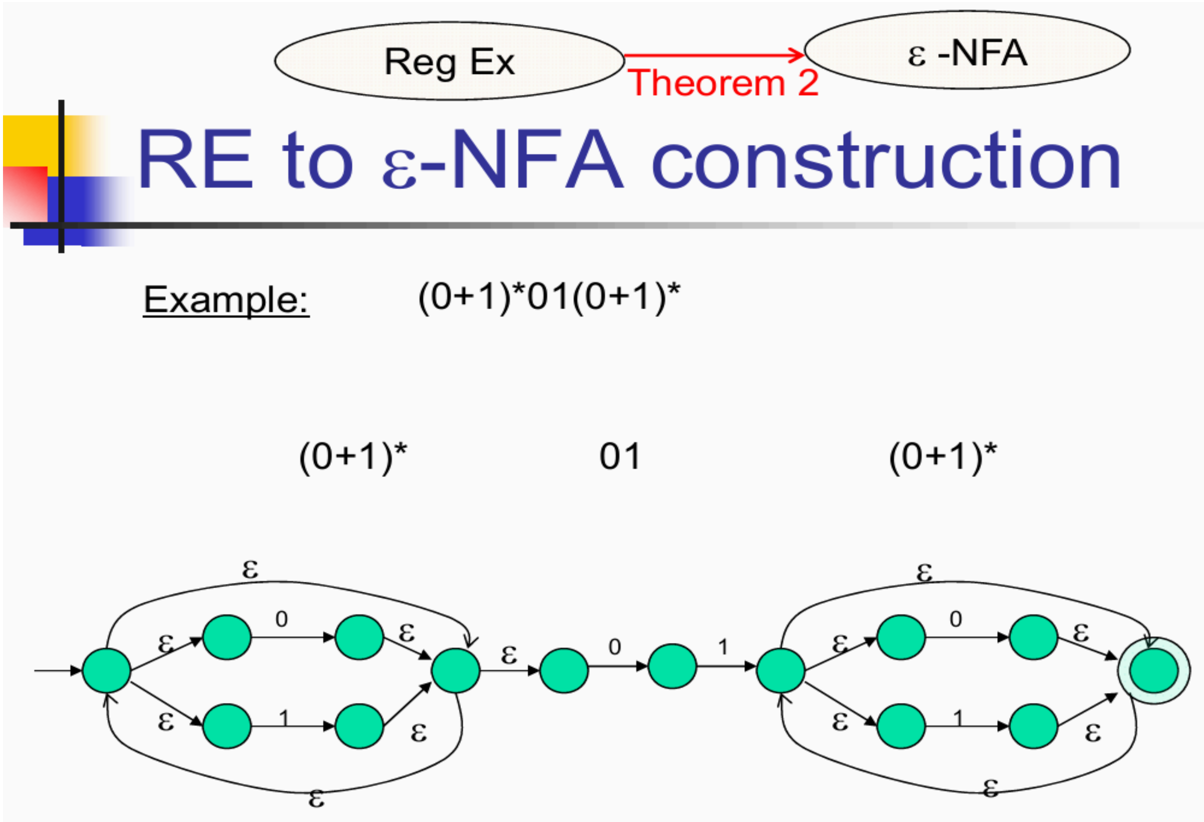
<u>Example:</u>



Q) What is the language

11

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



8.b) Consider the regular expression (a(cd)*b)*

(i) Find a string over {a, b, c, d}^4 which matches the expression.

(ii) Find a string over {a, b, c, d}^4 which does not match the expression

(i) A string over `{a, b, c, d}^4` that matches the regular expression `(a(cd)*b)*` is **acdb**.

(ii) A string over `{a, b, c, d}^4` that does not match the regular expression `(a(cd)*b)*` is **abcd**.

Regex: (a(cd)∗b)∗. Each block ba(cd)^k has length 2+2k (even, ≥2). A length-4 string can be either one block with k=1 or two blocks with k=0

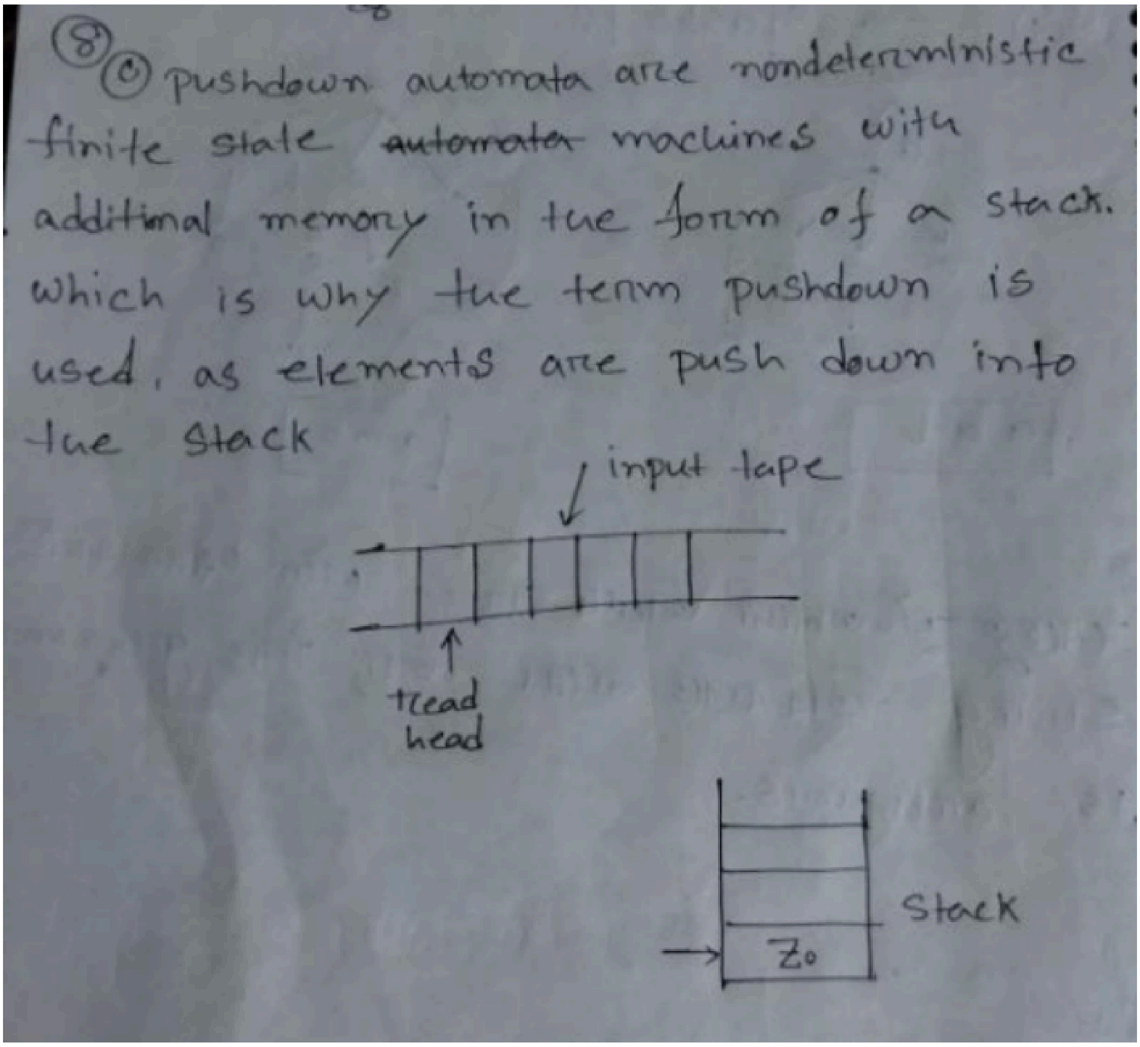
(i) **Matches:** acdb

Reason: acdb = a · (cd) · b (one block with k=1).

(ii) **Does not match:** aabb

Reason: any block must begin with a and end with b and the middle must be repetitions of cd; aabb cannot be decomposed into such blocks (neither aabb = a(cd)^k b nor as concatenation of ab-style blocks because aa / bb are invalid).

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



Does a Pushdown Automaton (PDA) have memory?

Answer: Yes.

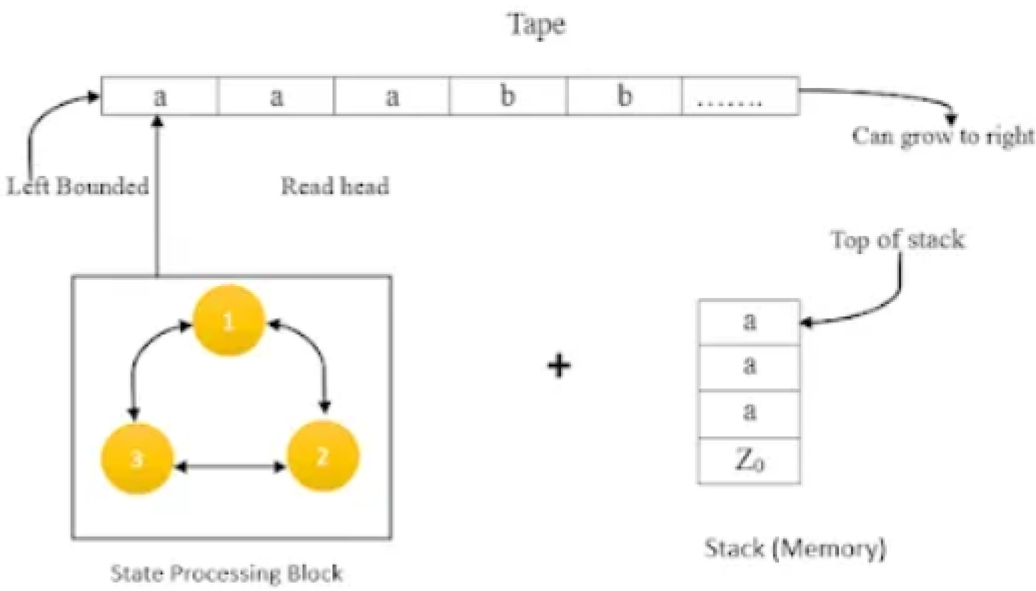
1. **PDAs have a stack.**
    - A stack is like a vertical pile of boxes where you can **put things on top (push)** or **take things off (pop)**.
2. **The stack remembers information.**
    - Unlike a simple finite automaton that “forgets” everything except its current state, a PDA can **remember many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



a) Find DFA's which accepts the following languages:

(i) Strings over {a, b} ending in aa.

(ii) String over {a, b} containing three consecutive a's (that is, contains the substring aaa)

(iii) All strings over {a, b} where each string of length 5 contains at least two a's.