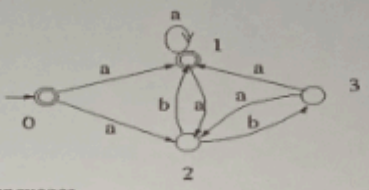
Department of Computer Science and Engineering
University of Barisal
3rd Year 2nd Semester Final Examination, 2024

Course Code: CSE-3203 | **Course Title: Theory of Computation**

Total time: 3.00 hours | **Total marks: 60.0**

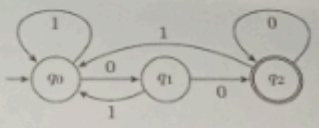
(Note: Answer any five set of questions from the followings)

1. a) Define Turing Machine. Explain pushdown automaton. 4
   b) Prove that every language accepted by a multitap TM is recursively enumerable. 6
   c) What do you know about context-free grammars? 2
2. a) Define with examples: Alphabet, String, Language 6
   b) Give a regular expression for the following language B over the alphabet {a, b}. 4
   B = {w | w does not contain the substring aaa}.
   c) When is an expression called regular expression? 2
3. a) Use the pumping lemma to prove that the language $A = \{ 0^{2n} 1^{3n} 0^{n} \mid n \geq 0 \}$ is not context free. 3
   b) Convert the following NFA to DFA. 6



   c) State the Pumping Lemma for regular languages. 3

4. a) Summarize the principal closure properties for regular languages. 3
   b) Convert DFA's to regular expressions by eliminating states (using an arbitrary example). 5
   c) Prove that if L is a regular language over alphabet Σ, then $\bar{L} = \Sigma^* - L$ is also a regular language. 4
5. a) Construct a context-free grammar for the following DFA: 5



   b) Show that the grammar ({S}, {a, b}, R, S) with rules R = S → aS | aSbS | ∈ is ambiguous. 4
   c) Explain inverse homomorphism. 3
6. a) Elaborate the three operations on languages that the operations of regular expression represent. 4
   b) Prove that if we add a finite set of strings to a regular language, the result is a regular language. 5
   c) Write the closure properties of regular languages. 3

7. a) Prove that the following are not regular languages: 2+2
   (i) The set of strings of 0's and 1's beginning with 1, such that when interpreted as an integer, that integer is a prime.
   (ii) The set of strings of the form $0^i 1^j$ such that the greatest common divisor of i and j is 1.
   b) State the properties of a parse tree. Construct a parse tree for the following string: 4
   $S \rightarrow SS+ \mid SS* \mid a$
   c) Prove that every language defined by a regular expression is also defined by a finite automata. 4

8. a) State formal definition of a finite automata. Give an example. 4
   b) Consider the regular expression $(a(cd)^*b)^*$. 2+2
      (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.
   c) Build an ε-NFA for the following language: 4

1. a) answer:

Sure, Emon 👍
Let's go step by step —

---

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:$M=(Q,\Sigma,\Gamma,\delta,q0,qaccept,qreject)$

Where:

Symbol	Meaning
Q	Finite set of states
Σ (Sigma)	Input alphabet (symbols allowed in input)
Γ (Gamma)	Tape alphabet (includes Σ plus blank symbol '_')
δ (delta)	Transition function: $\delta(q, X) \rightarrow (p, Y, D)$ where $D \in \{L, R\}$ (Left or Right movement)
q_0	Start state
q_accept	Accepting (final) state

q_reject	Rejecting (final) 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**.

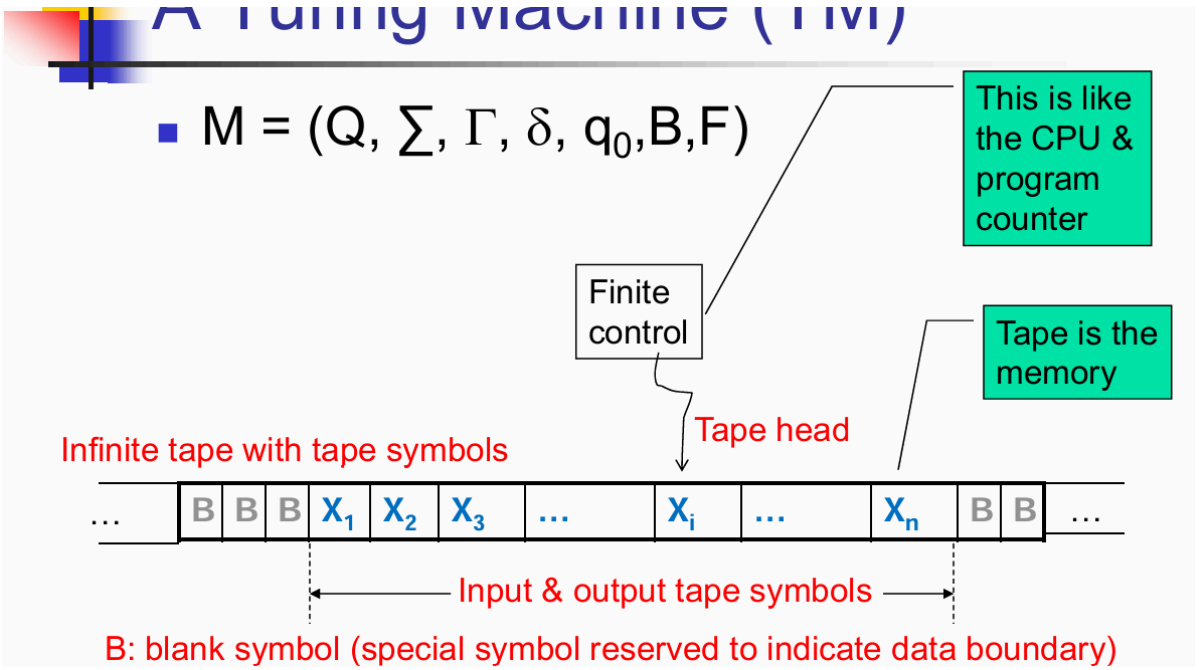


Diagram (conceptually):

```
... | 1 | 0 | 1 | 1 | _ | _ | ...
      ↑
   Tape Head
```

At each step:

$$\delta(q, X) = (p, Y, D)$$

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.

---

2. Pushdown Automaton (PDA): Explanation

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.**

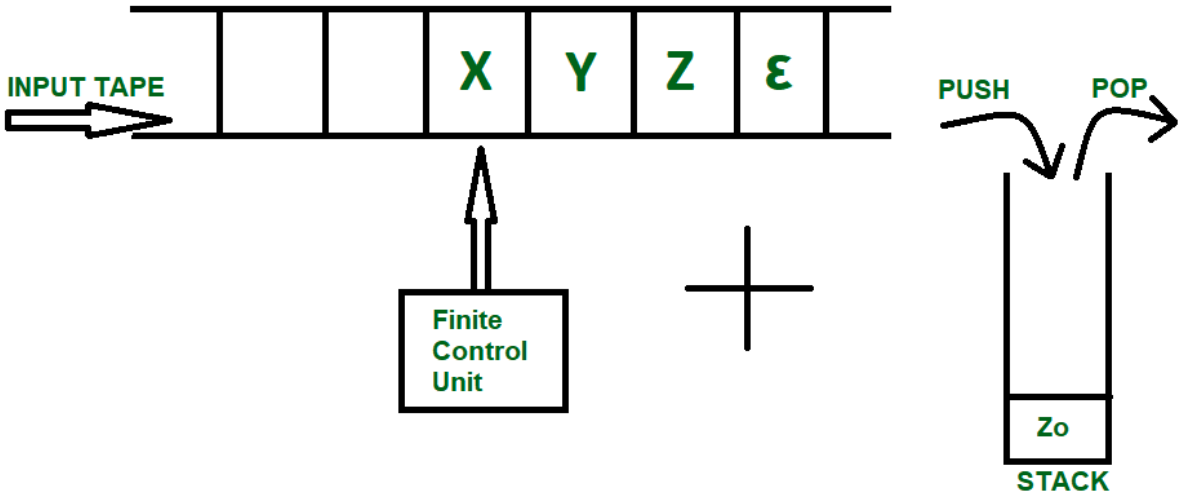


Diagram (conceptually):
Input Tape → a b b a
Stack → Z0 ↓
States → q0, q1, qf

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.

Comparison Summary:

Feature	Turing Machine	Pushdown Automaton
Memory	Infinite Tape	Stack
Power	Recognizes **Recursively Enumerable Languages**	Recognizes **Context-Free Languages**
Reversibility	Can move left or right	Reads input one-way (left to right)

Acceptance	Final state (or halting)	Final state or empty stack
Example	Simulates any algorithm	Recognizes balanced parentheses, $a^n b^n$

B.Answer: Prove that every language accepted by a multitap TM is recursively enumerable.

Proof: Suppose language L is accepted by a k-tape TM M. We simulate M with a one-tape TM N whose tape we think of as having 2k tracks. Half these tracks hold the tapes of M, and the other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located. Figure assumes K = 2. The second and fourth tracks hold the contents of the first and second tapes of M, track 1 holds the position of the head of tape 1 and track 3 holds the position of the second tape head.

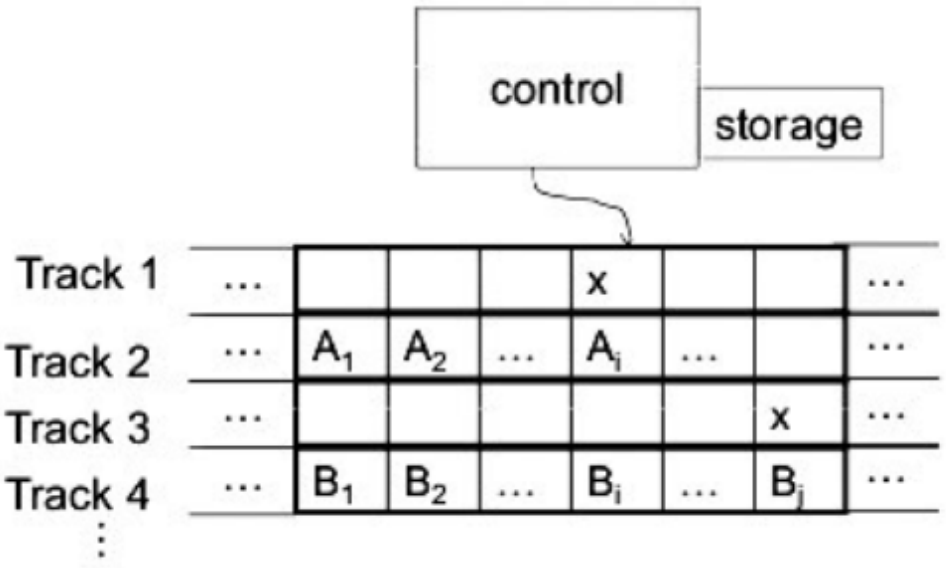


Figure: Simulation of a two-tape Turing machine by a one-tape Turing machine.

To simulate a move of M, N's head must visit the K head markers. So that N not get lost, it must remember how many head markers are to its left at all times; that count is stored as a component of N's finite control. After visiting each head marker and storing the scanned symbol in a component of its finite control, N knows what tape symbols are being scanned by each of M's heads. N also knows the state of M, which it stores in N's own finite control. Thus, N knows what move M will make.
N now revisits each of the head markers on its tape, changes the symbol in the track representing the corresponding tapes of M, and moves the head markers left or right, if necessary. Finally, N changes the state of M as recorded in its own finite control. At this point, N has simulated one move of M.

We select as N's accepting states all those states that record M's state as one of the accepting states of M. Thus, whenever the simulated M accepts, N also accepts and M does not accept otherwise.

Multitape TMs ≡ Basic TMs

- Theorem: Every language accepted by a k-tape TM is also accepted by a single-tape TM
- Proof by construction:
  - Construct a single-tape TM with 2k tracks, where each tape of the k-tape TM is simulated by 2 tracks of basic TM
  - k out the 2k tracks simulate the k input tapes
  - The other k out of the 2k tracks keep track of the k tape head positions

C.Answer:What do you know about context-free grammars?

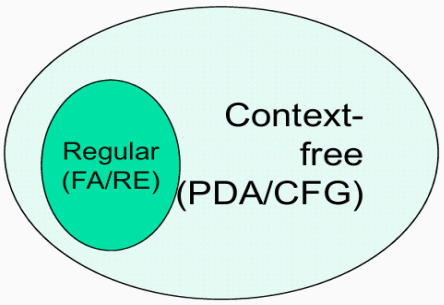
Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

A **CFG** is defined as a 4-tuple:

G=(V,Σ,R,S) Where:

Symbol	Meaning
V	Finite set of **variables** (non-terminal symbols)
Σ	Finite set of **terminals** (alphabet symbols)
R	Finite set of **production rules** of the form A → α
S	Start symbol (one of the variables in V)

- A language class larger than the class of regular languages
- Supports natural, recursive notation called "context-free grammar"
- Applications:
  - Parse trees, compilers
  - XML



In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

4.2 Context-Free Grammars

In this section, we shall introduce context-free grammars to generate a class of languages larger than the class of regular languages over the same alphabet.

Definition 3: A **context-free grammar**(CFG) is denoted by G = (V, T, P, S), where

V is a finite set of variables, S∈ V is the start symbol,

T is a finite set of terminals and T ∩ V = ∅, and

P is a finite set of productions or rewriting rules, each production is of the form:

$A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.

2.
a) Define with examples: Alphabet, String, Language

🧩 a) Definitions with Examples

1️⃣ Alphabet (Σ)

Definition:

An **alphabet** is a **finite, non-empty set of symbols**.

These symbols are the **basic units** from which strings and languages are formed.

Notation:

The alphabet is usually denoted by **Σ** (Greek letter "Sigma").

Example:

- Σ= {0, 1} → binary alphabet
- Σ= {a, b, c} → alphabet of letters
- Σ= {x, y, z, +, -, *, / } → alphabet for arithmetic expressions

---

2️⃣ String (or Word)

Definition:

A **string** is a **finite sequence of symbols** taken from an alphabet.

Notation:

If (Σ= {a, b}):

- Some possible strings:
    - a, b, ab, ba, aab, bba, etc.
- The **empty string** (no symbols) is denoted by **ε (epsilon)** or sometimes **λ**.

Examples:

- Over (Σ = {0,1}):
    - 010, 1110, ε are strings.
- Length of a string:
    - |010| = 3
    - |ε| = 0

3️⃣ Language

Definition:

A **language** is a **set of strings** formed from an alphabet **Σ** that satisfies certain rules or patterns.

In other words,

$L \subseteq \Sigma*$
where **Σ*** is the set of **all possible strings** (including ε) that can be made from Σ.

Examples:

1. If (**Σ** = {a, b})
    - (L_1 = {a, ab, abb}) → a finite language
    - L2={a^nb^n | n≥1}={ab,aabb,aaabbb,...} → an infinite language
2. Over (**Σ** = {0,1})
    - L={w | w has even number of 0s}

Relations Between Them

Concept	Definition	Example
Alphabet (Σ)	Set of symbols	{0, 1}
String (w)	Sequence of symbols	0101
Language (L)	Set of strings	{01, 0011, 000111}

Visual Summary:

Alphabet (Σ): {a, b}
Strings: ε, a, b, ab, ba, aab, abb, ...
Language (L): {ab, aabb, aaabbb, ...}

---

b) Give a regular expression for the following language B over the alphabet (a, b). B={ww does not contain the substring aaa)

We can describe the set of valid strings as the strings built from the alphabet $V = \{a, b\}$ which contain at most one or two consecutive a's.

> These are strings
>
> - starting with zero or more b's:
>
> $$b^*$$
>
> - followed by zero or more occurrences of a or aa each followed by one or more b's:
>
> $$(ab^+|aab^+)^*$$
>
> - and terminating with zero, one or two a's
>
> $$(\varepsilon|a|aa)$$
>
> We obtain
>
> $$b^*(ab^+|aab^+)^*(\varepsilon|a|aa) \tag{1}$$

> **Add-on:** The regular expression (1) generates all valid words in a *unique* manner. In such cases we can use it to derive a generating function
>
> $$A(z) = \sum_{n=0}^{\infty} a_n z^n$$
>
> with a_n giving the number of valid words of length n.

In order to do so all we need to know is the *geometric series expansion* since the ***star*** operator

$$a^* = (\varepsilon|a|a^2|a^3|\cdots) \qquad \textbf{translates to} \qquad 1 + a + a^2 + a^3 + \cdots = \frac{1}{1-a}$$

Accordingly $a^+ = aa^*$ translates to $\frac{a}{1-a}$ and alternatives like $(\varepsilon|a|aa)$ can be written as $1 + a + a^2$.

> We obtain by translating the regular expression in the language of generating functions (and by mixing up somewhat the symbolic to provide some intermediate steps)
>
> $$\begin{aligned} b^*\left(ab^+|aab^+\right)^*(\varepsilon|a|aa) &\longrightarrow \quad \frac{1}{1-z}\left(\frac{z^2}{1-z}\middle|\frac{z^3}{1-z}\right)^*\left(1+z+z^2\right) \\ &\longrightarrow \quad \frac{1}{1-z}\left(\frac{z^2+z^3}{1-z}\right)^*(1+z+z^2) \\ &\longrightarrow \quad \frac{1}{1-z}\cdot\frac{1}{1-\frac{z^2+z^3}{1-z}}(1+z+z^2) \\ &= \frac{1+z+z^2}{1-z-z^2-z^3} \end{aligned}$$

> We conclude: The number of valid words is given by the generating function $A(z)$
>
> $$A(z) = \frac{1+z+z^2}{1-z-z^2-z^3} = 1 + 2z + 4z^2 + 7z^3 + 13z^4 + 24z^5 + 44z^6 + 81z^7 + \cdots$$
>
> The expansion was done with the help of Wolfram Alpha. We see that e.g. the number of valid words of length 5 is 24.

So, out of $2^5 = 32$ binary words of length 5 there are 8 invalid words marked blue in the table below.

aaaaa	*abaaa*	*baaaa*	*bbaaa*
aaaab	*abaab*	*baaab*	*bbaab*
aaaba	*ababa*	*baaba*	*bbaba*
aaabb	*ababb*	*baabb*	*bbabb*
aabaa	*abbaa*	*babaa*	*bbbaa*
aabab	*abbab*	*babab*	*bbbab*
aabba	*abbba*	*babba*	*bbbba*
aabbb	*abbbb*	*babbb*	*bbbbb*

c)When is an expression called regular expression?

An **expression is called a regular expression** when it is used to describe a **set of strings (a language)** that can be defined using specific **rules and operators**.

Formally,
👉 A **regular expression (RE)** is a symbolic notation used to represent **regular languages** — the languages that can be recognized by a **finite automaton**.

💡 In simple terms:

A **regular expression** is a pattern that represents **a collection of strings** over some alphabet.

✅ Definition (Formal):

Let **Σ** be an alphabet (set of symbols). The **regular expressions over Σ** are defined as follows:

1. **Basic Regular Expressions:**
    - **ϕ (empty set)** → denotes the language containing no strings.
    - **ε (epsilon)** → denotes the language containing the empty string `{ε}`.
    - **a**, where `a ∈ Σ` → denotes the language `{a}` (a single symbol).
2. **Compound Regular Expressions:**
   If `r₁` and `r₂` are regular expressions, then:
    - **(r_1 | r_2)** → represents **union** (alternation): strings from either r_1 or r_2.
    - **(r_1r_2)** → represents **concatenation**: strings formed by joining a string from r_1 with one from r_2.
    - **(r_1*)** → represents **Kleene star**: zero or more repetitions of strings from r_1.

🧠 Example:

Let Σ = {a, b}

Regular Expression	Describes the Language
`a`	{ "a" }
`` `a ``	`b`
`ab`	{ "ab" }
`a*`	{ "", "a", "aa", "aaa", ... }
`` `(a ``	`b)*` ``

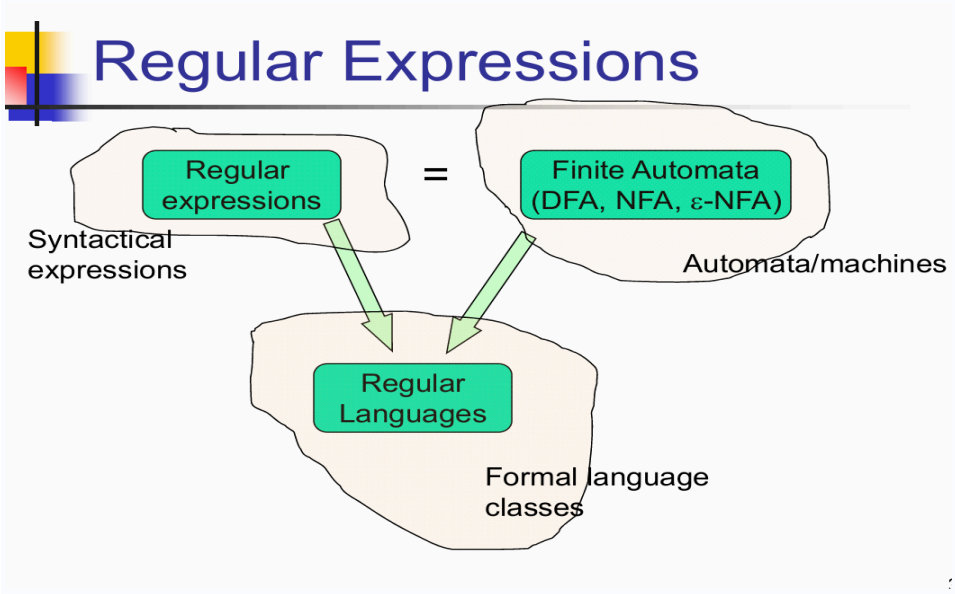
📘 In summary:

An **expression is called a regular expression** if it is built using:

- **symbols** from the alphabet,

- **operators** (union |, concatenation, and Kleene star *), and

- possibly **parentheses** for grouping —
  and it defines a **regular language** that can be recognized by a **finite automaton**.



3.a)Use the pumping lemma to prove that the language A={0^2n^3n 0^n| n≥0} is not context free.

- **Regular-language pumping lemma:** write $w = xyz$ with $|xy| \leq N,\ |y| > 0$ and $\forall k \geq 0\ xy^k z \in L$
- **Context-free pumping lemma (used here):** write $s = uvwxy$ with $|vwx| \leq N,\ |vx| > 0$ and $\forall k \geq 0\ uv^k wx^k y \in L$.

We want to show $A = \{0^{2n}1^{3n}0^n \mid n \geq 0\}$ is **not context-free**, so use the **CFL** version.

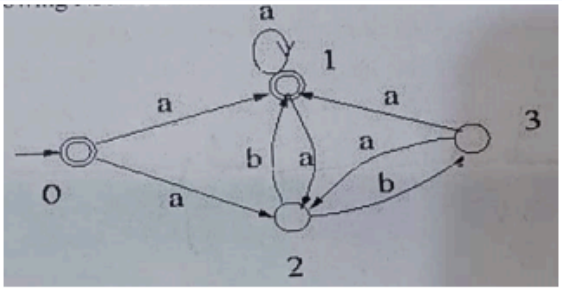
Short proof (CFL pumping lemma):

1. Assume A is context-free. Let pumping length be N.
2. Choose $s = 0^{2N}1^{3N}0^N \in A$ (so $|s| \geq N$). By the lemma write $s = uvwxy$ with
   - $|vwx| \leq N$,
   - $|vx| > 0$,
   - $\forall k \geq 0,\ uv^k wx^k y \in A$.
3. Because $|vwx| \leq N$, the substring vwx lies entirely inside one block (the first 0^{2N}, or the 1^{3N}, or the last 0^N) or crosses one boundary between two adjacent blocks.
4. In every possible placement pumping (take $k = 0$) changes the length(s) of at most the first or middle or last block(s) while leaving at least one block unchanged. But every string in A must have block lengths in ratio $2:3:1$ (first : middle : last). After pumping that ratio is violated (e.g. final block stays N while middle becomes $3N - b$ with $b \geq 1$, ↓ st is no longer twice the last, etc.).
5. Hence $uv^0wx^0y \notin A$, contradicting the lemma.

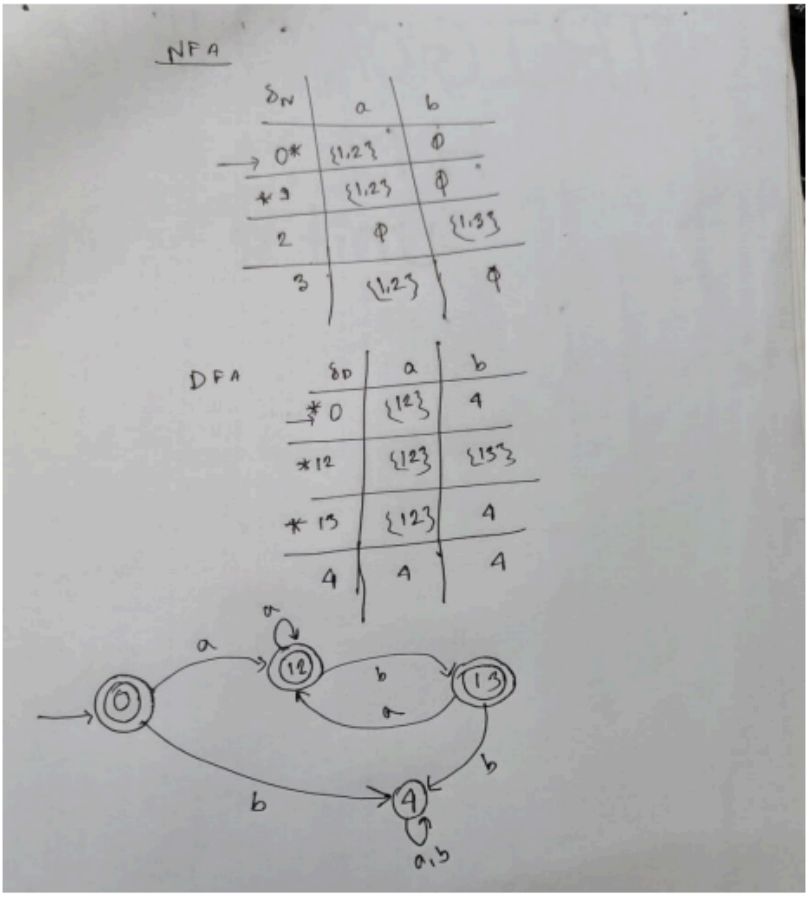
Therefore A is **not** context-free.

Regular pumping lemma (the w=xyz), note immediately that failing the regular lemma only shows "not regular" — it does **not** prove not context-free.That's why use CFL

b)Convert the following NFA to DFA



Sol:



c) State the Pumping Lemma for regular languages.

Then, there exists a constant $N > 0$ (called the *pumping length*) such that **every string** $w \in A$ with $|w| \geq N$ can be written as

$$w = xyz$$

such that the following three conditions hold:

1. $y \neq \varepsilon$ (the string y is not empty),
2. $|xy| \leq N$,
3. **For all** $k \geq 0$, $xy^k z \in A$.

Meaning:

If a language is regular, then any sufficiently long string in the language can have a middle part y that can be **"pumped"** — i.e., repeated any number of times (including 0) — and the resulting string will **still belong** to the language.

Use:

The **pumping lemma** is often used to **prove that a la** **uage is *not* regular** by showing that **no such decomposition** $w = xyz$ can satisfy all three conditions.

Pumping Lemma provides a method to prove that certain languages are not regular. The Pumping Lemma states that for any regular language, there exists a length such that any string longer than this length can be divided into three parts, and by repeating or removing the middle part, the resulting string will also be in the language.

Extra:

Example 1

Prove that L = {a^i^2 | i ≥ 1} is not regular.

Assume the set L is regular. Let n be the number of states of the FA accepting the set L.

Let w = a^{n^2}. The length of w is n^2, which is greater than n, the number of states of the FA accepting L. By using the Pumping Lemma, we can write w = xyz with $|xy| \leq n$ and $|y| > 0$.

Take **i = 2**, so the string will become $\mathbf{xy^2z}$.

$$|\mathbf{xy^2z}| = |\mathbf{x}| + 2|\mathbf{y}| + |\mathbf{z}|$$

$$= |\mathbf{x}| + |\mathbf{y}| + |\mathbf{z}| + |\mathbf{y}|$$

$$= \mathbf{n^2} + |\mathbf{y}|$$

Since |xy| n, |y| n, therefore $|xy^2z|$ $n^2 + n$

From the previous derivations, we can write −

$$\mathbf{n^2} < |\mathbf{xy^2z}| \leq \mathbf{n^2} + \mathbf{n} < (\mathbf{n} + \mathbf{1})^2$$

Hence, $|xy^2z|$ lies between n^2 and $(n + 1)^2$. They are the squares of two consecutive positive integers. In between the squares of two consecutive positive integers, no square o a positive integer belongs.

But a^{i^2}, where $i \geq 1$, is a perfect square of an integer. So, the string derived from it, i.e., $|xy^2z|$ is also a square of an integer, which lies between the squares of two consecutive positive integers. This is not possible.

So, $\mathbf{xy^2z} \in \mathbf{L}$. This is a contradiction and **L** is not regular.

Example 2

Prove that L = { a^p | p is prime } is not regular.

Solution

Assume the set L is regular. Let n be the number of states of the FA accepting the set L.

Let p be a prime number greater than n. Let the string w = a^p, $w \in L$. By using the Pumping Lemma, we can write w = xyz with $|xy| \leq n$ and $|y| > 0$. As the string w consists of only 'a's, x, y, and z are also strings of 'a's. Let us assume that y = a^m for some m with $1 \leq m \leq n$.

Let us take, i = p + 1.

$$|\mathbf{xy^iz}| = |\mathbf{xyz}| + |\mathbf{y^{(i-1)}}|$$

$$\mathbf{p} + (\mathbf{i} - \mathbf{1})\,|\mathbf{y}|$$

$$(\mathbf{p} + (\mathbf{p} + \mathbf{1} - \mathbf{1})\mathbf{m})$$

$$= \mathbf{p} + \mathbf{pm}$$

$$\mathbf{p}(\mathbf{1} + \mathbf{m})$$

p(1 + m) is not a prime number as it has factors p and (1 + m), including 1 and p(1 + m). So, $xy^{(p+1)}z \notin L$. This is a contradiction.

Prove that L = {a^n b^n | n ≥ 1} is not regular.

Solution

Assume the set L is regular. Let n be the number of states of the FA accepting the set L.

Let w = a^n b^n, where |w| = 2^n. By the Pumping Lemma, we can write w = xyz with |xy| ≤ n and |y| > 0.

We want to find a suitable i so that x y^i z ∉ L.

The string **y** can be one of the following –

- y is a string of only 'a's, so $y = a^k$ for some $k \geq 1$.
- y is a string of only 'b's, so $y = b^k$ for some $k \geq 1$.
- y is a string of both 'a's and 'b's, so, $y = a^k b^l$ for some $k, l \geq 1$.

For case (i), take i = 0. As $xyz = a^n b^n$, $xy^0 z = xz$ will be $a^{n-k} b^n$.

$$\text{As } k \geq 1, (n - k) \neq n, \text{ so } xy^0z \notin L$$

For case (ii), take i = 0. As $xyz = a^n b^n$, $xy^0 z = xz$ will be $a^n b^{n-k}$.

$$\text{As } k \geq 1, (n - k) \neq n, \text{ so } xy^0z \notin L$$

For case (iii), take **i = 2**. As $xyz = a^n b^n$,

$$xy^2z = xyyz'$$

We know,

$$xyz = a^n b^n = a^{n-k} a^k b^l b^{n-l}$$

So, xyyz will be,

$$a^{n-k} a^k b^l a^k b^l b^{n-l} = a^n b^l a^k b^n$$

Which is not in the form $a^n b^n$, so $\mathbf{xy^2 z \notin L}$.

For all three cases, we find a contradiction. Therefore, L is not regular.

4.

a) Summarize the principal closure properties for regular languages.

✅ Principal 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 ($\neg L$ 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.

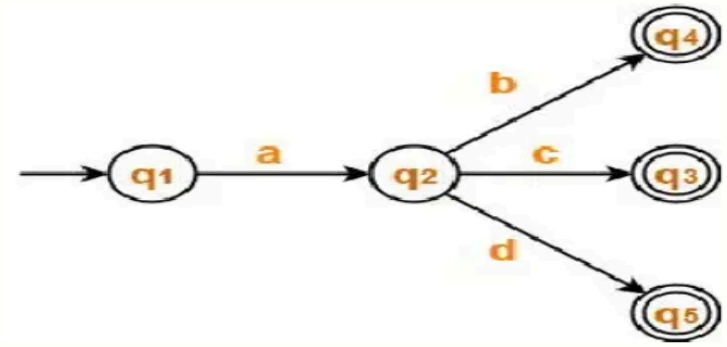
💡 **In short:**

> **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.

b) Convert DFA's to regular expressions by eliminating states (using an arbitrary example).

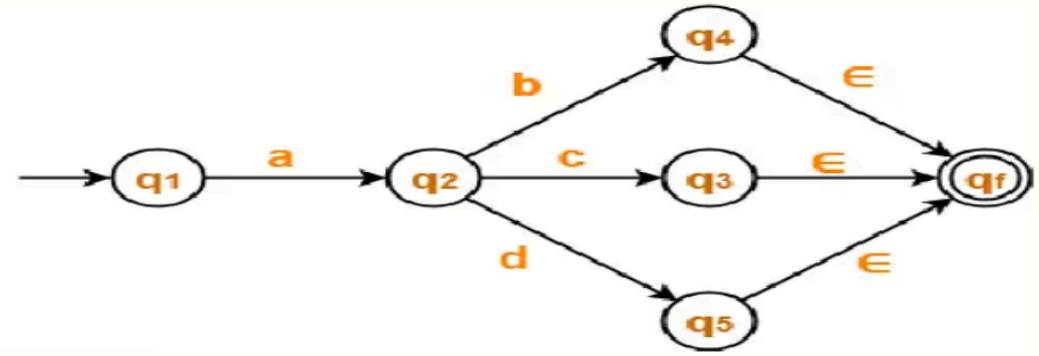
Find regular expression for the following DFA-



Step1:

- There exist multiple final states.
- So, we convert them into a single final state.

The resulting DFA is-

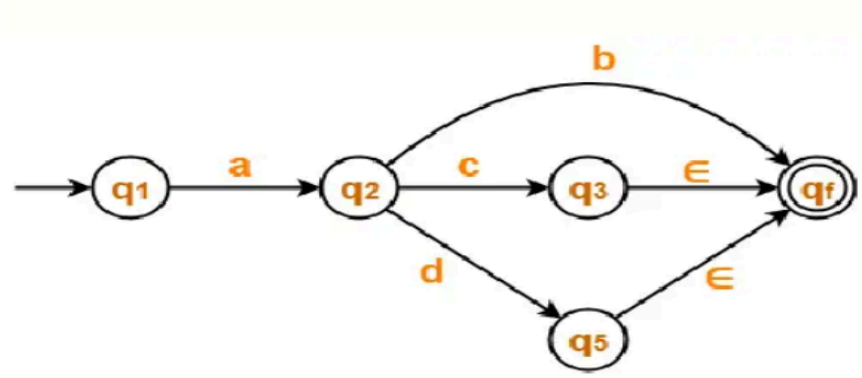


Step-02:

Now, we start eliminating the intermediate states.

First, let us eliminate state q_4.

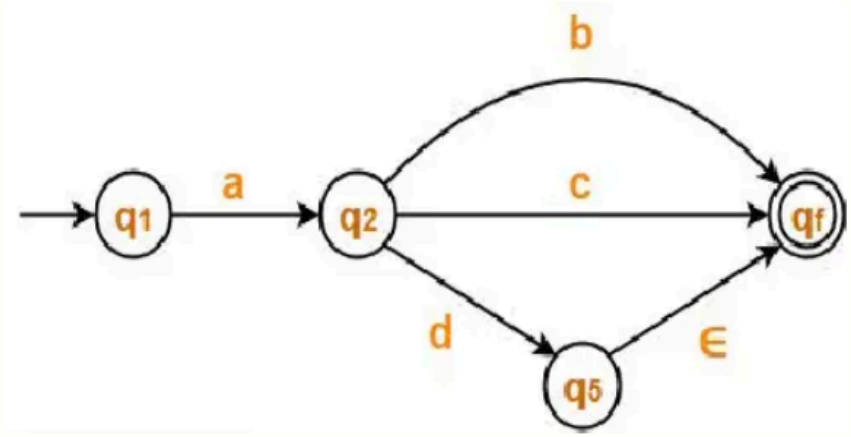
- There is a path going from state q_2 to state q_f via state q_4.
- So, after eliminating state q_4 , we put a direct path from state q_2 to state q_f having cost $b.\in = b$.



Step-03:

Now, let us eliminate state q_3.

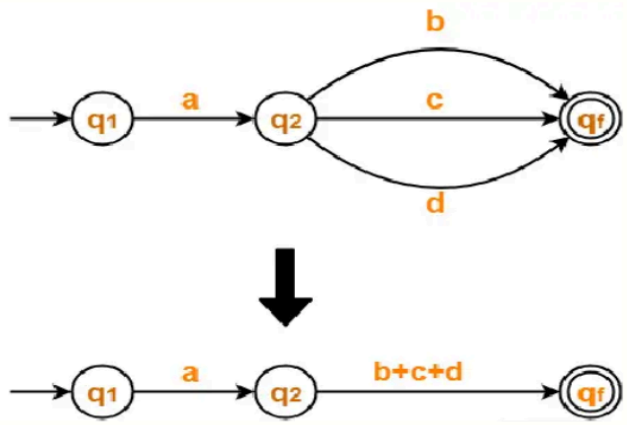
- There is a path going from state q_2 to state q_f via state q_3.
- So, after eliminating state q_3 , we put a direct path from state q_2 to state q_f having cost $c.\in = c$.



Step-04:

Now, let us eliminate state q_5.

- There is a path going from state q_2 to state q_f via state q_5.
- So, after eliminating state q_5 , we put a direct path from state q_2 to state q_f having cost d.∈ = d.



Step-05:

Now, let us eliminate state q_2.

- There is a path going from state q_1 to state q_f via state q_2.
- So, after eliminating state q_2 , we put a direct path from state q_1 to state q_f having cost a. (b+c+d).

From here,

Regular Expression = a(b+c+d)

Here's the **state elimination method in one line per step**:

1. **Add new start & final states** with ε-transitions.

2. **Select a non-start/non-final state** to eliminate.
3. **Update transitions**: new path = incoming × (self-loop)* × outgoing.
4. **Repeat elimination** until only start → final remains.
5. **The resulting edge label** = the DFA's regular expression.

C)Prove that if L is a regular language over alphabet Σ, then L ber = Σ* - L is also a regular language

If L is a regular language over alphabet Σ then $\overline{L} = \Sigma^* \setminus L$ is also regular.

Proof: Let L be recognized by a DFA

$$A = (Q, \Sigma, \delta, q_0, F).$$

Then $\overline{L} = L(B)$ where B is the DFA

$$B = (Q, \Sigma, \delta, q_0, Q \setminus F).$$

That is, B is exactly like A, but the accepting states of A have become the nonaccepting states of B and vice-versa.

Then w is in $L(B)$ iff $\hat{\delta}(q_0, w)$ is in $Q \setminus F$, which occurs iff w is not in $L(A)$.

Step 1: Start with DFA for L

Since L is regular, there exists a DFA

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

that accepts L, where:

- Q = set of states
- Σ\SigmaΣ = alphabet
- δ = transition function
- q0= start state
- $F \subseteq Q$= set of final states

Step 2: Construct DFA for complement

To get L ber, define a new DFA:

$M'=(Q,\Sigma,\delta,q0,Q-F)$

- Same states, alphabet, start state, and transitions
- New final states = all states that were not final in MMM

Step 3: Justification

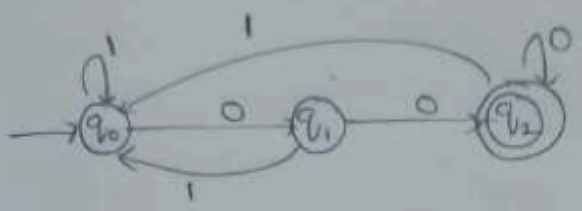
- Any string $w \in L$ is accepted by $M \rightarrow$ ends in some $q_f \in F \rightarrow$ rejected by M' because $q_f \notin Q - F$
- Any string $w \notin L$ ends in a state $q \notin F \rightarrow$ accepted by M' because $q \in Q - F$

Hence, M' accepts exactly all strings **not in L**, i.e., $L^{\text{ber}} = \Sigma^* - L$.

Step 4: Conclusion

Since M' is a DFA, L^{ber} is **regular**. ✅

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



solution

step 1: Assign a variable for Each state

$q_0 \rightarrow R$

solution

step 1: Given states are q_0, q_1 and q_2.

Transactions are 0 and 1.

step 2: Add a Rule $q_i \rightarrow a q_j$, if there is a Transaction from q_i to q_j

$q_0 \rightarrow 0q_1 \mid 1q_0$

$q_1 \rightarrow 0q_2 \mid 1q_0$

$q_2 \rightarrow 0q_2 \mid 1q_0$

step 3: If q_i is an ~~accept~~ Final state, then add a Rule $q_i \rightarrow \epsilon$

Here q_2 is the ~~acc~~ Final state

$\therefore q_2 \rightarrow \epsilon$

step 4:

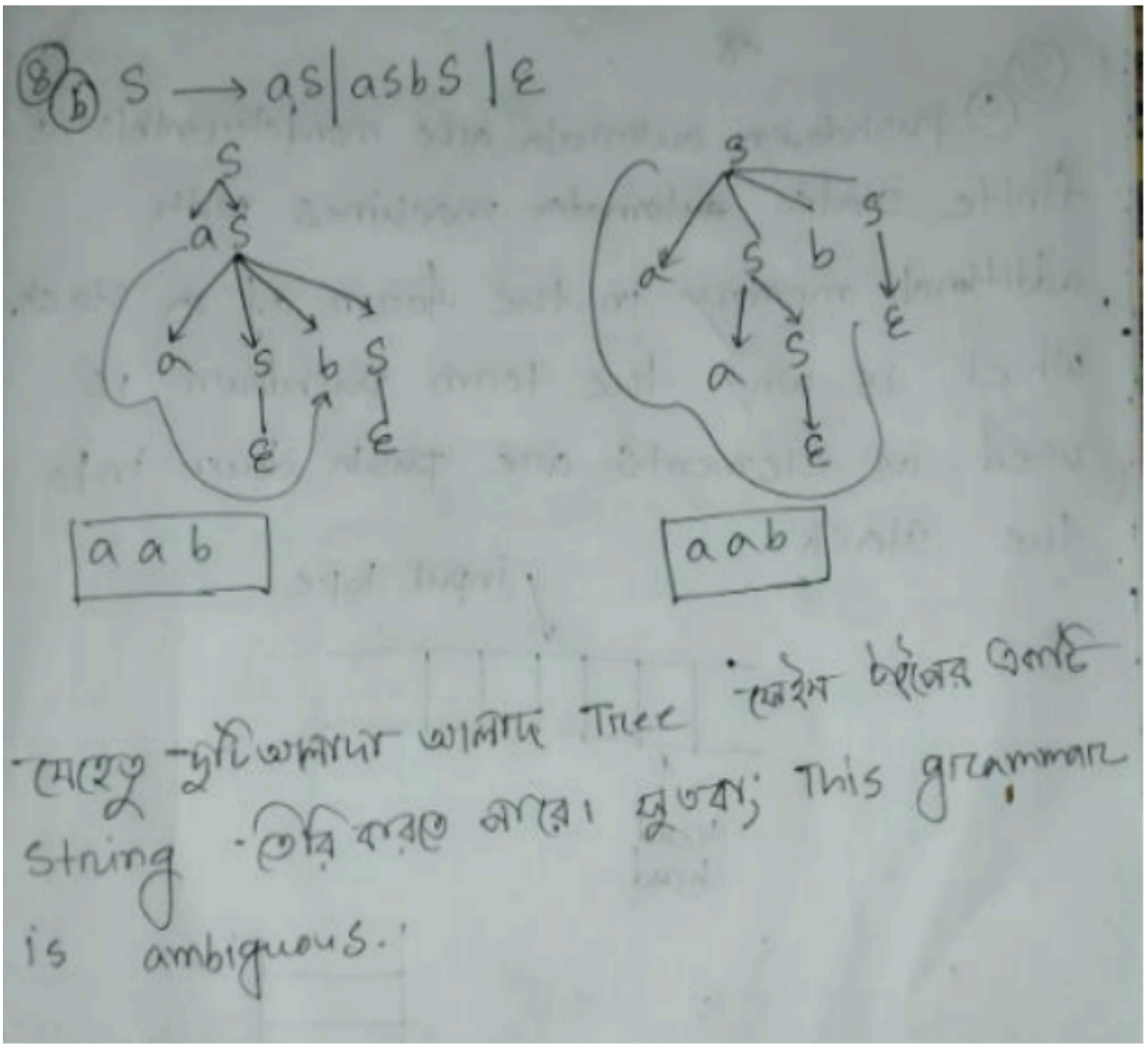
$q_0 \rightarrow 0q_1 \mid 1q_0$

$q_1 \rightarrow 0q_2 \mid 1q_0$

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

1. For each DFA state q create a grammar variable A_q.
2. Set the start variable $S = A_{q_0}$ where q_0 is the DFA start state.
3. For every DFA transition $\delta(p, a) = q$ add the production $A_p \to a\, A_q$.
4. For every DFA final state f add the production $A_f \to \varepsilon$.
5. If multiple transitions share a symbol, include all corresponding productions (union handled by multiple rules).
6. Optionally remove unreachable variables and useless productions.
7. The resulting CFG generates the same language as the DFA.

b)Show that the grammar ((S), (a, b), R, S) with rules R-SaS | aSbS € is ambiguous



Step 1: Choose a string

Let's test the string "**ab**" — it can be generated in **two ways**.

Step 2: Derivation 1

$$S \Rightarrow aSbS \Rightarrow a\varepsilon b\varepsilon = ab$$

Step 3: Derivation 2

$$S \Rightarrow aS \Rightarrow aaSbS \Rightarrow a(ab) \ (\text{using } S \Rightarrow aSbS \text{ then } S \Rightarrow \varepsilon)$$

Simplifying gives the same "**ab**" string but through a **different parse structure**.

Step 4: Conclusion

Since the same string "ab" can be derived using **two different parse trees (derivations)**, the grammar is **ambiguous**. ✅

Hence, the grammar $S \rightarrow aS \mid aSbS \mid \varepsilon$ is ambiguous.

🔷 Derivation 1 (using S → aSbS first)

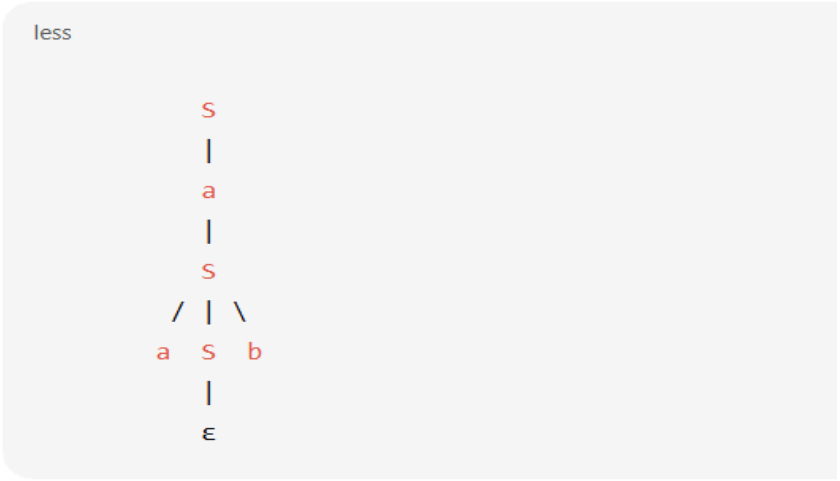
less

```
      S
    / | \
   a  S  b
      |
      ε
```

String formed: a + ε + b = **ab**

🔷 Derivation 2 (using S → aS first, then S → aSbS)

less



String formed: a + (a + ε + b) = **ab**

C)Explain inverse homomorphism.

An inverse homomorphism is the process of finding all strings in an alphabet that, when transformed by a given homomorphism, result in a string within a specific language

Inverse homomorphism is a closure property of regular languages. It allows us to map the input and output elements of a homomorphic language back to the original language.

We can say that if L is a regular language, its inverse homomorphism H^{-1} is as follows:

$$H^{-1}(L) = x \mid H(x) \in L\}$$

> **Note:** The inverse homomorphism of a regular language is also regular.

Example

Let's suppose there is a language L that has inputs in the set $\Sigma\{a, b, c\}$, and we have mapped it with $\Gamma\{0, 1\}$ homomorphically.

First, we determine the homomorphic images of each input symbol/alphabet. In this case, we may take:

$$H(a) = 0\ ,\ H(b) = 1\ and\ H(c) = 01$$

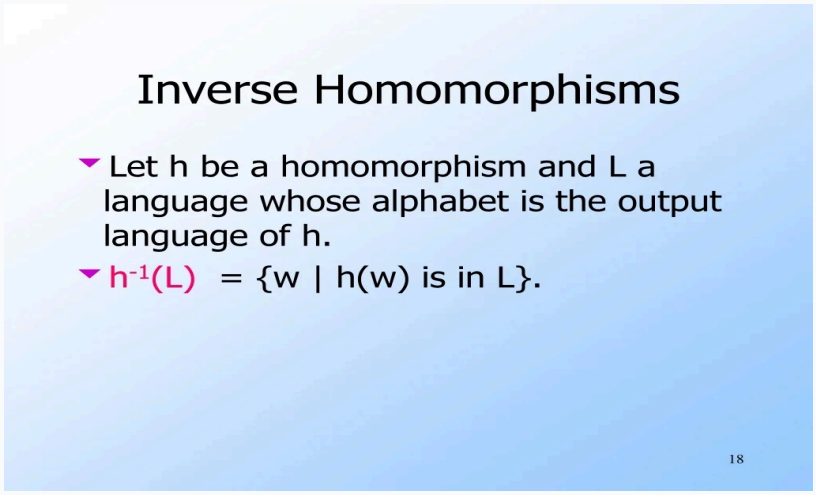
If we take $L = \{0011001\}$, then:

$$H^{-1}(L) = \{aabbaab, aabbac, acbaab, acbac\}$$

We see that the inverse homomorphism of such a language may have many translations/instances.

Note: The homomorphism of an inverse homomorphism of a language is a subset of the original language

$H(H\text{^}{-}1(L)) \subseteq L.$



Example:

Let
$\Sigma_1 = \{a, b\}$,
$\Sigma_2 = \{0, 1\}$,
and define
$h(a) = 0$,
$h(b) = 11$.

Now, let $L = \{011\} \subseteq \Sigma_2^*$.

We find $h^{-1}(L)$:

$$h^{-1}(L) = \{w \in \{a, b\}^* \mid h(w) = 011\}$$

Check possible strings over {a, b}:

- $h(ab) = 0\,11 = 011$ ✅
- $h(ba) = 110 = 110$ ❌
- $h(aab) = 0011 = 0011$ ❌

So,

$$h^{-1}(L) = \{ab\}$$

If **L is a regular language**, then **h⁻¹(L)** is also **regular**.

✅ Regular languages are **closed under inverse homomorphism**.

6.a)Elaborate the three operations on languages that the operations of regular expression represent.

🎯 Regular Expressions Represent Three Basic Operations on Languages

A **regular expression** describes a **regular language** using **three fundamental operations**:

Operation	Regular Expression Symbol	Description
1️⃣**Union (OR)**	+ or ∪ or `	`
2️⃣ **Concatenation**	*juxtaposition* (no symbol)	Represents **joining** two languages end-to-end
3️⃣**Kleene Star**	*	Represents **zero or more repetitions** of a language

---

1️⃣Union (Alternation / OR Operation)

Definition:

If (L_1) and (L_2) are two languages,
then their **union** is:

$$L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$$

Regular Expression Form:

If (R_1) and (R_2) are regular expressions for (L_1) and (L_2),
then ($R_1 + R_2$) (or ($R_1 \mid R_2$)) represents ($L_1 \cup L_2$).

Example:

- (R = a + b)
- (L(R) = { a, b })

➡ It accepts either ‘a’ **or** ‘b’.

2️⃣ Concatenation

Definition:

If (L_1) and (L_2) are two languages,
then their **concatenation** is:

$$L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

Regular Expression Form:

If (R_1) and (R_2) represent (L_1) and (L_2),
then (R_1R_2) represents (L_1L_2).

Example:

- (R = a b)
- (L(R) = { ab })

➡ It accepts the string formed by **‘a’ followed by ‘b’**.

3️⃣ Kleene Star (Repetition)

Definition:

If (L) is a language,
then the **Kleene closure** is:

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots$$

This means **zero or more repetitions** of strings from L.

Regular Expression Form:

If (R) represents L, then (R^*) represents (L^*).

Example:

- (R = a^*)
- (L(R) = {ε , a, aa, aaa, …….})

➡ It accepts **any number of a's**, including none.

🧠 Summary Table

Operation	Symbol in RE	Meaning	Example RE	Example Language
Union	+, ∪, ` `		Choice between strings	`a + b`
Concatenation	(no symbol)	Joining strings	`ab`	`{ab}`
Kleene Star	`*`	Repetition (0 or more times)	`a*`	`{ε, a, aa, aaa, …}`

💬 In Short:

> Regular expressions describe regular languages using three main operations:

- **Union** → choice between patterns
- **Concatenation** → sequence of patterns
- **Kleene Star** → repetition of patterns

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

Claim. If L is a regular language over alphabet Σ and F is a finite set of strings from Σ^*, then $L \cup F$ is regular.

We give two short, standard proofs.

Proof A — using closure properties (direct)

1. For any single string $w \in \Sigma^*$, the singleton language $\{w\}$ is regular.
   Reason: a regular expression w (or a DFA that reads the symbols of w along a chain of states and accepts only at the end) defines $\{w\}$.
2. A finite set $F = \{w_1, w_2, \ldots, w_n\}$ is the finite union

$$F = \{w_1\} \cup \{w_2\} \cup \cdots \cup \{w_n\}.$$

   Each $\{w_i\}$ is regular by step 1. Regular languages are closed under finite union (closure under union follows from closure under binary union applied repeatedly).
3. Therefore F is regular.
4. Regular languages are closed under union, so $L \cup F$ is regular (union of regular L and regular F).

This completes the proof.

Proof B — constructive via DFA/NFA (induction)

1. Construct a DFA (or NFA) M_L that recognizes L (exists since L is regular).
2. For a single string $w = a_1 a_2 \cdots a_k$, construct an NFA M_w that accepts exactly w: chain $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} q_k$ with q_k accepting.
3. To accept a finite set $F = \{w_1, \ldots, w_n\}$, form an NFA that has a new start state with ε-transitions to the start states of each M_{w_i}. That NFA accepts exactly F. (Equivalently add n disjoint chains.)
4. To accept $L \cup F$ form an NFA that has ε-transitions from a new start to the start of M_L and to each M_{w_i}. The resulting NFA accepts $L \cup F$. Converting this NFA to a DFA (subset construction) yields a DFA for $L \cup F$.

Hence $L \cup F$ is regular.

c) Write the closure properties of regular languages 1.Union(U)

If L_1 and L_2 are regular,
then $L_1 \cup L_2$ is also regular.

Example:
Let $L_1 = \{a^n \mid n \geq 0\}$ and $L_2 = \{b^n \mid n \geq 0\}$.
Then,

$$L_1 \cup L_2 = \{a^n \mid n \geq 0\} \cup \{b^n \mid n \geq 0\} = \{a^*, b^*\}$$

is regular.

◆ 2. Concatenation (·)

If L_1 and L_2 are regular,
then $L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\}$ is regular.

Example:
$L_1 = \{a^*\}, L_2 = \{b^*\}$

$$L_1 L_2 = \{a^n b^m \mid n, m \geq 0\}$$

is regular.

◆ 3. Kleene Star (*)

If L is regular,
then $L^* = \{w_1 w_2 ... w_k \mid w_i \in L, k \geq 0\}$ is regular.

Example:
If $L = \{a\}$,
then $L^* = \{a^n \mid n \geq 0\} = a^*$,
which is regular.

◆ 4. Intersection (∩)

If L_1 and L_2 are regular,
then $L_1 \cap L_2$ is regular.

Example:
$L_1 = (a+b)^*a$ → all strings ending with 'a'
$L_2 = a(a+b)^*$ → all strings starting with 'a'
Then

$$L_1 \cap L_2 = a(a+b)^*a$$

◆ 5. Complement (¯)

If L is regular,
then its **complement** $\bar{L} = \Sigma^* - L$ is also regular.

Example:
If $L = (a+b)^*a$ (strings ending with 'a'),
then

$$\bar{L} = (a+b)^*b$$

(strings ending with 'b') is regular.

◆ 6. Difference (−)

If L_1 and L_2 are regular,
then $L_1 - L_2 = \{w \mid w \in L_1, w \notin L_2\}$ is regular.

Example:
$L_1 = (a+b)^*$, $L_2 = (a+b)^*b$
Then

$$L_1 - L_2 = (a+b)^*a$$

◆ 6. Difference (−)

If L_1 and L_2 are regular,
then $L_1 - L_2 = \{w \mid w \in L_1, w \notin L_2\}$ is regular.

Example:
$L_1 = (a+b)^*$, $L_2 = (a+b)^*b$
Then

$$L_1 - L_2 = (a+b)^*a$$

(strings ending with 'a') is regular.

◆ 7. Reversal (R)

If L is regular,
then the **reversal** of L, denoted $L^R = \{w^R \mid w \in L\}$, is regular.

Example:
If $L = \{ab, baa\}$,
then

$$L^R = \{ba, aab\}$$

is regular.

◆ 8. Homomorphism and Inverse Homomorphism

- **Homomorphism (h):**
  If L is regular, then $h(L)$ is regular.
- **Inverse Homomorphism (h^{-1}):**
  If L is regular, then $h^{-1}(L)$ is regular.

Example:
Let $h(a) = 0, h(b) = 11$, and $L = \{011\}$.
Then $h^{-1}(L) = \{ab\}$ is regular.

7.

a) Prove that the following are not regular languages:

(i) The set of strings of 0's and 1's beginning with 1, such that when interpreted as an integer, that integer is a prime.

(ii) The set of strings of the form 0 i1 j such that the greatest common divisor of i and j is 1.

(i) $L_1 = \{$ binary strings that start with `1` and represent a prime number$\}$ — not regular (Pumping Lemma sketch)

1. Suppose L_1 is regular. Let p be the pumping length from the Pumping Lemma.
2. Pick a prime q so large that its binary representation w begins with `1` and $|w| > p$ (there are infinitely many primes so we can).
3. By the Pumping Lemma write $w = xyz$ with $|xy| \leq p$, $|y| \geq 1$, and $xy^k z \in L_1$ for all $k \geq 0$. Note y lies inside the first $|w|$ bits, so pumping changes the numerical value in a controlled, linear-in-k way:

$$n_k = \mathrm{val}(xy^k z) = A \cdot 2^{t(k-1)} + B$$

for some integers A, B, t (here $t = |y|$). Thus $\{n_k\}_{k \geq 0}$ is an arithmetic-type progression in k (values differ by a fixed positive increment when k changes).
4. Any such infinite progression of integers contains composite numbers (one can choose k so that n_k is divisible by some fixed small prime). Hence for some k the pumped string $xy^k z$ represents a composite number and so is **not** in L_1.
5. This contradicts the Pumping Lemma requirement that $xy^k z \in L_1$ for all k. Therefore L_1 is not regular.

(ii) $L_2 = \{0^i 1^j \mid \gcd(i, j) = 1\}$ — not regular (Myhill–Nerode / distinguishability)

1. For each $n \geq 0$ consider the prefix $u_n = 0^n$. We will show the infinitely many u_n are pairwise distinguishable — which by Myhill–Nerode implies non-regularity.
2. Take $m \neq n$. Choose a prime p that divides one of m, n but not the other (such a prime exists). Without loss assume $p \mid m$ and $p \nmid n$.
3. Use the suffix $s = 1^p$. Then:
   - $\gcd(m, p) \geq p > 1$ so $u_m s = 0^m 1^p \notin L_2$.
   - $\gcd(n, p) = 1$ so $u_n s = 0^n 1^p \in L_2$.

   Thus u_m and u_n are distinguishable by the same suffix s.
4. Since this works for any distinct m, n, there are infinitely many pairwise distinguishable prefixes 0^n. By Myhill–Nerode the language L_2 is not regular.

Final summary (one-line each)

- (i) Pumping an appropriately long binary prime yields an arithmetic family of values; some pumped member is composite → contradiction, so L_1 is not regular.
- (ii) For prefixes 0^m and 0^n choose a prime dividi one but not the other; the suffix 1^p distinguishes them; infinitely many distinguishable prefixes → L_2 not regular.

b) State the properties of a parse tree. Construct a parse tree for the following string:S->S S +|SS*|a

Properties of a parse tree (brief)

1. **Root** is the start symbol (here `S`).
2. **Interior nodes** are nonterminals; **leaf nodes** are terminals (or ε).
3. The **children of a nonterminal** correspond (in order) to the symbols of the right-hand side of a production used to expand that nonterminal.
4. The **yield (frontier)** — the left-to-right concatenation of the leaves — equals the derived terminal string.
5. The tree encodes a specific derivation; different parse trees mean different derivations (and show ambiguity if more than one parse tree exists for the same string).
6. A parse tree is **finite** and its depth equals the maximum length of derivation steps along any branch.

Grammar

$$S \to S\,S+ \mid S\,S* \mid a$$

(This is a postfix-expression style grammar: two subexpressions followed by `+` or `*`, or the terminal `a`.)

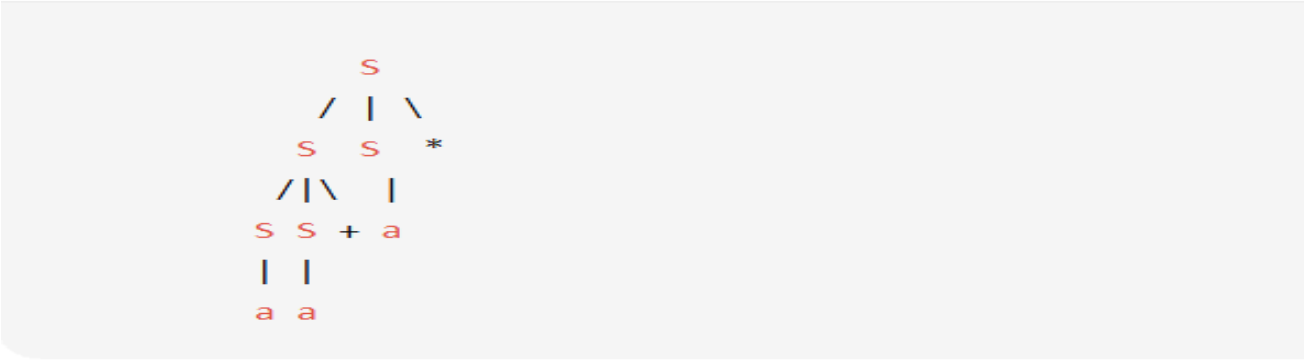
Construct a parse tree for the string `aa+a*`

I will build a leftmost derivation and the corresponding parse tree. The target string is the postfix `aa+a*`.

Leftmost derivation

```
S
⇒ S S *              (use S → S S *)
⇒ (S S +) S *        (expand the leftmost S by S → S S +)
⇒ (a a +) S *        (replace the leftmost two S by a, a)
⇒ (a a +) a *        (replace the remaining S by a)
```

Yield (concatenate leaves left→right): `a a + a *` → `aa+a*`.



More clearly (each production shown as children in order):

- Root: `S`
  - left child: `S`
    - left child: `S` → leaf `a`
    - middle child: `S` → leaf `a`
    - right child: terminal `+`
  - middle child (right child of root before `*`): `S` → leaf `a`
  - rightmost child of root: terminal `*`

Yield check: leaves left-to-right = `a a + a *` ⟹ `aa+a*` (matches).

---

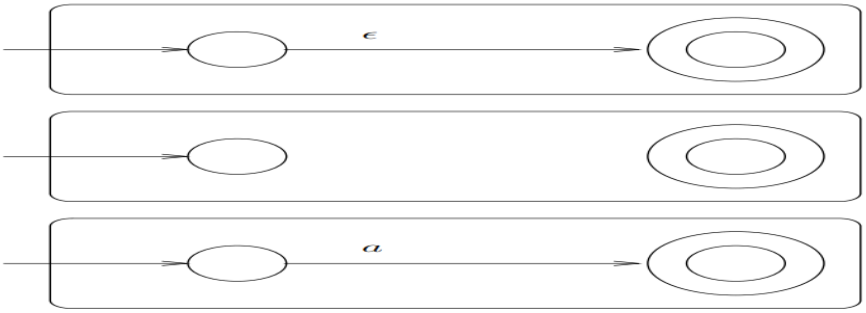
c) Prove that every language defined by a regular expression is also defined by a finite automata.

Every language defined by a regular expression is also defined by a finite automaton.

- Suppose $L = L(R)$ for a regular expression R. We show that $L = L(E)$ for some $\epsilon - NFA\ E$ with
  1. Exactly one accepting state
  2. No arcs into the initial state
  3. No arcs out of the accepting state
- The proof is by structural induction on R, following the recursive definition of regular expressions.

Basis

The basis of the construction of fsa from regular expressions:

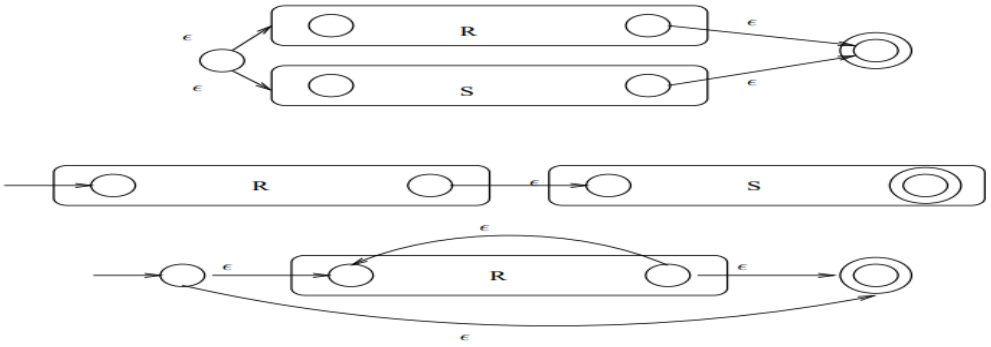


1. Expression ϵ: the language of the FSA is $\{\epsilon\}$.
2. Expression $\emptyset$: $\emptyset$ is the language of FSA.
3. Expression **a**: the language of the FSA is $\{a\}$.

All these automata satisfies the three initial conditions

Induction

The inductive step of the construction of fsa from regular expressions



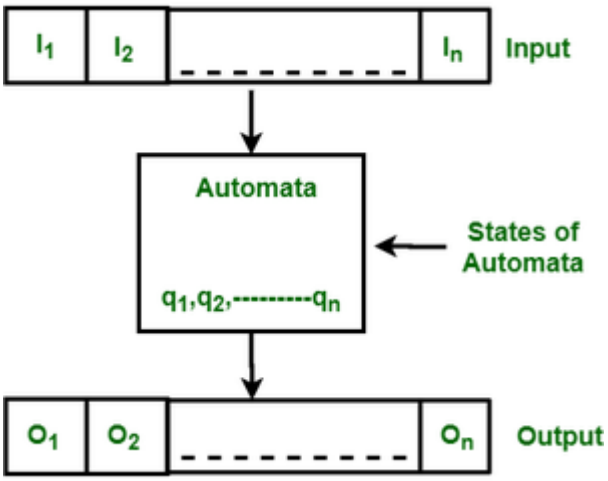
1. The expression is $R + S$ for some smaller expressions R and S.
2. The expression is RS for some smaller expressions R and S.
3. The expression is R^* for some smaller expression R.
4. The expression is (R) for some smaller expression R.

8.

a) State formal definition of a finite automata. Give an example

Finite automata are abstract machines used to recognize patterns in input sequences, forming the basis for understanding regular languages in computer science.

- Consist of states, transitions, and input symbols, processing each symbol step-by-step.
- If ends in an accepting state after processing the input, then the input is accepted; otherwise, rejected.
- Finite automata come in deterministic (DFA) and non-deterministic (NFA), both of which can recognize the same set of regular languages.
- Widely used in text processing, compilers, and network protocols.



Formally, a **finite automaton** is a **5-tuple**:

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

where:

1. **Q** → A finite set of **states**.
2. **Σ** → A finite set called the **input alphabet**.
3. **δ** → The **transition function**,

$$\delta : Q \times \Sigma \to Q$$

   (defines the next state for each pair of current state and input symbol).
4. **q_0** → The **start state**, where $q_0 \in Q$.
5. **F** → A set of **accepting (final) states**, where $F \subseteq Q$.

1. Deterministic Finite Automata (DFA)

A DFA is represented as {Q, ∑, q, F, δ}. In DFA, for each input symbol, the machine transitions to one and only one state. DFA does not allow any null transitions, meaning every state must have a transition defined for every input symbol

Example:

Construct a DFA that accepts all strings ending with 'a'.

Given:

Σ = {a, b},

Q = {q0, q1},

F = {q1}

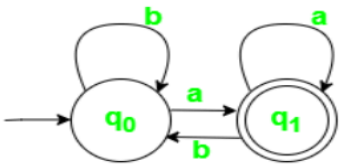


Fig 1. State Transition Diagram for DFA with Σ = {a, b}

State\Symbol	a	b
q0	q1	q0
q1	q1	q0

In this example, if the string ends in 'a', the machine reaches state q1, which is an accepting state.

2) Non-Deterministic Finite Automata (NFA)

NFA is similar to DFA but includes the following features:

- It can transition to multiple states for the same input.
- It allows null (ϵ) moves, where the machine can change states without consuming any input.

Example:

Construct an NFA that accepts strings ending in 'a'.

Given:

Σ = {a, b},

Q = {q0, q1},

F = {q1}

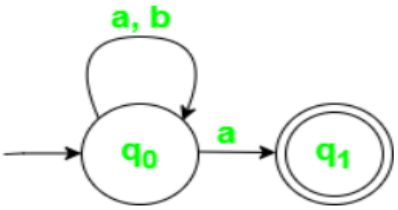


Fig 2. State Transition Diagram for NFA with Σ = {a, b}

State Transition Table for above Automaton,

State\Symbol	a	b
q0	{q0,q1}	q0
q1	φ	φ

In an NFA, if any transition leads to an accepting state, the string is accepted.

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`

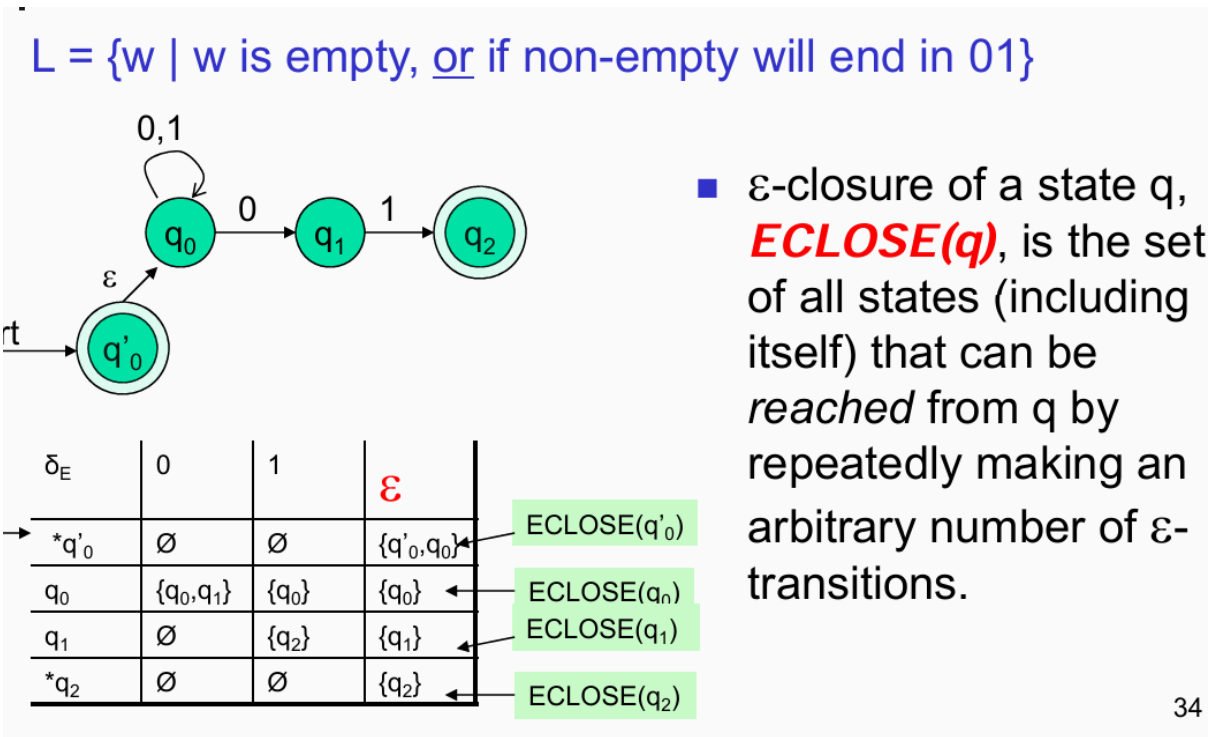
Reason: `acdb = a·(cd)·b` (one block with k=1).

(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)Build an e-NFA for the following language:

L = {w | w is empty, or if non-empty will end in 011}



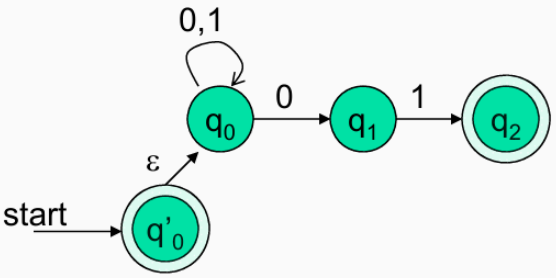
To simulate any transition:

Step 1) Go to all immediate destination states.

Step 2) From there go to all their ε-closure states as well.

Example of an ε-NFA

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



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

Simulate for w=101: