

Properties of set operations:

- ① Identity: $A \cup A = A$, $A \cap A = A$
- ② Associativity: $(A \cup B) \cup C = A \cup (B \cup C)$, $(A \cap B) \cap C = A \cap (B \cap C)$
- ③ Commutativity: $A \cup B = B \cup A$, $A \cap B = B \cap A$
- ④ Distributivity: $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$
- ⑤ Absorption: $(A \cup B) \cap A = A$
- ⑥ De Morgan's law: $A - (B \cap C) = (A - B) \cup (A - C)$

Disjoint sets:

$A = \{a, b, c, d\}$, $A \cap B = \phi$
 $B = \{x, y, z\}$

Cardinality:

$A = \{a, b, c, d\}$
 $(A) = \text{no. of elements in set } A = 4$

★ Relative function:

Cartesian Product:

Let A & B be two sets, then the set of all ordered pairs (x, y) where $x \in A$ & $y \in B$ is called the Cartesian product of set A & B . It is denoted by $A \times B$.

i.e. $A \times B = \{(x, y) : x \in A \text{ & } y \in B\}$

Relation:

A relation on set S & T is a set of ordered pair.

- ① $S \in S$ (S is a number of S)
- ② $t \in T$ (t is a number of T)
- ③ S & T need not be different

Eg: Suppose S is the set $\{a, b, c, d, e\}$ and set T is $\{x, y, z\}$



Then a relation on S & T is:

★ Type of Relation:

(1) Reflexive:

A relation $R \subseteq A \times A$ is reflexive if $(a, a) \in R$ for each $a \in A$.

The directed graph has a loop from each node to itself.

Eg: $A = \{1, 2, 3\}$, then

$R = \{(1, 1), (2, 2), (3, 3)\}$

(2) Symmetric / Anti-Symmetric:

A relation $R \subseteq A \times A$ is symmetric if $(b, a) \in R$ whenever $(a, b) \in R$.

Eg: Let $A = \{1, 2, 3\}$. Then

$R = \{(1, 2), (2, 1), (2, 3), (3, 2)\}$

(3) Transitive Relation:

A binary relation R is transitive if whenever $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$. The relation is transitive since, if a is ancestor of b and b is ancestor of c . Then, a is an ancestor of c .

Let, $A = \{1, 2, 3\}$
then, $R = \{(1, 1), (2, 2), (1, 3)\}$ is a transitive relation defined on set A .

④ Equivalence Relation:

A subset of $A \times A$ is called an equivalence relation on A if R satisfies the following conditions:

- ① If R is reflexive.
- ② If R is symmetric
- ③ If R is transitive.

Eg:- Let $A = \{1, 2, 3\}$
then, $R = \{(1, 1), (2, 2), (3, 3), (1, 2), (2, 1), (2, 3), (3, 2), (1, 3), (3, 1)\}$

(5) Partial order Relation:

A relation R on set S is said to be a partial order if R is reflexive, anti-symmetric & transitive. Let, $A = \{1, 2, 3\}$
Then, $R = \{(1, 1), (2, 2), (3, 3), (1, 2), (2, 3)\}$

★ Function:

- Suppose every element of S occurs exactly once as the first element of an ordered pair then the relation is called a function.
- The function otherwise known as mapping is every element in S ie every element of S is mapped to some element in the range.

★ Function as Relations:

A function $f: A \rightarrow B$ is a relation from A to B i.e subset of $A \times B$ such that each $a \in A$ belongs to a unique ordered pair (a, b) in f .

★ Function Type:

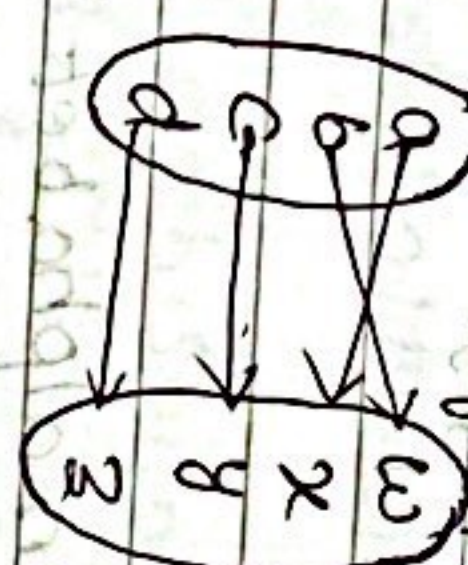
1) One to one function (Injection):
A function $f: A \rightarrow B$ is said to be one-to-one function if different elements in the domain A have distinct images in the range.

A function f is one-to-one if $f(a) = f(a')$ implies $a = a'$



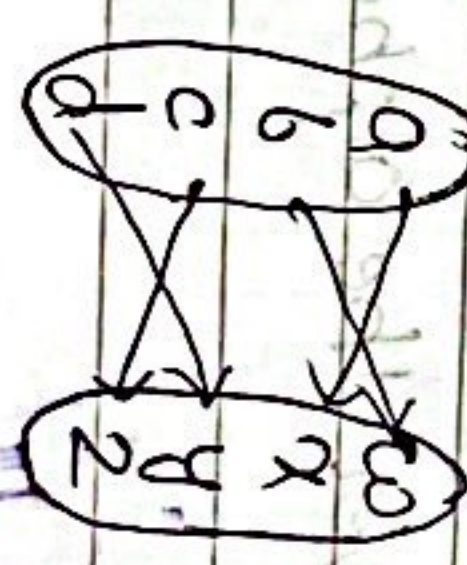
2) Onto function (Surjection):

If each element of B is the image of the some element of A , then it is called onto function. i.e $f: A \rightarrow B$ is onto if the image of f is the entire co-domain.



3) One-to-onto function (Bijection):

A function that is both one-to-one and onto is called a bijection. Such function maps each and every element of A to exactly one element of B with no element left over.



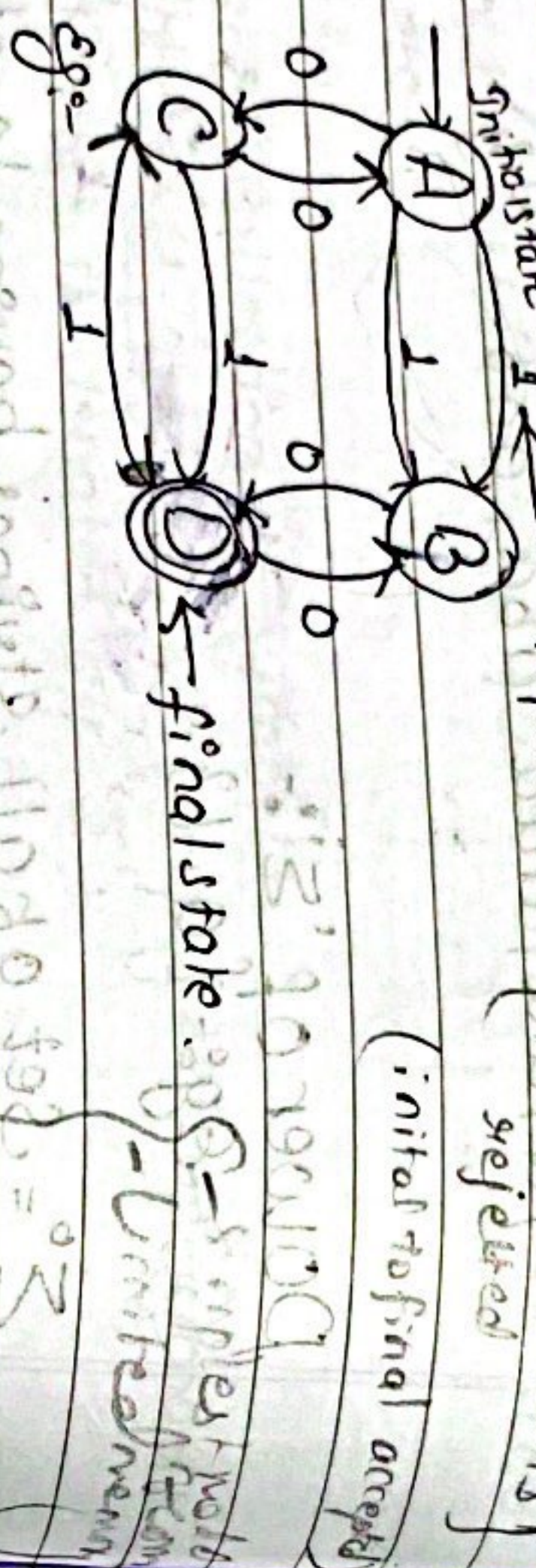
Finite Automata and Regular Expression

2.1. Finite Automata (finite state machine):

FA with output → FA without output.

Moore machine → DFA, NFA, E-NFA, machine (deterministic, non-deterministic, F.A.)

2.1.1 DFA (Deterministic Finite Automata):



Formal definition of DFA:-

A DFA is defined as a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$.

where,

Q = set of finite states.

Σ = Input alphabets / finite of I/P system.

q_0 = Start / initial state; $q_0 \in Q$.

F = Set of final state / accept state.

δ = transition function; $Q \times \Sigma \rightarrow Q$.

Example:

$$Q = \{A, B, C, D\}$$

$$\Sigma = \{0, 1\}$$

$$q_0 = \{A\}$$

$$F = \{D\}$$

	0	1
A	C	B
B	D	A
C	A	D
D	C	C

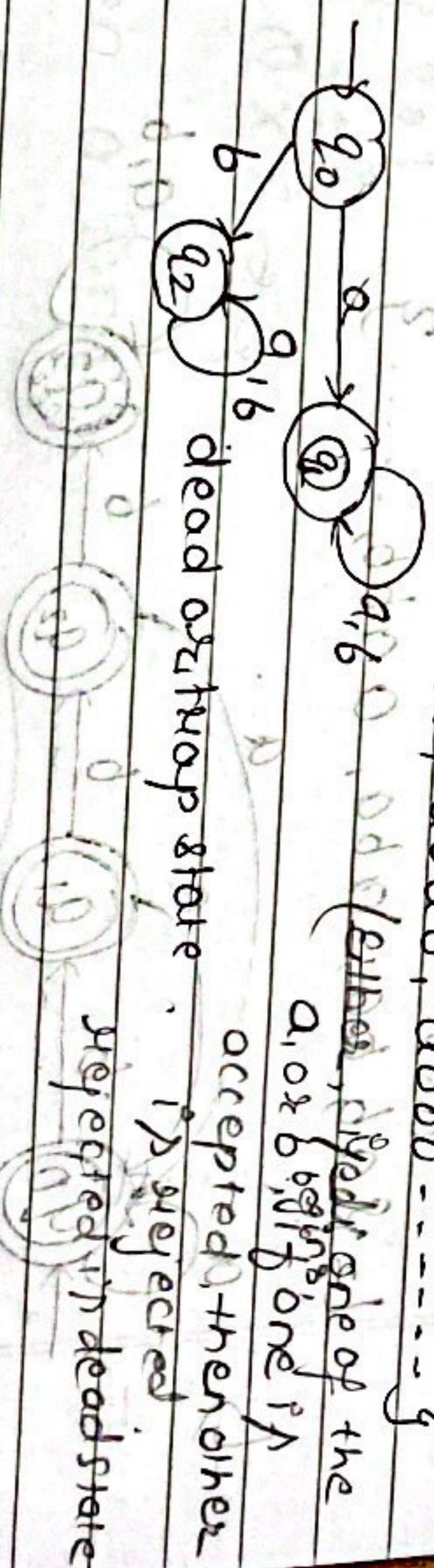
Eg: DFA with multiple ending machine, i.e.,

- Acceptance of string by DFA:-
- Scan the entire string, if we reach a final state from a initial state,
- Then we can say that the given string is accepted by the DFA.

Language Accepted:-

A FA is said to accept a language, if all the string in the language are accepted and all the string not in the language are rejected.

Eg: $L = \{a, aa, aab, ab, aaaa, abab, abbb, \dots\}$



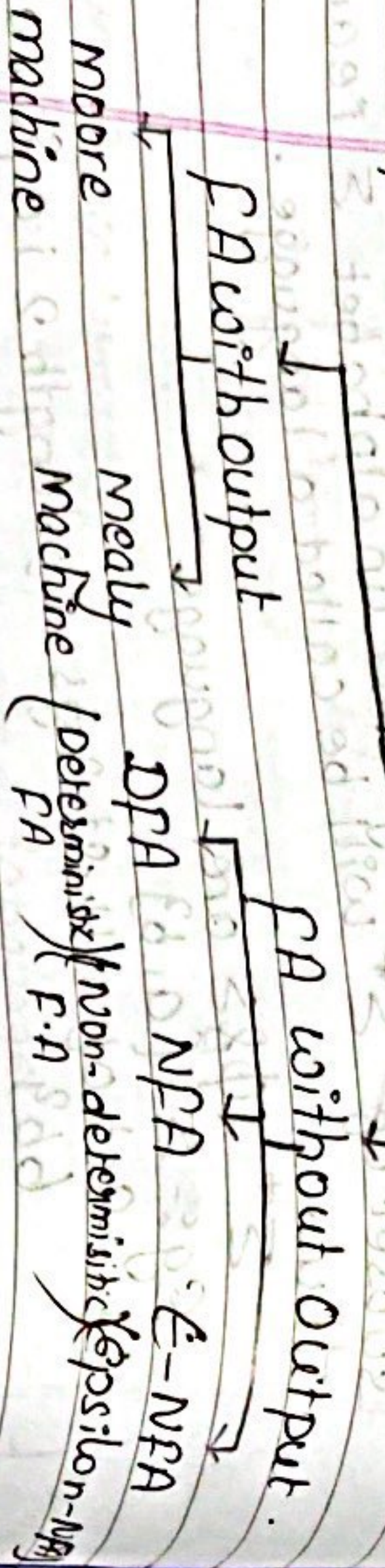
$L = \{a, aa, aab, ab, aaaa, abab, abbb, \dots\}$

$L = \{a, aa, aab, ab, aaaa, abab, abbb, \dots\}$

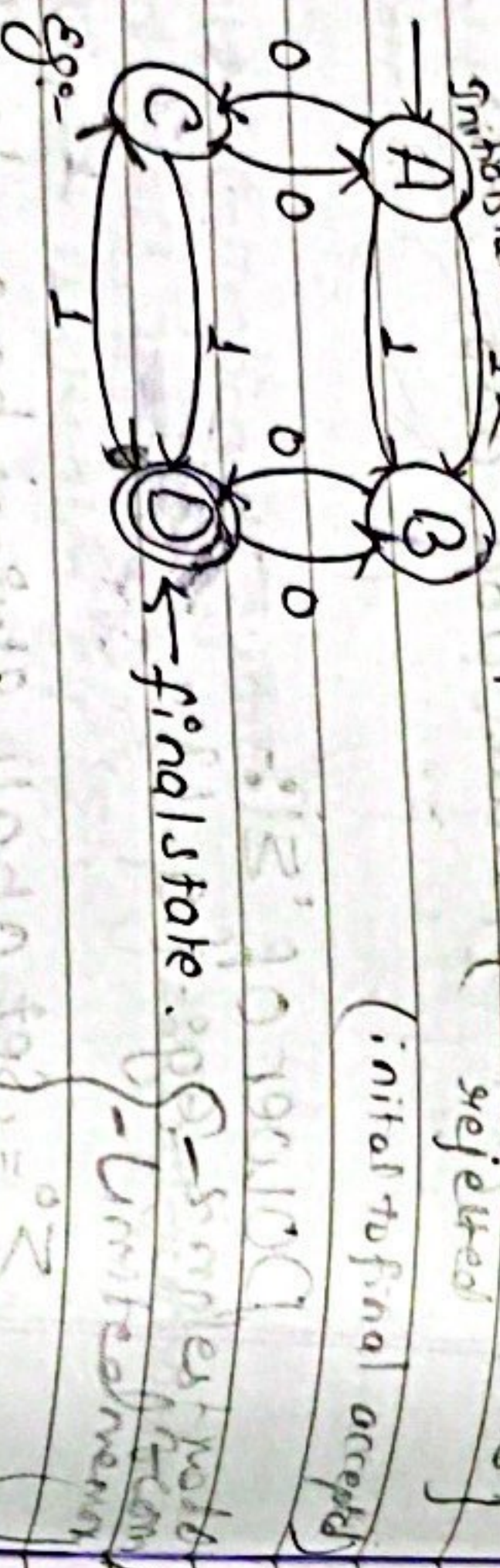
	a	b
q0	q1	
q1	q2	
q2	q3	
q3	q4	
q4	q5	
q5	q6	
q6	q1	

Finite Automata and Regular Expression

2.1 Finite Automata (finite state machine):



2.1.1 DFA (Deterministic Finite Automata):



Formal definition of DFA:-

A DFA is defined as a 5-tuple.

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

Where,

Q = Set of finite states.

Σ = Input alphabets / finite of i/p system.

q_0 = Start / initial state; $q_0 \in Q$

F = Set of final state / accept state.

δ = transition function; $Q \times \Sigma \rightarrow Q$.

from eg:-

$$Q = \{A, B, C, D\}$$

$$\Sigma = \{0, 1\}$$

$$q_0 = \{A\}$$

$$F = \{D\}$$

	0	1
A	C	B
B	D	A
C	A	D
D	C	C

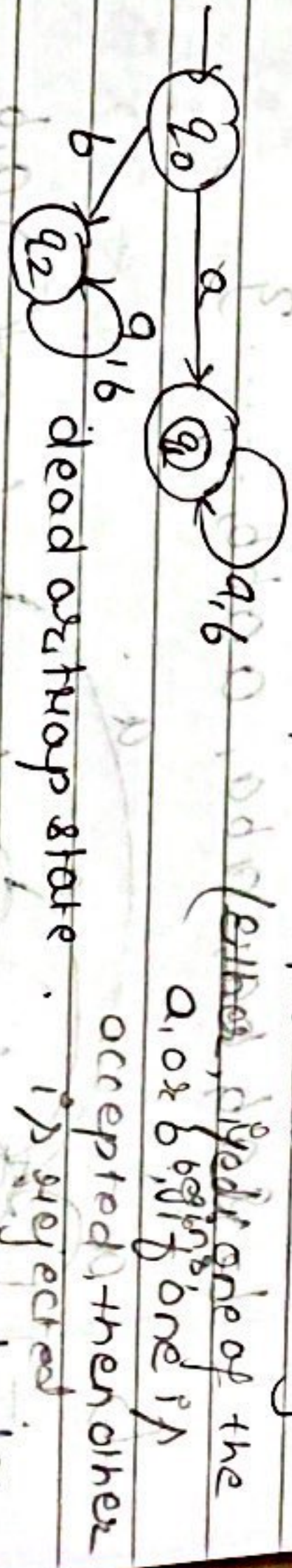
eg: DFA with prediction machine, but:

- Acceptance of string by DFA:-
- Scan the entire string, if we reach a final state from a initial state,
- Then we can say that the given string is accepted by the DFA.

Language Accept:-

A FA is said to accept a language, if all the string in the language are accepted and all the string not in the language are rejected.

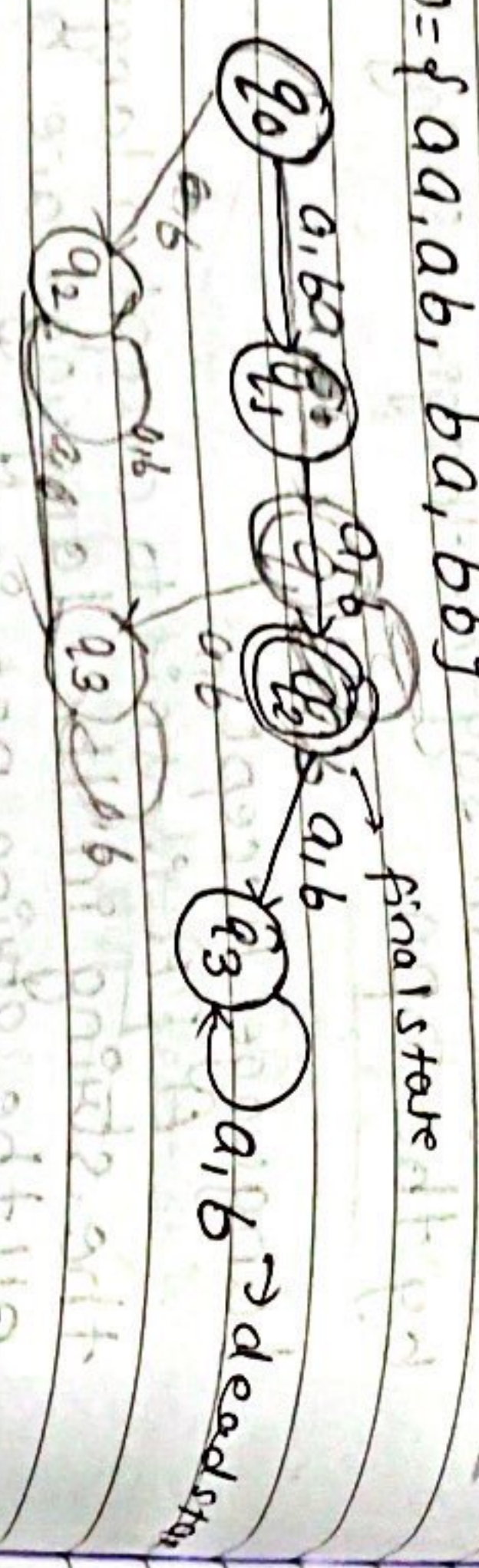
eg:-
 L_g = Set of all string that begins with 'a' over alphabet $\{a, b\}$
 $= \{a, aa, aab, ab, aaaa, abab, abbb, \dots\}$



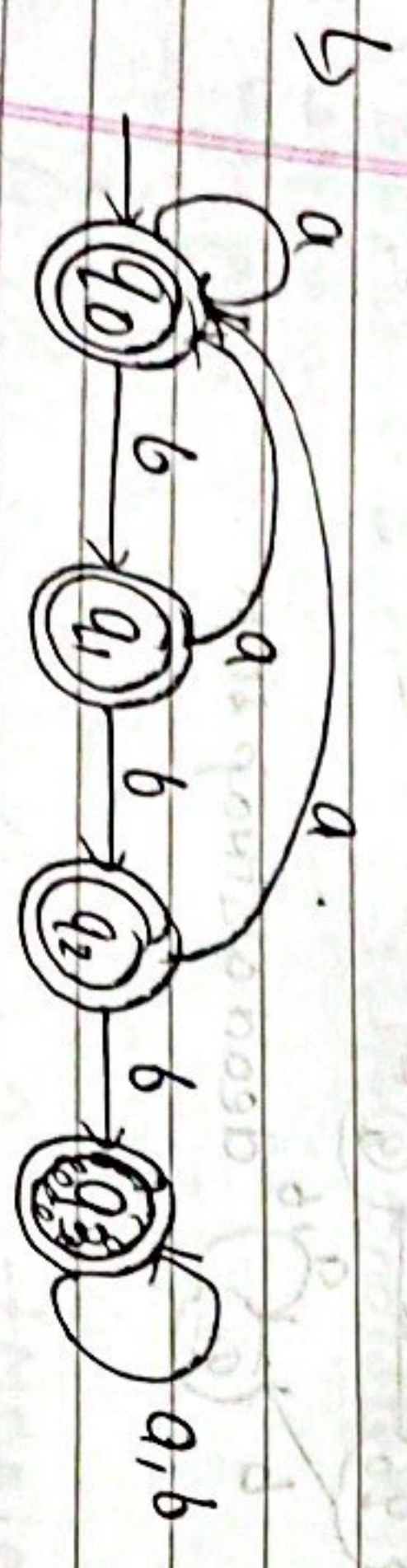
	a	b
q0	q1	q2
q1	q1	q2
q2	q2	q2

Design a DFA which accept the strings over alphabet $\{a, b\}$ having length 2.

$L = \{aa, ab, ba, bb\}$



Q Design a DFA that accept the language given $L = \{w \in \{a, b\}^* : w \text{ doesn't contain three consecutive } b's\}$
 $L = \{ababba, abba, aabbb, \dots\}$



The above DFA can be formalize defined.

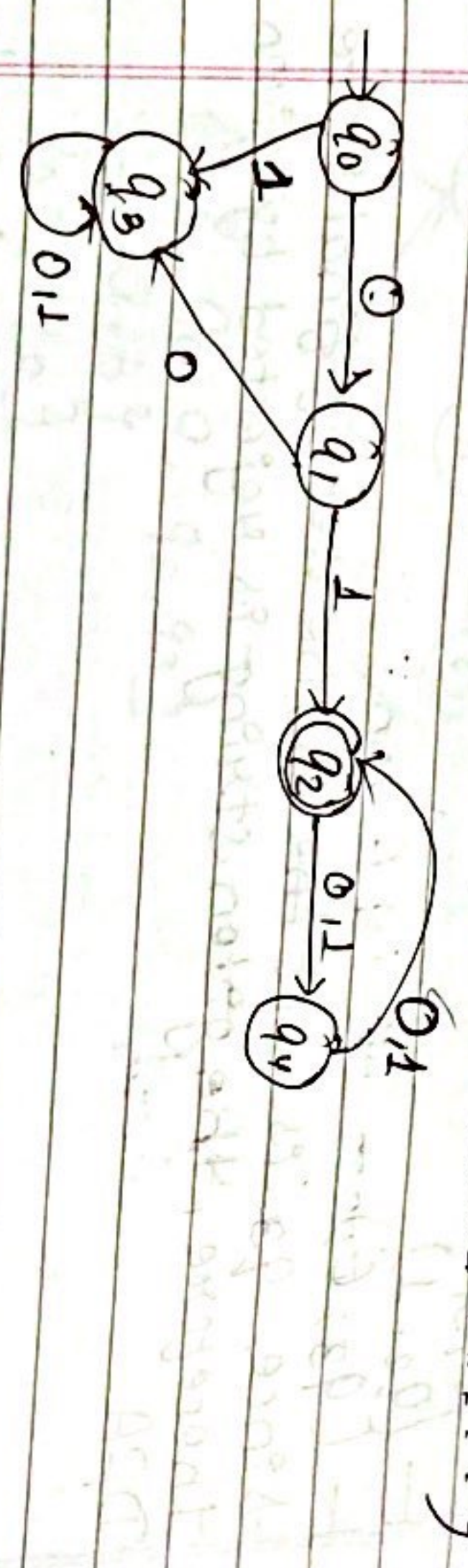
$M = (Q, \Sigma, q_0, f, \delta)$
 where $Q = \{q_0, q_1, q_2, q_3\}$
 $\Sigma = \{a, b\}$
 $q_0 = q_0$
 $f = \{q_3\}$
 $\delta = \{ (q_0, a) \rightarrow q_0, (q_0, b) \rightarrow q_1, (q_1, a) \rightarrow q_1, (q_1, b) \rightarrow q_2, (q_2, a) \rightarrow q_3, (q_2, b) \rightarrow q_4, (q_3, a) \rightarrow q_3, (q_3, b) \rightarrow q_4, (q_4, a) \rightarrow q_4, (q_4, b) \rightarrow q_4 \}$

Transition table $\rightarrow$

	q0	q1	q2	q3
q0	q0	q1		
q1	q1	q1	q2	
q2	q3	q3	q3	
q3	q3	q3	q3	q3

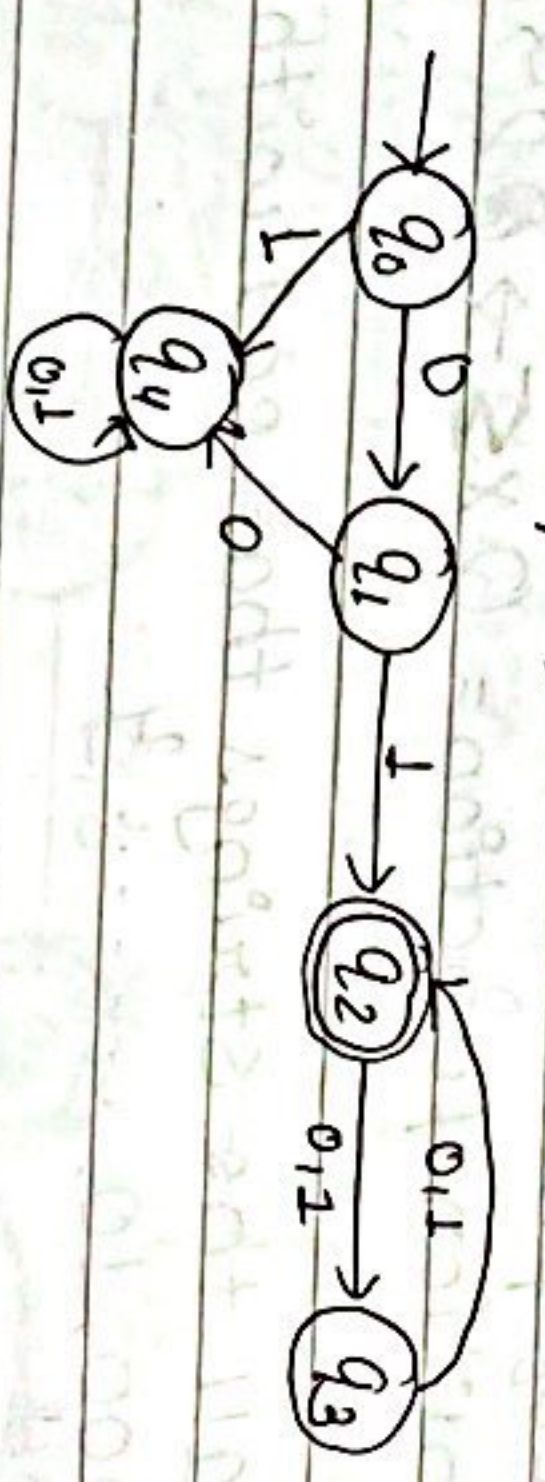
'f' turnstile, $\{S, T\} \rightarrow T$ is provable from S .

Q Design a DFA that accept the language given $L = \{w \in \{0, 1\}^* : w \text{ start with } 01 \text{ having even length}\}$
 $L = \{01, 0111, 010100, 0111101, \dots\}$



$Q = \{q_0, q_1, q_2, q_3, q_4\}$
 $\Sigma = \{0, 1\}$
 $q_0 = q_0$
 $f = \{q_2\}$
 $\delta = \{ (q_0, 0) \rightarrow q_1, (q_0, 1) \rightarrow q_3, (q_1, 1) \rightarrow q_2, (q_1, 0) \rightarrow q_4, (q_2, 0) \rightarrow q_2, (q_2, 1) \rightarrow q_2, (q_3, 0) \rightarrow q_4, (q_3, 1) \rightarrow q_4, (q_4, 0) \rightarrow q_3, (q_4, 1) \rightarrow q_3 \}$

Q Given a DFA,



$L = \{w : \{0, 1\}^* : w \text{ are all the string starting with } 01 \text{ having even by the length}\}$
 Check whether the given string 0111011 is accepted or rejected.
 0111011
 0111011
 0111011

$\rightarrow (q_2, 11011)$
 $\rightarrow (q_3, 1011)$
 $\rightarrow (q_3, 011)$
 $\rightarrow (q_3, 11)$
 $\rightarrow (q_2, 1)$
 $\rightarrow (q_3, \epsilon)$
 Since, q_3 is not the in the set of final state.
 Therefore, the given string is rejected by given DFA.

* Non-Deterministic Finite Automata (NFA)

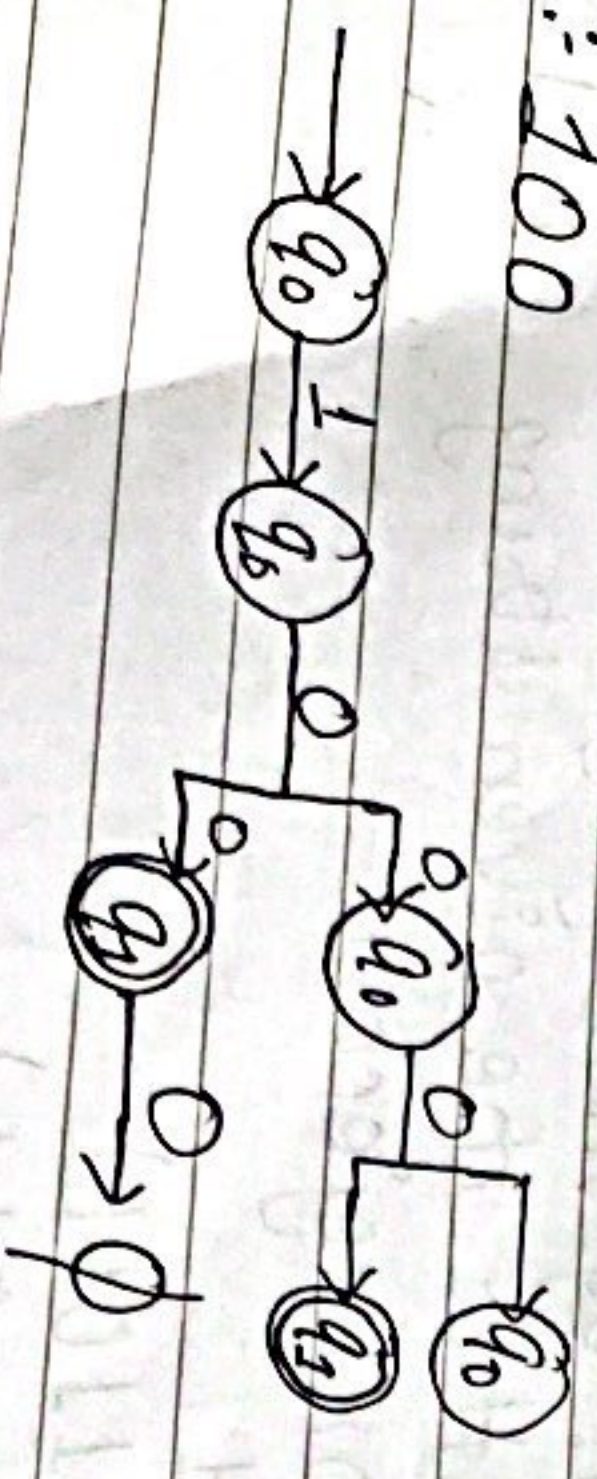
NFA is defined by 5-tuple $\{Q, \Sigma, q_0, F, \delta\}$

- where,
- $Q = \{ \text{No. of finite states} \}$
- $\Sigma = \{ \text{finite no. of input symbols / alphabets} \}$
- $q_0 = \text{initial / start state}$
- $F = \{ \text{Set of final states} \}$
- $\delta = \text{Transition function} = Q \times \Sigma \rightarrow Q$

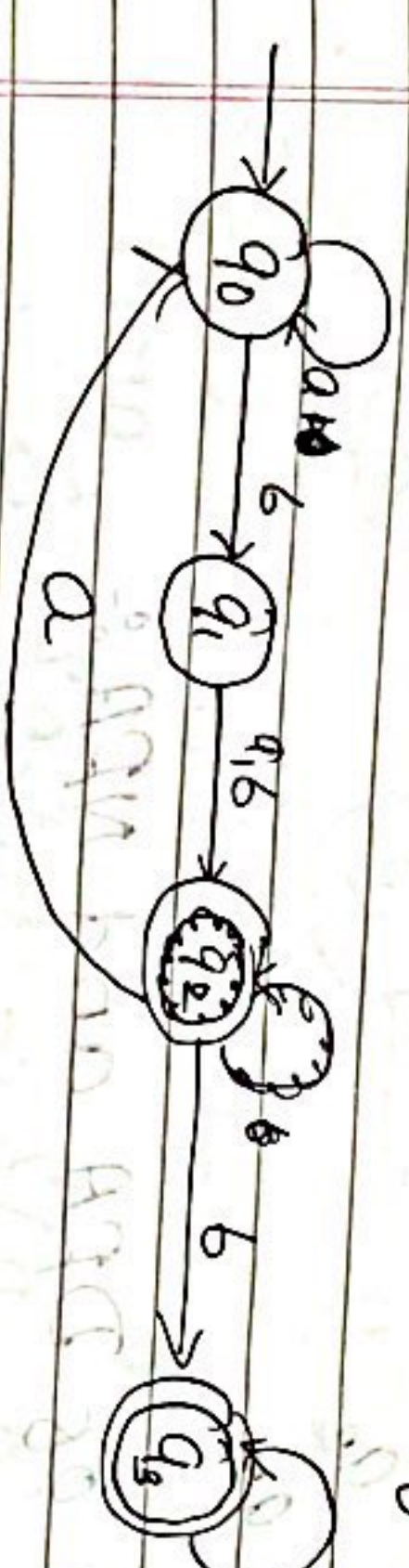
$L = \{ \text{Set of all the strings that end with '0'} \}$



eg: 100



Design a NFA that accepts the set of strings containing the occurrence of string 'abb' or 'bab'.



$Q = \{ q_0, q_1, q_2, q_3 \}$
 $\Sigma = \{ a, b \}$
 $q_0 = \text{start state}$
 $F = \{ q_4 \}$

q_0 to q_1 on 'a'
 q_1 to q_2 on 'b'
 q_2 to q_3 on 'b'
 q_3 to q_4 on 'a'

Design a NFA over $\{a, b\}$ that accepts strings having 'aaba' as substring.



$Q = \{ q_0, q_1, q_2, q_3 \}$
 $\Sigma = \{ a, b \}$
 $q_0 = \text{start state}$
 $F = \{ q_4 \}$

q_0 to q_1 on 'a'
 q_1 to q_2 on 'a'
 q_2 to q_3 on 'b'
 q_3 to q_4 on 'a'

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

Equivalence of DFA and NFA :-

For each NFA, there is an equivalent deterministic finite automata.

- NFA can always be converted into a equivalent deterministic one.
- Any language accepted by a NFA can also be accepted by some DFA. For this we describe an algorithm that takes any NFA and convert it into a DFA that accept the same language. This algorithm is called Subset Construction algorithm.

Algorithm steps :-

- Observe all the transitions from starting state q_0 for every symbol in Σ .
- For new states from (1) check all transitions for every Σ .
- Repeat step 2 till new steps are appeared.

Formally, we say that two finite automata M_1 and M_2 (NFA & DFA) are equivalent if $L(M_1) = L(M_2)$. Thus, two automata are considered to be equivalent if they accept the same language even though they may use different method to do so.

Eg:- Given a NFA :-



Step-1 :- Initial state q_0 for new DFA $\{q_0\}$
 Now

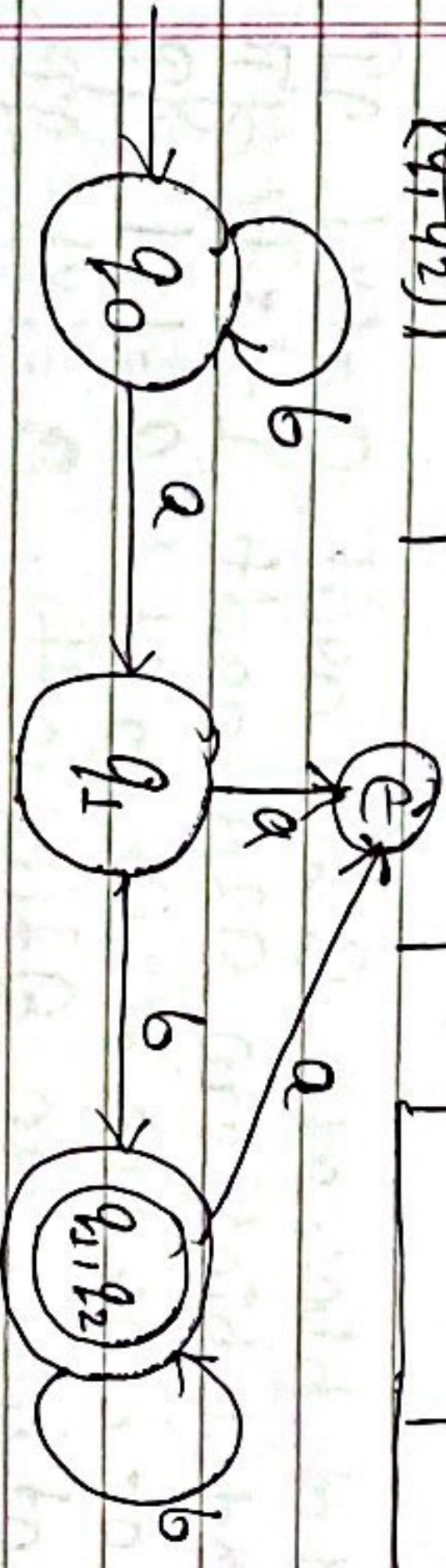
$\delta'(\{q_0\}, a) \Rightarrow \delta(q_0, a) = \{q_1\} \Rightarrow$ new state.
 $\delta'(\{q_0\}, b) \Rightarrow \delta(q_0, b) = q_0$

Step-2 :- For new state $\{q_1\}$

$\delta'(\{q_1\}, a) \Rightarrow \delta(q_1, a) = \emptyset$
 $\delta'(\{q_1\}, b) \Rightarrow \delta(q_1, b) \Rightarrow \{q_1, q_2\}$ new state

Step-3 :- For new state $\{q_1, q_2\}$
 $\delta'(\{q_1, q_2\}, a) \Rightarrow \delta(q_1, a) \cup \delta(q_2, a) = \emptyset \cup \emptyset = \emptyset$
 $\delta'(\{q_1, q_2\}, b) \Rightarrow \delta(q_1, b) \cup \delta(q_2, b) = \{q_1, q_2\} \cup \emptyset = \{q_1, q_2\}$ # Transition table

State \ Symbol	a	b
q_0	q_1	q_0
q_1	$\emptyset$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	$\emptyset$	$\{q_1, q_2\}$



Difference between DFA and NFA

DFA

1. Dead configuration is not allowed.

2. Multiple choice are not available corresponding to an input.

3. ϵ -move is not allowed.

4. Digital computer are deterministic.

5. Designing and understanding is difficult.

NFA

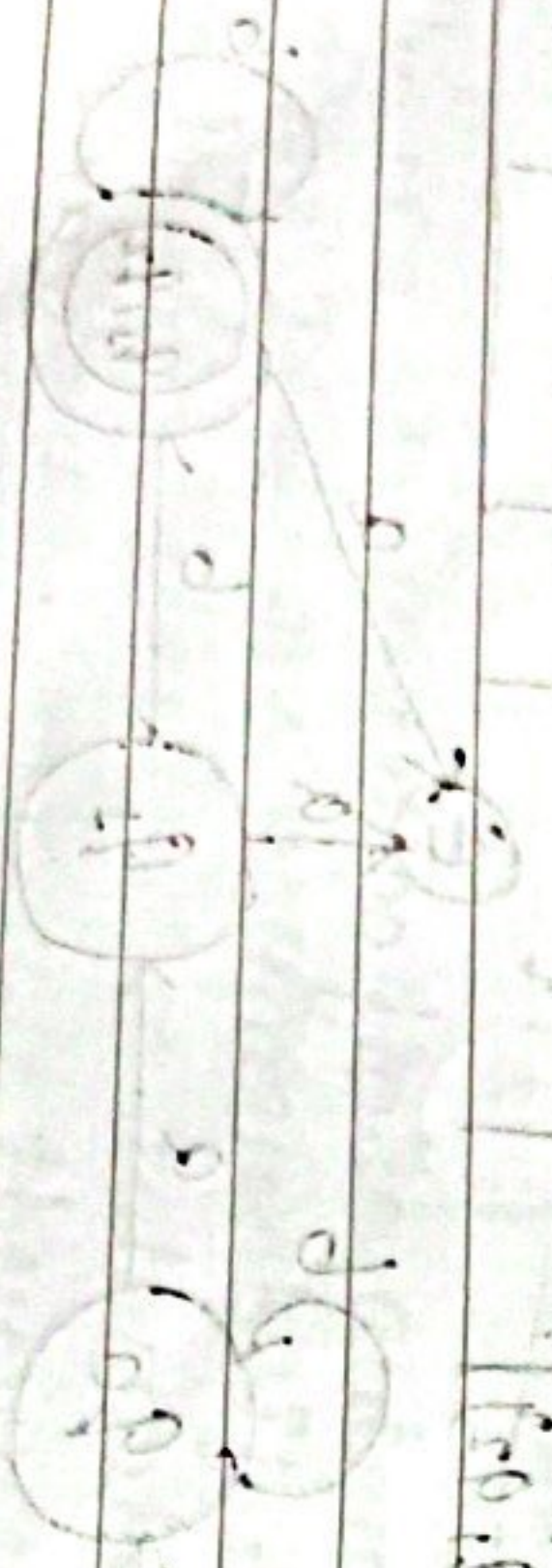
- 1) Dead configuration is allowed.

- 2) Multiple choices are available corresponding to an input.

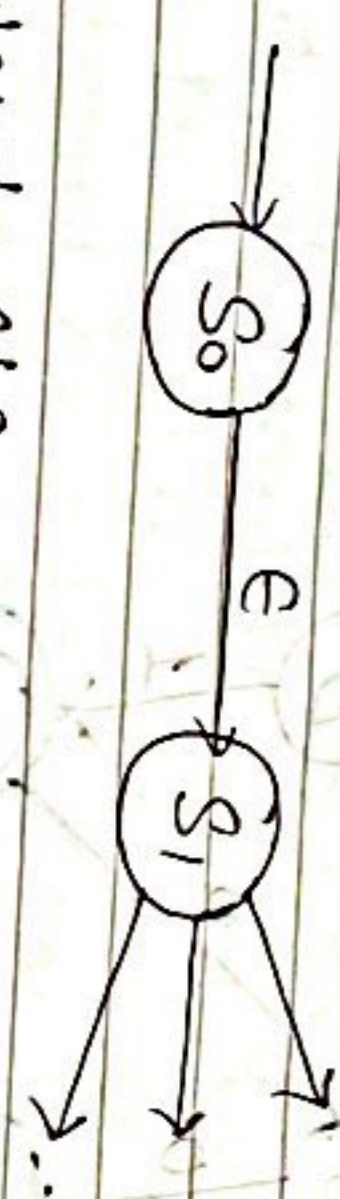
- 3) ϵ -move is allowed.

- 4) Non-Deterministic feature is not associated with real computers.

- 5) Designing and understanding is easy.



Epsilon-NFA (ϵ -NFA) (Eliminating ϵ -moves)



Rules to eliminate ϵ -moves:

- 1) Find all the edges starting from S_i .
- 2) Duplicate all edges to S_0 without changing edge.
- 3) If S_0 is initial state, make S_i initial.
- 4) If S_i is final state make S_0 final state.
- 5) Remove dead states.

Minimization of DFA:-

Minimization of DFA is required to obtain the minimal version of any DFA, which consists of the minimum number of possible states.

Need to combine two states if and only if two states are equivalent.

Two states 'A' and 'B' are said to be equivalent if $\delta(A, X) \rightarrow F$ and $\delta(B, X) \rightarrow F$ or $\delta(A, X) \rightarrow F$ and $\delta(B, X) \rightarrow F$.

where 'X' is any possible string.

If $|X|=0$, then A & B are said to be '0' equivalence.
If $|X|=1$, then A & B are said to be '1' equivalence.
If $|X|=2$, then A & B are said to be '2' equivalence.
If $|X|=n$, then A & B are said to be 'n' equivalence.

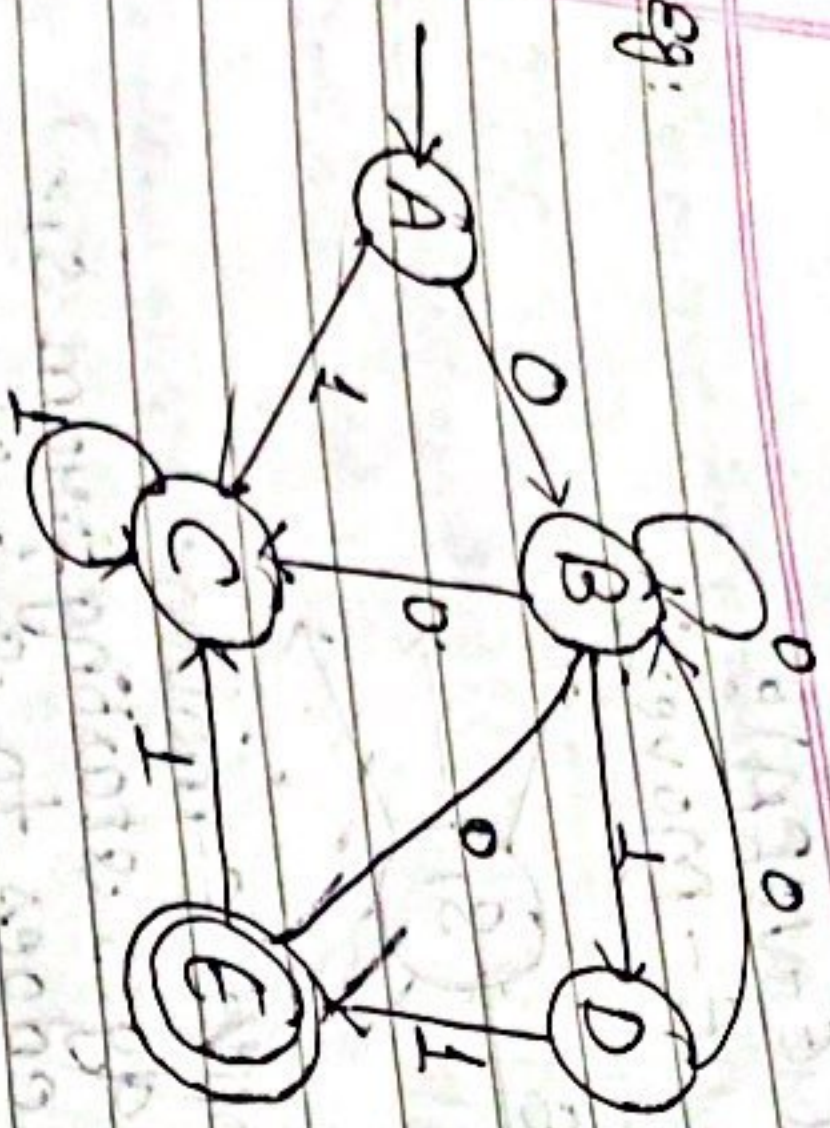


Λ (empty string).

Regular Expressions

- An expression used to generate string for finite automata (FA) is called regular expression.
- It is a formula for representing a language in terms of a form combined using three operations i.e. union, concatenation & closure.
- It describes a language exclusively by means of single symbols and ϕ , combined perhaps with symbols $|$ and $*$. Possibly with the help of parentheses.
- We give a formal definition of regular expressions over Σ as follows:
 1. Any terminal symbol ϵ is an element of Σ .
 2. The union of two regular expressions R_1 and R_2 written as $R_1 + R_2$ is also a regular expression.
 3. The concatenation two regular expressions R_1 and R_2 written as $R_1 R_2$ is also a regular expression.
 4. The iteration (closure) of a regular expression R , written as R^* is also a regular expression.
 5. If R is a regular expression then (R) is also a regular expression.
 6. The regular expression over Σ are precisely those obtained recursively by the application of the above rules several times.

eg:



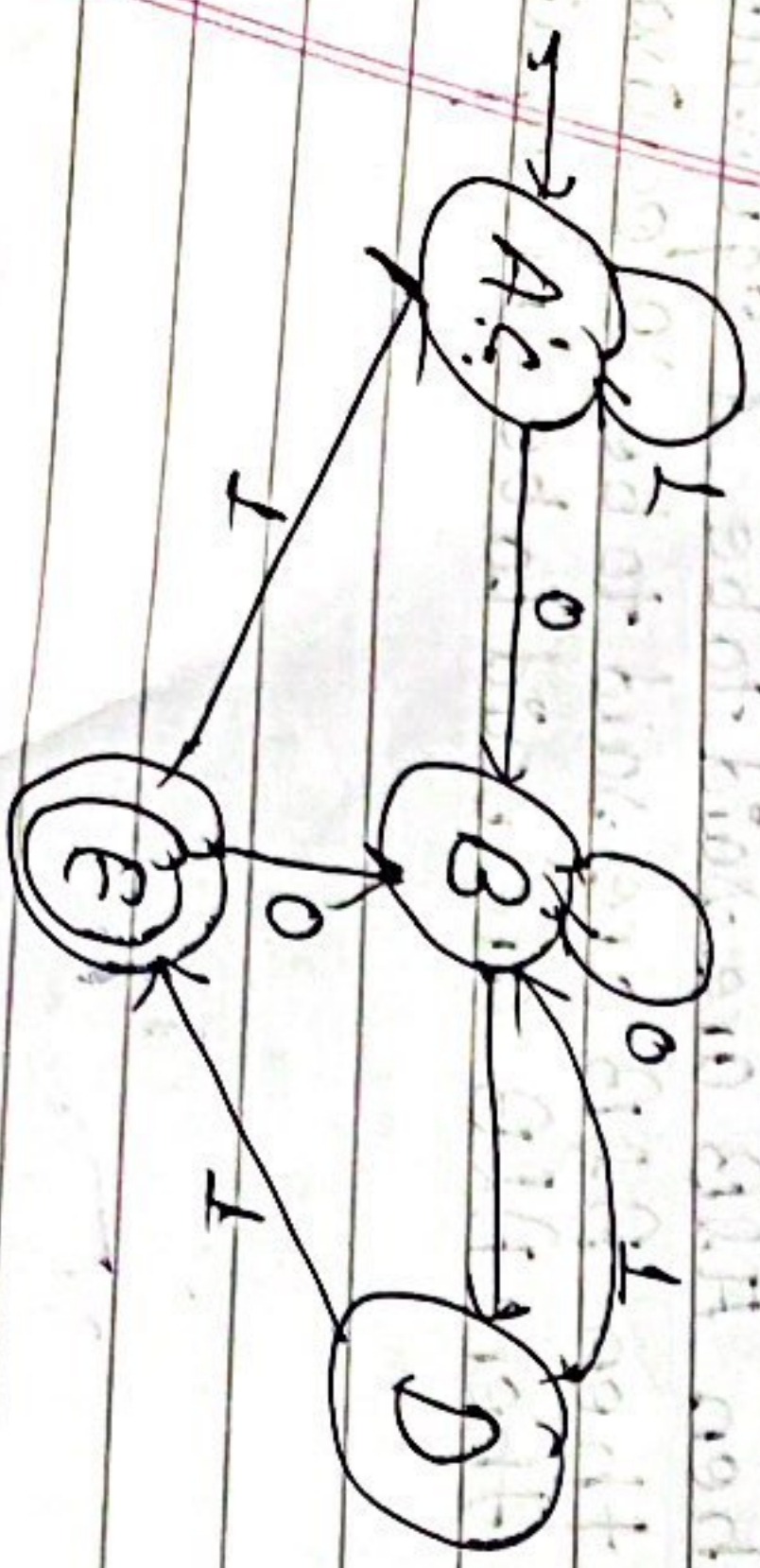
Start Symbol

$\rightarrow A$	0	1
B	B	D
C	B	C
D	B	E
E	B	C

Steps:-

non-final final

1. 0' Regu: $\{A, B, C, D\}^*$ $\{E\}$
2. 1' Regu: $\{A, B, C, D\}^* \{E\}$
3. 2' Regu: $\{A, C\}^*, \{B\}^*, \{D\}^*, \{E\}^*$ same consecutive step.
4. 3' Regu: $\{A, C\}^*, \{B\}^*, \{D\}^*, \{E\}^*$ same consecutive step.



eg 1:- $L = \{a^n b^m : n \text{ is even, } m \text{ is odd}\}$
followed by odd of 'b's'
 $L = \{b, aab, aabbb, aaaaab, \dots\}$
 $\therefore L = (aa)^* (b)^+$

closure iteration	
$a^0 = \epsilon$	$(aa)^0 = \epsilon$
$a^1 = a$	$(aa)^1 = aa$
$a^2 = aa$	$(aa)^2 = aaaa$
$a^3 = aaa$	$(aa)^3 = aaaaaa$
$a^4 = aaaa$	$(aa)^4 = aaaaaaaa$

eg 2:- $L = \{a^n b^m : n \text{ has strings of 0's and 1's ending in 00}\}$

$L = \{0100, 110100, 1001100, \dots\}$
 $\therefore L = (01)^* 00$

$L = \{a^n b^m : n \text{ has strings of 0's and 1's beginning with '0' and ending with '1'}\}$
 $L = \{01, 001, 010011, 0011111, \dots\}$

$\therefore L = 0(01)^*$

The relation between r.e. and the language L. Such that if α is any r.e. then $L(\alpha)$ is language represented by α .

- The function L is defined as follows:-
 - $L(\emptyset) = \emptyset$ and $L(a) = \{a\}$ for each $a \in \Sigma$
 - If α and β are r.e. then $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$
 - If α and β are r.e. then $L(\alpha \beta) = L(\alpha)L(\beta)$
 - If α is regular expression then $L(\alpha^*) = L(\alpha)^*$

Regular Operator:- Operator used to generate the language that are regular.

Union:-

If L_1 and L_2 are any two regular languages.

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

eg:-

$L_1 = \{00, 11\}$, $L_2 = \{e, 10\}$
 $L_1 \cup L_2 = \{00, 11, e, 10\}$

Concatenation:-

If L_1 and L_2 are both regular language.

$L_1 \cdot L_2 = \{s_1 s_2 : s_1 \in L_1 \text{ and } s_2 \in L_2\}$

eg:-

$L_1 = \{00, 11\}$ and $L_2 = \{e, 01\}$
 $L_1 \cdot L_2 = \{00, 011, 11, 1101\}$

Regular language:-

Let Σ be an alphabet, the class of regular language over Σ is defined as:-

- ϕ is a reg. language representing empty language
- $\{e\}$ is a reg. language representing language of empty string
- For each $a \in \Sigma$, $\{a\}$ is a regular language
- If $L_1, L_2, \dots, L_n$ is a regular language then $L_1 L_2 L_3 \dots L_n$ is a regular language
- If L is a regular language, then L^* is a regular language

Identifies theorem of Regular expression

- 1) $\phi + R = R$
- 2) $\phi R + R\phi = \phi$
- 3) $\phi R + R\phi = R$
- 4) $\phi^* = \phi$ & $\phi^* = \phi$
- 5) $R + R = R$
- 6) $R^* R^* = R^*$
- 7) $R R^* = R^* R$
- 8) $(R^*)^* = R^*$
- 9) $\epsilon + R R^* = \epsilon + R^* R = R^*$
- 10) $(p q)^* = p (q p)^* q$
- 11) $(p + q)^* = (p^* q^*)^* = (q^* p^*)^*$
- 12) $(p + q) R = p R + q R$ and $R (p + q) = R p + R q$

★

Arden's Theorem:-
Let 'p' and 'q' be the two regular expression over alphabet ' Σ '. if 'p' doesn't contain any empty string then,

Proof
 $x = q + x p$ has a unique soln $x = q p^*$

Let, $x = q + x p$ — (1)
Now, put the value of x again in right side of the equation (1)

$$x = q + (q + x p) p = q + q p + x p^2 \text{ — (II)}$$

again putting the value of 'x' in eqn (II)

$$x = q + q + q p + (q + x p) p^2$$

$$= q + q p + q p^2 + x p^3$$

Combining in the same way, we will get as

$$x = q + q p + q p^2 + q p^3 + \dots$$

$$x = q (\epsilon + p + p^2 + p^3 + \dots)$$

$$x = q p^*$$

Hence proved

Uses of Arden's rule to find the R.E for DFA
 ↳ To convert the given DFA into R.E, here are some of assumptions regarding the transition system.

- The transition diagram should not have the ϵ -transition.
- There must be only one initial state.
- The vertex or the state in the DFA are $q_1, q_2, \dots, q_n$ (Any q_i is final state).
- w_{ij} denotes the R.E representing the set of labels of the edges from q_i to q_j .

then,

$$q_1 = q_1 w_{11} + q_2 w_{21} + q_3 w_{31} + \dots + q_n w_{n1} + \epsilon$$

$$q_2 = q_1 w_{12} + q_2 w_{22} + q_3 w_{32} + \dots + q_n w_{n2} + \epsilon$$

$$q_n = q_1 w_{1n} + q_2 w_{2n} + q_3 w_{3n} + \dots + q_n w_{nn} + \epsilon$$

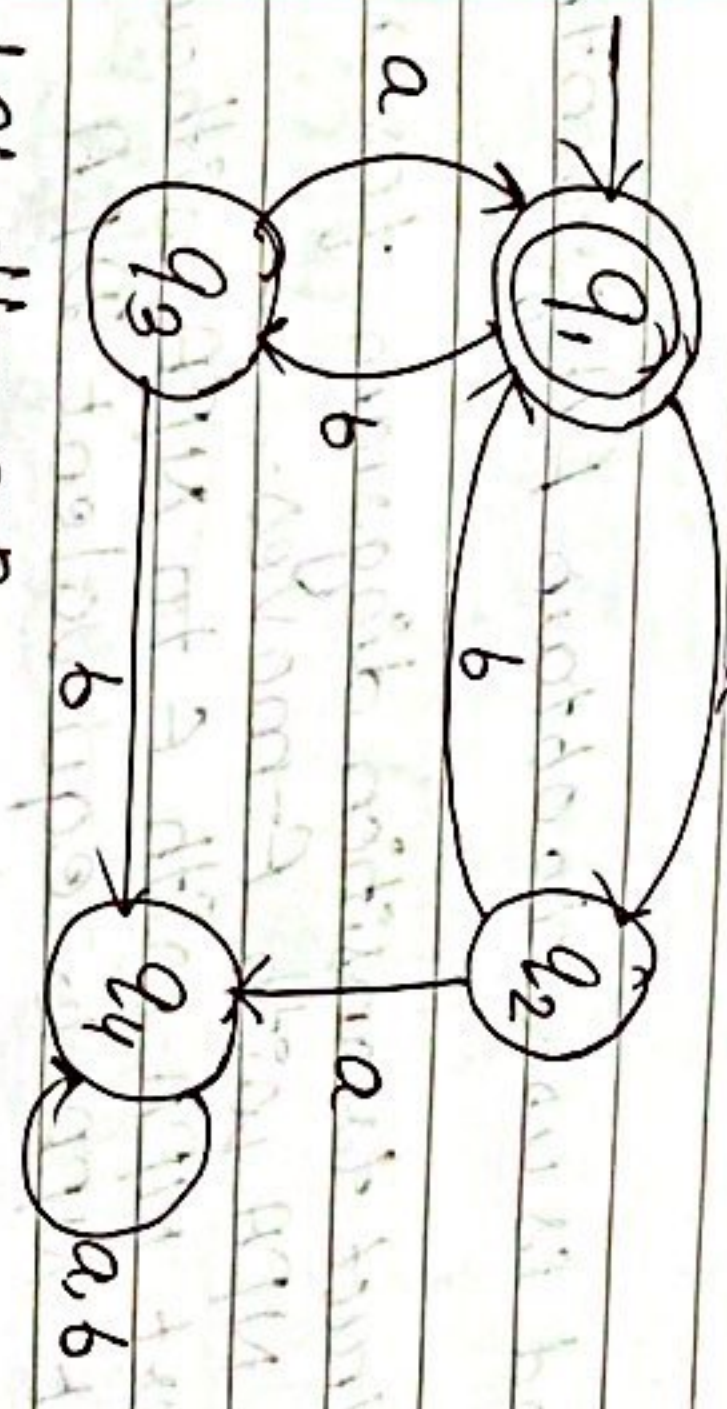
(i) $q_1 + q_2 + q_3 + \dots + q_n = x$

$$q_1 = q_1 w_{11} + q_2 w_{21} + q_3 w_{31} + \dots + q_n w_{n1} + \epsilon$$

$$q_2 = q_1 w_{12} + q_2 w_{22} + q_3 w_{32} + \dots + q_n w_{n2} + \epsilon$$

$$q_n = q_1 w_{1n} + q_2 w_{2n} + q_3 w_{3n} + \dots + q_n w_{nn} + \epsilon$$

Convert the following DFA into regular expression



Let the eqn.

$$q_1 = \epsilon + q_2 b + q_3 d \quad \text{--- (I)}$$

$$q_2 = q_1 d \quad \text{--- (II)}$$

$$q_3 = q_1 b \quad \text{--- (III)}$$

$$q_4 = q_2 a + q_3 b + q_4 a + q_4 b \quad \text{--- (IV)}$$

Take q_1 , because it is final state.

$$q_1 = \epsilon + q_2 b + q_3 a$$

Substituting the value of q_2 and q_3 from (II) & (III) in eqn (I)

$$q_1 = \epsilon + q_1 d b + q_1 b a$$

$$q_1 = \epsilon + q_1 (d b + b a)$$

$$R = \epsilon (d b + b a)^*$$

$$R = (a b + b a)^*$$

Conversion rules for Regular Expression to finite Automata.

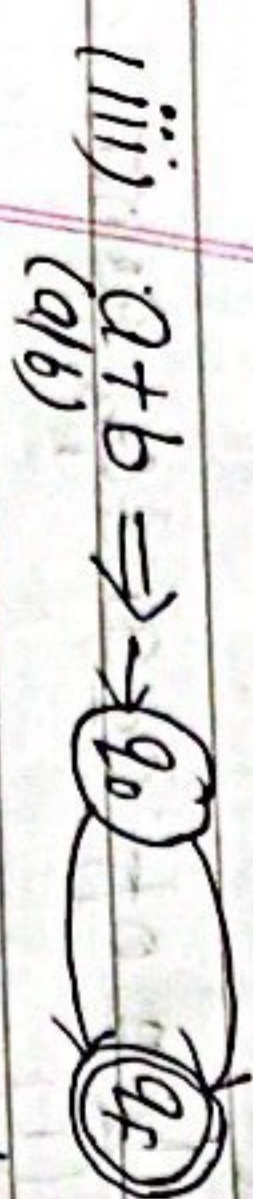
⇒ subset method is used to obtain FA from the given RE.

Step 1: Construct transition diagram for a given RE. by using NFA with ϵ -moves.
Step 2: Convert NFA with ϵ to NFA without ϵ .
Step 3: Convert NFA to equivalent DFA.

Some basic rules are as follows:

i) $\phi \Rightarrow$ 

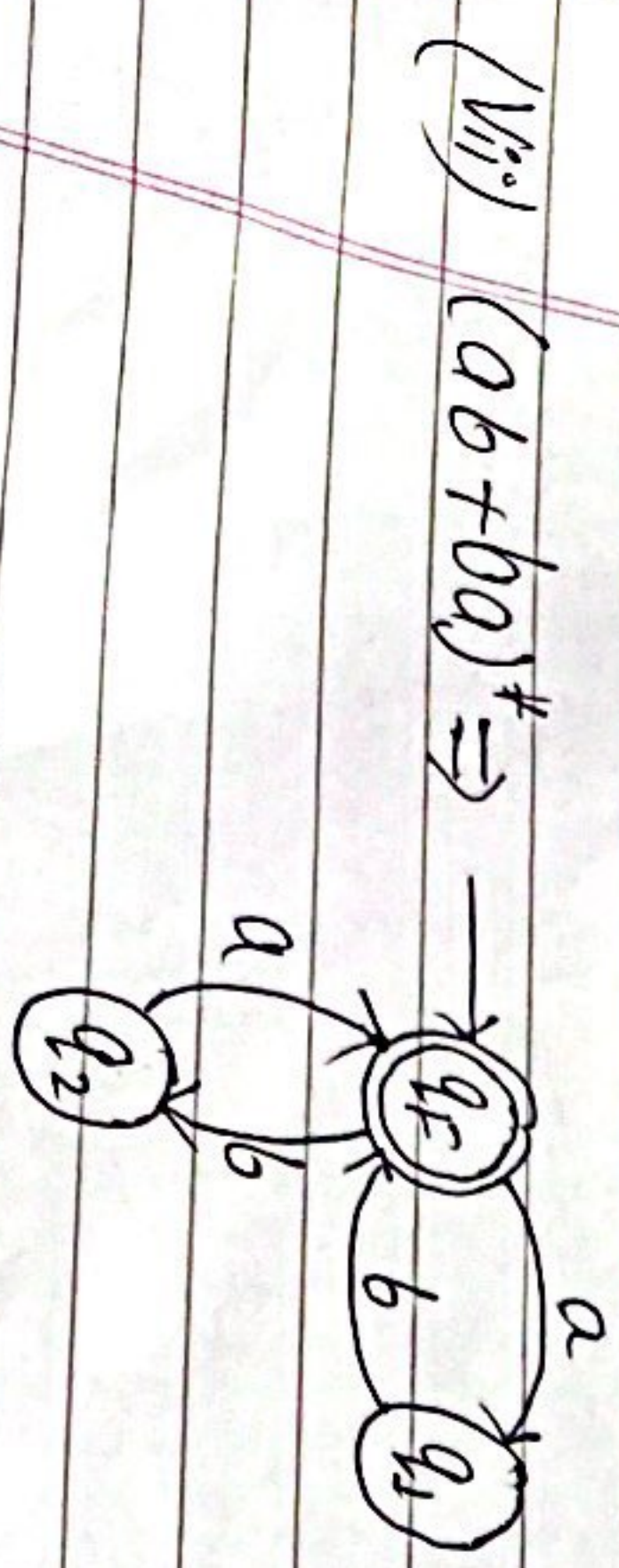
ii) $a \Rightarrow$ 

iii) $a+b \Rightarrow$ 

iv) $ab \Rightarrow$ 

v) $ab^* \Rightarrow$ 

vi) $(ab)^* \Rightarrow$ 

vii) $(a+b)^* \Rightarrow$ 

$b^* = \{ \epsilon, b, bb, bbb, \dots \}$
positive closure $b^+ = \{ b, bb, bbb, \dots \}$

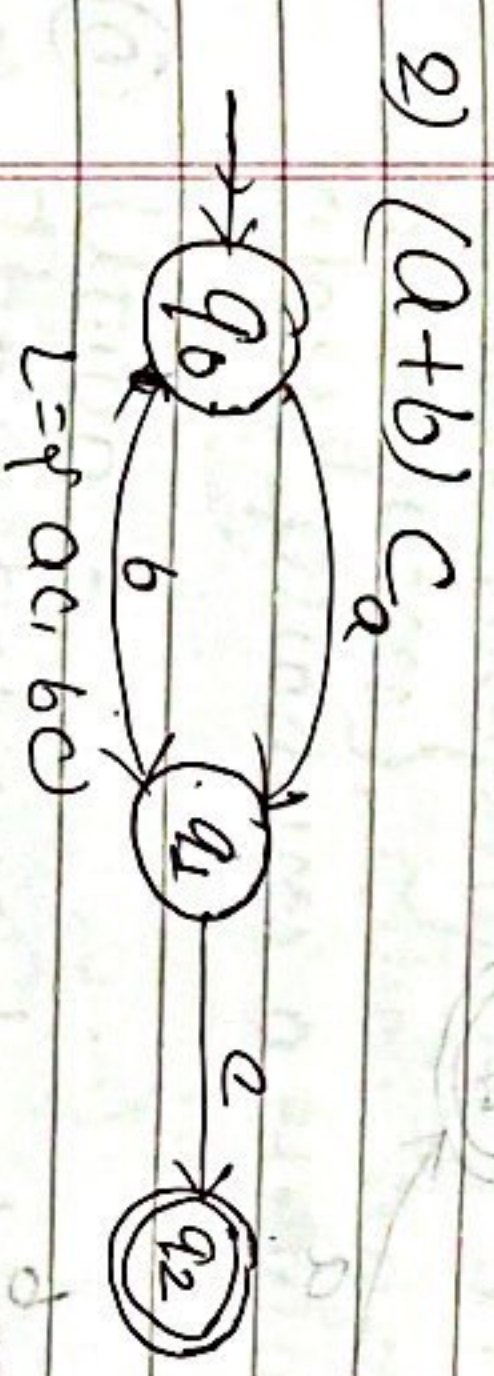
1) Convert RE to DFA.

a) ab^*c



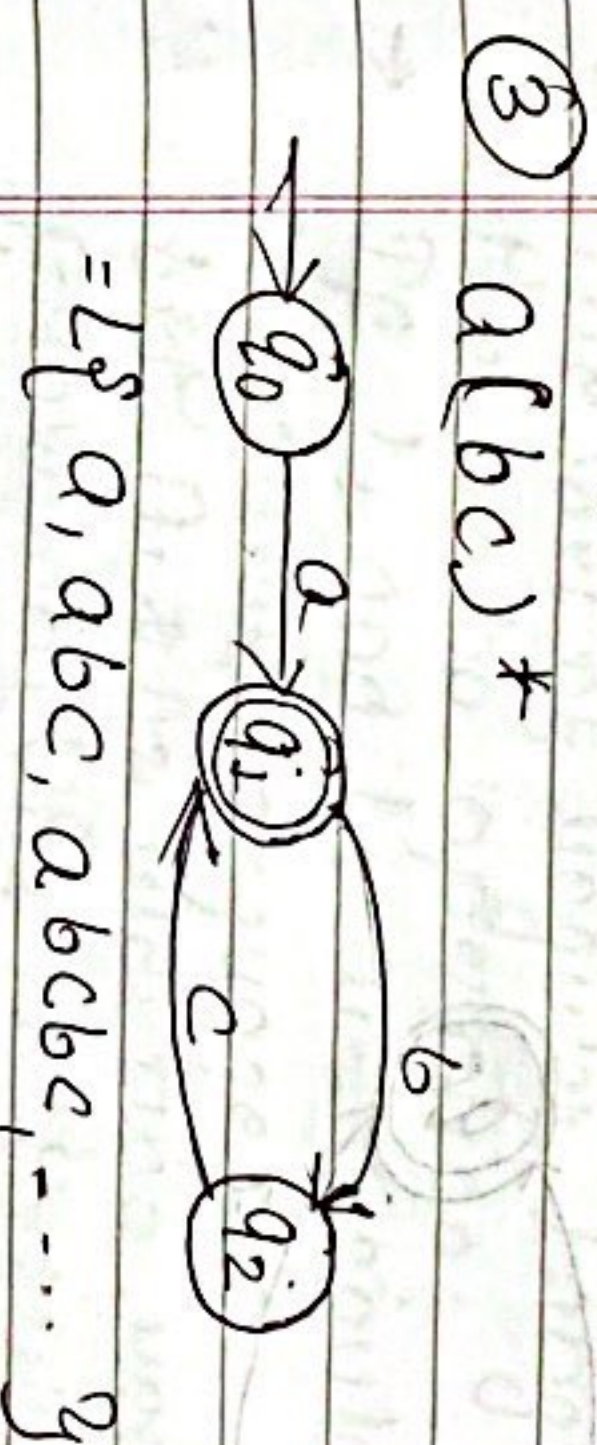
$L = \{ ac, abc, abbc, abbbcc, \dots \}$

2) $(a+b)^*c$



$L = \{ cac, bac, abac, abbbac, \dots \}$

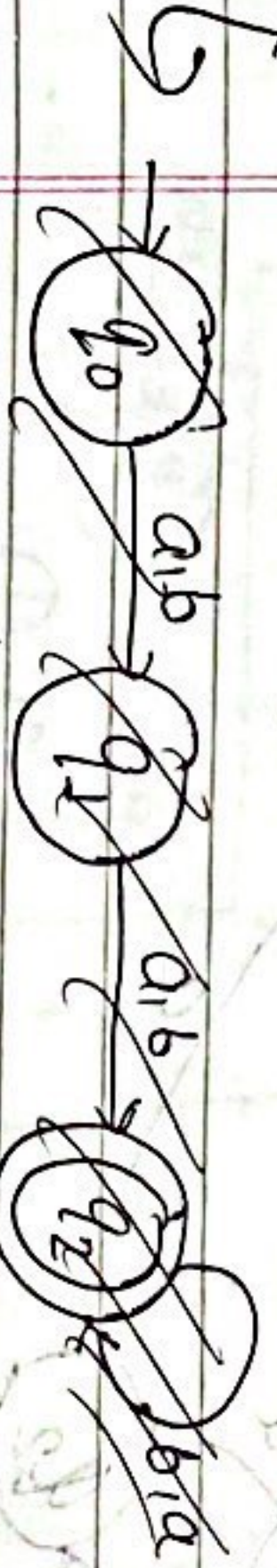
3) $a(bca)^*$



$L = \{ a, abca, abcbca, abcbcbca, \dots \}$

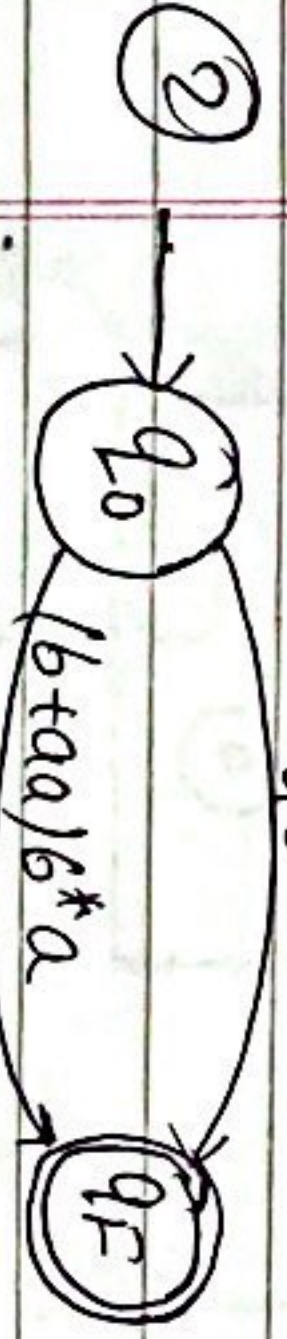
Design FA from given RE

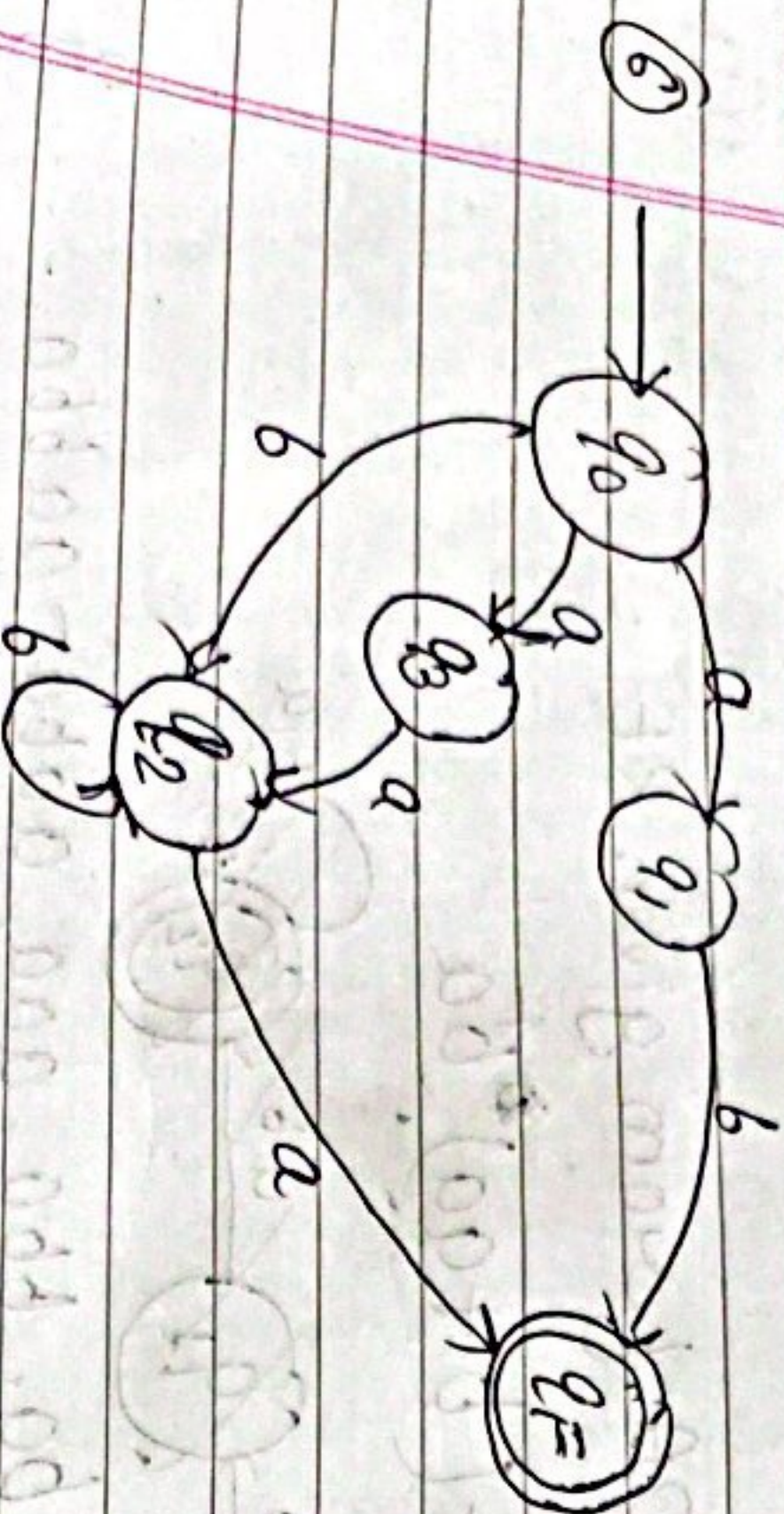
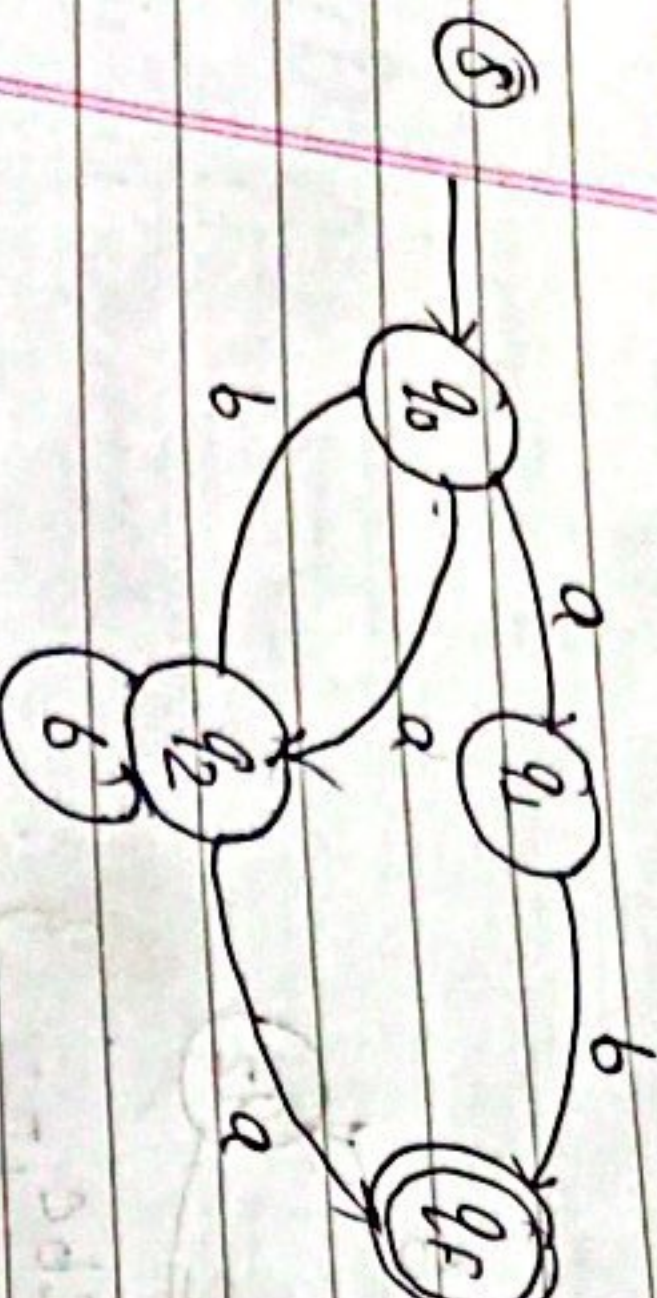
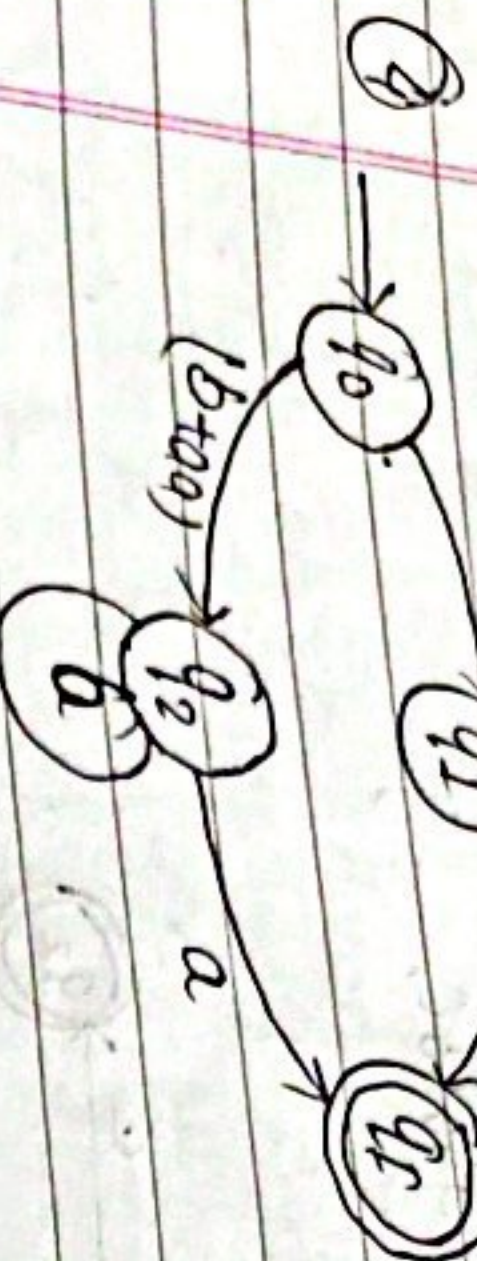
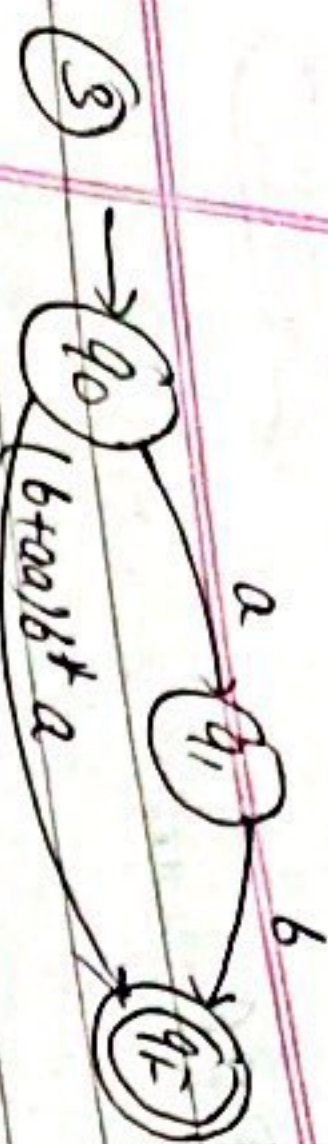
4) $RE = ab + (b+aa)^*ba$



$L = \{ ab, ba, bba, aaa, aaba, aabba, \dots \}$

1) $ab + (b+aa)^*ba$





★

Properties of Regular language :-
(1) Closure properties of Regular languages :-

If a set of regular languages are combined using an operator, then the resulting language is also regular.
→ Closure property is a statement (certain operations on languages), when applied to languages in a class, produces a result is also in that class.

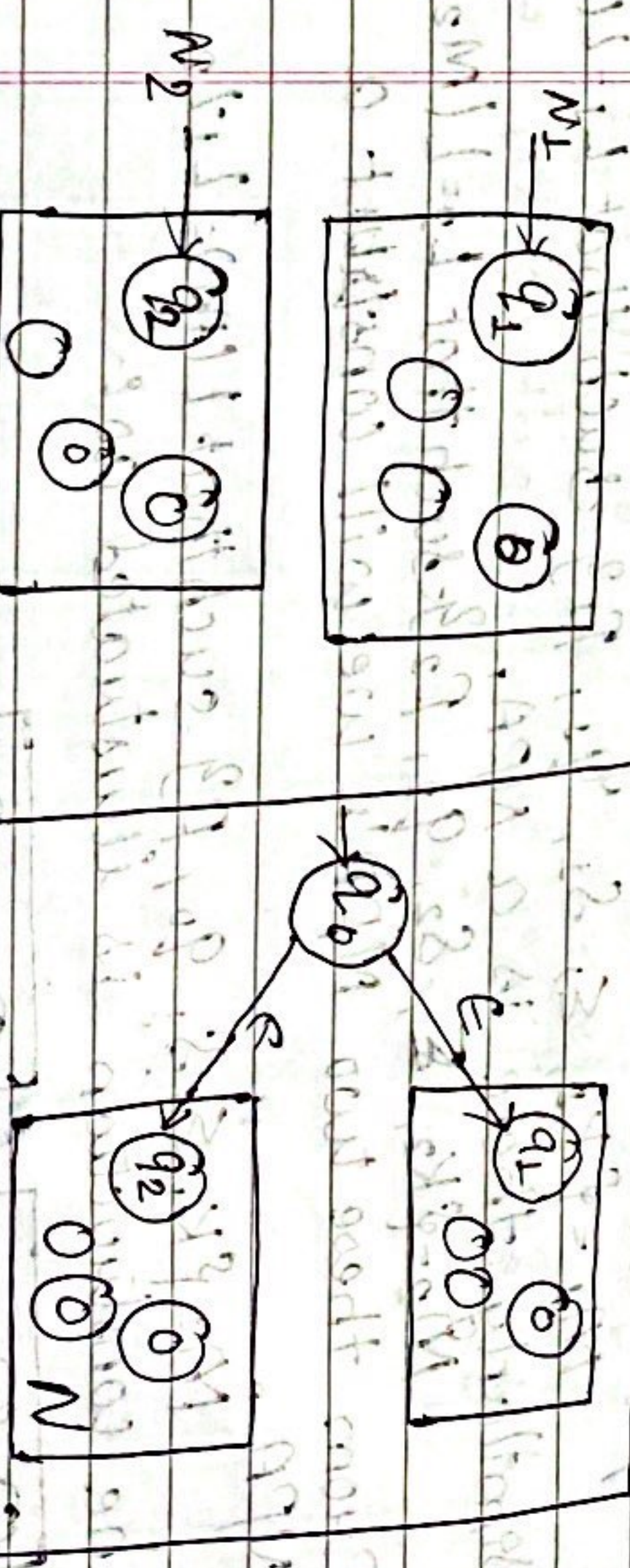
① Union :- If L_1 and L_2 are regular, then $L_1 \cup L_2$ is also a regular language.

→ The set of regular languages is closed under the union operation.

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

Let A_1 and A_2 are two regular languages, now we remove A_1 , A_2 is regular.

Here idea is, take two NFAs N_1 and N_2 for A_1 and A_2 and combine them to one. When NFA N_1 such that it accepts input i.e either accepted by N_1 and N_2 .



Then NFA 'N' accepts $L(N_1) \cup L(N_2)$.

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes A_1
Let $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes A_2
Let $N = (Q, \Sigma, \delta, q_0, F)$ is such that?

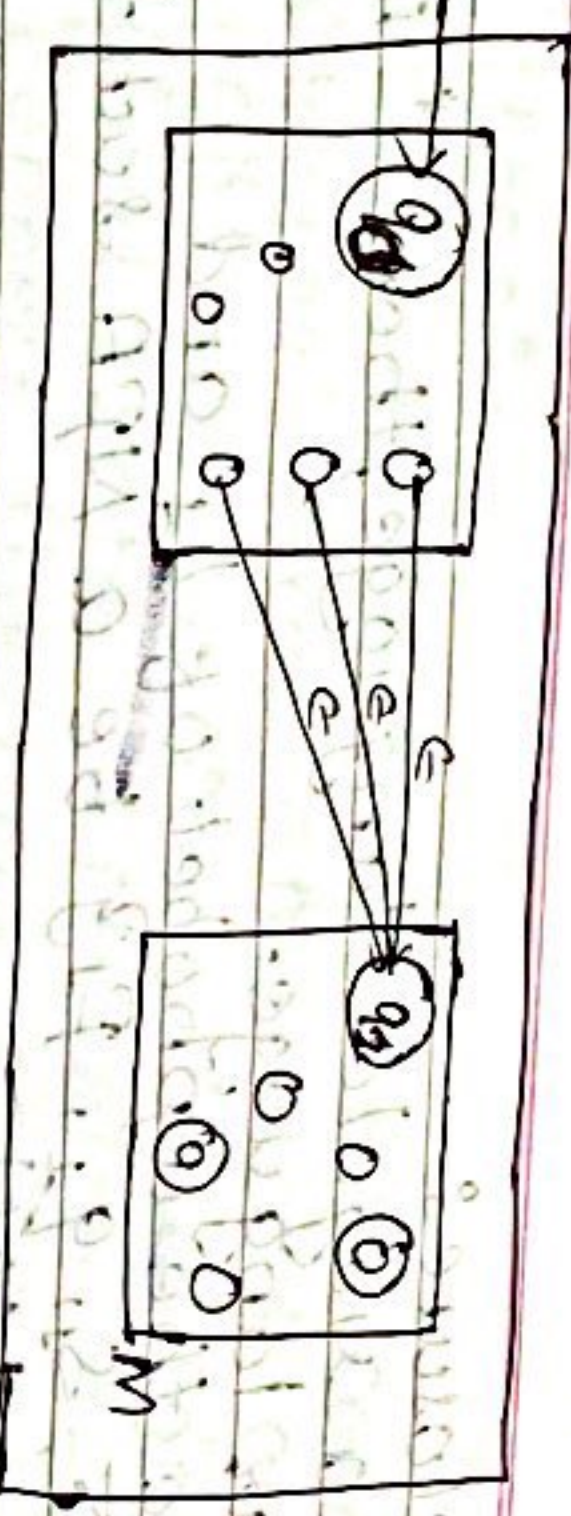
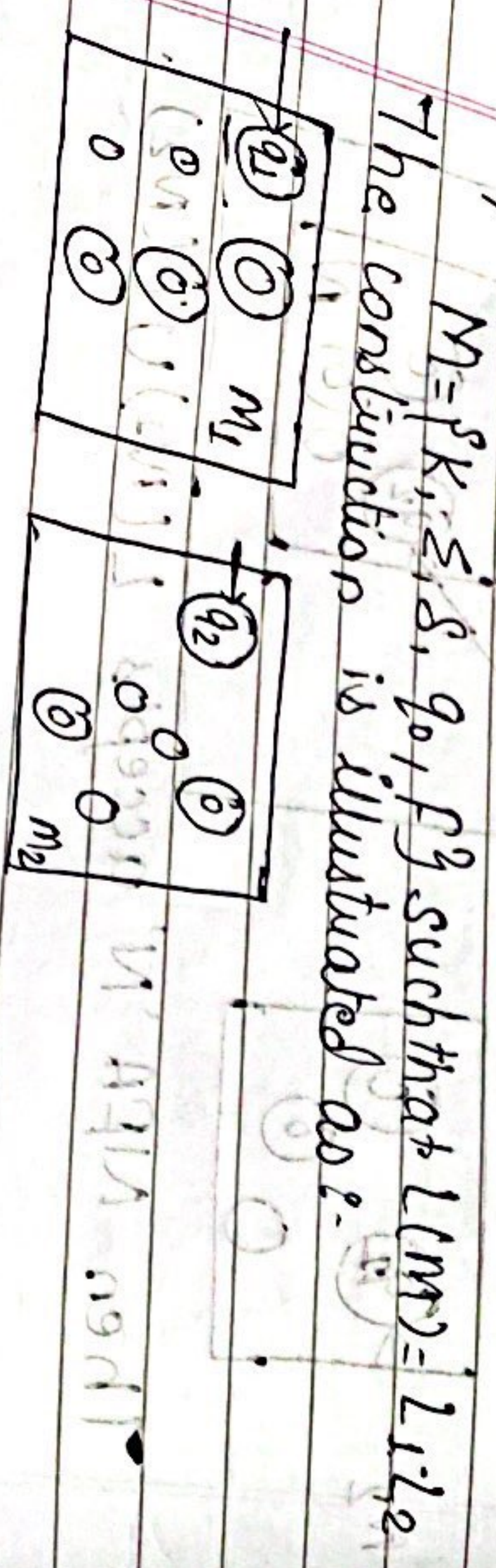
Now, $N = A_1 \cup A_2$
 $L(N) = A_1 \cup A_2$
Here, $Q = Q_1 \cup Q_2$; $q_0 = \text{start state of } N$
 $F = F_1 \cup F_2$

$\delta(q, a) = \delta_1(q, a)$ if $q \in Q_1$
 $\Rightarrow \delta_2(q, a)$ if $q \in Q_2$
 $\Rightarrow q_1, q_2$ if $q = q_0 \& a = \epsilon$
 $\Rightarrow \emptyset$ if $q = q_0 \& a \neq \epsilon$

(b) Concatenation:-

The set of regular language is closed under the concatenation operation i.e. if L_1 and L_2 are regular language over the same alphabet, then $L_1 L_2$ is also a regular language.

Proof:- Since L_1 is regular there is a NFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ such that $L_1 = L(M_1)$
Similarly, there is a NFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ such that $L_2 = L(M_2)$
From these two NFAs, we will construct a NFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $L(M) = L_1 L_2$
The construction is illustrated as:-



The NFA 'M' accepts $L(M_1) \cdot L(M_2)$

The NFA M is defined as follows:-

$K = K_1 \cup K_2$

$q_0 = q_1$ is the start state of M.

$F = F_2$ (same as accept state of M_2)

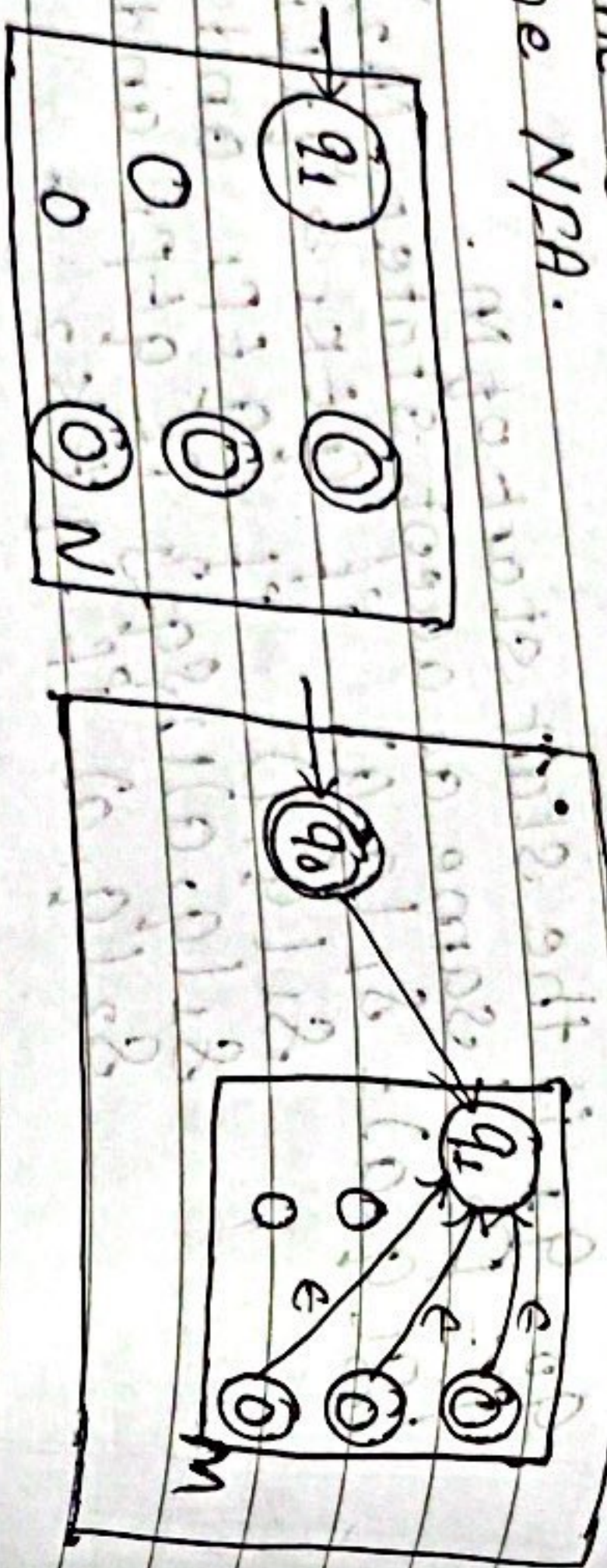
$\delta(q, a) = \delta_1(q, a)$ if $q \in K_1$ & $q = q_1$
 $\delta_2(q, a)$ if $q \in K_2$ and $a = \epsilon$
 $\delta_2(q, a)$ if $q \in K_1$ and $a = \epsilon$

Steps in concatenation of L_1 & L_2 :-
Put a ϵ -transition from each final state of L_1 to the initial state of L_2 .
Make the original final state of L_2 final.

Ex: $L_1 = \{a, aa, aaa, \dots\}$
 $L_2 = \{b, bb, bbb, \dots\}$
 $L_1 L_2 = \{ab, aabb, aaabbb, \dots\}$

Kleene closure: If L is a regular language, then L^* is also a regular language. Let Σ be the alphabet of L and $N = \{K, \delta, q_0, F\}$ be a NFA such that $L = L(N)$.

We will construct a NFA, $M = \{K', \Sigma, \delta', q_0', F'\}$ such that $L(M) = L^*$. The construction is illustrated in figure below.



The NFA M' accepts $L(N)^*$.

$K = K \cup \{q_0', q_n'\}$, where q_0' is the new start state and q_n' is the start state of N . $F' = q_0' \cup F$ since q_0' has to be an final state.

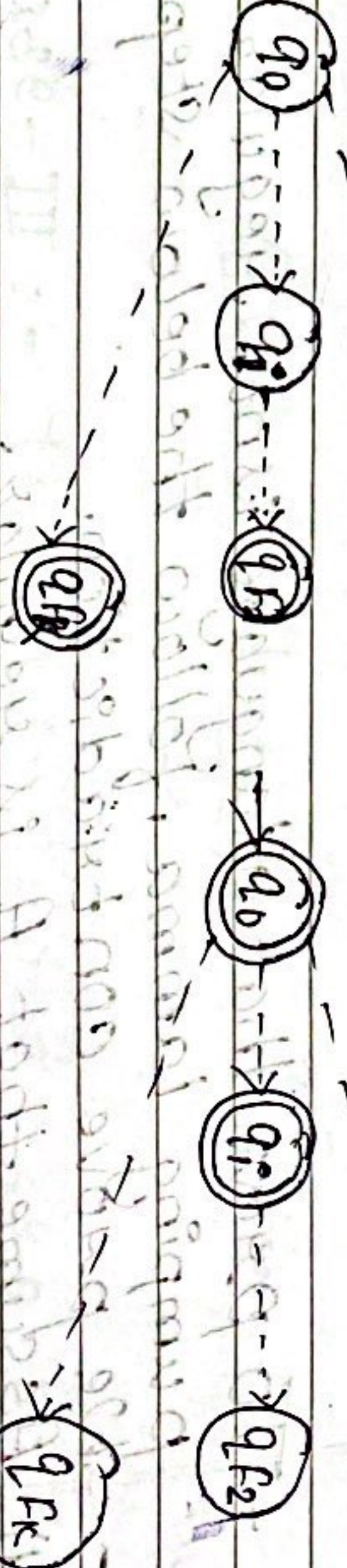
δ' for any $q \in K$ and $a \in \Sigma$

$$\begin{aligned} \delta'(q, a) &= \delta_1(q, a) \text{ if } q \in K_1 \text{ \& } q \in F_1 \\ &= \delta_1(q, a) \text{ if } a \in F_1 \text{ \& } a \notin F_1 \\ &= \delta_1(q, a) \cup \{q_0'\} \text{ if } q \in K_2 \text{ \& } q = F_2 \\ &= \emptyset \text{ if } q = q_0' \text{ \& } a \in \Sigma \\ &= \emptyset \text{ if } q = q_0' \text{ \& } a \notin \Sigma \end{aligned}$$

- Steps:-
- 1) Make a new start state, connect it to the original start state with a ϵ -transition.
 - 2) Make a new final state, connect the original final state to it with ϵ -transitions.
 - 3) Connect the new start state and final state with a pair of ϵ -transitions.

Complement: The set of regular language is closed under the complement operation. If L is a regular language over Σ , then the complement of L is $\Sigma^* - L$.

To show the complement of L is also a regular convert every final state.



Let $M = \{K, \Sigma, \delta, q_0, F\}$ be a DFA. Then the complementary language $L = \Sigma^* - L(M)$ is accepted by DFA $M' = \{K', \Sigma, \delta', q_0', F'\}$ i.e. M' is identical to M except that the final and non-final states are interchanged.

classmate
Date _____
Page _____

~~V14~~ Pumping lemma for regular languages

- ↳ Pumping Lemma is used to prove that a language is not regular.
- ↳ It cannot be used to prove that a language is regular.

If A is a regular language, then A has a pumping length p such that any string w where $|w| \geq p$ may be divided into three parts xyz such that the following conditions must be true.

- 1) $xyz \in A$ for every $p \geq 0$
- 2) $|y| > 0$
- 3) $|xy| \leq p$

To prove that a language isn't regular using pumping lemma, follow the below steps:-
We prove contradiction.

- 1) $xy'z \in A$ for every $p \geq 0$
- 2) $|y| > 0$
- 3) $|xy| \leq p$

- 3) Assume that A is regular.
- 4) All strings longer than p can be pumped $|s| \geq p$.
- 5) It has to have a pumping length (say p).
- 6) Now, find a string s in A such that $|s| \geq p$.
- 7) Divide s into xyz i.e. $|x|, |y|, |z| \geq 1$.
- 8) Show that $xy^2z \notin A$ for some i .

- 7) Then consider always that S can be divided into xy^2 .
- 8) Show that none of these can satisfy all the 3-pumping conditions at the same time.
- 9) S cannot be pumped - contradiction.

Eg:- Using pumping lemma show that the language $\{a^n b^n\}$ is not regular.

$A = \{a^n b^n \mid n \geq 0\}$ is not regular.

Assume that A is regular and pumping length $= p$

Let, $p=7$, $S=aaaaabbbbb$

Case-I:- The 'y' is in the 'a' parts:-

$\frac{2}{x} + \frac{2}{y} + \frac{2}{z}$

Case-II: The y is in the 'b' parts

$$\begin{array}{r} 22 \\ 22 \\ \hline 44 \\ 22 \\ \hline 66 \\ 22 \\ \hline 88 \\ 22 \\ \hline 100 \end{array}$$

Case-III :- The y is in the 'a.e.b' parts.

$\frac{aaaggaabbbbbb}{x \quad y \quad z}$

$$\Rightarrow xy^2z \Rightarrow xy^2z \quad (a=1, b=2)$$

⇒ aa aaaaaa a b b b b b b b b
⇒ 11 7 7

- 3) Converting among representation :-
 (i) Convert NFA to DFA and vice versa
 (ii) Convert automata to regular expression and vice versa

Chapter - 3

Context-Free Language (CFL)

- # Context free Grammar (CFG)
 In formal language theory, a context free language is a language generated by some context free Grammar (CFG).
 - It is more powerful method of describing language.
 - It can describe certain features that have a recursive structure.
 - An important application of CFG occurs in languages and compilation of programming language.
 - The collection of languages associated with CFG are called context free languages (CFL).

Context free grammar is a language generator to some set of rules.
 eg:- $A \rightarrow AB$
 $B \rightarrow \#$

- Here,
 - Grammar consists of collection of substitution rules called production.
 - Each rule comprise a symbol and string separated by arrow.
 - Symbol is called variable and string consisting that consists of variable and other symbols are called terminals.
 - One variable is called variable that usually occurs on the left hand side of the to most rule.

eg. $A \rightarrow 0A1$

$00A111 (A \rightarrow 0A1)$
 $000A1111 (A \rightarrow 0A1)$
 $000B1111 (A \rightarrow B)$
 $000\#1111 (B \rightarrow \#)$

Formal Definition of CFG:-

Context free Grammar is defined by four tuples: $\langle V, \Sigma, S, P \rangle$ where

- V = set of variables or Non-terminal symbols
- Σ = set of terminal symbols
- S = start symbol
- P = Production rule

$V = \{A, B\}$

$\Sigma = \{0, 1, \#\}$

$S = \{A\}$

$P \Rightarrow A \rightarrow 0A1/B$

$B \rightarrow \#$

Some Terminologies:-

1) Derivation:-

Let $G = \langle V, \Sigma, S, P \rangle$ be context free Grammar.
 If $w_1, w_2, \dots, w_n$ are string over Variable V such that $w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n$ then say w_n is derivable.

The sequence of substitution to obtain a string is called derivation say w_1 is derived from w_0 is called

derivation.

i.e. $w_1 \xRightarrow{*} w_n$ then, the sequence of steps to obtain w_n .

2)

Language of CFG $L(G)$:-
 If $G = \langle V, \Sigma, S, P \rangle$ is a CFG, then the language of G denoted by $L(G)$ is the set of terminal strings that have derivations from start symbol.
 i.e. $L(G) = \{ w \in \Sigma^* : S \Rightarrow w \}$

3)

Sequential form :-
 Derivations from the start symbol produce strings that have a special rule. This is called sequential form. This is i.e. $G = \langle V, \Sigma, P, S \rangle$ is a CFG then any string α is in $L(G)$ iff $\alpha \in \Sigma^*$ such that $S \xRightarrow{*} \alpha$ is a sequential form.

It $S \xRightarrow{*} \alpha$, then α is a left sequential form.
 If $S \xRightarrow{*} \alpha$ then α is a right sequential form.

Eg:-

$E \rightarrow I$

$E \rightarrow E + E \mid E * E \mid (E)$

$I \rightarrow a \mid b \mid I_1 I_2 \mid I_3 I_4$

Also,

$E * (I + E)$ is a sequential form as there is

$E \rightarrow E * E$

$\Rightarrow E * (E)$

$\Rightarrow E * (I + E)$

$\Rightarrow E * (I + E)$

① Consider a^*b

$$E \xrightarrow{m} E^*E$$

$$\Rightarrow I^*E$$

$$\Rightarrow a^*E$$

② Consider $E^*(E+E)$

$$E \Rightarrow E^*(E)$$

$$\Rightarrow E^*(E+E)$$

Let $G = (V, \Sigma, R, S)$ where

$$V = \{a, b\}$$

$$\Sigma = \{a, b\}$$

$$R = \{S \rightarrow aSb, S \rightarrow ab\}$$

$$\text{Find } L(G)$$

① $S \rightarrow aSb$

$$S \rightarrow ab$$

② $S \rightarrow ab$

$$S \rightarrow aSb \Rightarrow aabbb$$

③ $S \rightarrow aSb$

$$\Rightarrow aaaSbbb$$

$$\Rightarrow aaaSbbb$$

$S \rightarrow aSb$

$\Rightarrow aaaSbbb$

$\Rightarrow aaaSbbb$

$\Rightarrow a^{n-1}Sb^{n-1}$

$\Rightarrow a^{n-1}ab^{n-1}$

$\Rightarrow a^n b^{n+1}$

$$L(G) = \{a^n b^{n+1} : n \geq 1\}$$

Let $G = (V, \Sigma, R, S)$ where

$$V = \{a, b\}$$

$$\Sigma = \{a, b\}$$

$$R = \{S \rightarrow aAs, S \rightarrow bs\}$$

$$\text{Find } L(G)$$

① $S \rightarrow aAs$

$$S \rightarrow b$$

② $S \rightarrow b$

$$S \rightarrow aAs \Rightarrow aab$$

③ $S \rightarrow aAs$

$$S \rightarrow aAs \Rightarrow aaaS$$

$$= aaaS$$

$a^n b$

$$L(G) = \{a^n b : n \geq 0\}$$

Let: $G = (V, \Sigma, R, S)$ where

$$V = \{a, b\}$$

$$\Sigma = \{a, b\}$$

$$R = \{S \rightarrow aA|E, A \rightarrow bs\}$$

$S \rightarrow aA|E$

$$A \rightarrow bs$$

$S \rightarrow aA$

$$A \rightarrow bs \Rightarrow aabs$$

$$A \rightarrow bs \Rightarrow aabs$$

string $\rightarrow$ terminal $\rightarrow$ reverses eg 54209

classmate
Date _____
Page _____

(11) $S \rightarrow aA$ (11) $S \rightarrow aA$ (11) $S \rightarrow aA$
 $S \rightarrow abS$ $\rightarrow abS$ $\rightarrow abS$
 $S \rightarrow ab$ $\rightarrow ab$ $\rightarrow ab$

$L(G) = \{ (ab)^n : n \geq 0 \}$

Construct a CFG for language $L = \{ w^2 : w \in \{a,b\}^* \}$

$L = \{ aca, bcb, aaca, bbcb, abca, abcb, aabcb, \dots \}$

$S \rightarrow aSa$ (1)
 $S \rightarrow bSb$ (2)

$S \rightarrow \epsilon$ (3)

Here, the grammar is $G = (V, \Sigma, S, P)$

where,

$V = \{ S, a, b \}$

$\Sigma = \{ a, b \}$

$S \rightarrow \epsilon$

$P = \{ S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \epsilon \}$

$S \rightarrow aSa$

#

$S \rightarrow aSa$

add

add

Construct a CFG for language $L = \{ a^n b^n : n \geq 0 \}$

$L = \{ a^2 b^2, a^3 b^3, a^4 b^4, \dots \}$

$S \rightarrow aAB$ $S \rightarrow aab$
 $A \rightarrow aAa$ $S \rightarrow aAaAa$
 $B \rightarrow aBb$ $A \rightarrow bAa$

classmate
Date _____
Page _____



$a^3 b^3$
 $a^2 b^2$
 $a^4 b^4$

Derivation Tree/parse Tree:-

→ It is a tree representation of strings of terminals using the productions defined by the grammar.

→ It pictorially shows how the start symbol of a grammar derive a string in the language.

→ It is an ordered tree that shows the essential

of a derivation.

→ Formally, given a CFL $G = (V, \Sigma, R, S)$, a parse tree has a following properties.

(i) The root is labeled by start symbol.

(ii) Each interior node of parse tree are variable.

(iii) Each leaf node of parse is labeled by a terminal symbol or ϵ .

(iv) If an interior node is labeled with a non-terminal A and its childrens are $X_1, X_2, \dots, X_n$ then there is a production P as:

$$A \rightarrow X_1, X_2, \dots, X_n \text{ for each } X_i \in T$$

Example:-1.

Consider the grammar

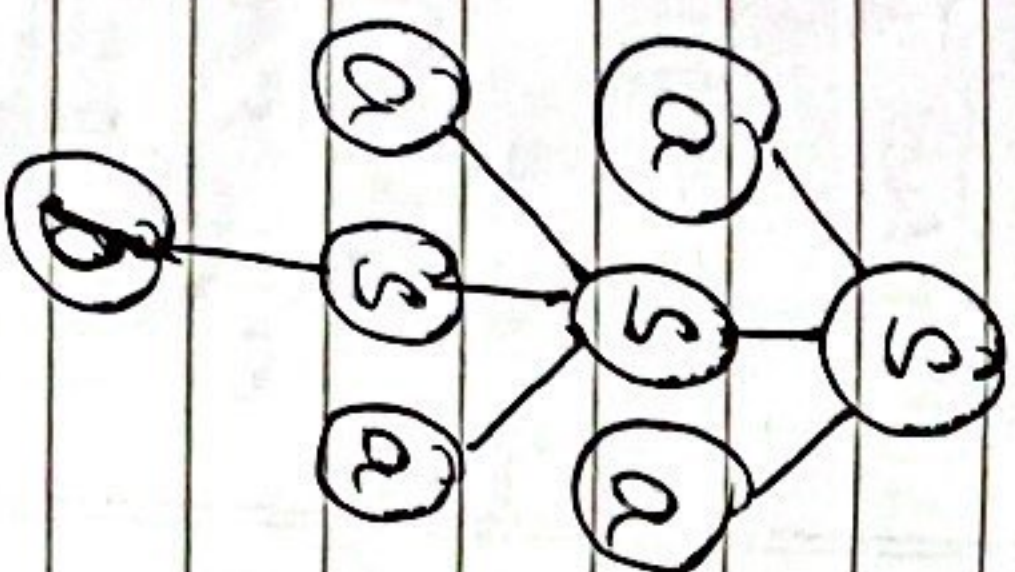
$$S \rightarrow a s d / a / b / \epsilon$$

Now, for the string aabaa

$$S \rightarrow a s d$$

$$\rightarrow a a s a a$$

$$\rightarrow a a b a a$$



Example:-2

Consider the grammar

$$S \rightarrow A1B$$

$$A \rightarrow 0A/\epsilon$$

$$B \rightarrow 0B/1B/\epsilon$$

Construct the parse tree for string 00101.

$$S \rightarrow A1B$$

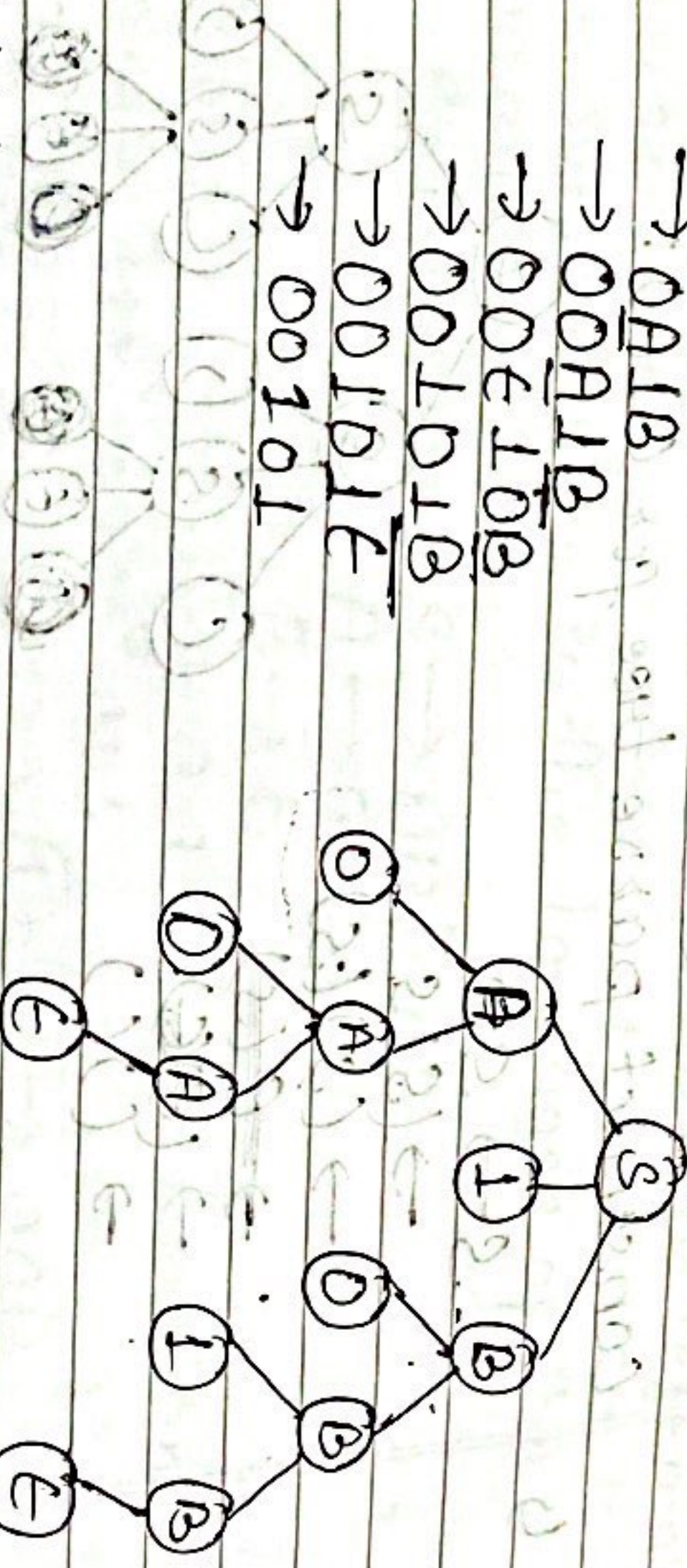
$$\rightarrow 0A1B$$

$$\rightarrow 00A1B$$

$$\rightarrow 001B$$

$$\rightarrow 0010B$$

$$\rightarrow 00101$$



$$000111$$

$$S \rightarrow A1B$$

$$\rightarrow 0A1B$$

$$\rightarrow 00A1B$$

$$\rightarrow 000A1B$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

$$\rightarrow 000111$$

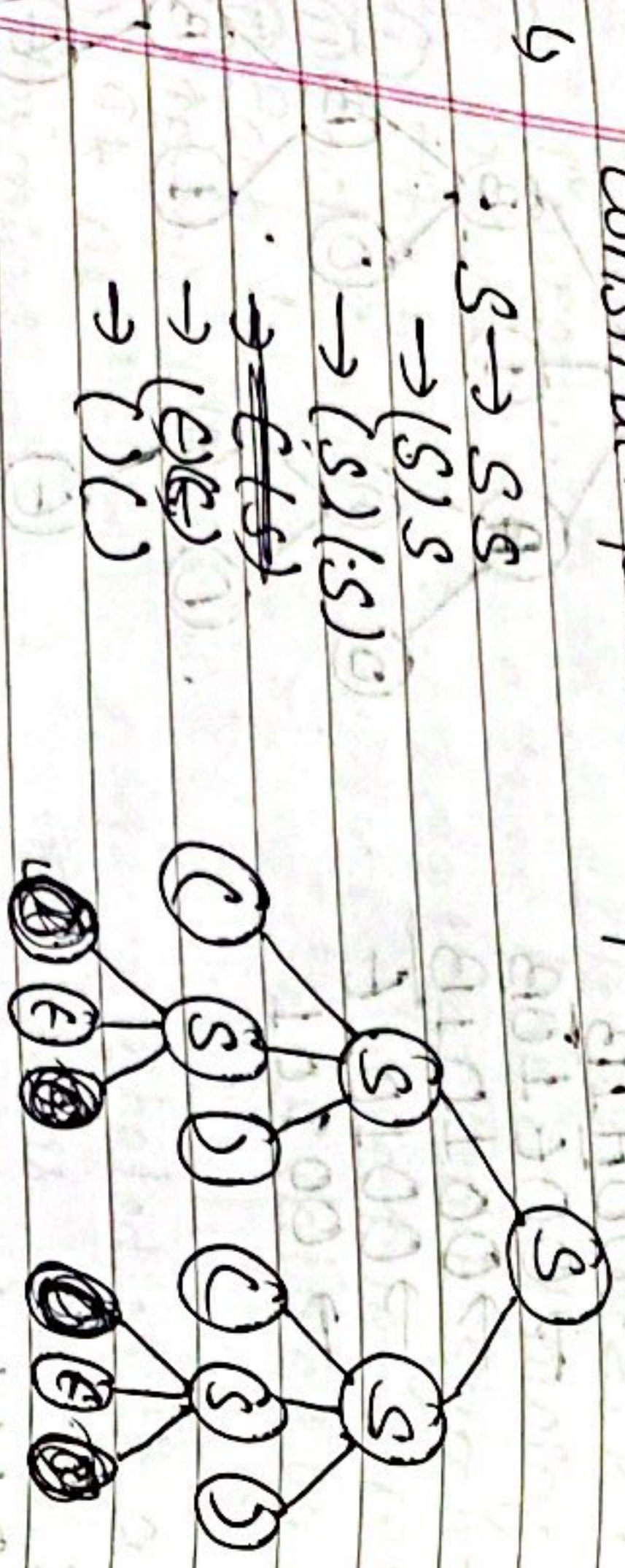
Derivation Tree / parse tree :-

Q Let $G = (V, \Sigma, R, S)$ where:

$$V = \{S, \epsilon, \lambda, \gamma\}$$

$$R = \{S \rightarrow S\lambda S, S \rightarrow \epsilon\}$$

Construct parse tree for $\lambda\lambda\lambda$:



Left-most and Rightmost Derivation :-

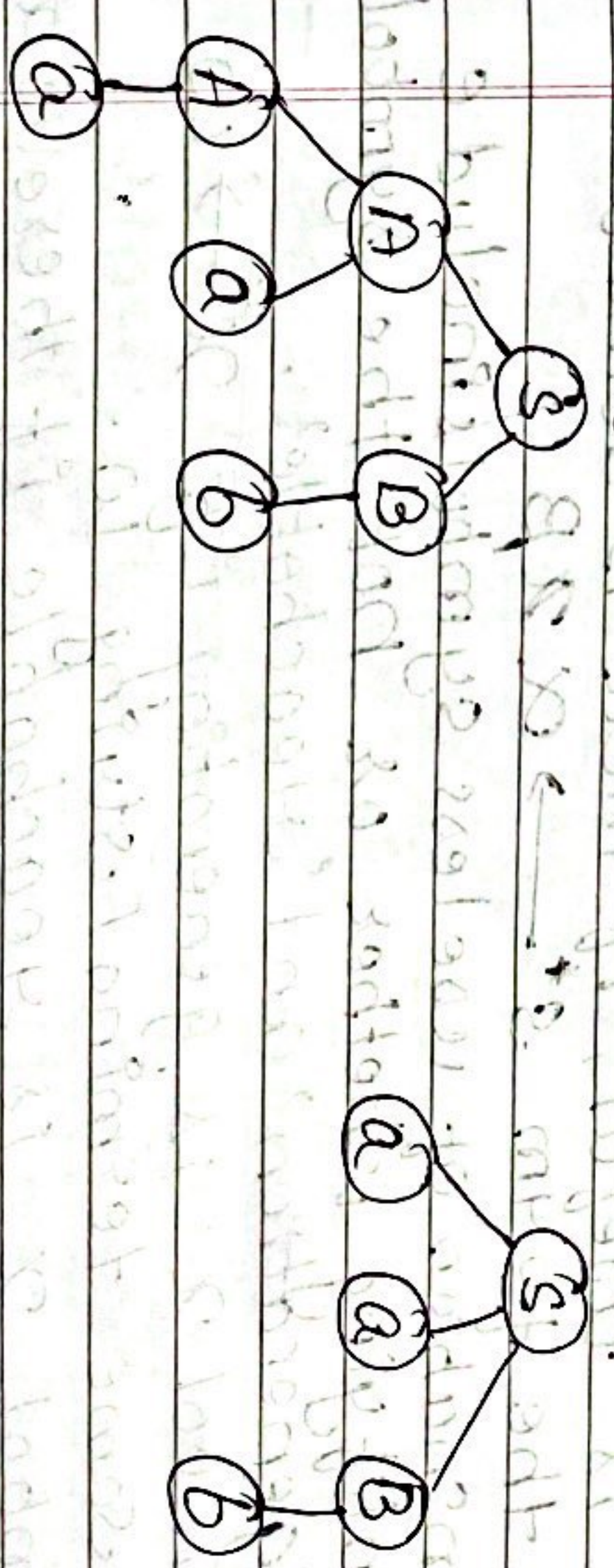
- In left most derivation, left most variable is replaced first.
- A tree which is obtained by applying production to the left most variable in each step is called left derivation tree.
- In right most derivation, right most variable is replaced first.
- A right derivation tree is obtained by applying production to the right most variable in each step.

Ambiguity in Grammar :-

A grammar $G = (V, T, P, S)$ is said to be ambiguous if there is a string $w \in L(G)$ for which we can derive two or more distinct derivations rooted at S and yielding w . In other words, a grammar is ambiguous if it can produce more than one leftmost or more than one rightmost derivation for the same string in the language of the grammar.

Example :- $S \rightarrow AB/aB$
 $A \rightarrow a/Aa$
 $B \rightarrow b$

Now for string aab we have two left most derivation as :-



Simplification of CFG:

In CFG, sometimes all the production rules and symbols are not needed for the derivation of strings. Besides this, there may also be some NULL productions and unit productions. Elimination of these production & symbols is called simplification of CFG. We must:

- ① Eliminate useless symbols.
- ② Eliminate empty productions.
- ③ Eliminate unit productions.

① Eliminate useless symbols: These variables or terminals that do not appear in any derivation of a terminal string from the start symbol. For a grammar, $G = (V, \Sigma, R, S)$, a symbol x is useful, if there is some derivation of the form $S^* \rightarrow \alpha x \beta$.

→ Elimination of useless symbols include identifying whether or not the symbol is 'generating' and 'reachable'.

→ A symbol x is generating if $x^* \rightarrow w$ for some terminal string w .

→ A symbol x is reachable if there is a derivation $S^* \rightarrow \alpha x \beta$ from some 'α' and 'β'.

→ Thus, we eliminate non-generating and then non-reachable symbols.

eg: Consider a grammar rule:

$A \rightarrow abS | abA | abB$

$B \rightarrow aB$

$C \rightarrow dc$

In above, eg: production rule $C \rightarrow dc$ is useless, because the variable C will never occur in derivation of any string. The other production are written in such a way that variable ' C ' can never reach from the start symbol ' S '.

Production $B \rightarrow aB$ is also useless, because there is no way it will ever terminate, then, it can never produce a string. To remove useless production, we first find all the variables which will never lead to a terminal string such as variable B . We then, remove all the production in which variable B occurs.

So,

$S \rightarrow abS | abA$

$A \rightarrow cd | abA$

$C \rightarrow dc$

We then try to identify all the variable that can never be reached from the starting variable such as variable ' C '. We then remove all the production in which var ' C ' occurs. Then, below grammar is now free of useless production/symbols.

$S \rightarrow abS | abA$

$A \rightarrow cd | abA$

② Eliminating Empty (ϵ) Productions:-
A grammar is said to have empty production if there is a production of the form $A \rightarrow \epsilon$.

- Here, we have to discover nullable variable.
- A variable 'A' is nullable if $A^* \rightarrow \epsilon$.

Procedure for removal:-
step 1:- To remove $A \rightarrow \epsilon$; look for all productions whose right side contains A.

step 2:- Replace each occurrence of 'A' in each of these production with ϵ .

step 3:- Add the resultant productions to the grammar.

eg:- Remove (ϵ) null production from the following grammar.

$S \rightarrow ABAC$
 $A \rightarrow aA/\epsilon$
 $B \rightarrow bB/\epsilon$
 $C \rightarrow c$

Soln

Here we have two ϵ productions to remove i.e. $A \rightarrow \epsilon$, $B \rightarrow \epsilon$

To eliminate, $A \rightarrow \epsilon$, production which contain A in right side are:

$S \rightarrow ABAC$
 $A \rightarrow aA$
Add ϵ to the right side of A
 $b \rightarrow \epsilon$

① $S \rightarrow ABAC$
 $\rightarrow BAC/ABC/c$

② $A \rightarrow aA$
 $\rightarrow a$

$S \rightarrow ABAC/BAC/ABC/c$
 $A \rightarrow aA/a$
 $B \rightarrow bB/b$
 $C \rightarrow c$

To eliminate $B \rightarrow \epsilon$,
productions which contain B at right side.

① $S \rightarrow ABAC \rightarrow AAC$ ② $S \rightarrow BAC \rightarrow AC$ ③ $S \rightarrow ABC \rightarrow AC$
④ $S \rightarrow BC \rightarrow c$ ⑤ $B \rightarrow bB \rightarrow b$

finally we get:-

$S \rightarrow ABAC/BAC/ABC/AC/c$
 $A \rightarrow aA/a$
 $B \rightarrow bB/b$
 $C \rightarrow c$

$a \rightarrow a, b \rightarrow b, c \rightarrow c, d \rightarrow d, e \rightarrow e, f \rightarrow f, g \rightarrow g, h \rightarrow h, i \rightarrow i, j \rightarrow j, k \rightarrow k, l \rightarrow l, m \rightarrow m, n \rightarrow n, o \rightarrow o, p \rightarrow p, q \rightarrow q, r \rightarrow r, s \rightarrow s, t \rightarrow t, u \rightarrow u, v \rightarrow v, w \rightarrow w, x \rightarrow x, y \rightarrow y, z \rightarrow z$

Eliminating unit productions

Any production rule of the form $A \rightarrow B$, where $A, B \in \text{Non-Terminals}$ is called unit production.

Procedure for Removal:

Step 1: To remove $A \rightarrow B$, add production $A \rightarrow \alpha$ to the grammar where $\alpha \in \text{Terminal}$, $\alpha \neq B$.

Step 2: Delete $A \rightarrow B$ from the grammar.

Step 3: Repeat from Step 1 until all unit productions are removed.

Example:

Remove unit productions from the grammar whose production rule is given by:

$P: S \rightarrow xy$
 $x \rightarrow a$

$y \rightarrow z/b$

$z \rightarrow m$

$m \rightarrow N$

$N \rightarrow a$

Here we have three unit productions:

$y \rightarrow z$

$z \rightarrow m$

$m \rightarrow N$

① Since, $N \rightarrow a$, we add $m \rightarrow a$

$P: S \rightarrow xy, x \rightarrow a, y \rightarrow z/b, z \rightarrow m, m \rightarrow a$

② Since, $m \rightarrow a$, we add $z \rightarrow a$

$P: S \rightarrow xy, x \rightarrow a, y \rightarrow z/b, z \rightarrow a, m \rightarrow a, N \rightarrow a$

③ Since, $z \rightarrow a$, we add $y \rightarrow a$

$P: S \rightarrow xy, x \rightarrow a, y \rightarrow a/b, z \rightarrow a, m \rightarrow a, N \rightarrow a$

Removing the unreachable symbols:

$P: S \rightarrow xy$

$x \rightarrow a$

$y \rightarrow a/b$

Normal Forms of CFG

Since in CFG, the production rule is of the form $\alpha \rightarrow \beta$, where $\alpha \in V$ and $\beta \in (V \cup \Sigma)^+$. LHS of production rule in CFG have only non-terminal and RHS may have empty string, terminal, non-terminal or the combination of both terminal and non-terminals.

Then, when we apply some restrictions on the RHS of the CFG for defining new production, then grammar is said to be in normal form.

Among several normal forms we deal with the following two normal forms:

① Chomsky Normal Form (CNF)

② Greibach Normal Form (GNF)

$A \rightarrow BC$ $AB \rightarrow A$ $AB \rightarrow A$ $AB \rightarrow A$

(1) Chomsky Normal Form (CNF) :-

→ In the CNF, we have restrictions on the length of RHS, and the nature of symbol.
 In the RHS of production is that every node has either two internal vertices (i.e. two non-terminals only) or a single leaf (i.e. exactly one terminal).
 A CFG 'G' is in CNF, if every production is of the form:
 $A \rightarrow BC$
 Here, A, B & C are non-terminal symbols and 'a' is a terminal symbol.

Q Convert the following CFG into its equivalent CNF.

Soln,
 Here,
 $S \rightarrow aAS/as$
 $A \rightarrow aABb$
 $B \rightarrow bB/bAa$

1. $S \rightarrow aAS$ gives $S \rightarrow C_1AS$ $\therefore C_1 \rightarrow a$
 $S \rightarrow C_1AS$ gives $S \rightarrow C_1C_2$ $\therefore C_2 \rightarrow AS$

2. $S \rightarrow as$ gives $S \rightarrow C_1's$ $\therefore C_1 \rightarrow a$

3. $A \rightarrow aBb$ gives $A \rightarrow C_1Bb$ $\therefore C_1 \rightarrow a$
 $A \rightarrow C_1Bb$ gives $A \rightarrow C_1BC_3$ $\therefore C_3 \rightarrow b$
 $A \rightarrow C_1BC_3$ gives $A \rightarrow C_1C_4$ $\therefore C_4 \rightarrow BC_3$

4. $B \rightarrow bB$ gives $B \rightarrow C_3B$ $\therefore C_3 \rightarrow b$
 5. $B \rightarrow bAa$ gives $B \rightarrow C_3AC_1$ $\therefore C_3 \rightarrow b, C_1 \rightarrow a$
 $B \rightarrow C_3AC_1$ gives $B \rightarrow C_3C_4$ $\therefore C_4 \rightarrow AC_1$

Let $G' = (V', \Sigma, R', S')$ be the equivalent CNF where,

$V' = \{S, A, B, C_1, C_2, C_3, C_4, C_5\}$
 $\Sigma = \{a, b\}$
 $R' = \{S \rightarrow C_1C_2, C_1C_3, A \rightarrow C_2C_4, B \rightarrow C_3C_4, C_3C_5, C_4 \rightarrow C_5C_6, C_5 \rightarrow C_3A, C_3 \rightarrow b, C_4 \rightarrow a, C_5 \rightarrow b\}$

$C_4 \rightarrow C_5C_6$
 $C_5 \rightarrow C_3A$
 $C_1 \rightarrow C_2C_4$
 $C_3 \rightarrow b$

Q Convert the following CFG into its equivalent CNF.

$G \Rightarrow \{V, \Sigma, R, S\}$
 $V = \{S, A, B\}$
 $\Sigma = \{a, b\}$
 $R = \{S \rightarrow aAB/AAB/BA, A \rightarrow aA/a, B \rightarrow ab/bA\}$

$G(A, B) = TM$
 $G(A, B) = TM$
 $G(A, B) = TM$
 $G(A, B) = TM$

(2) Greibach Normal form (GNF) is quite useful.

GNF is another normal form quite useful in some proofs and constructions.

A CFG is in GNF if every production is in the form:

$A \rightarrow a\alpha$ where $a \in V^*$ and α is only one terminal $a \in \Sigma$.

Here, any CFG will be in GNF if it is in the form:

non-terminal $\rightarrow$ exactly one terminal followed by any no. of non-terminals including null.

Eg: Given CFG $S \rightarrow abasa/aba$

construct equivalent GNF.

Soln
Here, we have $S \rightarrow abasa/aba$,

let us introduce new variables A and B and productions $A \rightarrow a$ and $B \rightarrow b$ and substitute into the given grammar as:

$S \rightarrow aBASA/aBA$

$A \rightarrow a$ and $B \rightarrow b$

Here, let $G' = (V', \Sigma, R', S')$ be newly constructed

GNF equivalent to given CFG.

where, $V' = \text{set of NT} = \{S, A, B\}$

$\Sigma = \{a, b\}$

$S = S$

$R \rightarrow S \rightarrow aBASA/aBA$

$A \rightarrow a, B \rightarrow b$

Eg: Given CFG

$S \rightarrow aA/aB$

$B \rightarrow b/aB$

$C \rightarrow b$

$A \rightarrow a$

Convert equivalent GNF.

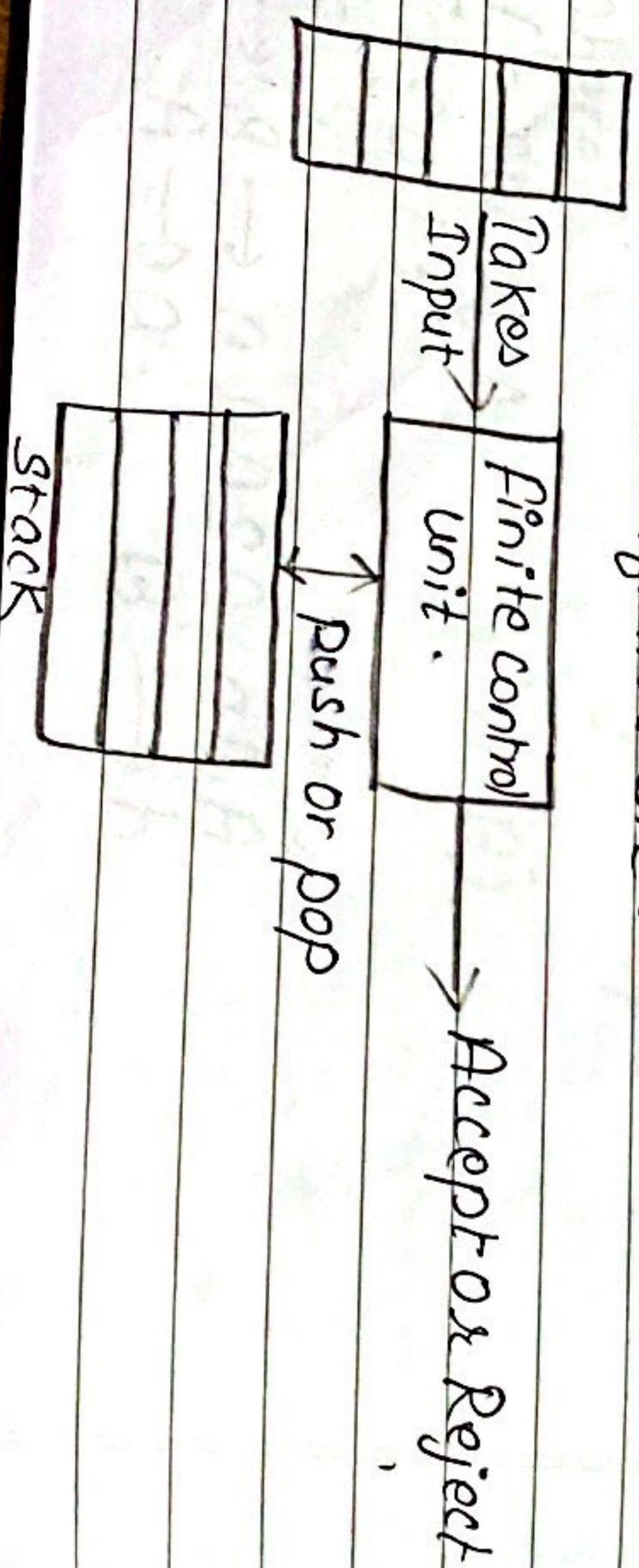
Push Down Automata (PDA)

* Introduction:-

- A PDA is a way to implement a CFG in a similar way we design FA for regular grammar.
- A DFA can remember a finite amount of information but a PDA can remember an infinite amount of information.
- It is more powerful than FSM.
- FSM has a very limited memory but PDA has more memory.
- Basically, a Pushdown automata is:- "finite state machine + a stack"
- A stack is a way we arrange elements one on top of another.
- A stack does two basic operations
PUSH: A new element is added at the top of the stack.
POP: The top element of the stack is read and removed.

A PDA has 3-components:-

1. An input tape.
2. A finite control unit
3. A stack with infinite size.



Formal Definition of PDA:-

A PDA is formally defined by 7 tuples as shown.

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where,

- Q = A finite set of states
- Σ = A finite set of input symbols.
- Γ = A finite stack Alphabet.
- δ = The transition function
- q_0 = The start state
- Z_0 = The start stack symbol
- F = The set of final/accepting states

- δ : Takes an argument a triple $\delta(q, a, x)$ where,
- q is a state in Q
- a is either an input symbol in Σ or $a = \epsilon$
- x is a stack symbol, that is a member of Γ .

The output of δ is finite set of pairs (p, r) where:

p is a new state.

r is a string of stack symbols that replaces x at the top of the stack.

Eg: If $r = \epsilon$ then the stack is popped.

If $r = x$ then the stack is unchanged.

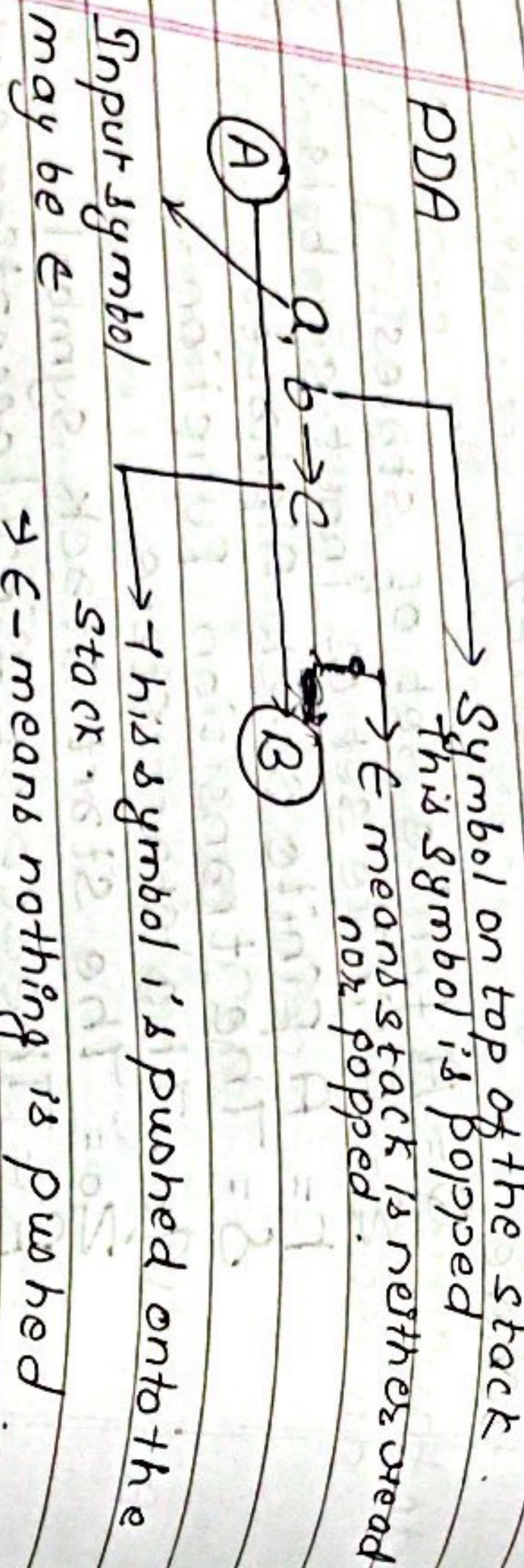
If $r = yz$, the x is replaced by z and y is pushed onto the stack.

$0 \rightarrow \epsilon \rightarrow pop$
 $same \rightarrow \epsilon \rightarrow push$

push pop skip.

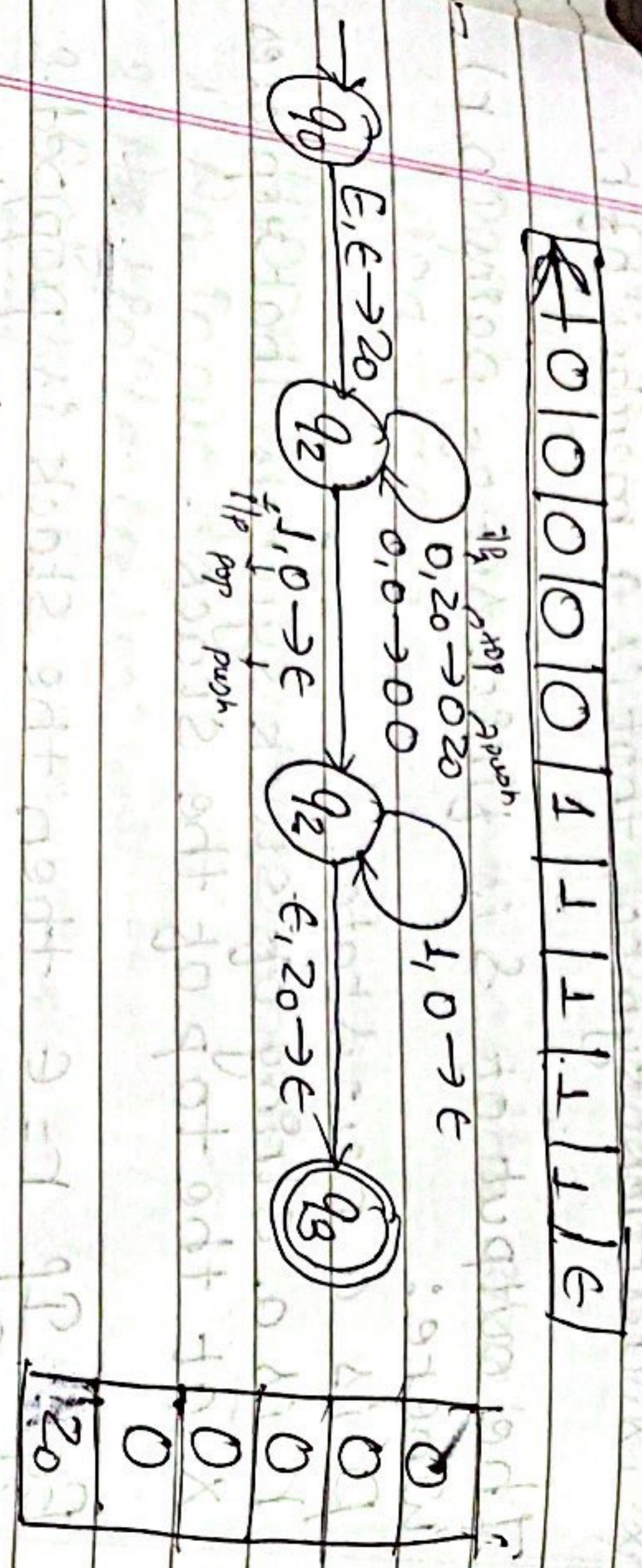
(pop push)
 $0 \rightarrow \epsilon$
 (pop push)
 (pop push)

Graphical Notation of PDA



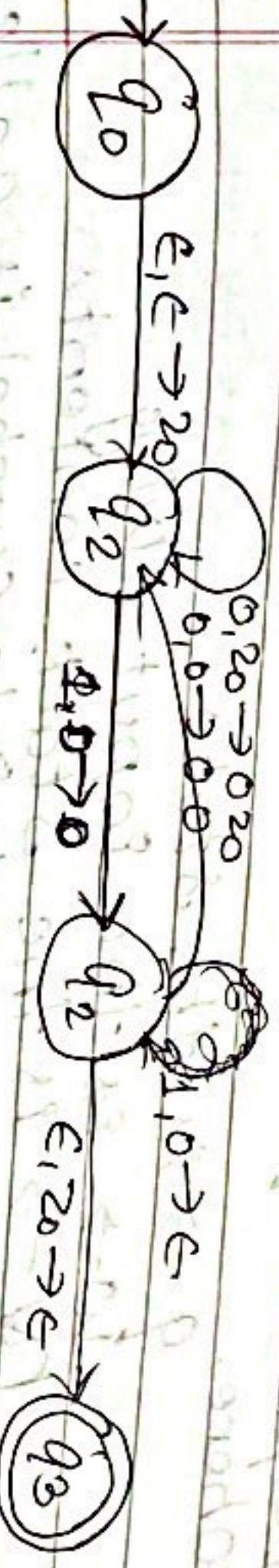
Construct a PDA that accepts;

$L = \{0^n 1^n \mid n \geq 1\}$



Construct the PDA of language.

$L = \{0^n 1^n \mid n \geq 1\}$



Construct the PDA of language

$L = \{w \mid w \in \{0,1\}^*$

$L = \{0^i 1^j \mid i, j \geq 1\}$

Construct a PDA that accepts

✓

Instantaneous Description of PDA (ID)

→ Any config of an PDA can be describe by a tuple (q, w, z) where,

$q \rightarrow$ is the state
 $w \rightarrow$ is the input string
 $z \rightarrow$ is the stack contents.

such description by tuple symbol is called an ID of PDA. For PA just δ is sufficient to represent sequence of ID.

→ For PDA, we need a notation that describe changes in the state, input and stack, thus we adopt the notation of ID.

let $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ be a PDA. Then we define a relation $\vdash$ yields as $(q, w, z) \vdash (p, w, \beta z)$ if $\delta(q, a, z)$ contains $(p, \beta, \text{and may be } \epsilon)$.

This move reflects the idea the by consuming 'a' from the input and replacing z on the top of the stack by β , we can go from state q to p .

We can define acceptance of any string by PDA in two ways.

- 1) Acc Acceptance by final state:
→ Given PDA, P , the language accepted by final state is,
 $L(P) = \{w \mid (q, w, z_0) \vdash^* (p, \epsilon, r) \text{ where } p \in F, r \in \Gamma^*\}$

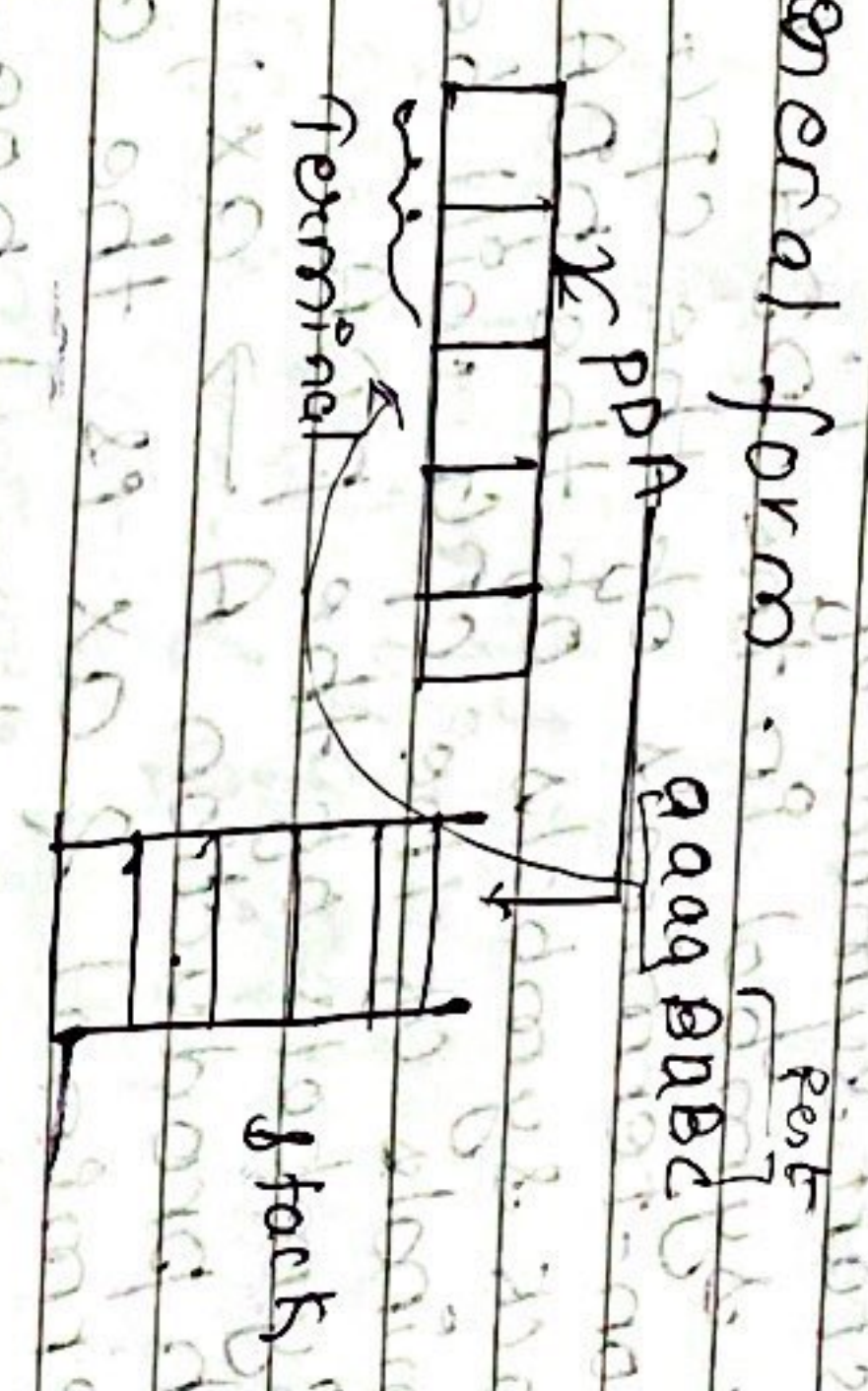
2) Acceptance by empty stack:

$\{w \mid (q, w, z_0) \vdash^* (p, \epsilon, \epsilon) \text{ where } p \in Q, (p, \epsilon, \epsilon) \in F\}$

Equivalence of CFL and PDA:

Theorem:

A language is context free if some push down automata recognizes it. Some push down part is given a CFL, show how to construct a PDA that recognizes it. Part 2: Given a PDA, show how to construct a CFL that recognizes the same language. General form.



- At each step expand left most derivation.
- match stack top of a Rule
- Pop stack
- push right hand side of rule onto the stack

Equivalence of PDA and CFL

→ To show the language defined by PDA are exactly the context free language.
→ Main goal is to prove that CFL & PDA accept the same class of language.

CFL to PDA conversion procedure:-

① Input: A CFL, $G = (V, T, P, S)$
Output: equiv. PDA, $M = (Q, \Sigma, S, \delta, q_0, T, F)$

Step 1: Convert the productions of the CFL into GNF.

Step 2: The PDA will have only state q_0 .

Step 3: The start symbol of CFL will be the start symbol in the PDA, M .

Step 4: All non-terminals of the CFL will be the stack symbols of the PDA and will be the terminals of the CFL will be the input symbols of the PDA.

Step 5: For each production $A \rightarrow \alpha X$, where 'A' is terminal & αX is the combination of terminals and non-terminals make a transition, $\delta(q, a, A) \rightarrow (q, X)$.

From CFL to PDA:

Given CFL $G = (V, T, P, S)$ we construct a PDA, M that accepts language generated by G .

$L(M) = L(G)$

The machine can be defined as:
 $M = (Q, \Sigma, T, V, U, \delta, q_0, S, F)$
where, $Q = \{q_0\}$ (is only the state in the PDA)
 $\Sigma = T$
 $V = U = \emptyset$ (i.e. PDA uses terminals and variable of G as stack symbols).

$Z_0 = S$ (Initial stack symbol is start symbol in G (grammar)).

$F = \emptyset$ (Final state is start symbol in G (grammar)).

$q_0 = q_0$

δ can be defined as:

- $\delta(q, \epsilon, A) \rightarrow (q, A)$ where $A \rightarrow \alpha$ is a production of G .
- $\delta(q, a, a) \rightarrow (q, \epsilon)$ for all $a \in T$.

Here, transition occurs in only one state q_0 .

PDA = $(Q, \Sigma, T, Z_0, F, q_0, \delta)$

Alternatively:

We can define PDA for the CFL with 2 states 'P' and 'q', where 'P' is start state. There idea is that the stack symbol initially is supposed to be ϵ and at first PDA start with state 'P' and reading ϵ , it insert

start symbol 's' at left into stack and change the state to 'q'. Then all transitions occur in state 'q'.

i.e. PDA can be defined as:

$M = (\{q_0, q_1, \dots, q_n\}, \Sigma, \Gamma, \delta, q_0, \{q_f\})$; stack top is 's'.

The δ can be defined as:

$\delta(q, \epsilon, \epsilon) = (q, s)$ / s is start symbol of a.

$\delta(q, a, a) = (q, \epsilon)$, for all $a \in \Sigma$.

Example:-

Let $G = (V, T, P, S)$ where, $V = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$

$P \Rightarrow E \rightarrow T \mid E + T$

$T \rightarrow F \mid T * F$

$F \rightarrow a \mid (E)$

We can define a PDA eq^u to this grammar as:

$M = (\{q_0, q_1, \dots, q_n\}, \Sigma, \Gamma, \delta, q_0, \{q_f\})$

where, δ can be defined as:

$\delta(q_0, \epsilon, \epsilon) = (q_0, s)$

$\delta(q_0, a, a) = (q_0, \epsilon)$

$\delta(q_0, a, \epsilon) = (q_0, a)$

$\delta(q_0, \epsilon, s) = (q_0, \epsilon)$

$\delta(q_0, \epsilon, s) = (q_0, \epsilon)$

$\delta(q_0, \epsilon, s) = (q_0, \epsilon)$

Now with this PDA, M let us assume the string $\delta(q_0, a + (a * a), s) \vdash (q_0, a + (a * a), s + T)$

$\vdash (q_0, a + (a * a), T + T)$

$\vdash (q_0, a + (a * a), F + T)$

$\vdash (q_0, a + (a * a), F * F)$

$\vdash (q_0, a + (a * a), (T * F))$

$\vdash (q_0, a + (a * a), T * F)$

$\vdash (q_0, a + (a * a), F * F)$

$\vdash (q_0, a + (a * a), T + T)$

$\vdash (q_0, a + (a * a), F + T)$

$\vdash (q_0, a + (a * a), F * F)$

$\vdash (q_0, a + (a * a), (T * F))$

$\vdash (q_0, a + (a * a), T * F)$

$\vdash (q_0, a + (a * a), F * F)$

$\vdash (q_0, a + (a * a), T + T)$

$\vdash (q_0, a + (a * a), F + T)$

$\vdash (q_0, a + (a * a), F * F)$

$\vdash (q_0, a + (a * a), (T * F))$

$\vdash (q_0, a + (a * a), T * F)$

$\vdash (q_0, a + (a * a), F * F)$

$\vdash (q_0, a + (a * a), T + T)$

$\vdash (q_0, a + (a * a), F + T)$

Q. Show that $L = \{a^n b^n | n \geq 0\}$ is not context free language.

Assume that the above language L is context free language.

Then, L must have pumping length (say) p .

Now, we take a string s such that $|s| \geq p$.

We divided the string s into $UVXY^2Z$.

Let, say $p=4$, so, $s = a^4 b^4 c^4$.

Case-I: V and Y each contain only one type of symbol.

UV^2XY^2Z

$aaabbbcccg$

Now, we have to further divide and pump statement (s.t)

Let, $i=2$

UV^2XY^2Z

$aaaaabbbccccc$

Case-II: Either the V and Y has more than one kind of symbol.

UV^2XY^2Z

$aaabbbcccg$

Now, we have to further divide and pump st.

UV^2XY^2Z

$aaabbbcccg$

Since, UV condition of pumping lemma is not satisfy. So, we can say that, the

Q. Show that $L = \{a^n b^n | n \geq 0\}$ is not context free language.

Assume that the above language L is context free language.

Then, L must have pumping length (say) p .

Now, we take a string s such that $|s| \geq p$.

We divided the string s into $UVXY^2Z$.

Let, say $p=4$, so, $s = a^4 b^4 c^4$.

Case-I: V and Y each contain only one type of symbol.

UV^2XY^2Z

$aaabbbcccg$

Now, we have to further divide and pump statement (s.t)

Let, $i=2$

UV^2XY^2Z

$aaaaabbbccccc$

Case-II: Either the V and Y has more than one kind of symbol.

UV^2XY^2Z

$aaabbbcccg$

Now, we have to further divide and pump st.

UV^2XY^2Z

$aaabbbcccg$

Since, UV condition of pumping lemma is not satisfy. So, we can say that, the

Closure properties of context free languages (CFLs):

Context free languages are closed under:

- Union
- Concatenation
- Kleene star.

Suppose $G_1 (V_1, \Sigma_1, R_1, S_1)$ and $G_2 (V_2, \Sigma_2, R_2, S_2)$

Example for G_1 :

$S_1 \rightarrow a^m b^n$ eqv: $S_1 \rightarrow a^m b^n / \epsilon$

$S_1 \rightarrow \epsilon$

For G_2 :

$S_2 \rightarrow \epsilon$ and eqv: $S_2 \rightarrow c^m d^n / \epsilon$

$S_2 \rightarrow \epsilon$

Let L_1 & L_2 be the CFLs generated by G_1 & G_2 then,

$L_1 = L(G_1) = \{a^m b^n : m \geq 0\}$

$L_2 = L(G_2) = \{c^m d^n : m \geq 0\}$

① Union:

If L_1 and L_2 are two CFLs, then union of L_1 and L_2 i.e. $L_1 \cup L_2$ is also context free.

Union of L_1 and L_2 :

$L = L_1 \cup L_2 = \{a^m b^n \mid m \geq 0\} \cup \{c^m d^n \mid m \geq 0\}$

$= \{a^m b^n \mid m \geq 0, n \geq 0\}$ is also context free.

$G = (V_1 \cup V_2 \cup \{S\}, \Sigma, U, \Sigma_2, R, S)$ where $R = R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}$ and S is a

new symbol

$S \rightarrow S_1 / S_2$

$S_1 \rightarrow a^m b^n / \epsilon$

$S_2 \rightarrow c^m d^n / \epsilon$

Their union says either of two conditions to be true so it is also a CFL. So CFL are closed under union.

② Concatenation:

If L_1 and L_2 are two CFLs, their concatenation $L_1 \cdot L_2$ will also be context free.

eg: $L = L_1 \cdot L_2 = \{a^m b^n c^m d^n \mid m \geq 0, n \geq 0\}$ is also context free.

The corresponding grammar will be

$S \rightarrow S_1 \cdot S_2$

$S_1 \rightarrow a^m b^n / \epsilon$

$S_2 \rightarrow c^m d^n / \epsilon$

Their concatenation says 1st no. of a's should be equal to no. of b's, the no. of c's should be equal to d's. So, we can create a PDA, which will first push for a's, pop for b's, push for c's and pop for d's.

Now, it can be accepted by PDA, Hence CFLs are closed under concatenation.

③ Kleene Closure:

If L_1 is context free, its Kleene Closure L_1^* will be also be context free.

$S_1 \rightarrow a^m b^n / \epsilon$

$S \rightarrow S_1 S_1 / \epsilon$

Then,

$L^* = \{a^n b^n : n \geq 0\}$ is also context free.
So, CFL are closed under Kleene Closure.

Decision Algorithm for context free languages (CFL):

There are following decision algorithms for CFL.

1. Algorithm for deciding whether a CFL is empty or not (emptiness).
2. Algorithm for deciding whether a CFL is finite or not (finiteness).
3. Algorithm for deciding whether any string 'w' can be generated by the same cfl or not. (Membership).

#1) Algorithm for deciding whether a given CFL is empty or not (emptiness):

1. Checking whether the given cfl generates empty language or not.

Algorithm:

Step-1: Eliminate useless variables from the given grammar.

Step-2: If the starting symbol also includes the useless variable then the grammar generates empty language. Otherwise, non-empty language.

Eg:-

$S \rightarrow XY$
 $X \rightarrow AA \mid AX$
 $A \rightarrow a$
 $B \rightarrow BY$
 $Y \rightarrow b$

Since, $B \rightarrow BY$ doesn't use or not terminate so, eliminate useless variable.

$S \rightarrow XY$
 $X \rightarrow AA \mid AX$
 $A \rightarrow a$
 $Y \rightarrow b$

Since, the given grammar is non-empty.

Eg: $S \rightarrow aSb \mid Sb \mid Sa$

$\rightarrow aSb$
 $\rightarrow aSbb$
 $\rightarrow aSabb$

S does not end, so, empty language.

2) Finiteness: Checking whether the given grammar generates finite or infinite language.

Algorithm:

Step-1: Simplify the grammar.

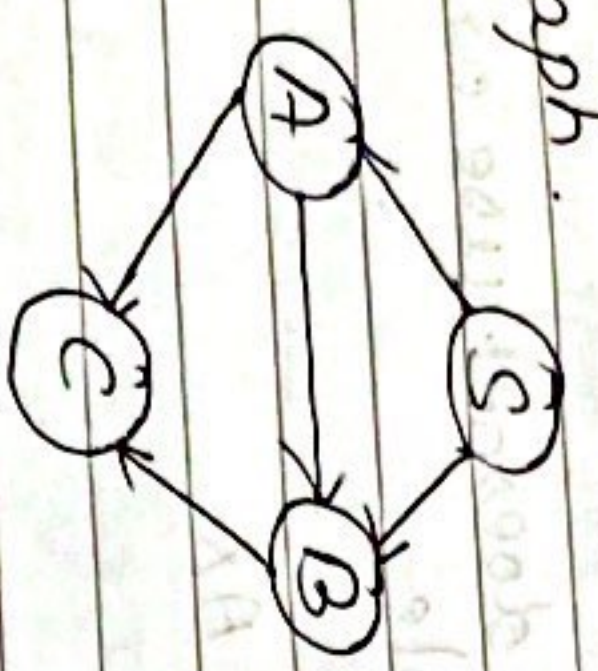
Step-2: Convert it into CNF.

Step-3: Construct the CNF graph (only variables)

Step-4: If the graph contains any loop or cycle then the grammar generates infinite language.

eg: $S \rightarrow AB$
 $A \rightarrow BC/a$
 $B \rightarrow cc/b$
 $C \rightarrow a$

CNF graph.



There is no loop/cycle hence, the given grammar is finite.

③ Membership Algorithm.

Checking whether the given string 'x' is a member of the given cfl or not.
This concept is also called as parsing algorithm.
→ Also done using CYK Algorithm.

Check whether the string 'baaba' is a member of the following grammar or not.

$S \rightarrow AB$	$BAABc$	$S \rightarrow BC$	$baac$
$\rightarrow BAB$	$baaba$	$\rightarrow AC$	baa
$\rightarrow BAC$	$baaba$	$\rightarrow BA$	baa
$\rightarrow baaba$	$baaba$	$\rightarrow AB$	baa
$\rightarrow baaba$	$baaba$	$\rightarrow BA$	baa
$\rightarrow baaba$	$baaba$	$\rightarrow BA$	baa

$$P(m) = n+13$$

$S \rightarrow BC$
 $\rightarrow BAB$
 $\rightarrow BACC$
 $\rightarrow baaba$

Q: Design a Turing machine that accepts the language $L = \{a^n b^n \mid n \geq 1\}$ over the alphabet $\Sigma = \{a, b, c\}$ [Sol: fall/spring] [2013 Fall]



language $L = \{a^n b^n \mid n \geq 1\}$ over the alphabet $\Sigma = \{a, b, c\}$ [Sol: fall/spring] [2013 Fall]

classmate
Date _____
Page _____

$L(n) = n+1$
 input $n=2 \Rightarrow$ unary system = 11

8	7	6	5	4	3	2	1	0

 $\frac{1}{2}$

Diagram illustrating a sequence of states q_0, q_1, q_2, q_3 connected by transitions labeled $1/1, R$ and $1/1, L$. The sequence is shown as a horizontal chain of circles representing states, with arrows indicating transitions between them. The transitions are labeled with the input/output pair and the action (R for Right, L for Left). The sequence starts at q_0 , goes to q_1 (labeled $1/1, R$), then to q_2 (labeled $1/1, R$), and finally to q_3 (labeled $1/1, L$). There is also a self-loop on q_1 labeled $1/1, R$.

PC
① $\rightarrow q_0$ $(a+ab+aa)^* b^* a^* b^* a$ $\rightarrow q_f$

