1) write the algebraic laws for regular expression ?

Algebraic laws of Regular Expressions.

let

R, S, T . Regular expressions

ϵ = Empty string

ϕ = Empty set

→ The algebraic laws for regular expressions are fundamental properties that govern their manipulation and simplification in automata and compiler design. These laws allow for the transformation of regular expressions while preserving the language they denote

Fundamental Laws :

Commutative Law for Union :

→ $R+S = S+R$

→ The order of union (OR operation) does not affect the resulting language.

Associative laws :

→ For Union : $(R+S)+T = R+(S+T)$

→ For concatenation : $(R \cdot S) \cdot T = R \cdot (S \cdot T)$

→ Grouping of terms in union or concatenation does not affect the resulting language. for union or concatenation.

Distribute laws:

→ $R \cdot (S+T) = (R \cdot S)+(R \cdot T)$

→ $(S+T) \cdot R = (S \cdot R)+(T \cdot R)$

→ Concatenation distributes over union.

Identity laws:

→ union : $R+\phi = \phi+R = R$.

→ The empty set (ϕ) is the identity element for union.

Concatenation : $R \cdot \epsilon = \epsilon \cdot R = R$

→ The empty string (ϵ) is the identity element for concatenation.

Annihilator law : $R \cdot \phi = \phi \cdot R = \phi$

→ Concatenating with the empty set results in the empty set.

Idempotent law: $R+R = R$

→ Union with itself does not change the language.

Laws involving kleene closure :

closure of Empty string :

$\epsilon^* = \epsilon$

• Zero or more repetitions of the empty string is the empty string.

closure of Empty set :

$\phi^* = \epsilon$

→ Zero or more repetitions of the empty set is the empty string.

Double closure:

$(R^*)^* = R^*$

→ Applying kleene closure twice is equivalent to apply it once.

Relation between closure and Positive closure :

$R^+ = R \cdot R^* = R^* \cdot R$

↦ positive closure (one or more repetitions) can be expressed using kleene closure.

closure as union with Empty string:

$R^* = R^+ + \epsilon$

↦ kleene closure includes the empty string, while positive closure does not.

2) Abstract model of finite automaton.

In compiler design, a finite automaton is an abstract mathematical model with a finite number of states and rules for transitioning between them based on input symbols.

↦ A Finite Automation (FA) is a 5-tuple mathematical model defined as:

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

where,

$Q \rightarrow$ Finite set of states

Σ (sigma) $\rightarrow$ Finite set of input symbols (alphabet).

δ (delta) $\rightarrow$ Transition function: $Q \times \Sigma \rightarrow Q$

↦ defines how the automation moves from one state to another

$q_0 \rightarrow$ Initial state

$F \rightarrow$ set of final (accepting) states $F \subseteq Q$

↦ working principle:

↦ the FA reads input string symbol by symbol

• At each step, based on current state and input symbol, transition function δ decides the next state.

- After the entire input is consumed.
  - If the FA ends is an accepting state (EF) → string is accepted.
- otherwise → string is accepted.

Types of Finite Automata:

1) Deterministic Finite Automaton (DFA):
   - For each state and input, exactly one transition exists
   - No ε-moves
   - used in compiler design (lexical analysis)

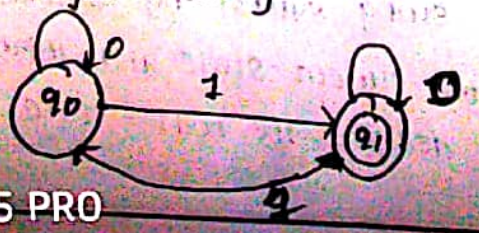
2) Non-Deterministic Finite Automata (NDFA):
   - For a state and input, multiple transitions are possible.
   - May include ε-transitions (move without input).
   - Easier to construct, later to converted to DFA.

Diagrammatic Representation:

FA can be represented as state transition diagram

- States = circles
- Start State = Arrow pointing into initial state
- Final States = Double circles
- Transitions = Directed edges labeled with input symbols.

Problem:. construct a DFA over the string $\Sigma = \{0,1\}$ of string length = 2, check given string is 1101 is 0,1 DFA.



Mathematical Model :

$M = (\{q_0, q_1\}, \{0,1\}, \delta, q_0, \{q_1\})$

- $Q = \{q_0, q_1\}$
- $\Sigma = \{0,1\}$
- q_0 = start state
- $F = \{q_1\}$

Transition Table

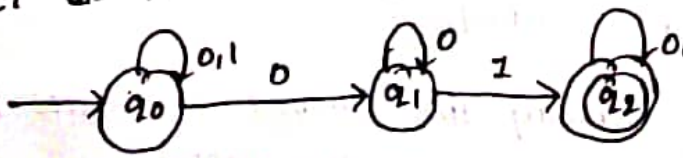
δ	0	1
q_0	q_0	q_1
q_1	q_1	q_0

$\delta(q_0,0) = q_0$
$\delta(q_0,1) = q_1$
$\delta(q_1,0) = q_1$
$\delta(q_1,1) = q_0$

$= \delta(1101)$
$= \delta(q_1 101)$
$= \delta(q_0 01)$
$= \delta(q_0 1)$
$= \delta(q_1)$

$\therefore$ DFA is satisfied.

construct a NFA string $\Sigma = \{0,1\}$



Mathematical Model:

$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- q_0 = starting state
- q_2 = final state.

Transition Table

δ	0	1
q_0	$\{q_0, q_1\}$	q_0
q_1	q_1	q_2
q_2	q_2	q_2

3) Minimization of finite Automata.

Introduction

↦ Finite Automata (FA) :- Machine that accepts or rejects strings over an alphabet.

↦ In practice (e.g., lexical analysis in compilers), we want the smallest DFA for efficiency

↦ processing of reducing the number of states in DFA without changing the language it accepts.

↦ unique minimal DFA (unique up to renaming of states)

Why Minimization

1. Optimization - Reduces memory and transition table size
2. Efficiency - Speeds up token recognition in compilers.
3. uniqueness - Minimal DFA is unique, helps in language comparison.
4. Simplicity - Easier to analyze and implement.

Steps in Minimization

- Remove unreachable states
- Partition the states

Method:

1. Initial partition (P0):
   - Final states (F)
   - Non-final states (Q-F)
2. Refinement:
   - Split groups further based on their transitions.
   - Continue until no further splitting is possible.

Construct the Minimal DFA:

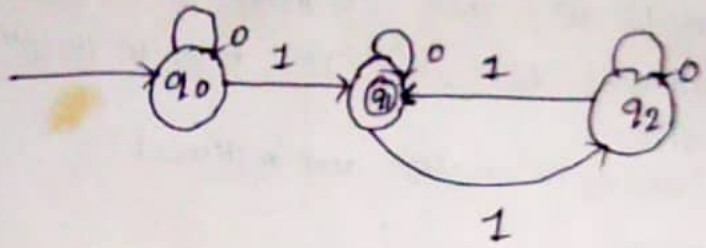
- Each equivalence class = one state in the new DFA.
- Define transitions between classes.

Minimization Algorithm.

1. Create a table with pairs of states
2. Mark pairs where one is final and other is non-final.
3. Iteratively mark pairs whose transitions go to already marked pairs.

(4) Remaining unmarked pairs = equivalent states → merge them.

Example: Minimization of automata which is finite



Step-1: Transition Table.

Q/Σ	0	1
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_1

Step-2 :- 0-Equivalence: we have to divide final states and non-final states.

$\{q_0, q_2\}$ are non-final states

$\{q_1\}$ is final state

Step-3: 1-equivalence

$\delta(q_0,0) = q_0$ $\quad$ $\delta(q_0,1) = q_1$

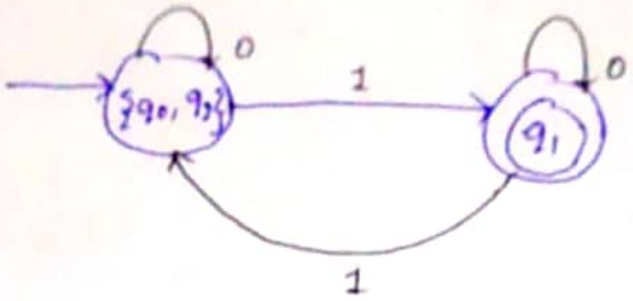
$\delta(q_2,0) = q_2$ $\quad$ $\delta(q_2,1) = q_1$

} → These two are equal.

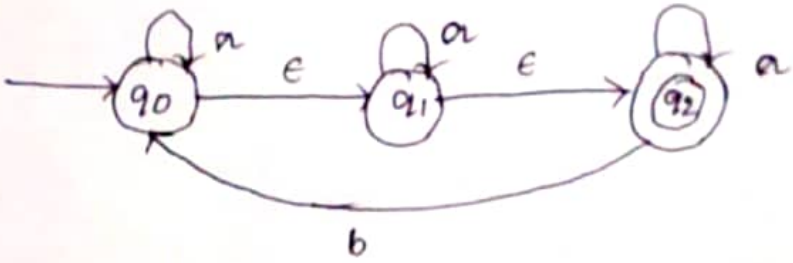
** when two consequences have same states we have to stop the problem and check for further level

so it has two states, they are

$\{q_0, q_2\}$, $\{q_1\}$

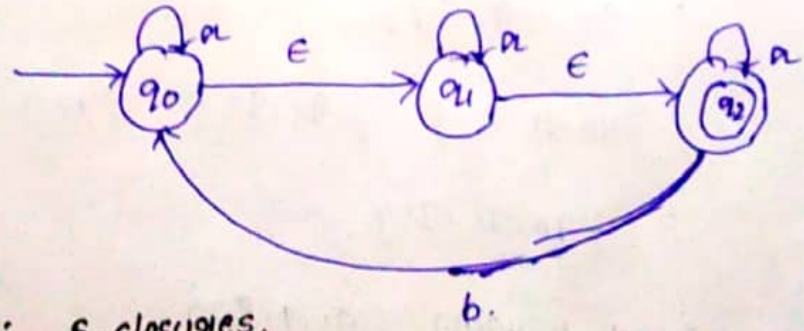


4) converting ε NFA to DFA example.



Step-1 :

Given



Step-2: ε-closures.

ε-closure (q0) = $\{q_0, q_1, q_2\}$

ε-closure (q1) = $\{q_1, q_2\}$

ϵ-closure $(q_1) = \{q_1\}$.

STEP-3: ϵ-NFA Transition table:

δ	a	b	ϵ
q_0	q_0	ϕ	q_1
q_1	q_1	ϕ	q_2
q_2	q_2	q_0	ϕ

STEP-4: DFA Transitions.

- On input a: using $\{q_0, q_1, q_2\}$

$$\delta(\{q_0, q_1, q_2\}, a)$$

$$= \delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a)$$

$$= q_0 \cup q_1 \cup q_2$$

$$= \{q_0, q_1, q_2\} \cup \{q_1, q_2\} \cup \{q_2\}$$

$$= \{q_0, q_1, q_2\}.$$

- on input b using $\{q_0, q_1, q_2\}$.

$$\delta(\{q_0, q_1, q_2\}, b)$$

$$= \delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b).$$

$= \phi \cup \phi \cup q_0$

$= \{q_0, q_1, q_2\}$.

on input a using $\{q_1, q_2\}$.

$\delta(\{q_1, q_2\}, a) = \delta(q_1, a) \cup \delta(q_2, a)$

$= q_1 \cup q_2$

$= \{q_1 q_2\} \cup \{q_2\}$

$= \{q_1, q_2\}$.

on input b using $(\{q_1, q_2\})$

$\delta(\{q_1, q_2\}, b) = \delta(q_1, b) \cup \delta(q_2, b)$

$= \phi \cup q_0$

$= \{q_0, q_1, q_2\}$.

on input a using $\{q_2\}$.

$\delta(q_2, a) = q_2 a = q_2$

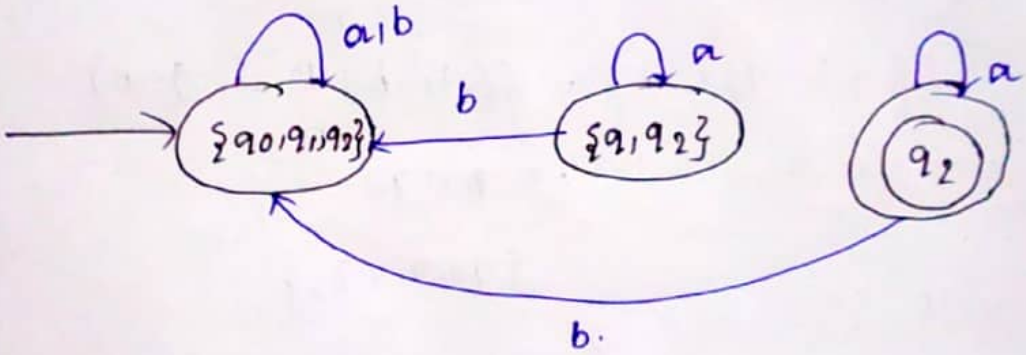
on input b using $\{q_2\}$

$\delta(q_2, b) = \{q_0\} = \{q_0, q_1, q_2\}$

Transition Table of DFA.

δ	a	b
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$
$\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
$\{q_2\}$	$\{q_2\}$	$\{q_0, q_1, q_2\}$.

Diagram:



5) Differences between NFA and DFA.

DFA	NFA. NFA.
1. Defination	
Each state has exactly one transition for each input symbol.	A state can have zero, one or multiple transitions for a given input symbol (including ε-transition)
2. Aspect.	
DFA (Deterministic Finite Automata)	NFA (Non-Deterministic Finite Automata)

	DFA	NFA
3. Transition function	$\delta : Q \times \Sigma \to Q$ single next state for each other	$\delta : Q \times \Sigma \to 2^{\wedge}Q$ set of possible next states for each symbol.
4. uniqueness of path.	For every input string, there is exactly one unique path in DFA	For every input string, there may be multiple paths to follow.
5. ϵ - Transitions.	DFA does not allow ϵ -transitions	NFA can have ϵ-transitions.
6. Complexity of construction.	More difficult to construct directly because every transition must be specified	Easier to construct (esp from regular expressions)
7. Speed of Executions.	Faster, because there is only one unique path to follow for any input	slower since it may explore multiple paths in parallel.

8. Memory Requirement:

Needs more states in many cases

Requires fewest states compared to DFA.

9. Acceptance of String

A string is accepted if the unique path ends in final state

A string is accepted if at leats one path in a final state

10. Implementation

Easier to implement programming

Harder to implement directly, usually converted into DFA first.

11. Equivalence

DFA and NFA are equivalent in power both accept the same regular languages

Same as DFA (no extra power, just more convenient representation