CO-1

Symbol :- a, b, c, 0, 1, 2 . . . .

Alphabet :- Σ - Collection of symbols Eg:- {a, b}, {d, e, f, g}

String - Sequence of Symbols. Eg:- a, b, 0, 1, aa, bb, ab, 01

Language - Set of strings.

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

$L_1 = \{00, 01, 10, 11\}$

$L_2 = \{000, 001, 010, 011, 100, 101, 110, 111\}$ } finite

L_3 = set of all string begin with 0

$L_3 = \{0, 00, 01, 000 \ldots\}$ Infinite

Powers of Σ :- $\Sigma = \{0, 1\}$

Σ^0 = set of all strings of length 0 : $\Sigma^0 = \{\epsilon\}$

Σ^1 = set of all strings of length 1 $\Sigma^1 = \{0, 1\}$

Σ^2 = length 2 ; $\Sigma^2 = \{00, 01, 10, 11\}$

Cardinality :- number of elements in a set

$\Sigma^n = 2^n$

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3$

$= \{\epsilon\} \cup \{0, 1\} \cup \{00, 01, 10, 11\} \cup \ldots$

= set of all possible strings of all lengths over $\{0, 1\}$

Finite Automata

- FA with o/p
  - moore machine
  - meeley machine.
- FA without o/p.
  - DFA
  - NFA
  - ϵ NFA

DFA = Deterministic finite Automata.

* Simplest model & limited memory.

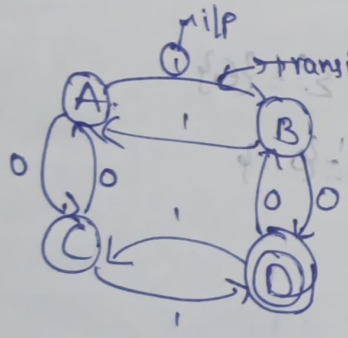
* Every DFA defined by 5 tuples

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

Q = set of all states Σ = inputs

q = start state. F = set of final states

δ = transition function.



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

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

$q_0 = A$

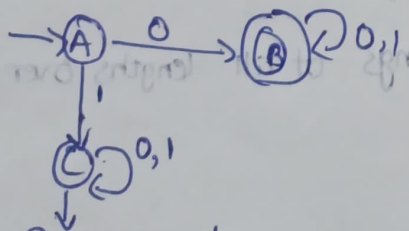
$F = \{D\}$

$\delta =$

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

DFA-1 :- L_1 = set of all strings that start with '0'

$= \{0, 00, 01, 000, 010, 011, \ldots\}$

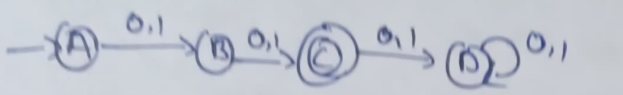


Dead state / trap state.

* The current state we know what next state will be
* It has one unique next state.
* It has no choices or randomness
* It is easy & simple to design

DFA-2 :- Construct a DFA that Accepts sets of all strings over {0,1} of length 2

$\Sigma = \{0,1\}$ $L = \{00, 01, 10, 11\}$

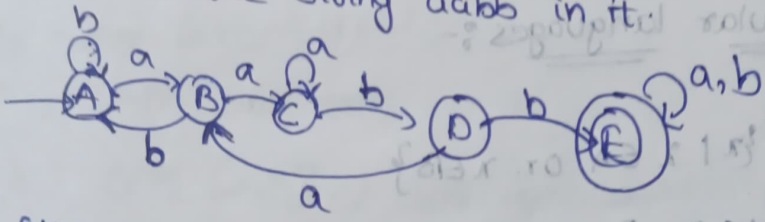


DFA-3 :- Construct a DFA that Accepts any strings over {a,b} that doesn't contain the aabb in it

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

Make above problem simple.

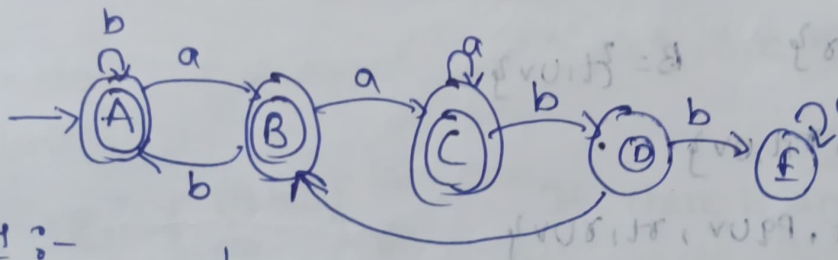
let us construct a DFA that Accepts all strings over {a,b} that contains the string "aabb" in it.



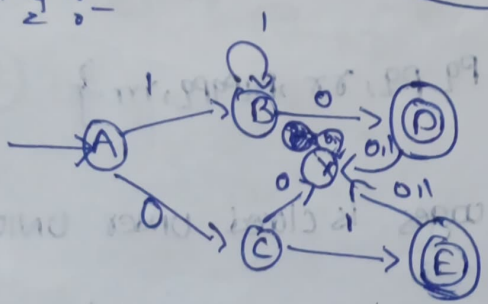
- flip the states.

to flip state - 1) To make final state into non-final state.

2) Make non-final state into final state.



DFA-4 :-



10

1 1 1 0 ✓

A B D

01 ✓

L = {Accepts the string 01 or a string of atleast one '1' followed by a 0}

one binary digit '1'

Regular Language :- A language is said to be RL if and only if some finite state machine recognizes it

What lang are not REG?

The languages which are not recognized by any FSM. which require memory.

① - memory of FSM is very limited

② - It cannot store or count strings.

Eg:- ① ababbabbb ② $a^N b^N$ aaa bbb

Operations on Regular languages :-

UNION - $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

CONCATENATION - $A \circ B = \{xy \mid x \in A \& y \in B\}$

STAR - $A^* = \{x_1, x_2, x_3 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

Eg:- $A = \{pq, r\}$ $B = \{t, uv\}$

$A \cup B = \{pq, r, t, uv\}$

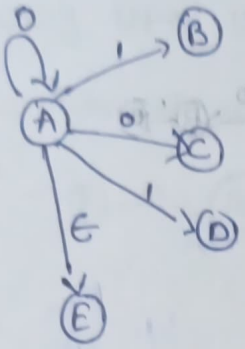
$A \circ B = \{pqt, pquv, rt, ruv\}$

$A^* = \{\epsilon, pq, r, pqr, rpq, pqpq, rr, pqpqpq, rrr \ldots\}$

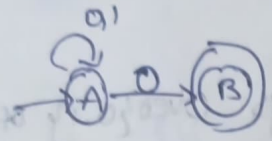
Theorm 1 :- The class of Regular languages is closed under UNION

Theorm 2 :- The class of Regular languages is closed under CONCATENATION.

NFA :-



* In NFA, the current state there could be multiple next states
* The next state may be chosen at random
* All the next states may be chosen in the parallel



L = {set of all strings that end with 0}.

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

Q - set of all states = {A, B}

Σ - inputs. = {0, 1}

q_0 - start state = A

f = set of final states = B

$\delta = Q \times \Sigma \rightarrow 2^Q$

$A \times 0 \rightarrow A$

$A \times 0 \rightarrow B$

$A \times 1 \rightarrow A$

$B \times 0 \rightarrow \phi$

$B \times 1 \rightarrow \phi$.

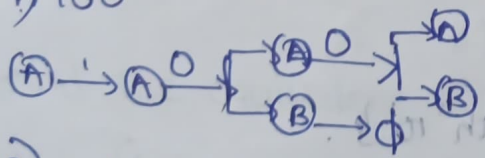
$A' \rightarrow A, B, AB, \phi$ - $2^2 = 4$.

3 states A, B, C. - $2^3 = 8$

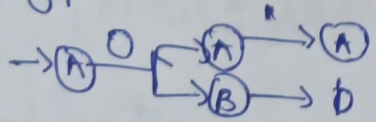
$A' \rightarrow A, B, C, AB, AC, BC, ABC, \phi$

Eg: for above

1) 100



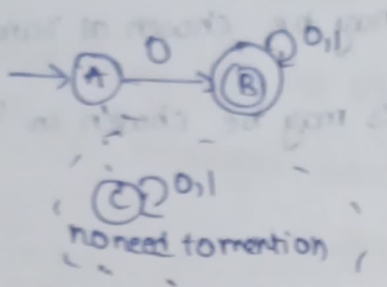
2) 01



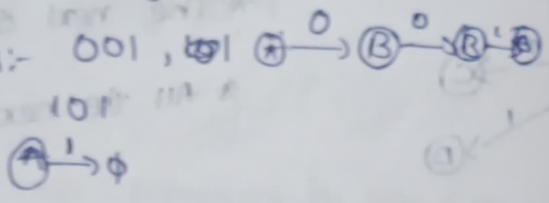
If there is any way to run the machine that ends in any set of states out of which atleast one state is a final state then NFA accepts.

NFA - 2 :- 1) L = {set of all strings that start with 0}

= {0, 00, 01, 000, ... }



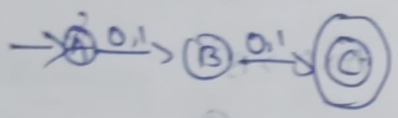
Eg:- 001, 101



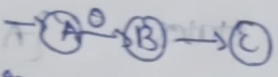
2) Construct a NFA that accepts sets of all strings over {0,1} of length 2

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

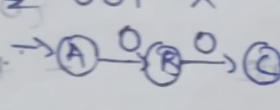
$L = \{00, 01, 10, 11\}$



Eg:- 00 ✓

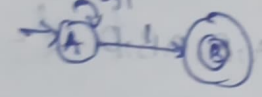


Eg:- 001 ×



NFA - 3 :- 1).

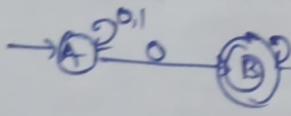
1) L_1 = {set of all strings that ends with '1'}



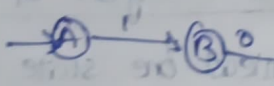
01, 001, 0001, 0*1, 1 ✓

101, 1101 ✓

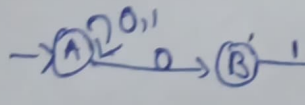
2) L_2 = {set of all strings that contain '0'}



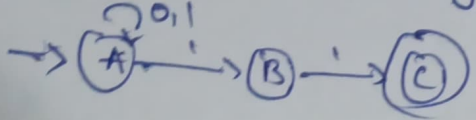
3) L_3 = {set of all strings start with '10'}.



4) L_4 = {set of all strings that contain '01'}.



5) L_5 = {set of all strings that ends with '11'}

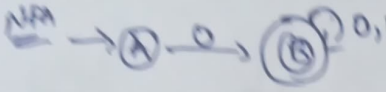


Conversion of NFA to DFA:

* Every DFA is a NFA, but not v/v

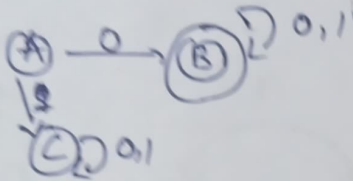
① L = {set of all strings (0,1) starts with '0'}

NFA



	0	1
A	B	ϕ
B	B	B

DFA

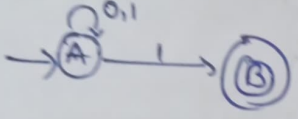


	0	1	
A	B	C	→ Dead state
B	B	B	
C	C	C	

② L = {set of all strings over (0,1) that ends with '1'}.

$\Sigma = 0,1$

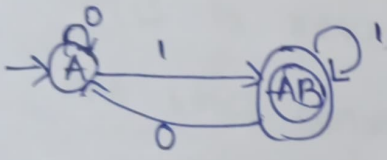
NFA



	0	1
A	{A}	{A,B}
B	ϕ	ϕ

{Subset Construction method}

DFA

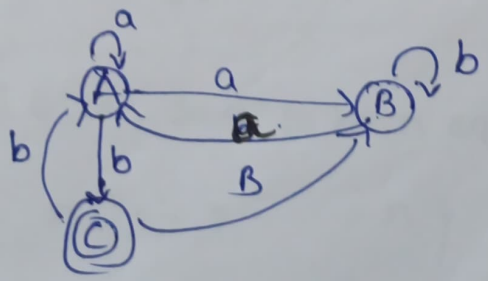


	0	1	
A	A	{AB}	→ Combined state
AB	A	{AB}	

③ find the equivalent DFA for the NFA given by

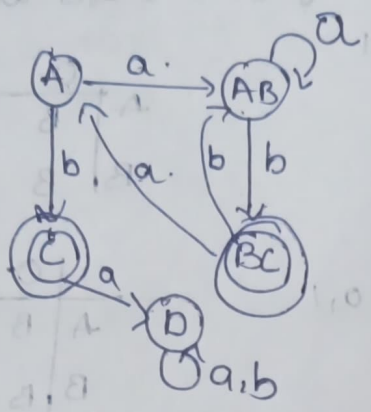
$M = [\{A,B,C\}, (a,b), \delta, A, \{C\}]$ where δ is given by:

	a	b
→ A	A,B	C
B	A	B
C	-	A,B

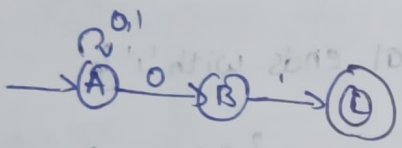


	a	b
A	AB	C
AB	AB	BC
BC	A	AB
C	D	AB
D	D	D

The state in NFA Ns final, in dFA all the in NFA are final.

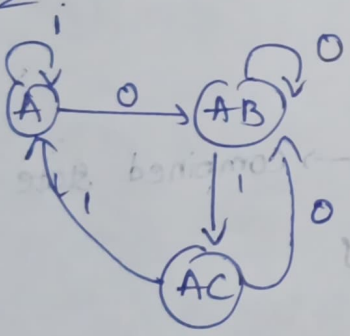


NFA-4:- L = { set of all strings over (0,1) that ends with '01' }



	0	1
A	A,B	A
B	ϕ	C
C	ϕ	ϕ

DFA

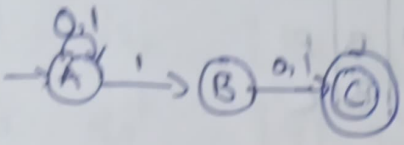


1101 ✓
1110 ✗

	0	1
A	AB	A
AB	AB	AC
AC	AB	A
B		

NFA 5°- Design an NFA for an language that accepts all strings over {0,1} in which the second last symbol always '1'. Then convert it into equivalent DFA

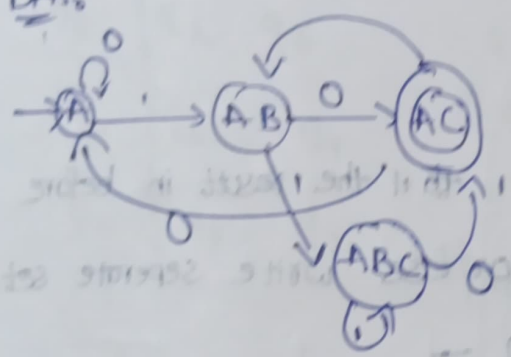
NFA:-



	0	1
A	{A}	{A,B}
B	C	C
(C)	ϕ	ϕ

Eg:- 1010

DFA:-



	0	1
A	A	AB
AB	AC	ABC
(AC)	A	AB
(ABC)	AC	ABC

1 0 1 0
1 0 1 0 1 0

Minimization of DFA :-

It is is required to obtain the minimal version of any DFA which consists of min no of states.

Two states 'A' and 'B' are said to be equivalent if

$\delta(A,x) \to$ final state
&
$\delta(B,x) \to$ final state

OR

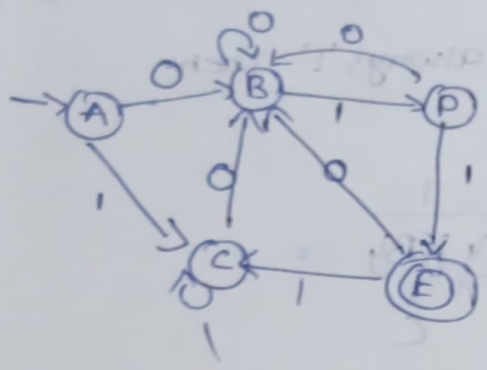
$\delta(A,x) \to f$
&
$\delta(B,x) \to f$

where 'x' is any input string

if $|x| = n$ (length of x) then A & B are said to be n equivalent

where $n = \{0,1,2,\ldots n\}$

minimize DFA :-



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

nonfinal final

0 equivalence - {A, B, C, D} {E}

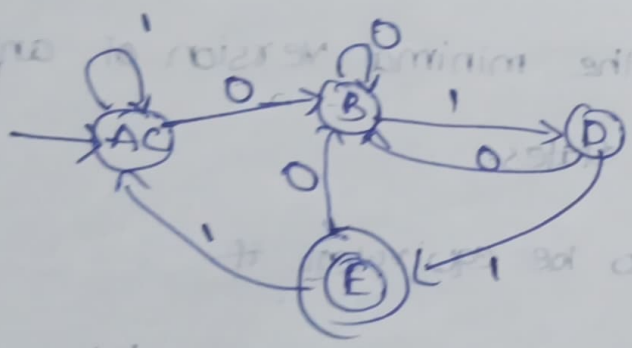
1 Equivalence - {A, B, C} {D} {E}

* check A, B where goes on 0 & 1 if the result in before (0 equivalance) write in the set or else write seperate set

2 equvilance - {A, C} {B} {D} {E}.

3 equvilance - {A, C} {B} {D} {E}

→ if two consecutive rows are same stop process



minimize DFA -28-

	0	1
q_0	q_1	q_5
q_1	q_6	q_2
q_2	q_0	q_2
q_3	q_2	q_0
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

0 -equivalance

$\{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}$ $\{q_2\}$

1-equivalence

$\{q_0, q_4, q_6\}$ $\{q_1, q_7\}$ $\{q_3, q_5\}$ $\{q_2\}$

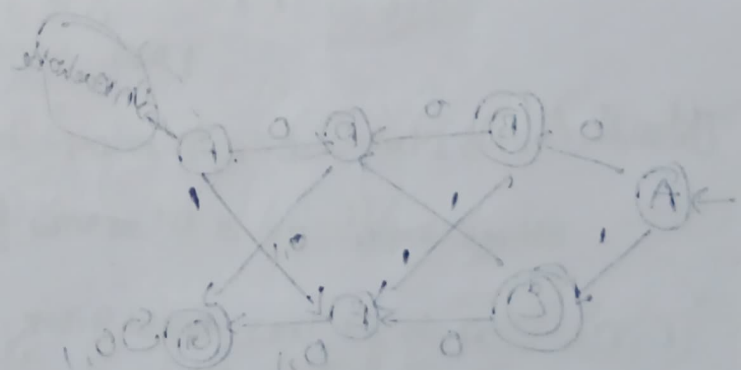
2-equivalence

$\{q_0, q_4\}$ $\{q_6\}$ $\{q_1, q_7\}$ $\{q_3, q_5\}$ $\{q_2\}$

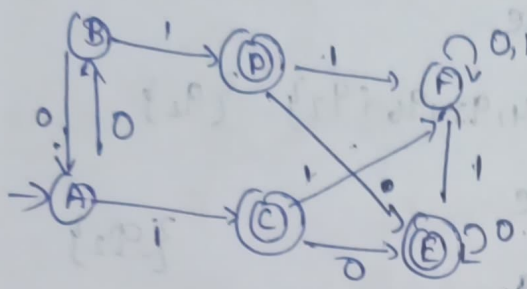
3-equivalence

$\{q_0, q_4\}$ $\{q_6\}$ $\{q_1, q_7\}$ $\{q_3, q_5\}$ $\{q_2\}$.

	0	1
$\Rightarrow\{q_0 q_4\}$	$\{q_1, q_7\}$	$\{q_3 q_5\}$
$\{q_6\}$	q_6	$\{q_0 q_4\}$
$\{q_1, q_7\}$	$\{q_6\}$	$\{q_2\}$
$\{q_3 q_5\}$	$\{q_2\}$	$\{q_0 q_4\}$
$\{q_2\}$	$\{q_0 q_4\}$	$\{q_2\}$.



minimize the following DFA :-

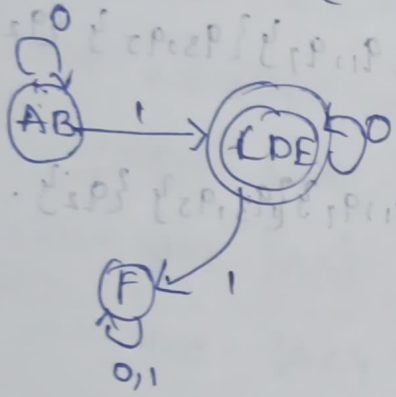


	0	1
→ A	B	C
B	A	D
C	E	f
D	E	f
E	E	f
F	f	f

0-Equivalence - {A, B, F} {C, D, E}

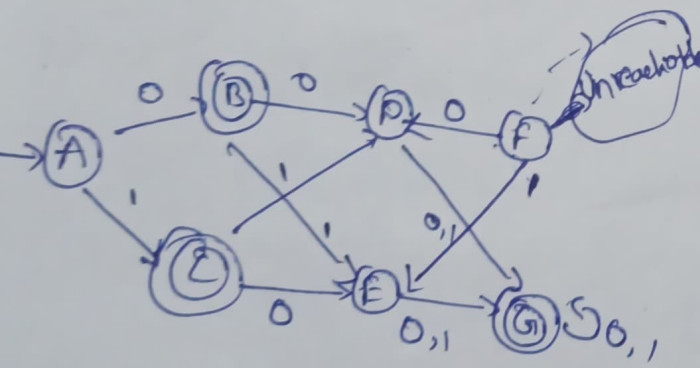
1-equivalence - {A, B} {F} {C, D, E}

2-equi - {A, B} {F} {C, D, E}



	0	1
→ {AB}	{AB}	{CDE}
{f}	f	f
{CDE}	CDE	f

minimize -4 :- When there are unreachable states involved



A state is said to be unreachable if there is no way it can be reached from the initial state.

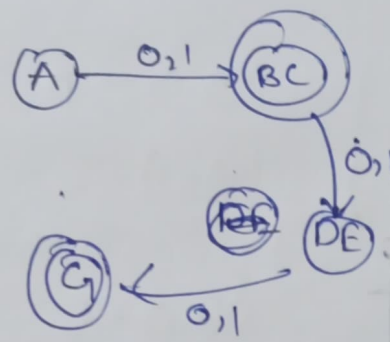
	0	1
→A	B	C
(B)	D	E
(C)	E	D
D	G	G
E	G	G
(G)	G	G

0-Equivalence : {A,D,E} {B C G}

1-equivalence :- {A,D E} {B,C} {G}

2-equivalence :- {A} {D,E} {B,C} {G}

3-equivalence :- {A} {D,E} {B,C} {G}



	0	1
A	B,C	BC
DE	G	G
(BC)	DE	DE
(G)	0	0

Myhill-Nerode.

Steps

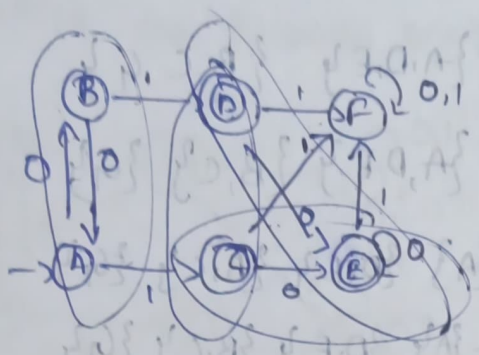
1) Draw a table for all pairs of states (P,Q)

2) Mark all pairs where P∈F and Q∉F

3) If there are any unmarked pairs (P,Q) such that [δ(P,x), δ(Q,x)] is marked, then mark [P,Q] where 'x' is an input symbol

REPEAT THIS UNTIL NO MORE MARKINGS CAN BE MADE.

4) Combine all unmarked pairs and make them a single state in minimized DFA



	A	B	C	D	E	F
A						
B						
C	✓	✓				
D	✓	✓				
E	✓	✓				
F	✓	✓	✓	✓	✓	

(B,A) - $\delta(B,0) = A$, $\delta(B,1) = D$
$\delta(A,0) = B$, $\delta(A,1) = C$

(D,C) - $\delta(D,0) = E$, $\delta(D,1) = F$
$\delta(C,0) = E$, $\delta(C,1) = F$ (Repeat again)

(E,C) - $\delta(E,0) = E$, $\delta(E,1) = f$
$\delta(C,0) = E$, $\delta(C,1) = f$ (still no change)

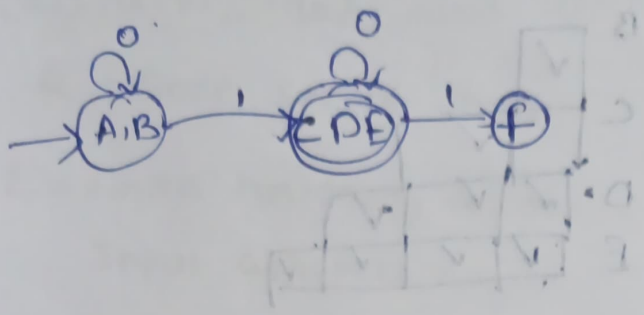
✓ (E,D) - $\delta(E,0) = E$, $\delta(E,1) = f$
$\delta(D,0) = E$, $\delta(D,1) = f$

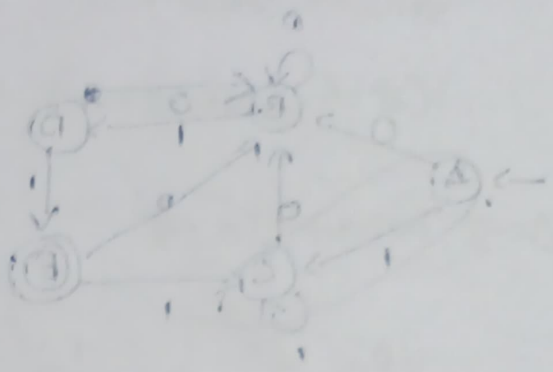
(F,A) - $\delta(F,0) = F$, $\delta(F,1) = f$
$\delta(A,0) = B$, $\delta(A,1) = C$

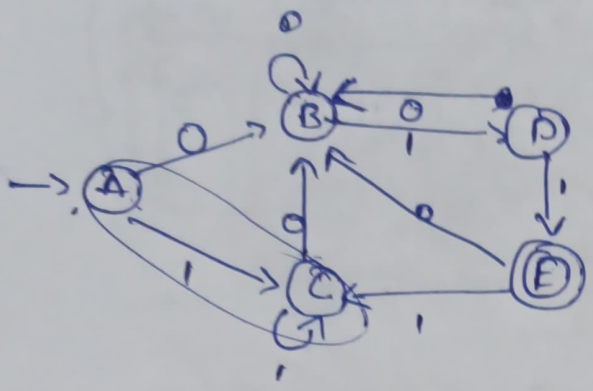
(F,B) - $\delta(F,0)$ F
$\delta(B,0)$ A

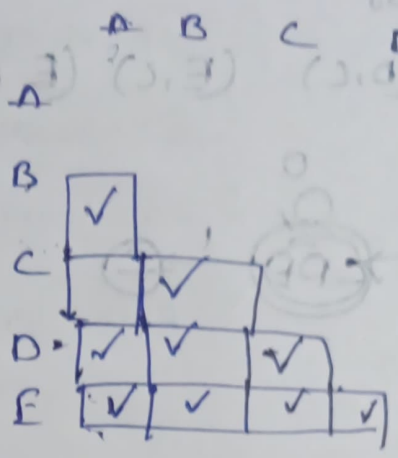
Unmark pairs:-

(A,B) (D,C) (E,C) (E,D)









BA = $\delta(B,0) - B$ | $\delta(B,1) - D$

$\delta(A,0) - B$ | $\delta(A,1) - C$

AC $\delta(A,0) - B$ | $\delta(A,1) - C$

$\delta(C,0) - B$ | $\delta(C,1) - C$

~~BB~~ $\delta(C,0) - B$ | $\delta(C,1) - C$

$\delta(B,0) - B$ | $\delta(B,1) - D$

DA $\delta(D,0) - B$ | $\delta(D,1) - E$

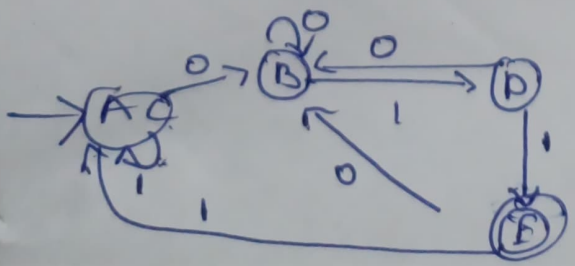
$\delta(A,0) - B$ | $\delta(A,1) - C$

DB $\delta(D,0) - B$ | $\delta(D,1) - E$

$\delta(B,0) - B$ | $\delta(B,1) - D$

D,C $\delta(D,0) - B$ | $\delta(D,1) - E$

$\delta(C,0) - B$ | $\delta(C,1) -$



FA with outputs

mealy machine

$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$

Q = Finite set of states

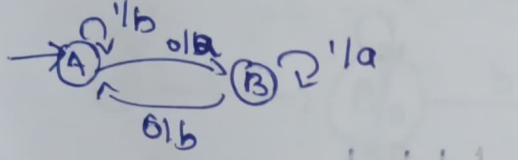
Σ = finite non-empty set of Input alphabets

Δ = The set of o/p Alphabet

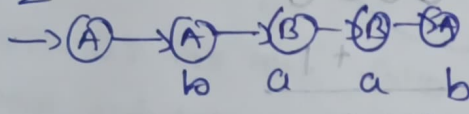
δ = Transition function: $Q \times \Sigma \to Q$

λ = o/p function $\Sigma \times Q \to \Delta$

q_0 = Initial state



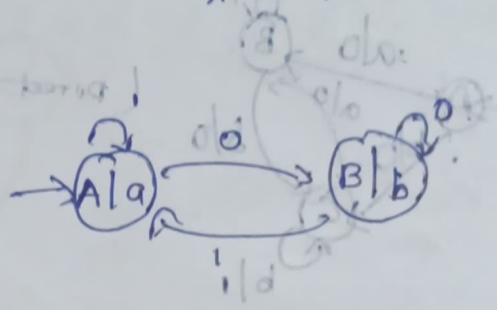
Eg:- 1010



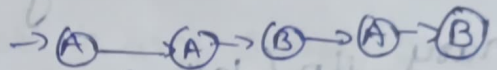
moore machine

$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$

λ = o/p function $Q \to \Delta$

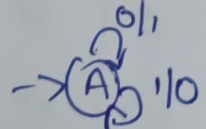


Eg:- 1010

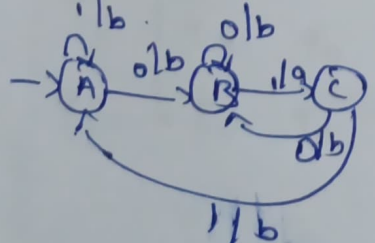


n->n+1.

mealy-1:- Construct a mealy machine that produces the 1's Complement of any binary input string.



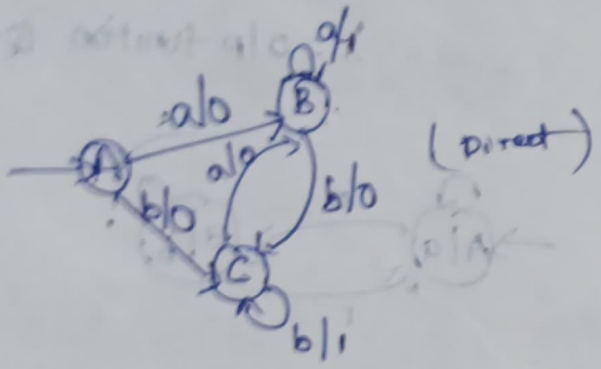
mealy-2:- Construct a mealy that print 'A' whenever the sequence '01' is encountered in any i/p Binary string.



0110
baabb

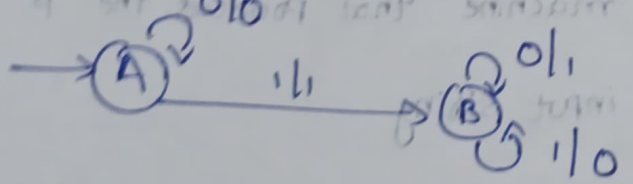
(First DFA then mealy)

Mealy-3 :- Design a mealy accepts language consist of string from Σ where $\Sigma = \{a, b\}$ and strings should end either aa (or) bb



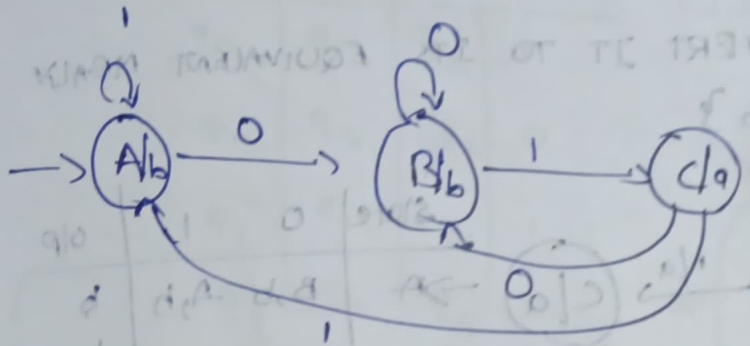
mealy-4 :- Construct mealy that gives 2's complement of any binary i/p (last carry bit is neglected)

10100	11100	1111
01011	00011	0000
+1	+1	+1
01100	00100	0001

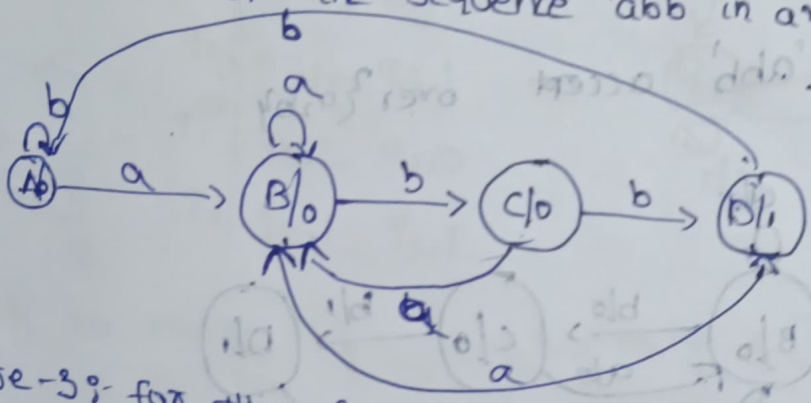


moore:- construct a more that Print 'a' whenever sequence 'or' is encountered. in any input binary string

$\Sigma = \{0,1\}$
$\Delta = \{a,b\}$



moore-2:- Construct a moore machine that counts the occurences of the sequence 'abb' in any strings over $\{a,b\}$.



moore-3:- for the following moore machine input alphabet is $\Sigma = \{a,b\}$ output alphabet $\Delta = \{0,1\}$ run the following i/p sequences find the respective outputs (i) aabab (ii) abbb (iii) abaabb

States	a	b	outputs
→ q_0	q_1	q_2	0
q_1	q_2	q_3	0
q_2	q_3	q_4	1
q_3	q_4	q_4	0
q_4	q_0	q_0	0

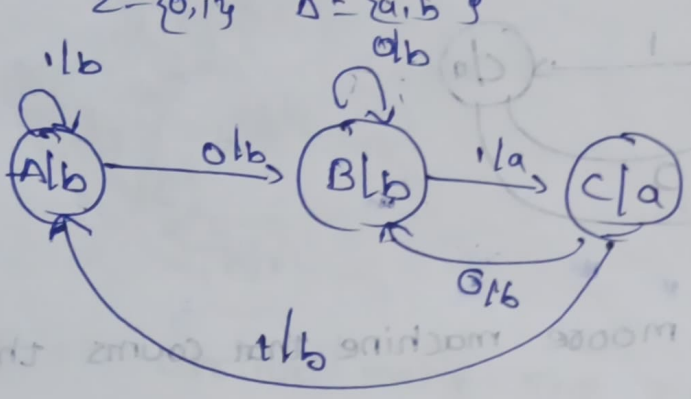
1) 0 0 1 0 0 0

2) 0 0 0 0 0 0

3) ~~a b~~ 0 0 0 0 0 1

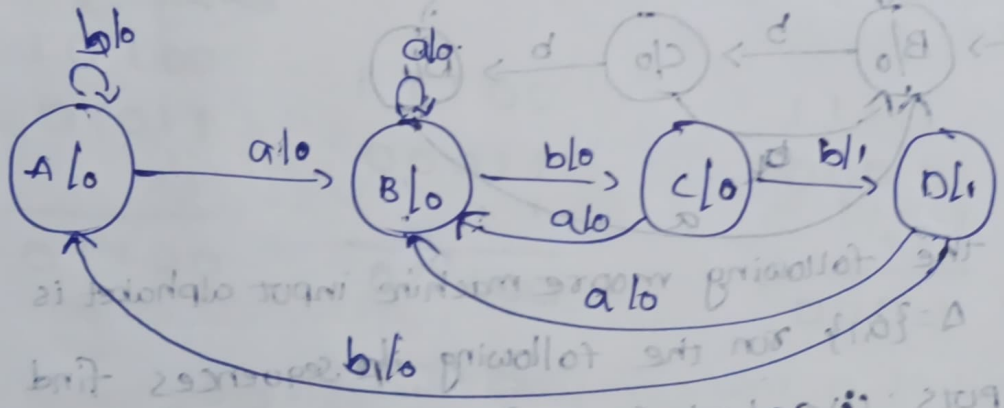
Conversion of moore to mealy① - Construct a moore that prints 'a' whenever the sequence '01' is encountered in any input binary string and then CONVERT IT TO ITS EQUIVALENT MEALY

$\Sigma = \{0,1\}$ $\Delta = \{a,b\}$



State	0	1	o/p
→A	B,b	A,b	b
B	B,b	C,a	b
C	B,b	A,b	a

moore to mealy② - 'abb' accept over $\{a,b\}$

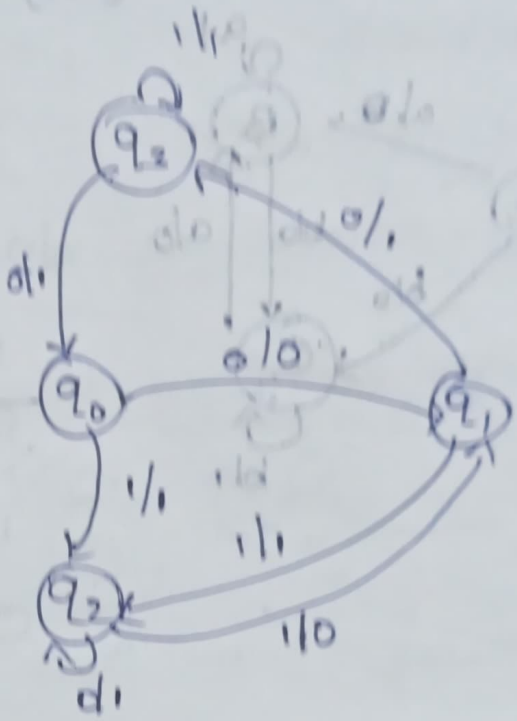


State	a	b
→ A	B,0	A,0
B	B,0	C,0
C	B,0	D,1
D	B,0	A,0

moore to mealy :- $\Sigma = \{0,1\}$ $\Delta = \{0,1\}$

moore:

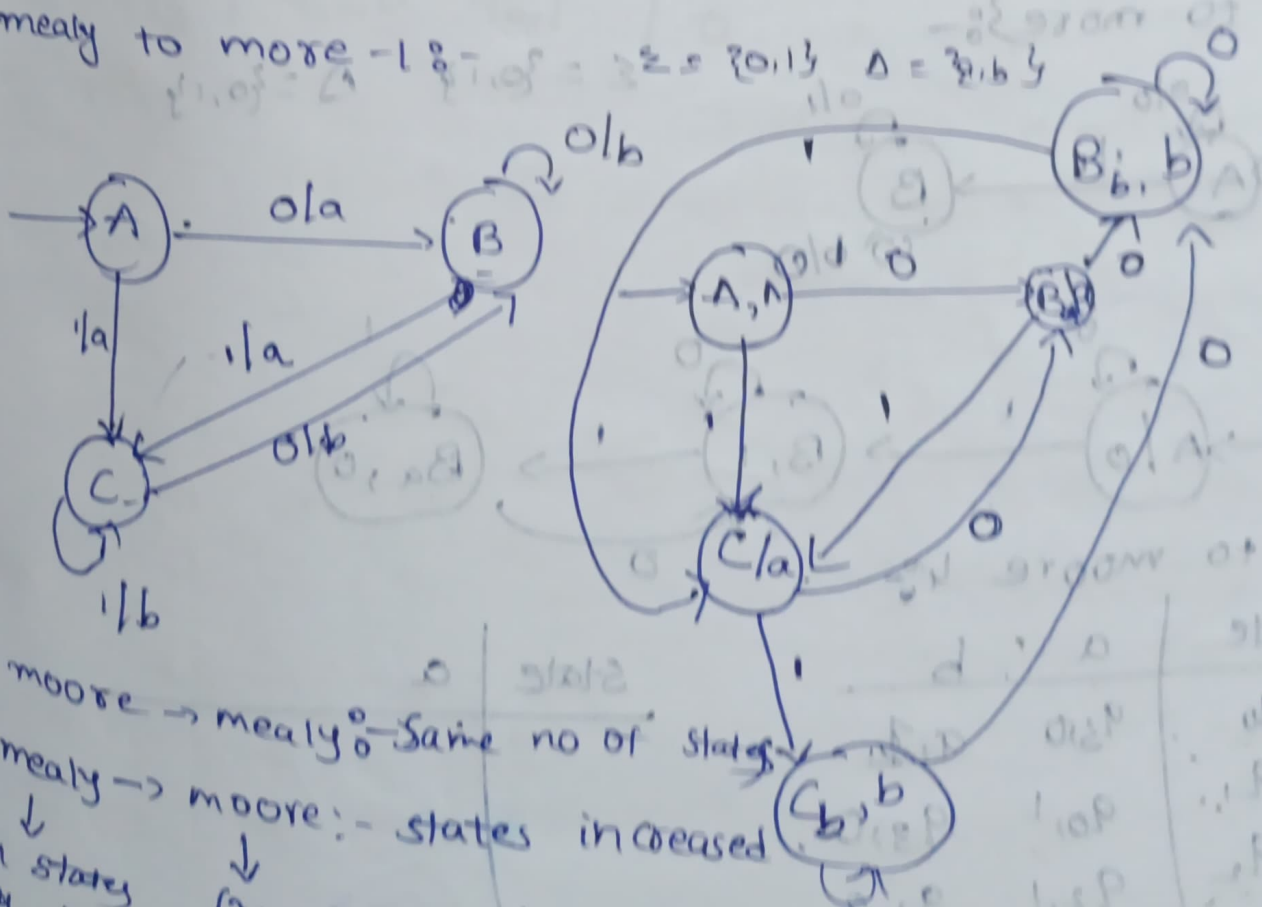
state	0	1	o/p
q_0	q_1	q_2	1
q_1	q_3	q_2	0
q_2	q_2	q_1	1
q_3	q_0	q_3	1



mealy

state	0	1
→ q_0	q_1,0	q_2,1
q_1	q_3,1	q_2,1
q_2	q_2,1	q_1,0
q_3	q_0,1	q_3,1

mealy to moore :- $\Sigma = \{0,1\}$ $\Delta = \{a,b\}$



moore → mealy :- Same no of states

mealy → moore :- states increased

↓

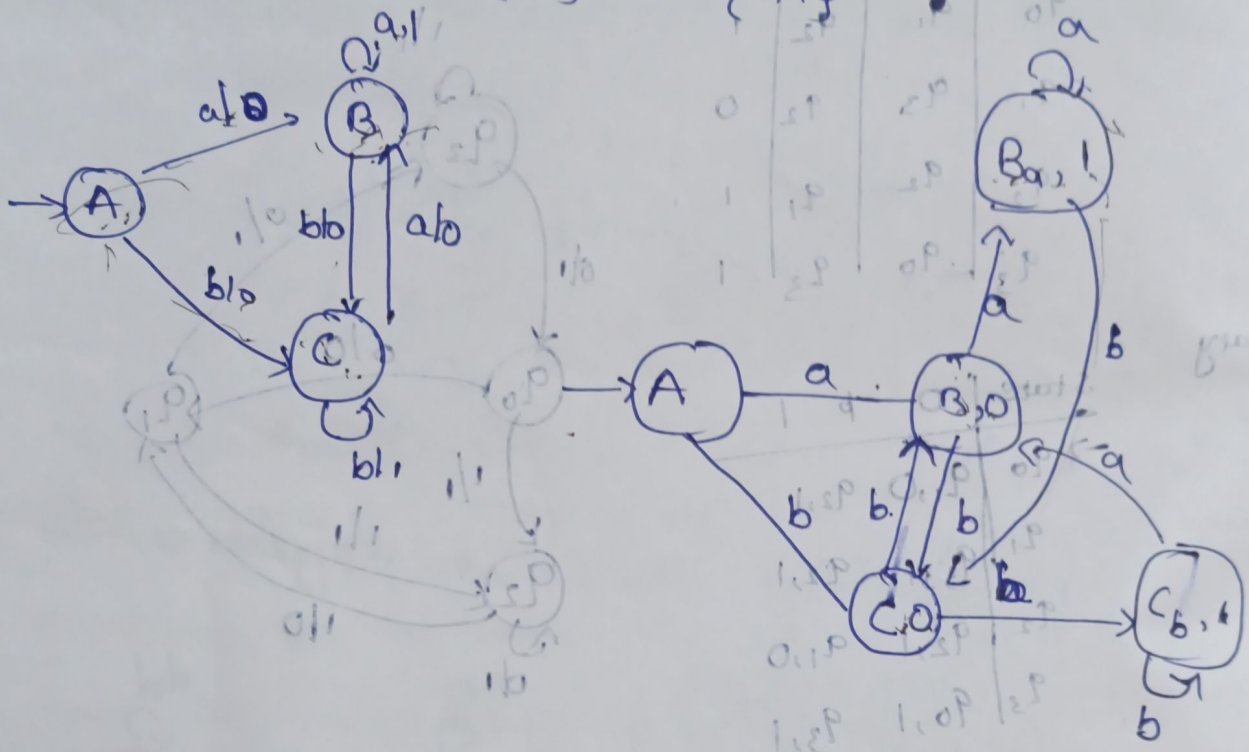
x states

y o/p

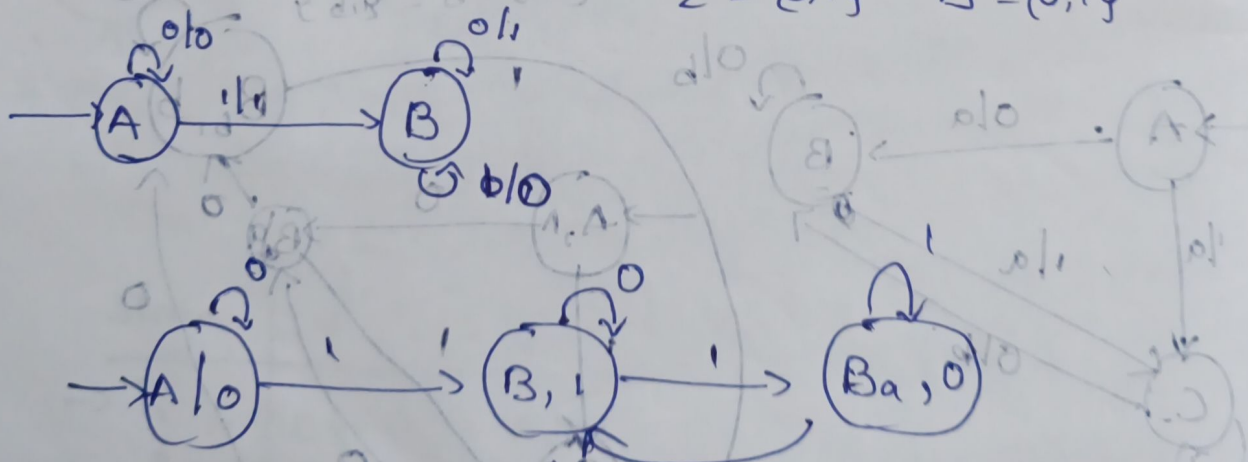
↓

(x × y) no of states of max

mealy to moore-2: given mealy prints '1' whenever sequence 'aa' or 'bb' is encountered in any input binary string from Σ^* where $\Sigma = \{a, b\}$ $\Delta = \{0, 1\}$



mealy to more3:- $\Sigma = \{0, 1\}$ $\Delta = \{0, 1\}$

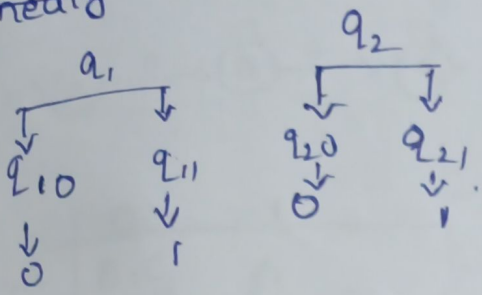


mealy to moore 4:-

State	a	b
→ q_0	q_3,0	q_1,1
q_1	q_0,1	q_3,0
q_2	q_2,1	q_2,0
q_3	q_1,0	q_0,1

State	a

meaty



State	a	b	o/p
→ q_0	$q_3, 0$	$q_{11}, 1$	1
q_{10}	$q_0, 1$	$q_3, 0$	0
q_{11}	$q_0, 1$	$q_3, 0$	1
q_{20}	$q_{21}, 1$	$q_{20}, 0$	0
q_{21}	$q_{21}, 1$	$q_{20}, 0$	1
q_3	$q_{10}, 0$	$q_0, 1$	0

Epsilon (ϵ) - NFA

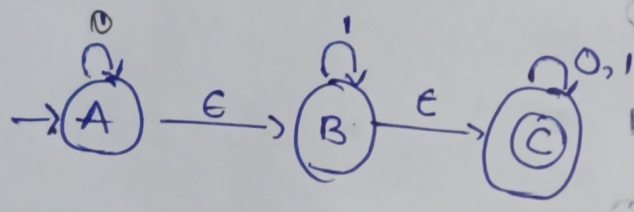
ϵ = Empty Symbol.

$\{Q, \Sigma, q_0, \delta, F\}$.

$\delta : Q \times \Sigma \cup \epsilon \rightarrow 2^Q$.

every state on ϵ goes to itself

Conversion of ϵ-NFA to NFA



ϵ-closure(ϵ^*) - All states that can be reached from a particular state only by seeing the ϵ symbol.

ϵ^* 0 ϵ^*

A — A — A B C

B — ϕ —

C — C — C

ϵ^* 1 ϵ^*

A — ϕ —

B — B — B, C

C — C — C.

ϵ^* 0 ϵ^*

B — ϕ —

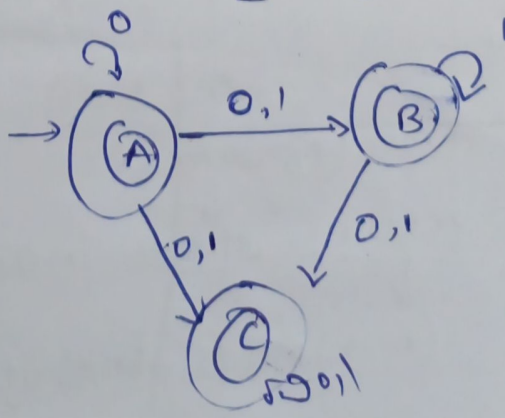
C — C — C ✓

ϵ^* 1 ϵ^*

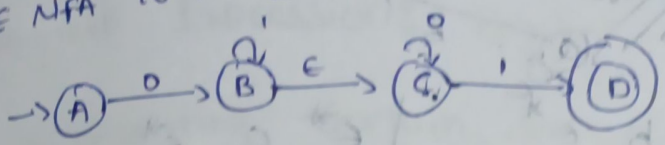
B — B — B, C

C — C — C

	0	1
→ A	A B C	B C
B	C	B C
C	C	C



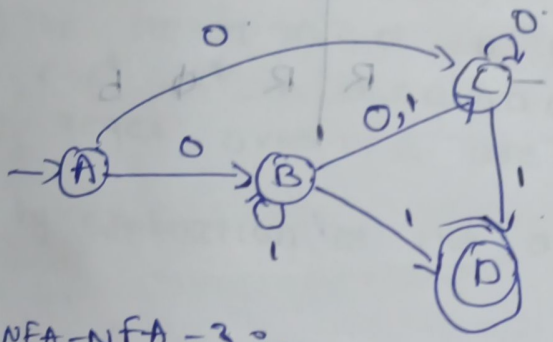
ε NFA to NFA 2 :-



	0	1
A	B, C	ϕ
B	C	BCD
C	C	D
D	ϕ	ϕ

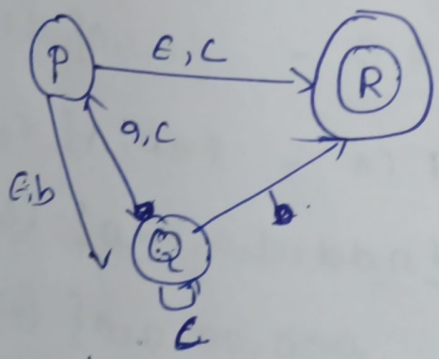
	ε*	0	ε*
A	A	B	B C
B	B	ϕ	ϕ
	C	C	C
C	C	C	C
D	D	ϕ	ϕ

	ε*	1	ε*
A	A	ϕ	ϕ
B	B	B	B C
	C	D	D
C	C	D	D
D	D	ϕ	ϕ



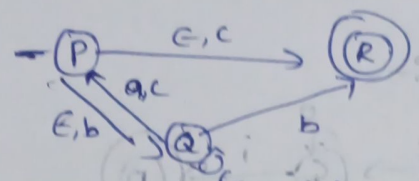
$\epsilon^* a \epsilon^*$ $\epsilon^* b \epsilon^*$ $\epsilon^* c \epsilon^*$

ε NFA - NFA - 3 :-



	a	b	c
P	PRQ	RQ	RQ
Q	PQ	R	Q
R			

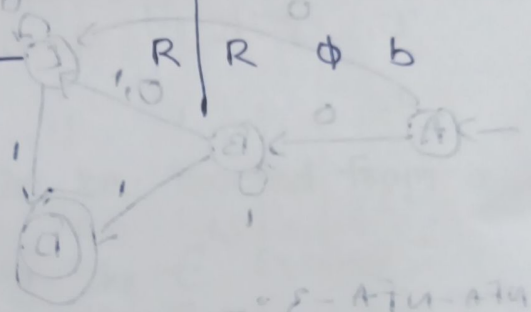
ϵNFA to NFA - 3°:-



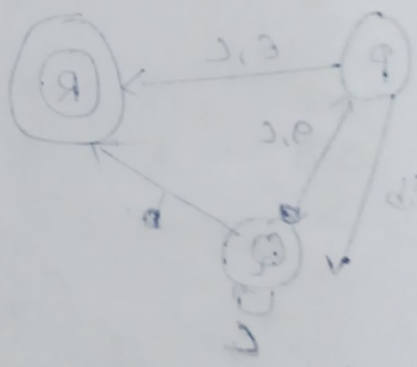
	ϵ^*	a	ϵ^*
P	P	ϕ	ϕ
Q	P	P	
R	ϕ	Q·R	
q	Q	P	PQR
∂R	R	ϕ	-

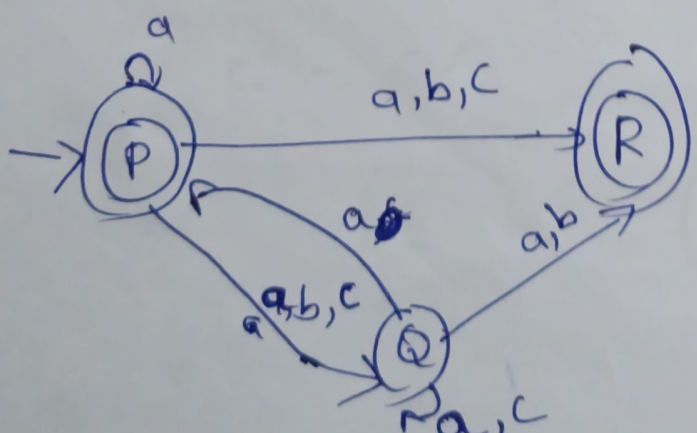
	ϵ^*	b	ϵ^*
P	P	Q	Q
Q	R	R	
R	ϕ	-	
q	Q	R	R
R	R	ϕ	-

	ϵ^*	c	ϵ^*
P	P	R	R
Q	Q	Q	
R	ϕ	-	
q	Q	Q	Q
R	R	ϕ	b



	a	b	c
→ P	PQR	QR	QR
Q	PQR	R	Q
R	ϕ	ϕ	ϕ





Regular Expressions:- Regular expressions are used for representing certain sets of strings in an algebraic fashion.

1) Any terminal symbol i.e. symbols $\in \Sigma$ including Λ and ϕ are regular expressions — $a, b, c \ldots, \Lambda, \phi$

2) The union of two reg. expressions is also a reg. exp.
$R_1, R_2 \; (R_1 + R_2)$

3) The concatination of two regex is also a regex
$R_1, R_2 (R_1 R_2)$ (or closure)

4) The iteration ($*$) of a regex is also a regex.
$R \to R^*$ $a^* = \Lambda, a, aa, aaa \ldots$

5) regex over Σ are precisely those obtained recursively by application of the above rules once or several times.

Q) Describe the following sets as Regex

1) $\{0, 1, 2\}$ — A) $R = 0 + 1 + 2$

2) $\{\Lambda, ab\}$ — A) $R = \Lambda ab$

3) $\{abb, a, b, bba\}$ — $R = abb + a + b + bba$

4) $\{\Lambda, 0, 00, 000, \ldots\}$ — $R = 0^*$ (closure of 0)

5) $\{1, 11, 111, 1111 \ldots\}$ — $R = 1^+$ (closure of 1 exclude Λ)

Identities of Regex:

1) $\phi + R = R$
2) $\phi R + R\phi = \phi$
3) $\epsilon R = R\epsilon = R$
4) $\epsilon^* = \epsilon$ and $\phi^* = \epsilon$
5) $R + R = R$
6) $R^* R^* = R^*$
7) $RR^* = R^* R$
8) $(R^*)^* = R^*$
9) $\epsilon + RR^* = \epsilon + R^* R = R^*$
10) $(PQ)^* P = P(QP)^*$
11) $(P+Q)^* = (P^* Q^*)^* = (P^* + Q^*)^*$
12) $(P+Q)R = PR + QR$ and
$R(P+Q) = RP + RQ$

ARDEN'S Theorm

If P and Q are two regular expressions over Σ, and if P doesn't contain ϵ, then following equation in R given by $R = Q + RP$ has a unique solution i.e, $R = QP^*$

$R = Q + RP \rightarrow (1)$

$= Q + QP^* P$

$= Q(\epsilon + P^* P)$

$= QP^*$

$\boxed{R = QP^*}$

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

$R = Q + RP$

$= Q + [Q + RP]P$

$= Q + QP + RP^2$

$= Q + QP + [Q + RP]P^2$

$= Q + QP + QP^2 + RP^3$

$= Q + QP \cdots \rightarrow QP^n + RP^{n+1}$ $\quad R = QP^*$

$= Q + QP + \cdots + QP^n + QP^* P^{n+1}$

$= Q\left[\epsilon + P + P^2 + \cdots + P^n + P^* P^{n+1}\right]$

$R = QP^*$

② $(1+00^*1) + (1+00^*1)(0+10^*1)^*(0+10^*1)$ is equal to $0^*1(0+10^*1)^*$

LHS $= (1+00^*1) + (1+00^*1)(0+10^*1)(0+10^*1)$

$= 1+00^*1\left[\epsilon + (0+10^*1)^*(0+10^*1)\right]$ $\quad \epsilon + R^*R = R^*$

$= (1+00^*1)(0+10^*1)^*$

$= (\epsilon + 00^*1)(0+10^*1)^*$ $\quad \epsilon \cdot R = R$

$= (\epsilon + 00^*)1(0+10^*1)^*$

$= 0^*1(0+10^*1)^* = RHS$

Designing Regex :- Design Regex for following languages over $\{a,b\}$

1) Language accepting strings of length exactly 2

$L_1 = \{aa, ab, ba, bb\}$

$R = aa + ab + ba + bb$

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

$= (a+b)(a+b)$

2) Language accept string atleast 2

$L_1 = [aa, ab, ba, bb, aaaa \ldots]$

$R = (a+b)(a+b)(a+b)^*$ → more than length 2

3) Language accept atmost 2

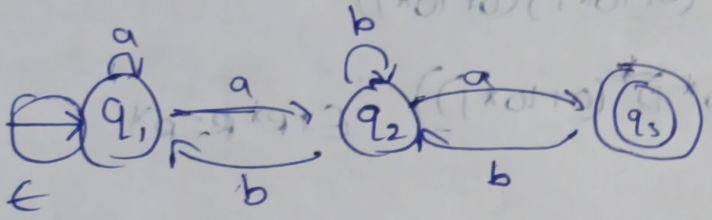
$L_1 = \epsilon, a, b, aa$

$L_1 = \{\epsilon, a, b, aa, ab, ba, bb\}$

$R = \epsilon + a + b + aa + ab + ba + bb$

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

~~Regex~~
NFA TO Regex -1°-



$q_3 = q_2 a$ — ①

② — $q_2 = q_1 a + q_2 b + q_3 b$

③ — $q_1 = q_1 a + q_2 b + \epsilon$

① $q_3 = q_2 a$

$= (q_1 a + q_2 b + q_3 b) a$

$= q_1 aa + q_2 ba + q_3 ba$ — ④

② → $q_2 = q_1 a + q_2 b + q_3 b$ $\quad q_3 = q_2 a$

$= q_1 a + q_2 b + q_2 ab$

$= q_1 a + q_2 b + q_2 ab$

$q_2 = q_1 a + q_2 (b + ab)$ $\quad R = Q + RP$

R Q R P $\quad R = QP^*$

$R_2 = q_1 a (b + ab)^*$ — ⑤

③ $q_1 = \epsilon + q_1 a + q_2 b$ ⑤ in this.

$q_1 = \epsilon + q_1 a + (q_1 a)(b + ab)^* b$ $\quad R = Q + RP$

$R = QP^*$

$q_1 = \epsilon + q_1 (a + a(b + ab)^* b)$

$q_1 = \epsilon (a + a(b + ab)^* b)^*$

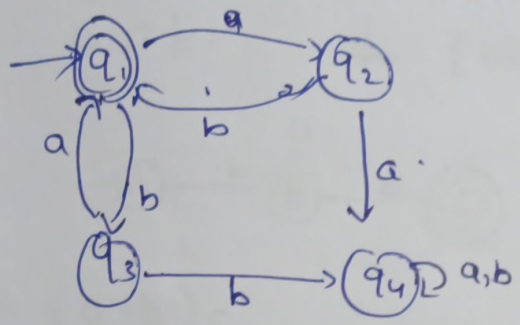
final state (q3)

$q_3 = q_2 a$

$= q_1 a (b + ab)^* a$. Puting q_2 from (5).

$= (a + a(b+ab)^* b)^* \; a(b+ab)^* a$. Puting of q_1 from (5).

DFA to Regex :-



$q_1 = \epsilon + q_2 b + q_3 a$ → (1)

$q_2 = q_1 a$ → (2)

$q_3 = q_1 b$ → (3)

$q_4 = q_3 b + q_2 a + q_4 a + q_4 b$ → (4)

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

q_2 and q_3 from (2) (3)

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

$q_1 = \epsilon + q_1 (ab + ba)$

$R \quad Q \quad R \quad P$

$R = Q + RP$

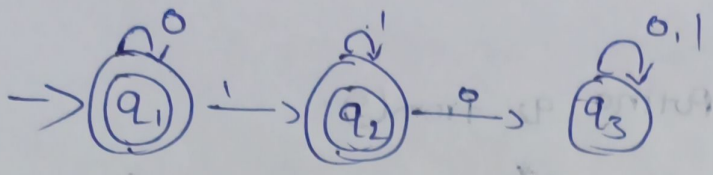
$R = QP^*$

$q_1 = \epsilon (ab + ba)^*$

$\epsilon R = R$

$q_1 = (ab + ba)^*$ – Regex.

DFA to Regex - 2 :-



$q_1 = \epsilon + q_1 0$ - (1)

$q_2 = q_1 1 + q_2 1$ - (2)

$q_3 = q_2 0 + q_3 0 + q_3 1 0$

final state (q_1)

(1) → $q_1 = \epsilon + q_1 0$

$R = Q + RP$

$R = QP^*$

$q_1 = \epsilon 0^*$

$\boxed{q_1 = 0^*}$ - (4)

final state (q_2)

$q_2 = q_1 1 + q_2 1$

$q_2 = 0^* 1 + q_2 1$ Put (4) in q_1

$R = Q + RP$

$R = QP^*$

$q_2 = 0^* 1 (1)^*$ - (3)

R = union of both final states

$R = 0^* + 0^* 1 1^*$

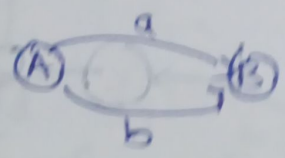
$= 0^* (\epsilon + 1 \cdot 1^*)$

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

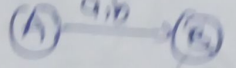
$\boxed{R = 0^* 1^*}$

Regex to FA :-

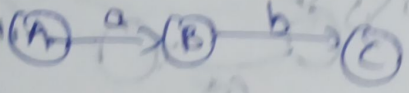
a+b



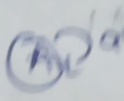
or



a.b



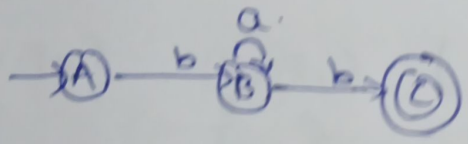
a^*



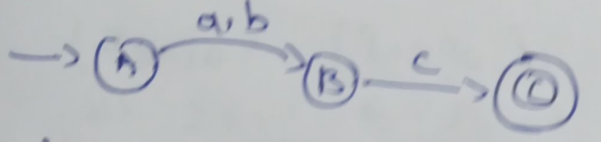
Regex to FA :-

i) ba^*b.

$L_w = \{bb, bab, baab, \ldots\}$



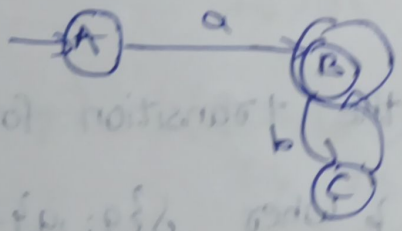
ii) $(a+b)c$



ac, bc.

iii) $a(bc)^*$

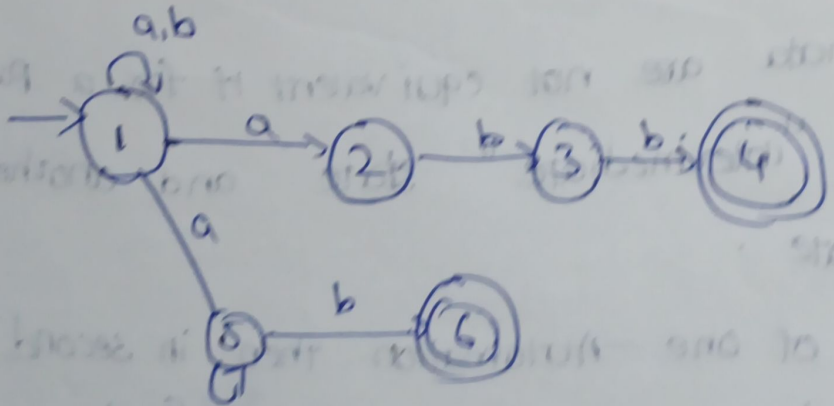
a, abc, abcbc, abcbcbc



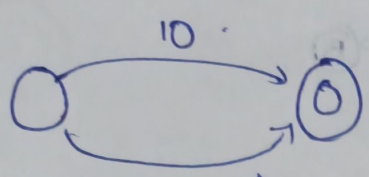
iv) $(a|b)^*(abb|a^+b)$

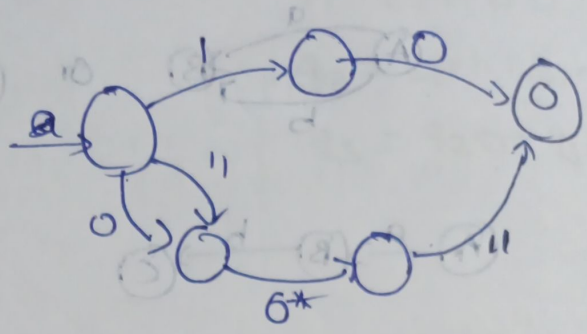
$a^+ = \{a, aa, \ldots\}$

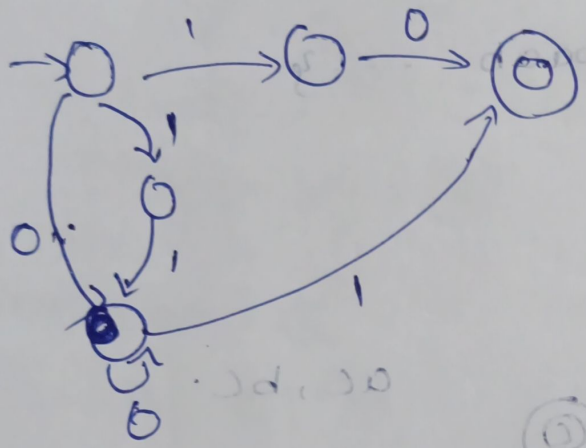
$a^* = \{\epsilon, a, aa, aaa, \ldots\}$



(5) $10 + (0+11)0^*1$







equivalence of Two FA:-

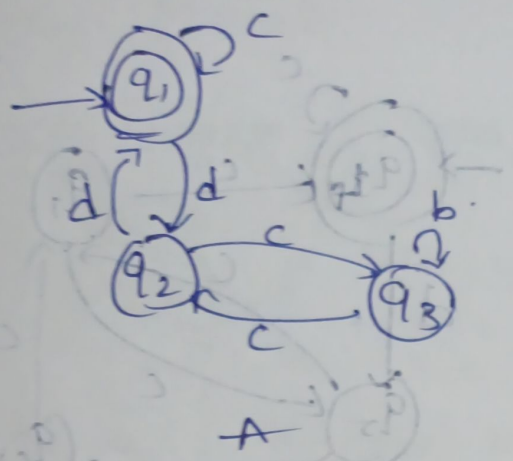
steps to identify equivalence.

1) For any pair of states $\{q_i, q_j\}$ the transition for input $a \in \Sigma$ is defined by $\{q_0, q_b\}$ wher $\delta\{q_i, a\} = q_a$ and $\{q_j, a\} = q_b$.

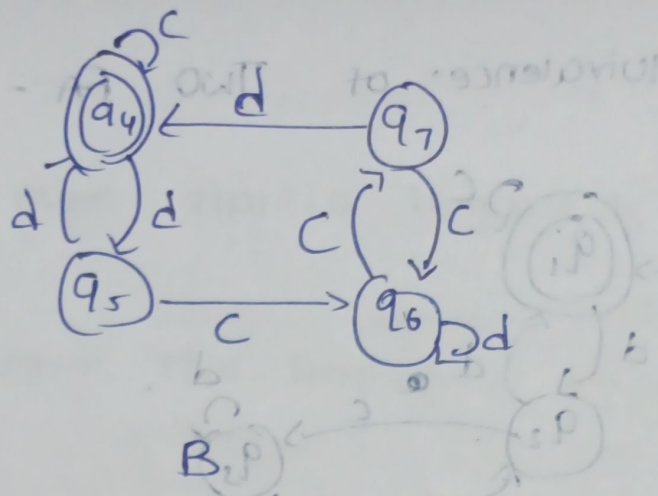
The two Automata are not equivalent if for a pair $\{q_a, q_b\}$ one is "intermediate" state and another is "final" state.

2) if Initial state of one Automation then in second Automation also initial state must be final ~~fin~~ state for them to be equivalent

eg:



A

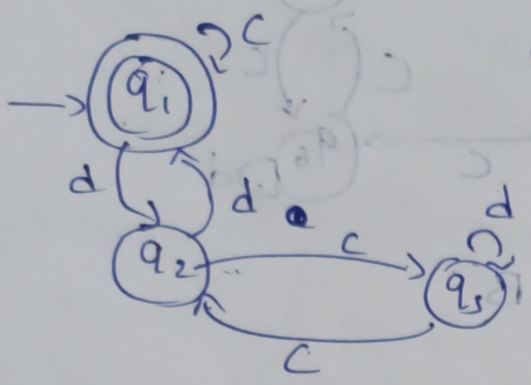


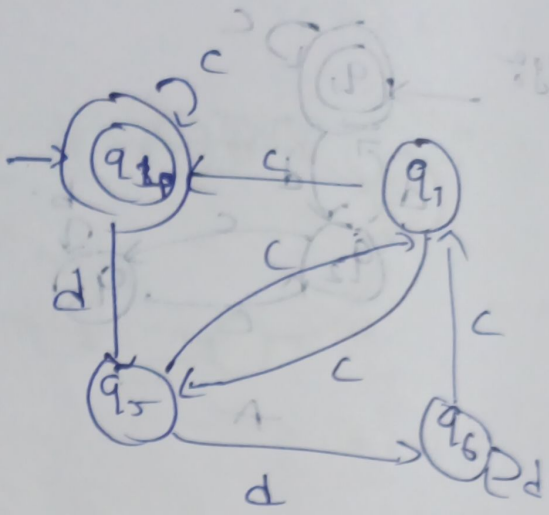
B

States	c	d
(q_1, q_4)	(q_1, q_4) fs fs	(q_2, q_5) Is Is
$(q_2\ q_5)$	(q_3, q_6) Is Is	(q_1, q_4) fs fs
(q_3, q_6)	(q_2, q_7) ← new pair Is, Is	(q_3, q_6) Is Is.
(q_2, q_7)	(q_3, q_6) Is Is	(q_1, q_4) fs fs

A & B are equivalent.

(equivalence of TWO FA - 2 :-





States	c	d
(q_1, q_4)	(q_1, q_4) Fs Fs	(q_2, q_5) Is Is.
(q_2, q_5)	(q_3, q_7) Is Is	(q_1, q_6) f Is (X)

NOT equivalent FA

Pumping lemma :-

* Pumping lemma is used to prove that a language is not regular
* It cannot be used to prove that language is regular.

If A is a Regular language, then A has a pumping length 'p' such that any string 's' where $|s| \geq p$ may be divided into 3 parts $s = xyz$ such that the following conditions must be true.

1) $xy^iz \in A$ for every $i \geq 0$
2) $|y| > 0$
3) $|xy| \leq p$

Steps to prove Pumping lemma. (Not Regex) (use contradiction)

1) Assume that A is regular.
2) It has to have a pumping length (say p).
3) All strings longer than p can be pumped $|s| \geq p$
4) Now find a string 's' in A such that $|s| \geq p$
5) Divide s into xyz
6) Show that $xy^iz \notin A$ for some i
7) Then consider all ways that s can be divided into xyz
8) Show that none of these can satisfy all 3 pumping conditions at the same time.
9) s cannot be pumped :: CONTRADICTION

Pumping lemma - 1 :-

using PL Prove that language $A = \{a^n b^n \mid n \geq 0\}$ is not Regex

Proof

1) Assume A is Regular

2) Pumping length = p

$S = a^p b^p$ $\Rightarrow S = aaaaaaaa\,bbbbbbbb$

x y z

Case-1 :- The y is in 'a' part

aa | aaaa | abbbbbbb

x y z

Case-2 :- The y is in 'b' part

aaaaaaaabb | bbbb | b

x y z

case-3 :- The y is in 'a' & 'b' Part

aaaaa | aabb | bbbbb

x y z

(1) $xy^iz \Rightarrow xy^2z$

(2) aaaaaaaaaabbbbbbbb

$|xy| \neq p$ ($|xy| \leq p$)

$xy^iz \Rightarrow xy^2z$

(3) aaaaaaaaaabbbbbbbb

$7 \neq 7$

$xy^iz \Rightarrow xy^2z \Rightarrow$ aaaaaa | aabbaabb | bbbbb

not following $a^n b^n$

Pumping lema - 2 :-

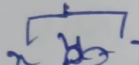
using Pumping lemma Prove that language

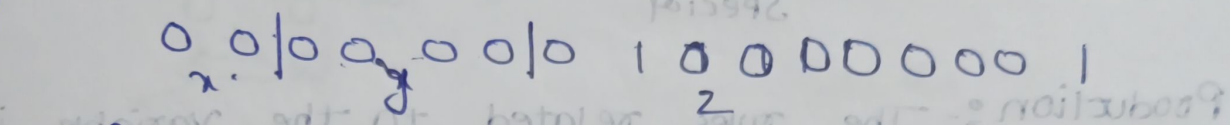
$A = \{yy \mid y \in \{0,1\}^*\}$ is not Regex.

Proof :- 1) Assume that A is Regex

2) then must have a Pumping length $\geq p$

$S = 0^p 1 0^p 1$. Let $p = 7$.





$xy^iz \Rightarrow xy^2z \Rightarrow$ 00 0000 0000 01 0000000 1 $\notin A$

$|y| > 0$

$|xy| \leq p$.

$10 \leq 7$ ✗ not Pump

A is not Regex.

CO-2

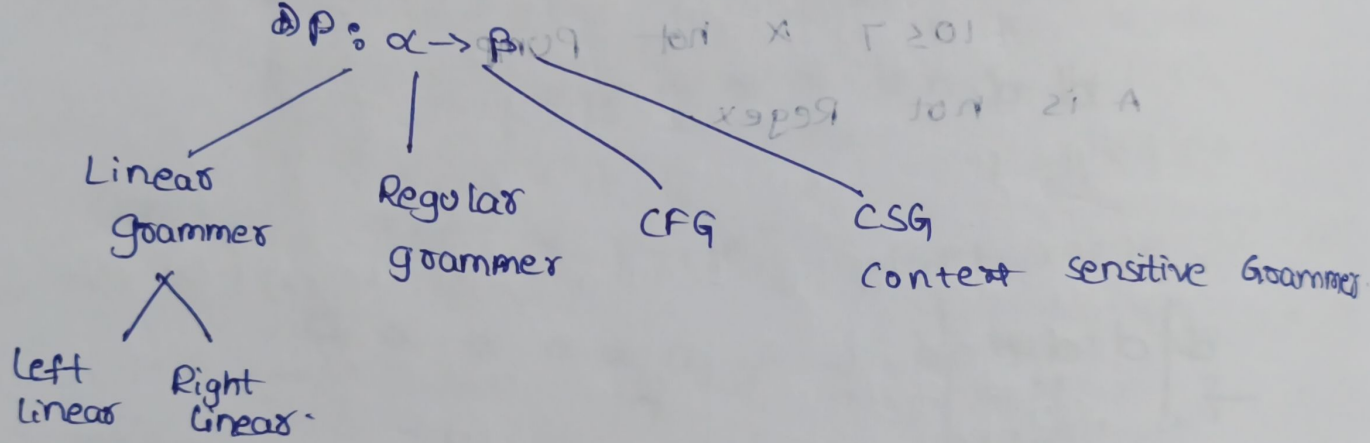
CFG:-

Context free grammer

Grammer:- A grammer is defined by $G = \langle V, T, P, S \rangle$

V-Variables (Capital) T-terminal (Small letters, 0-9, Special) P-Production $\alpha \rightarrow \beta$. S-starting Symbol

Production:- The rules related to the variable is called a production. based on this productions grammer can be classified into 4 types.



Left:- variables should always been left most

$\alpha \in V$

$\beta \in VT^* | T^*$.

Eg:- $S \rightarrow Aa | b$

Right:- variables should always been Right most

Eg: $S \rightarrow aA | a$

RG:- A grammer is said to be Regular if and only if it is either leftly engine or Rightly engine

CFG:- In this Productions are in the form

$\alpha \in V \quad \beta = (V \cup T)^*$

CSG:- In this grammer productions in the form

$\alpha \to \beta \qquad \alpha \in (V \cup T)^+ \qquad \beta \in (V \cup T)^+$

Derivations:- It is nothing but a generating a string in grammer with starting symbol S.

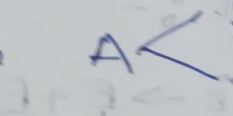
$S \overset{*}{\Rightarrow} w$

There are mainly two types 1) Left most 2) Right most

Left most :- Replacing the variable in w from left to right.

Right most :- Replacing ~~<->~~ right to left

Derivation tree/Parse tree :- A tree is a derivation tree or Parse tree for a grammer G

* starting symbol 'S' is the root node.
* the productions in the form  then where A is the parent node $x_1, x_2, \ldots x_n$ are child nodes. that are represented from Left to right
* Integer nodes of a tree is Varibles of grammer
* exterior nodes of a tree is terminals of grammer.

Find the left most derivation, Right most derivation & derivation tree for the following grammer P: {E → E+E - ①
E → E*E - ②
E → id } - ③

at string w = id+id * id+id.

A) Left most :- left to right. Derivation.

E → E+E - ①

E → id+E ③

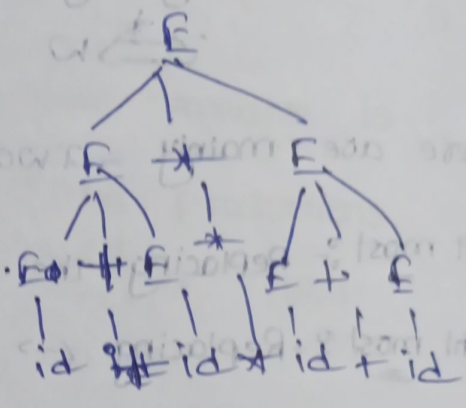
E → id+ E*E ②

E → id+ id* E

E → id+id* E+E

E → id+id * id+E

E → id+id *id+id.



Right most :-

E → E+E

E → E+id

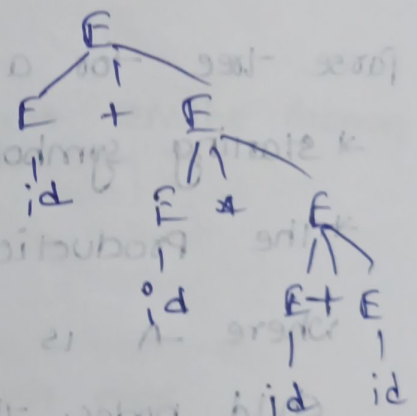
E → E*E+id

E → E*id+id

E → E+E*id+id

E → E+id*id+id

E → id+id*id+id



Find P: { S → aB | bA

A → a | aS | bAA

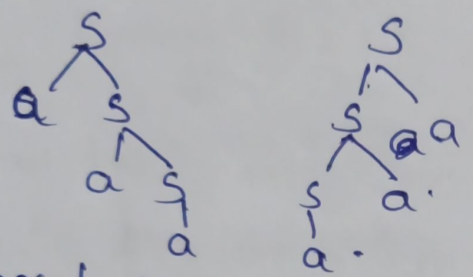
B → b | bS | aBB w = aabbabab

left most :=

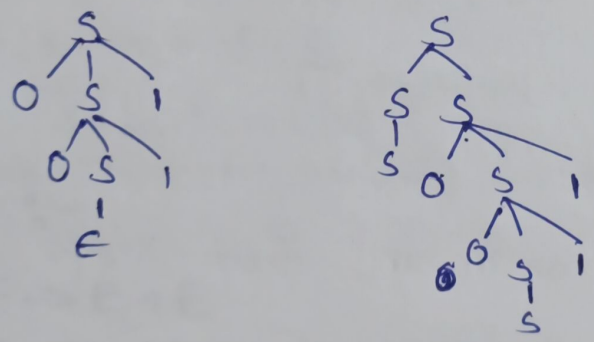
Ambiguity in CFG :- A CFG is called Ambigous atleast 1 word in the language that it generates two or more derivation trees. it means in the grammer G1 generates two or more different derivation trees for the same string is called Ambigous grammer.

show that the grammer is Ambigous

$S \rightarrow aS \mid Sa \mid a$ $\quad w = aaa$



$S \rightarrow 0S1 \mid SS \mid \epsilon$ $\quad w = 0011$



simplification of CFG:-

CFG is simplified into 3 types

1) elimination of useless symbols.
2) elimination of ε Production $(A \to \epsilon)$
3) elimination of Unit Pooduction $(A \to B)$

useless symbol:- if a variable does not deriving the terminal directly or indirectly

if a variable deriving terminal but it is not used. other productions. (Note:- Not applicable for start symbol 'S')

begin

1) Old V := ∅;
2) New V := {A | A → w for some w in T*};
3) while OLD V ≠ NEW V do

deegin

4) OLD V := NEW V;
5) NEW := OLD V ∪ {A | A → α for some α in (T ∪ OLD V)*}

end;

6) V' := NEW V

end.

eliminate usile.

eliminate useless for following grammer

{ S → AB | CA
A → a
B → BC | AB
C → bB | a }

① old V = ϕ
New = {A, C}

② old V = {A, C} ∪ {S}
= {S, A, C}

P' : { S → CA
A → a
C → a }

in above grammer B is useless symbol it doesn't derive terminal directly or indirectly

* elimination of ∈ production.

1) identify ∈ production.
2) Remove these productions after substituting in remaining productions where these variables are exist

Eg- Eliminat ∈ production for following grammer

P : { S → AB
A → aA | ∈
B → bB | ∈
}

Step 1 : A → ∈ B → ∈

Step-2 :- S → AB | B | A
S → AB | A | B

P' : { S → AB | A | B
A → aA | a
B → bB | b

3) Elimination of Unit Production

1) identify Unit productions.

2) Replace the productions of unit varibles if it not contain a unit production.

Eg- eliminate Unit Prod. for following grammer

P : {S → AB|CF
A → C|aA|a
B → bB|A|b
C → D|a
D → b
E → B|b }

Step-1 :- A → C
B → A
C → D
E → B

Step-2 A → C
A → D|a|aA|a
A → D|aA|a
A → b|aA|a
A → aA|a|b

Step-3
B → A
B → aA|a|b|bB|b
B → aA|bB|a|b

Step-4 :- C → D
C → b|a
C → a|b

Step-5 E → B
E → aA|bB|a|b

Normal forms:-

Classified into two types :- 1) CNF (Chomsky Normal Form)
2) GNF (GREIBACH NORMAL FORM)

*Any context free grammer without ϵ is generated by a grammer in which R productions of the form single terminal (or) product of two non-terminals

*In GNF grammer should be in single terminal or terminal followed by non-terminal.

$A \to a$ (or) $A \to a\alpha$

Grammer - CO_2

Type	Grammar accepted	Language Accepted	Automaton
Type-0	Unrestricted	Recursively enumerable	Turing machine
Type-1	Context ~~free~~ Sensitive	Context sensitive Language	Linear Bounded Automaton
Type-2	CFG	Context free Language	Pushdown Automata
Type-3	Regular	Regular language	F.S Automaton.

Grammer - G :-

A grammer 'G' can be formally described using tuples

$G = \{V, T, S, P\}$ where

V = "set of variables / non Terminal symbols."

T = set of terminal symbols.

S = start symbol

P = Production rules for Terminals and Non-Terminals

A production rules has form $\alpha \to \beta$ where α & β are strings on V, U, T and at least one symbol of α belongs to V

Example :- $G = (\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to a, B \to b\})$

$V = \{S, A, B\}$

$T = \{a, b\}$

$S = S$

$P = S \to AB, A \to a \quad B \to b$

Eg :- $S \to AB$
$\to aB$
$\to ab.$

Right linear Grammer

A grammer is to be right linear if all productions are of form

$A \to xB$

$A \to x$

where $A, B \in V$ and $x \in T$

Left linear Grammer

A gramer is said to be left liner if all productions are form of

$A \to Bx$

$A \to x$

where $A, B \in V$ and $x \in T$

Eg:- $S \to abS / b$ - Right Linear

$S \to Sab / b$ - Left linear

* DG-1:- The set of all strings that can be derived from a Grammer is said to be the LANGUAGE from that grammer

1) $G_1 = (\{S, A\}, \{a, b\}, S, \{S \to aAb, aA \to aaAb, A \to \epsilon\})$

$S \to aAb$

$\to aaAbb$ by $S \to aAb$

$\to aaaAbbb$ by $aA \to aaAb$

$\to aaabbbb$ by $aA \to aaAb$

$A \to \epsilon$

2) $G_2 = (\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to a, B \to b\})$

$S \to AB$

$\to ab$

$L(G_2) = \{ab\}$

Language generated by the grammer.

B) $G_3 = (\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to aA|a, B \to bB|b\})$

$S \to AB$
$\to ab$

$S \to AB$
$\to aAbB \to b$
$\to aabb$

$S \to AB$
$\to aAb$
$\to aab$

$S \to AB$
$\to abB$
$\to abb$

$L(G_3) = \{ab, a^2b^n, a^2b, ab^2 \ldots\}$

$= \{a^m b^n \mid m \geq 0 \text{ and } n \geq 0\}$

Context free languages:-

In formal language a context free language is a language generated by some context free grammer.

A set of all CFL is identical to the set of languages accepted by pushdown Automata.

This is defined by 4 tuples as $G = \{V, \Sigma, S, P\}$ where.

V = non-terminal Symbol

Σ = set of Terminal Symbol

S = start symbol

P = Production Rule.

*Context free Grammer has a production rule of the form $A \to a$

where, $a = \{V \cup \Sigma\}^*$ and $A \in V$

CFL to RL

$\{G = (S, A), (a, b), S \to \{aAb, A \to aAb|\epsilon\}\}$;

$S \to aAb$

$\Rightarrow aaAbb$ by $A \to aAb$.

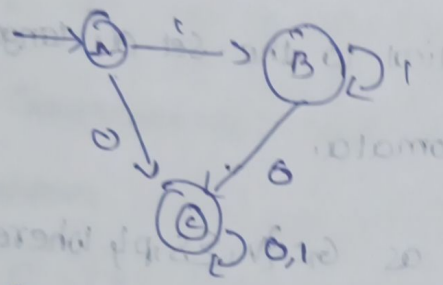
$\to aaaAbbb$ $\quad$ $A \to \epsilon$

$\Rightarrow aaabbb$

$\Rightarrow a^3b^3 \Rightarrow a^nb^n$

Regular Language and FA

* DFA can be expressed in regular language. * RL can be Context free. Complement of context-free is Context free



$C = A0 + B0 + C0 + C1 \quad -(1)$

$B = A1 + B1 \quad -(2)$

$A = \epsilon \quad -(3)$

(3) in (2)

$B = \epsilon 1 + B1$

$B = 1 + B1 \Rightarrow R = Q + RP$

$B = 1^*1 \quad R = QP^*$

(3) $\Rightarrow C = \epsilon 0 + 1^*10 + C(0+1)$

$\underset{R}{\underline{R}} = \underset{Q}{\underline{0 + 1^*10}} + \underset{R}{C}\underset{P}{\underline{(0+1)}}$

$C = (0 + 1^*10)(0+1)^* \quad R = QP^*$

method to find whether a string belong to grammer or not

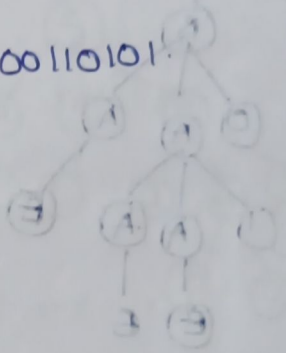
1) start with start symbol and choose closest production that matches to given string

2) Replace the variables with its most appropriate production.

Repeat the process until the string is generated until no other productions are left.

Example:- verify wheather the Grammer $S \to 0B|1A$, $A \to 0|0S|1AA|1$

$B \to 1|1S|0BB$ generates the string 00110101.



$S \to 0B$ $(B \to 0B)$

$\to 00BB$ $B \to 0BB$

$\to 001B$ $B \to 1$

$\to 0011S$ $B \to 1S$

$\to 00110B$ $S \to 0B$

$\to 00110\$S$ $B \to 1S$

$\to 001101OB$ $S \to 0B$

$\to 00110101$ $B \to 1$

Example:- Grammer $S \to aAb$, $A \to aAb|\wedge$ generates aabbb

$S \to aAb$

$\to aaAb$ $(A \to aAb)$

$\to aabb$ $(A \to \wedge)$

$aaaAbbb$ $(A \to aAb)$

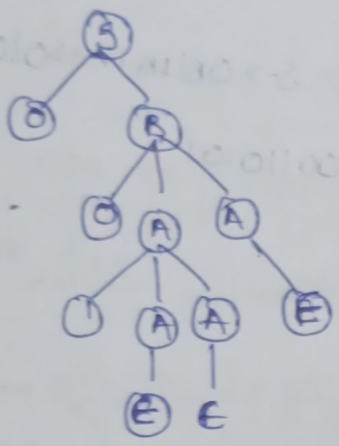
$aaabbb$ $(A \to \wedge)$

Derivation Tree:

A Derivaton Tree or parse Tree is an Ordered rooted tree that graphically represents the semantic information of strings derived from a CFG.

Example: for the Grammer $G = \{V, T, P, S\}$, whose

$S \to 0B$, $A \to 1A|1|\epsilon$, $B \to 0AA$



Root vertex :- Start symbol

Vertex: Non-terminal symbols

Leaves: Labelled by Terminal (or) ϵ

Left

Left Derivation.

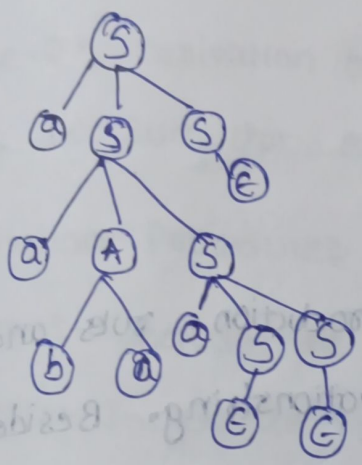
A left Derivation Tree is obtained by applying production to the left most variable in each step.

Right Derivation.

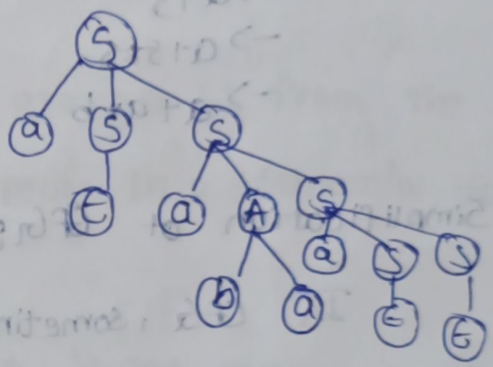
A right derivation tree is obtained by applying production to the right most variable in each step.

for generating string aabaa from gramer

S->aASlassle, A->SbA|ba.



aabaa



Ambigous Grammer:-

A Grammer is said to be ambigous if there exists two or more derivation tree for a string w

[two or more left derivation tree].

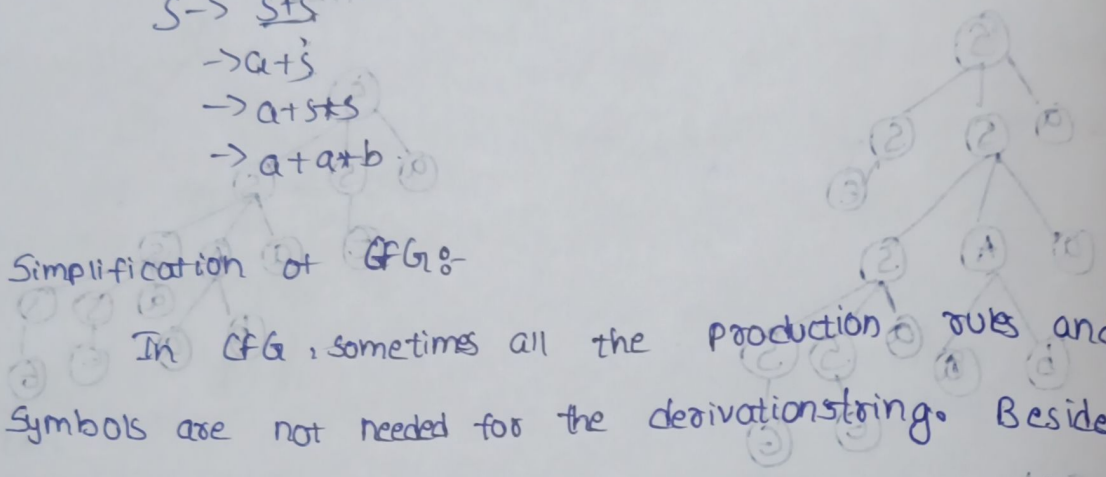
Example:- $G_1 = \{S, \{a+b, *\}, P, S\}$ where P consists of $S \to S+S \mid S*S \mid a \mid b$ then a+a*b

$S \to S+S$

$\to a+S$

$\to a+S*S$

$\to a+a*b$



Simplification of CFG:-

In CFG, sometimes all the production rules and symbols are not needed for the derivationstring. Besides this, there may also be some NULL productions and UNIT productions. Elimination of these productions and symbols is called simplification of CFG

1) Reduction of CFG
2) Removal of Unit productions
3) Removal of Null productions

1) Reduction of CFG :- Derivation of an equivalent grammer 'G' from the CFG, G, such that each variable derives

derivation procedure :-

step-1 :- Include all symbol w_1 that derives some terminal and initialize $i=1$

step-2 :- Include Symbols w_{i+1} that derives w_i

step-3 :- Increament i and repeat step 2, until $w_{i+1} = w_i$

step-4 :- Include all production rules that have w_i in it

phase-2 : Derivation of an equivalent grammer G', from the CFG G such that each symbol appears in a sentential form

Derivation Procedure :-

step-1 :- Include the start symbol in Y_1 and initalize $i=1$

step-2 :- Include all symbols Y_{i+1} that can be derived from Y_i and include all production rules that have been applied.

step-3 :- Increcment i and repeat step 2 until $Y_{i+1} = Y_i$

1 Find a reduced grammer equivalent to grammer G, having production rules

P: S → AC|B, A → a, C → c|BC, E → aAe.

Phase-1

$T = \{a, c, e\}$

$W_1 = \{A, C, E\}$

$W_2 = \{A, C, E, S\}$

$W_3 = \{A, C, E, S\}$

$G' = \{(A, C, E, S), (a, c, e), P, (S)\}$

P': S → AC, A → a, C → c, E → aAe

Phase-2

$Y_1 = \{S\}$

$Y_2 = \{S, A, C\}$

$Y_3 = \{S, A, C, a, c\}$

$Y_4 = \{S, A, C, a, c\}$

$G'' = \{(A, C, S), \{a, c\}, P, \{S\}\}$

P'': S → AC A → a C → c

Removal of CFG

Any production Rule of the form $A \to B$ where $A, B \in$ Non Terminals is called unit production

Procedure:-

step-1:- To remove $A \to B$ add production $A \to x$ to grammer rule whenever $B \to x$ occur in grammer [$x \in$ Terminal]

step-2:- Delete $A \to B$ from grammer

step-3:- Repeat from step 1 until all unit productions removed

Example:- remove unit production.

P: $S \to Xy$, $X \to a$, $Y \to Z|b$, $Z \to M$, $M \to N$, $N \to a$.

remove $Y \to Z$ $Z \to M$ $M \to N$:

1) $N \to a$ we add $M \to a$

P: $S \to Xy$, $X \to a$ $Y \to Z|b$ $Z \to M$ $M \to a$ $N \to a$.

2) $M \to a$ add $Z \to a$

P: $S \to Xy$, $X \to a$, $Y \to Z|b$, $Z \to a$; $M \to a$, $N \to a$.

3) Since $Z \to a$ add $Y \to a$

P: $S \to Xy$, $X \to a$, $Y \to a|b$, $Z \to a$, $M \to a$, $N \to a$.

remove un reachable Symbol.

P: $S \to Xy$ $X \to a$ $Y \to a|b$

1 Remove of Null Productions:-

In a CFG a non-Terminal symbol 'A' is a nullable variable if there is a production $A \to \epsilon$ or there is a derivation that start at 'A' and leads to ϵ (like $A \to \dots \to \epsilon$)

Procedural for removal:-

Step-1:- To remove $A \to \epsilon$ look for all productions whose right side contain A.

Step-2:- Replace each occurence of 'A' in each of these productions with ϵ.

Step-3:- Add the resultant productions to grammer.

Example:- $S \to ABAC$, $A \to aA \mid \epsilon$, $B \to bB \mid \epsilon$, $C \to c$.

remove $A \to \epsilon$, $B \to \epsilon$

1) to eliminate $A \to \epsilon$

$S \to ABAC$

$S \to ABC \mid BAC \mid BC$

$A \to aA$

$A \to a$.

New productions: $S \to ABAC \mid ABC \mid BAC \mid BC$ $A \to aA \mid a$, $C \to c$

2) eliminate $B \to \epsilon$

$S \to AAC \mid AC \mid \epsilon$ $B \to b$

new productions:- $S \to ABAC \mid ABC \mid BAC \mid BC \mid AAC \mid AC \mid c$

$A \to aA \mid a$

$B \to bB \mid b$

$C \to c$

Chomsky Normal form.

In CNF we have a restriction on length of RHS, which is elements in RHS should either be two variables or Terminal

A CFG is in chomsky Normal form if the productions are following forms:

$A \to a$

$A \to BC$.

where A, B, and C are non-terminals and a is a terminal.

Step-1:- If the start Symbol S occurs on some right side, create a new start Symbol S' and a new production

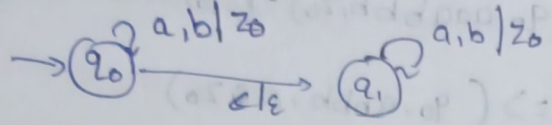
$S \to S'$

Step-2:- Remove Null & Unit Productions

Step-3:- Replace each production $A \to B_1 \dots B_n$ where $n > 2$ where $A \to B_1 C$ where $C \to B_2 \dots B_n$ repeat this step until for all productions having two or more symbols on right side.

Step-4:- If right side of any production is in the form $A \to aB$ where 'a' is terminal and A & B are non-terminals, then production is replaced by $A \to XB$ and $X \to a$

Repeat these steps for every production which is in form of $A \to aB$.



Greibach Normal form:

A CFG is in GNF if the productions are in following forms

$A \to b$

$A \to bC_1C_2 \cdots C_n$

where $A, C_1, C_2 \cdots C_n$ are non-Terminals and b is Terminal.

Steps to convert a given CFG to GNF:

Step-1:- Check if the given CFG has any unit or NULL production and remove them

Step-2:- Check whether the CFG is already in CNF and convert it

Step-3:- Change the names of non-terminals into some A_i in A0 order

Ex:- $S \to CA \mid BB$

$B \to b \mid SB$

(1,2) satisfied $C \to b$

$A \to a$

$A_1 \to A_2A_3 \mid A_4A_4$

$A_4 \to b \mid A_1A_4$

$A_2 \to b$;

$A_3 \to a$.

Replace S with A_1

C with A_2

A with A_3

B with A_4

Step-4:- After the rules so that non-Terminals are in Ascending Order such that if Productions is of form $A_i \to A_jx$ then $i < j$ and should never be $i \geq j$

$A_4 \to b | A_1 A_4$ $A_1 \to A_2 A_3$ & $A_1 \to A_4 A_4$.

$A_4 \to b | A_2 A_3 A_4 | A_4 A_4 A_4$ $A_2 \to b$

$A_4 \to b | b A_3 A_4 | A_4 A_4 A_4$.

left recursion.

step 5:- Remove left recursion.

Introduce a new variable to remove left recursion.

$A_4 \to b | b A_3 A_4 | A_4 A_4 A_4$

$Z \to A_4 A_4 Z | A_4 A_4$

$A_4 \to b | b A_3 A_4 | bZ | b A_3 A_4 Z$.

now grammer is

$A_1 \to A_2 A_3 | A_4 A_4$.

$A_4 \to b | b A_3 A_4 | bZ | b A_3 A_4 Z$

$Z \to A_4 A_4 | A_4 A_4 Z$

$A_2 \to b$

$A_3 \to a$.

GNF

$A_1 \to bA_3 | bA_4 | bA_3 A_4 A_4 | bZA_4 | bA_3 A_4 Z A_4$.

$A_4 \to b | bA_3 A_4 | bZ | bA_3 A_4 Z$.

$Z \to bA_4 | bA_3 A_4 A_4 | bZA_4 | bA_3 A_4 Z A_4 |$
$bA_4 Z | bA_3 A_4 Z A_4 | bZA_4 Z | bA_3 A Z A_4 Z$.

$A_2 \to b$

$A_3 \to a$.

Pumping lemma (for CFL)

Pumping lemma (for CFL) is used to prove that a language is NOT Context free.

If A is a context free language then A has pumping length 'p' such that any string 's' where $|s| \geq p$ may be divided into 5 pieces $s = uvxyz$ such that following conditions must be true.

(i) $uv^i xy^i z$ is in A for every $i \geq 0$

(ii) $|vy| > 0$

(iii) $|vxy| \leq p$.

* Assume A is CFL
* it has pumping length "p"
* All strings longer than p can be pumped $|s| \geq p$
* Now find a string 's' in A such that $|s| \geq p$.
* Divide s into uvxyz
* Show that $uv^i xy^i z \notin A$ for some i
* Then consider the ways that s can be divided into uvxyz.
* Show none of these ~~conditions~~ satisfy 3 pumping conditions.
* hence s cannot be pumped.

Eg: $L = \{a^N b^N c^N \mid N \geq 0\}$ is Not context free.

* Assume that L is context free of Pump length p.

$S = aPbPcP$.

divide S int uvxyz.

$P = 4$ So, $S = a^4 b^4 c^4$

Case I:- a|a a|a b b b b c|c c|c c|c
u v x y z.

v and y contain only one type of symbol

$uv^i xy^i z$ (Let $i = 2$)

a|aa aa b b b b c|c c c c c|c
u v^i x y^i z.

$a^6 b^4 c^5$ {This string doesn't belong to language}

Case-II:- either v and y has more than one kind of symbol

aa|aabb|b|b|cccc
u v x y z

$uv^i x y^i z$ $i = 2$.

aa aabb aabb b bbcccc {string doesn't belong to lang}

Case-III:- Stradles midpoint

00000 11|111 00|000 11111
u v x y z

00000 11|1111|1|0000|000 11111
u v x y z

$0^5 1^7 0^7 1^5 \notin L$.

Eg:- $L = \{ww \mid w \in (0,1)^*\}$ not CFL

$S = 0^p 1^p 0^p 1^p$

Case I :- vxy doesn't not straddle boundary.

$P=5$ so $S = 0^5 1^5 0^5 1^5$

00000 | 1111 | 0000 1111

u v x y z

$uv^i x y^i z$ $P=2 \Rightarrow uv^2xy^2z$.

000001111110000011111

$0^5 1^7 | 0^5 1^5$ { here 1st half ≠ 2nd half according to given CFL }.

Case II :- vxy straddle the first boundary

000|00 111|11 00000 11111

v x y z.

uv^2xy^2z.

000|0000|1|1111|100000 11111

$0^7 1^7 \; 0^5 1^5$ { 1st ≠ 2nd }.

Case II b :- vxy straddle the third boundary.

00000 11111 000|00 111|1

u v x y z.

00000 11111 000|0000|1|1111|1

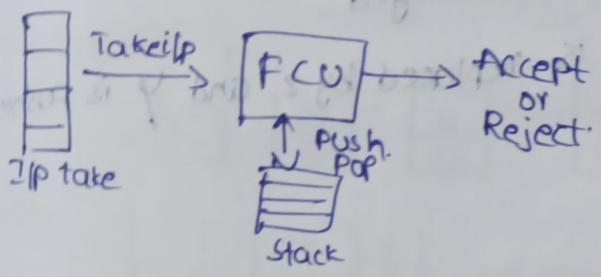
$[0^5 1^5] | 0^7 1^7$

Pushdown Automata :- PDA is a way to implement a CFG in similar way we design FA for regular grammer

* more powerful than FSM.
* FSM has a very limited memory but PDA has more memory
* PDA = Finite state Machine + A stack.

A PDA has 3 components

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



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

Q = finite set of states

Σ = finite set of ~~states~~ Ilp symbol.

Γ = A finite stack Alphabets

δ = Transition function

q_0 = start state

Z = State stack symbol

f = Final state.

δ takes as argument a triple $\delta(q, a, x)$ where.

(i) q is a state in Q

(ii) a is either a input symbol in Σ or $a = \epsilon$

(iii) x is a stack symbol, that is a member of Γ

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

p is a new state

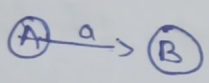
γ is a string of stack symbols that replace x at top of

Eg:- if $\gamma = \epsilon$ then Stack is Popped

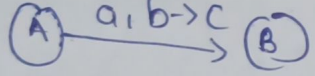
if $\gamma = x$ then Stack is unchanged

if $\gamma = yz$ then x is replaced by z and y is Pushed onto

FSM



PDA



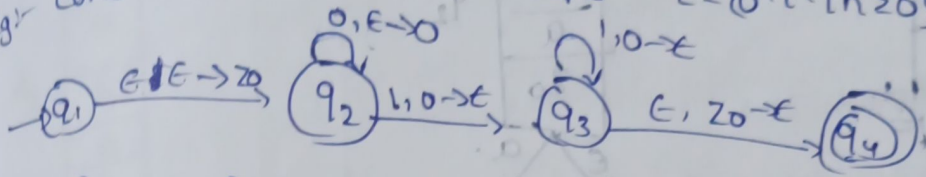
a:- ilp symbol {may be ϵ}

b:- Symbol on top of the stack. This symbol is Popped.

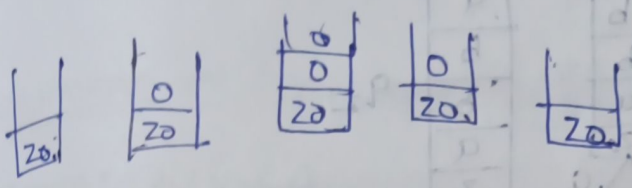
{'ϵ' means either the stack is neither read nor poped

c:- This symbol is pushed onto the stack {'ϵ' nothing is pushed}

Eg:- Construct PDA that accepts $L = \{0^n 1^n \mid n \geq 0\}$

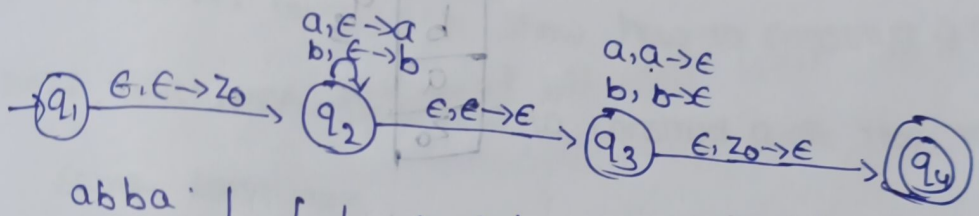


Eg:- 0011.

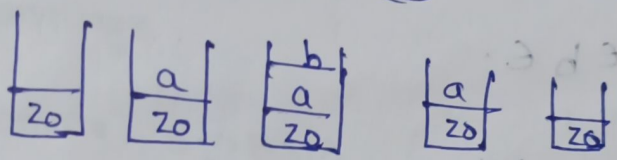


② Construct a PDA accept even palindromes of the form.

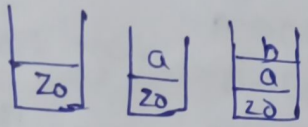
$$L = \{WW^R \mid W = (a+b)^+\}$$



abba.



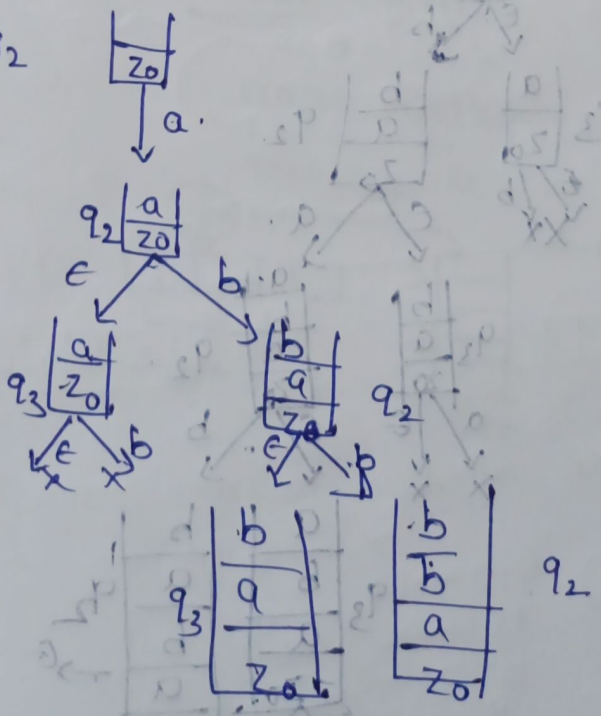
abab

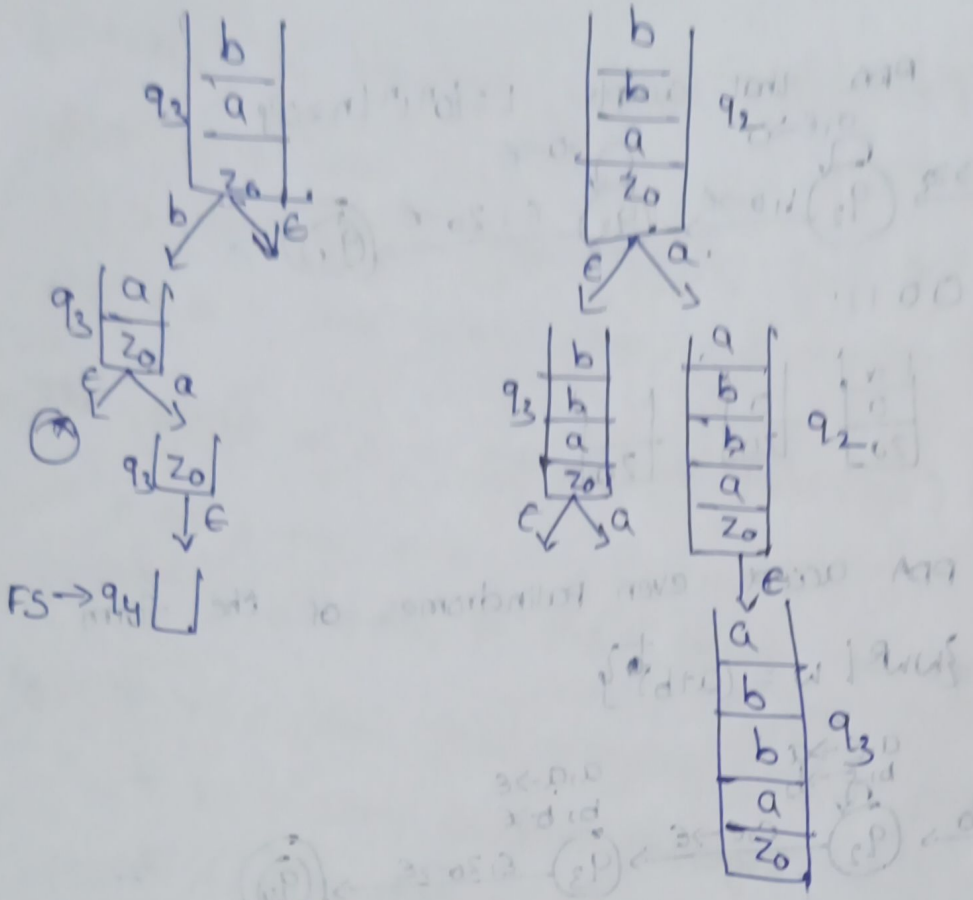


we cannot pop "a"

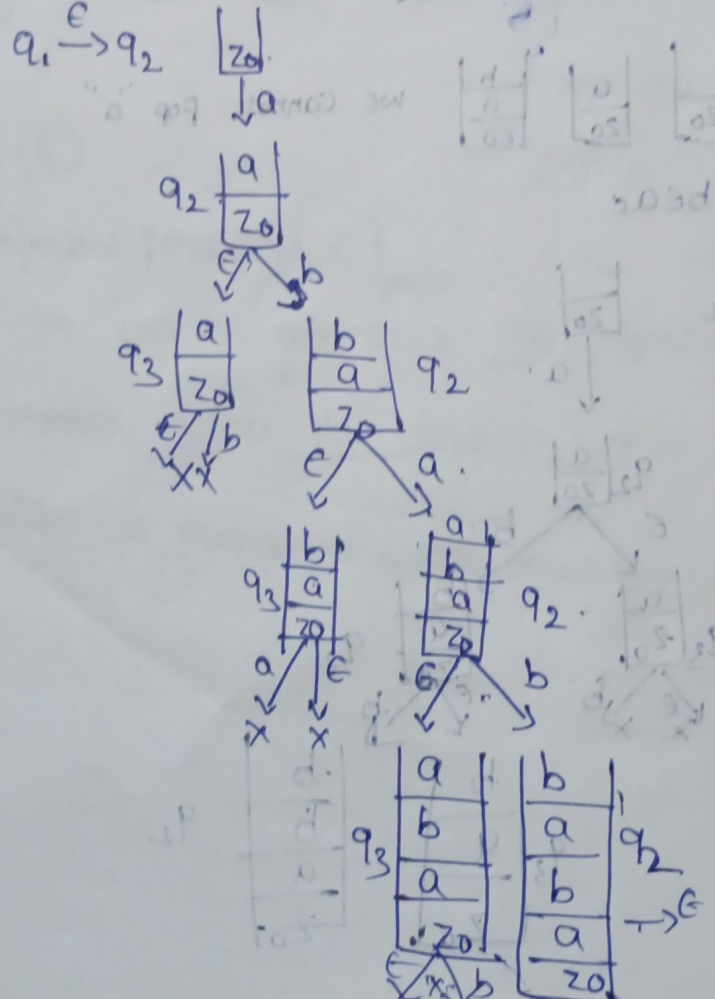
Eg:- ∈a∈b∈b∈a∈

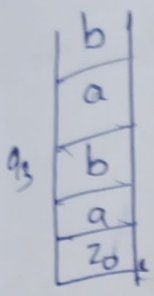
$q_1 \xrightarrow{\in} q_2$





eg:- εaεbεaεbε.





Equivalence of CFG and PDA :-

Theorm :- A language is context free iff some push down Automata recognizes it

proof :- Part1 :- Given CFG, Show how to construct a PDA recognize it

Part2 :- Given, PDA, Show how to construct a CFG that recognize. Same language.

Given a grammar S → BS|A
A → 0A|ε
B → BB|1|2. (Find or build a PDA)

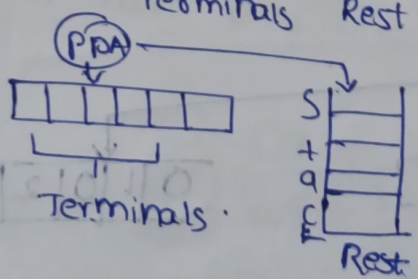
Construct left most derivation

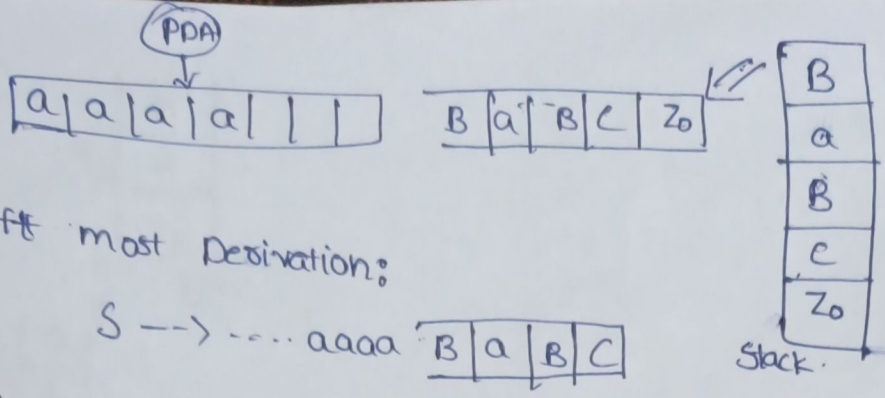
→ S
→ BS
→ BBS
→ 2BS
→ 22S
→ 22A
→ 22ε
→ 221

General form

aaaa BaBC

Terminals Rest.





Left most Derivation:

$S \rightarrow \cdots aaaa$ | B | a | B | C |

At each step expand left most derivation

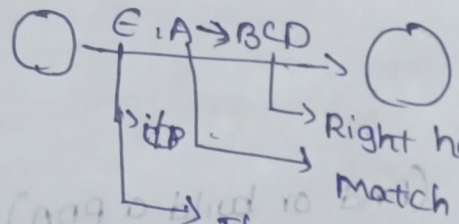
Eg:- Rule: $B \rightarrow ASA \times BA \rightarrow \cdots aaaa$ | A | S | A | x | B | A | a | B | C |

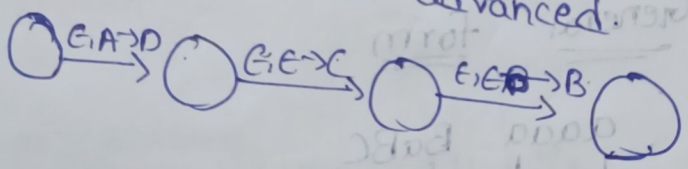
-> Match Stack Top at a Rule

-> Pop Stack

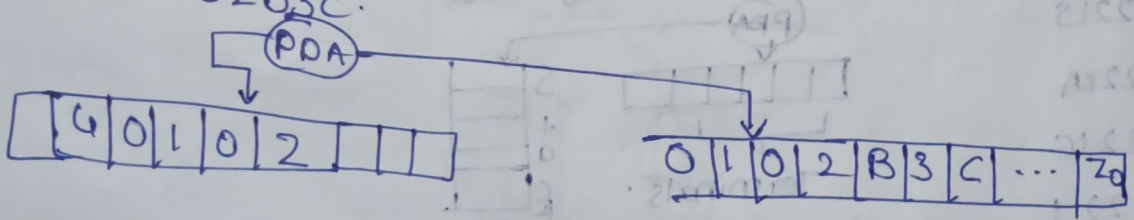
-> Push Right hand Side of Rule onto Stack.

Rule:
$A \rightarrow BCD$

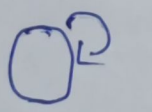




Rule: $A \rightarrow 0102B3C$.

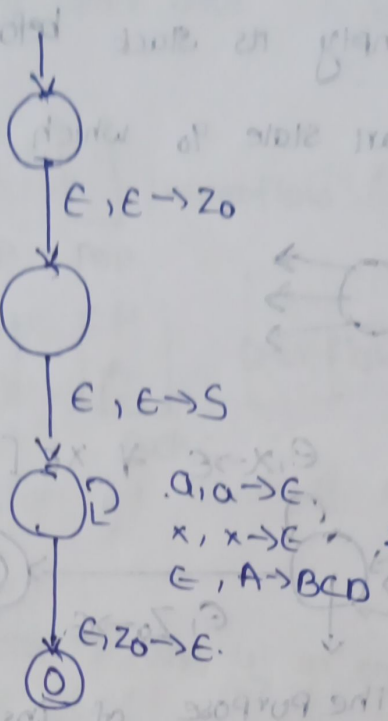


match Terminal Symbols to The stack top

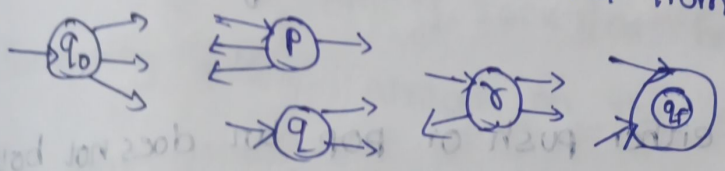


$0,0 \rightarrow \epsilon$
$1,1 \rightarrow \epsilon$
$Z, Z \rightarrow \epsilon$
$X, X \rightarrow \epsilon \quad \forall X \in Z$

The final PDA



b) Given a PDA → Build a CFG from it



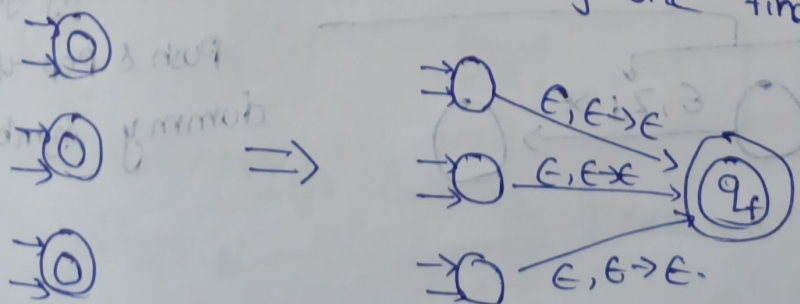
Step-1: Simplify PDA
Step-2:- Build CFG.

There will be a Non-Terminal for every pair of states:
A_{pq}, A_{q0}, A_{rq0}.

The starting Non-Terminal will be: $A_{q_0 q_f}$

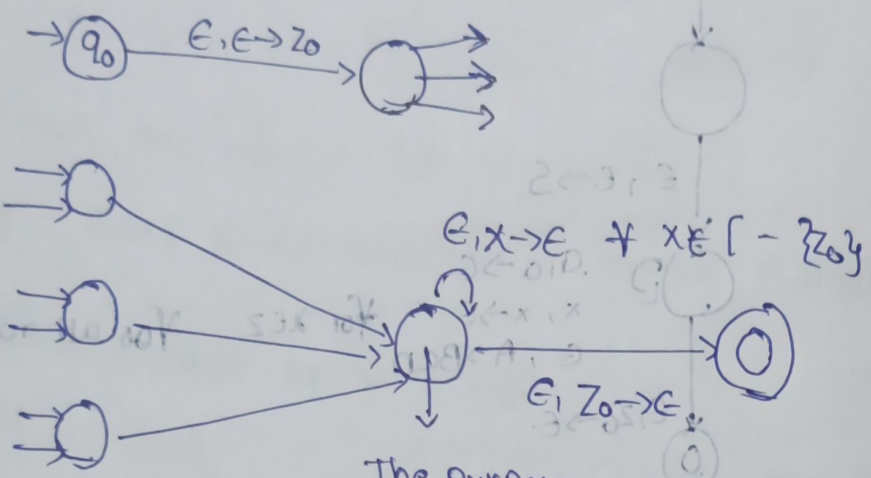
Simplification of PDA:-

1) The PDA should have only one final state.



2) The PDA should empty its stack before accepting

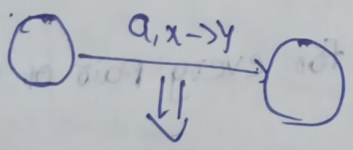
--> Creating a new start state q_0 which pushes Z_0 to stack.



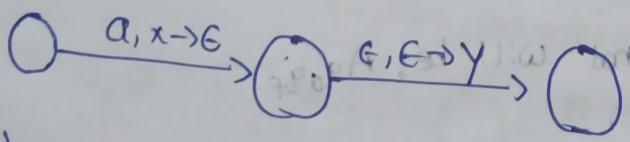
The purpose of inserting new state is to make stack empty

3) every transition do either push or pop but does not both.

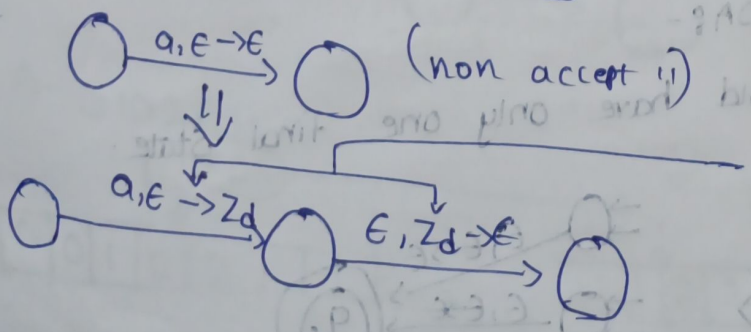
*



(not accept convert as below)



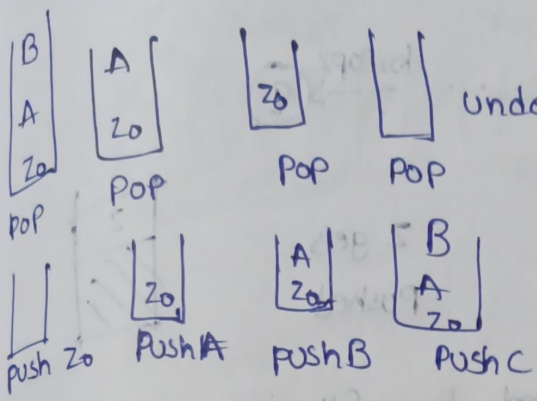
*



Z --> Dummy Symbol

Push & Pop the dummy symbol.

→ start with an empty stack and finish with an empty stack.



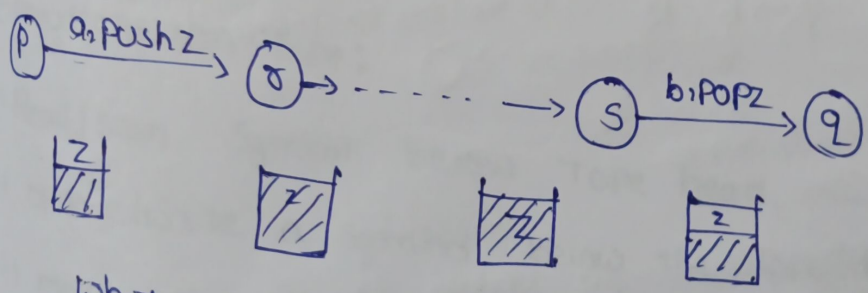
Build CFG

Consider two states p and q in the PDA

Could we go ~~from~~ to q without stack underflow and maintaining a empty stack at begining and end? OR if something where already on the stack

We will create a Non Terminal A_{pq} in our Grammer. A_{pq} will generate exactly those strings that will take us from p to q maintaining all the above stack conditions.

Case-1°-



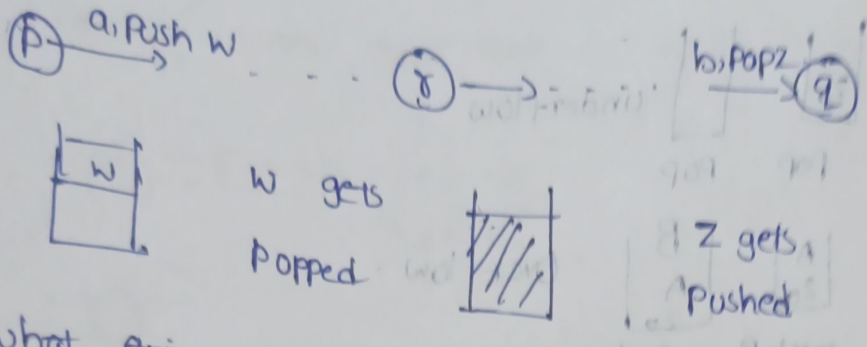
What strings can be generated by following this path.

--> 'a...b'

$A_{pq} \rightarrow a A_{rs} b$

--> This rule will generate exactly those strings

Case-2:



What strings can be generated by following this path?

$A_{pq} \rightarrow A_{pr} A_{rq}$

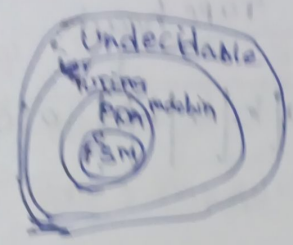
--> This rule will generate exactly those strings

-> To get from any state (p) to itself $A_{pp} \rightarrow \epsilon$

-> If the PDA accepts some strings, then there is a way to go from (q_0) to (q_f) that doesn't modify the stack

Our start Non-terminal is $A_{q_0 q_f}$.

Turing machine.



TM - Recursively enumerable language

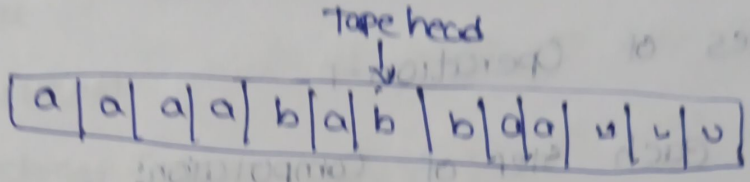
PDA - context free language

FSM - Regular language.

Data structure :-

TURING Machine :-

A Tape

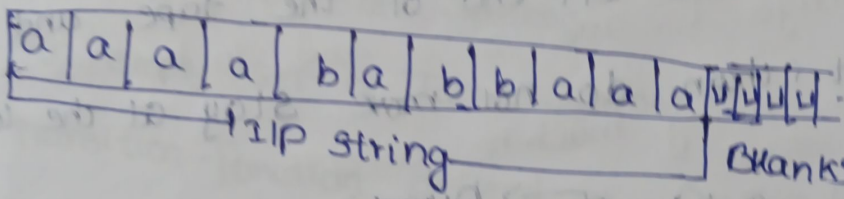


Tape Alphabets: $\Sigma = \{0, 1, a, b, x, \ldots\}$ (infinite set of i/p symbols)

u - Blank symbol is a special symbol.

The blank is a special symbol used to fill infinite Tape.

Initial Configuration :-



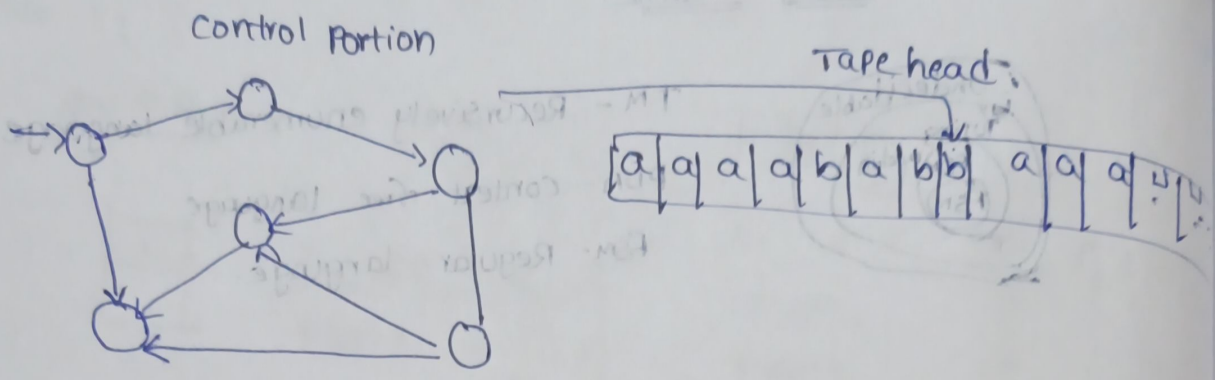
Operations on Tape :

(i) Read / scan symbol below Tape head

(ii) update / write a symbol below the Tape head.

(iii) Move the tape head one step LEFT

(iv) Move the tape head one step RIGHT



The Control portion Similar to FSM or PDA, [Deterministic]

Rules of Operation-1

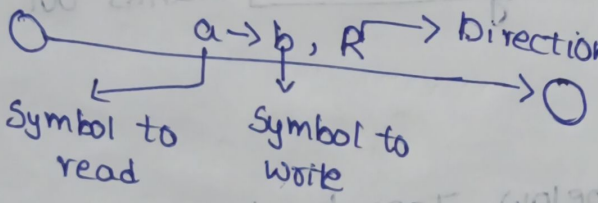
At each step of computation:

→ read current symbol

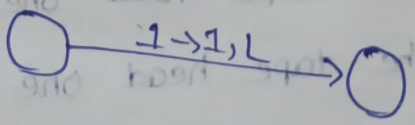
→ update the same cell

→ move exactly one cell either LEFT or RIGHT

If we are at the left end of the tape, and trying to move left, then do not move stay at the left end.



if you don't want to update the cell, just write the same symbol.



Rules of Operation-2

-> control is with a sort of FSM

->Initial state

->final state: (There are two final states)

1) Accept state

2) Reject state

->computation can either

1) HALT and accept

2) HALT and reject

3) Loop (the machine fails to HALT)

Definition:- A Turing machine is a set of 7 Tuples

$$(Q, \Sigma, \Gamma, \delta, q_0, b, F)$$

Q -> Non empty set of states

Σ -> Non empty set of symbols

Γ -> Non empty set of Tape symbols

δ -> Transition function defined as $Q \times \Sigma \to \Gamma \times (R/L) \times Q$.

q_0 -> Initial state

b -> blank symbol

f -> set of final states (Accept & Reject state)

$$\delta(q_0, a) \to (q_1, Y, R)$$

Turing Thesis:-

Turing Thesis states that any computation that can be carried out by mechanical means can be performed by some Turing machine.

few arguments for accepting this thesis are:-

(i) Anything that can be done on existing digital computers can also be done by Turing machine.

(ii) NO one has yet been able to suggest a problem solvable by what we consider an algorithm, for which Turing machine program cannot be written.

Recursively enumerable Language:-

A language L and Σ is said to be rec.EL if there exists a Turing machine accepts it.

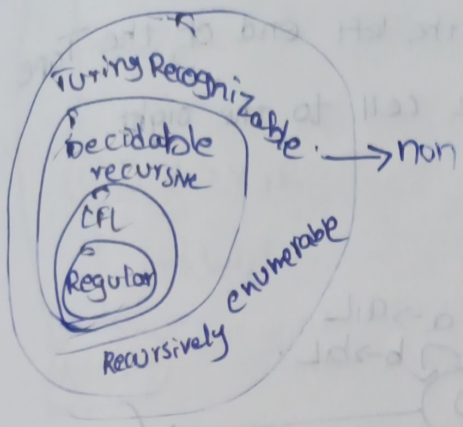
Variations of Turing machine.

* One Tape or many
* infinite on both ends
* Alphabets only {0,1} or more?
* Can the head also stay in same place?
* Allow non-determinism.

All variants are equivalent in computing capability

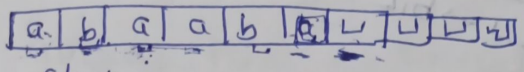
Turing machine and lambda calculus are equivalent in power

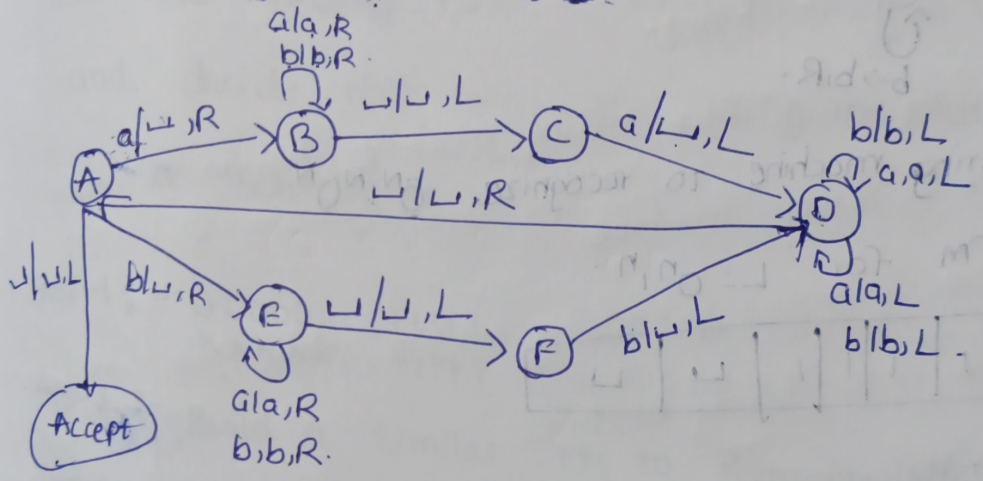
Algorithmically computable means. computable by Turing machine.



Construct a Turing machine for even palindrome.

~~s: a b~~

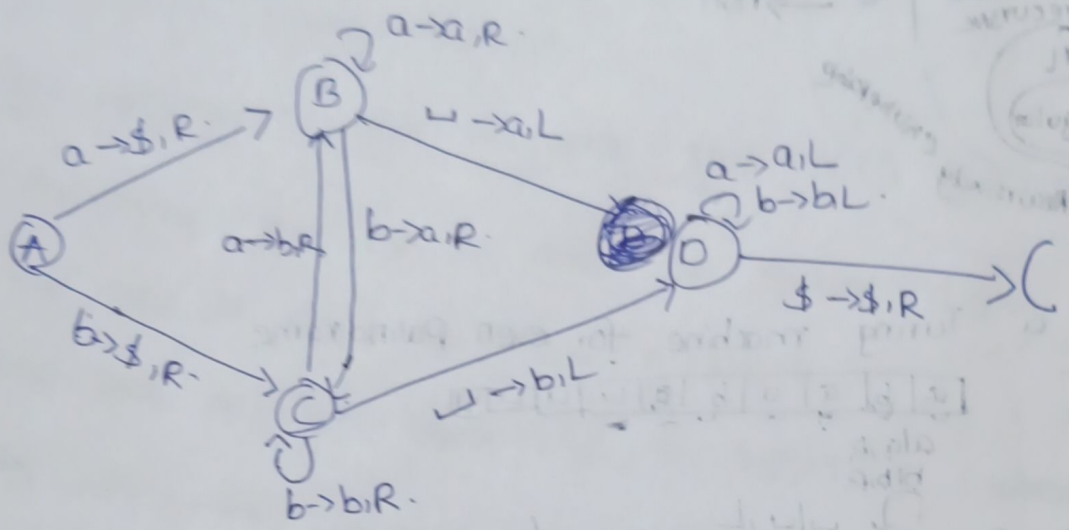




Problem :-

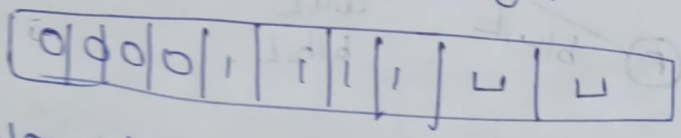
Problem:- how can we recognize the left end of the Tape of a Turing machine.

Solution :- Put a special symbol $ on the left end of the Tape and shift the input over one cell to the right.



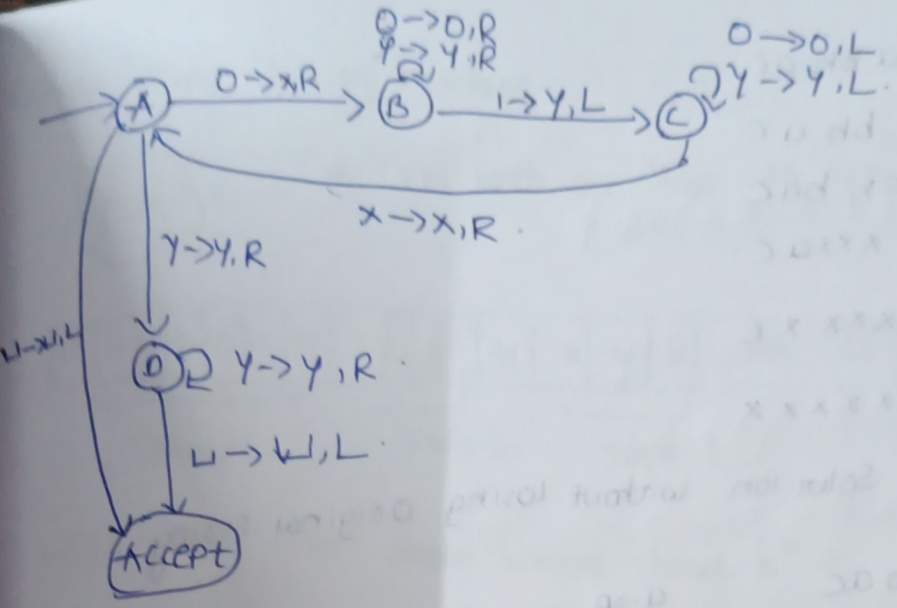
*Build a turing machine to recognize. $0^n 1^n 0^n$

Design TM for $L = 0^{n}1^{n}$.



Algorithm:-

* change "0" to "x"
* move RIGHT to FIRST "1"
  If None: REJECT
* change "1" to "y"
* move Left to leftmost "0"
* Repeat above steps until no more '0's
* make sure no more 1's remain.



$L = 0^n 1^n 0^n$

we have already have a turing machine to turn $0^n 1^n$ to $X^n Y^n$ and decide that language. Using this Turing machine as a subroutine.

Step-1 :- 0 0000 1 1111 0 00000
X X X X X Y Y Y Y Y 0 00000

Step-2 :- Build a similar TM to Recognize $Y^N 0^N$

Step-3 :- Build a final TM by combining these two smaller TM Together into one larger TM.

Comparing two strings :-

$\{wtw \mid w \in \{a,b\}^*\}$.

Sol:- Use a new symbol such as 'x'

*Replace each symbol into an x after it has been examined

a b b ac # a bb ac

x b b a c # x bb a c

x x bac # x x bac

x x x a c # x x x ac

x x x x c # x x x x c

x x x x x # x x x x x

to complete Solution without losing original string :-

a bbac # abb ac

P q q P r # P q q P r

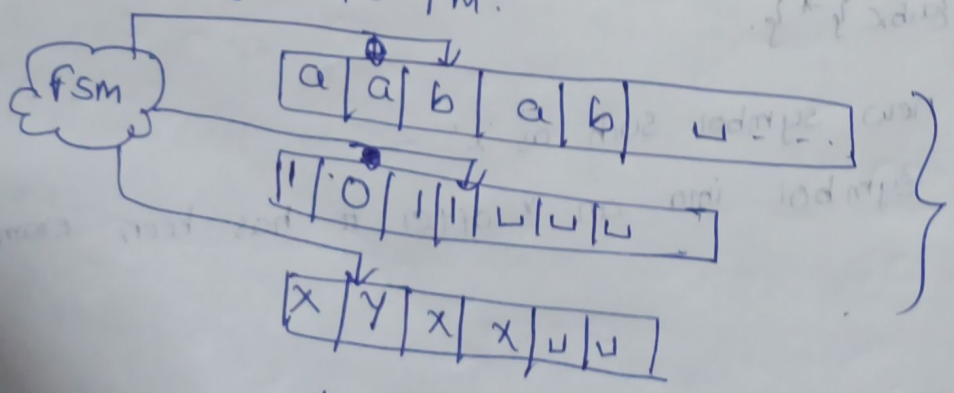
a → p
b → q
c → r

Multi-tape Turing machine:-

Theorm:- every multi-tape Turing machine has an equivalent Single tape Turing machine.

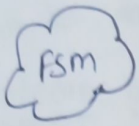
Proof :- Given a muttitape Turing machine show how to build a single tape Turing machine.

* need to store all tapes on a single tape.
* each tape has tape head
* need to transform a move in multi-tape TM into one or moves in Single tape TM.

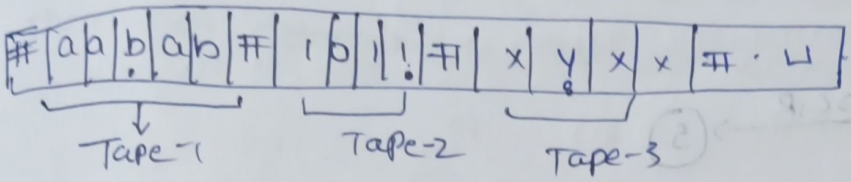


A example with k=3 TAPES

Single Tape Turing machine.



← A TM with one Tape to simulate above multitape TM



- Add "dots" to show where head "k" is
- To simulate a transition from state q, we must scan our Tape to see which symbols are under k Tape heads.
- Once we determine this and are ready to make Transition we must scan accross the tape again to update cells and move dots
- Whenever one head moves off the right end we must shift our tape so we can insert a '⊔'

Non deterministic Turing machine:-

A Turing machine can be defined as a set of 7 tuples

$$(Q, \Sigma, \Gamma, \delta, q_0, b, F)$$

$Q \rightarrow$ non empty set of states

$\Sigma \rightarrow$ non empty set of symbols

$\Gamma \rightarrow$ non empty set of Tape symbols.

$\delta \rightarrow Q \times \Sigma \rightarrow \Gamma \times (R|L) \times Q$

$q_0 \rightarrow$ Initial state

$b \rightarrow$ blank symbol

$F \rightarrow$ set of final states (Accept (or) Reject)

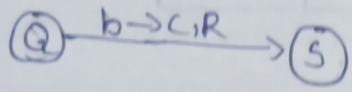
The production rule of Turing machine will be written as

$$\delta(q_0, a) \rightarrow (q_1, Y, R)$$

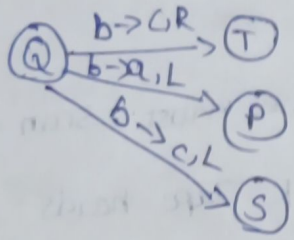
Transition function:-

$\delta : Q \times \Sigma \rightarrow P\{\Gamma \times (R/L) \times Q\}$

Deterministic :

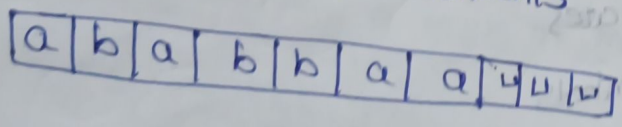


non-Deterministic :-



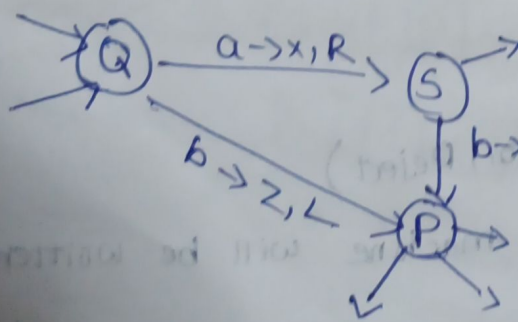
Configuration :-

- A way to represent the entire state of TM at a moment during computation.
- A string which captures:
  - * Current state
  - * Current position of the head.
  - * The entire TAPE Contents



a b a q b b a

Deterministic TM:-



computation history

C C Q a b b C

⇓

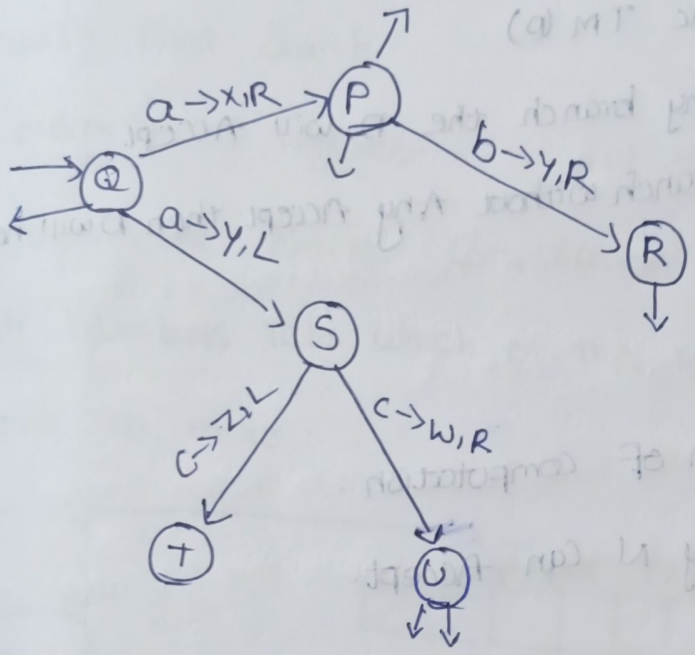
C C x S b b C

⇓

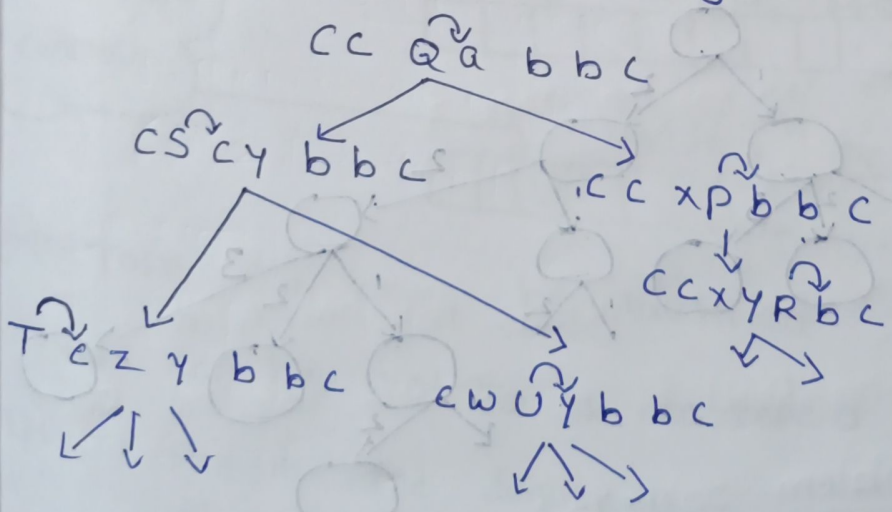
C C x y P b C

With non-determinism:- At each moment of computation there can be more than one successor configuration.

Outcome is Accept:- If any branch of computation accept, then non deterministic TM will Accept.



Computation history.



Reject:- If all branches of the computation HALT and Reject. (i.e, no branches accept, but all computations HALT) then the non-deterministic TM Rejects.

Theorm:- Every non-deterministic TM has equivalent Determinis

Proof:-

-Given a Non-deterministic TM (N) show how to construct an equivalent Deterministic TM (D)

-If N accepts on any branch the D will Accept

-if N halts on every branch without Any Accept then D will halt [Rej

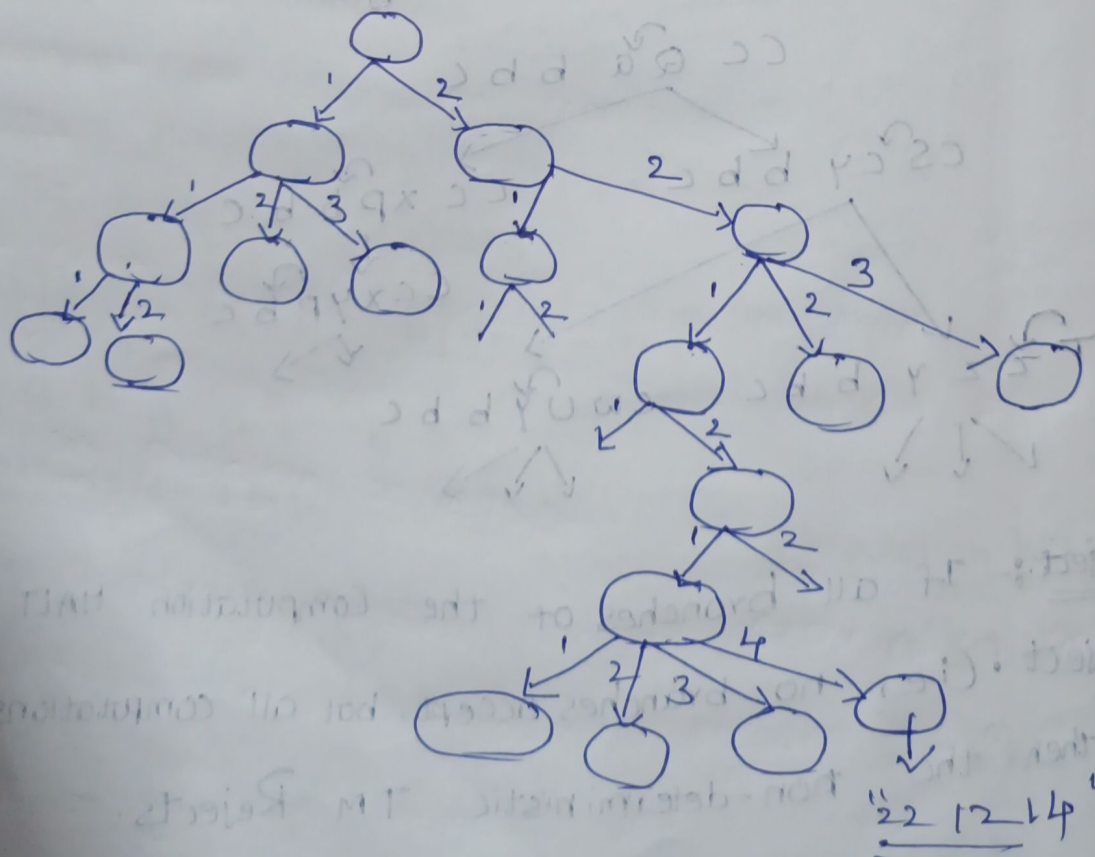
Approach:-

-Simulate N

-Simulate all branch of computation.

-Search for any way N can Accept.

Computational history:-



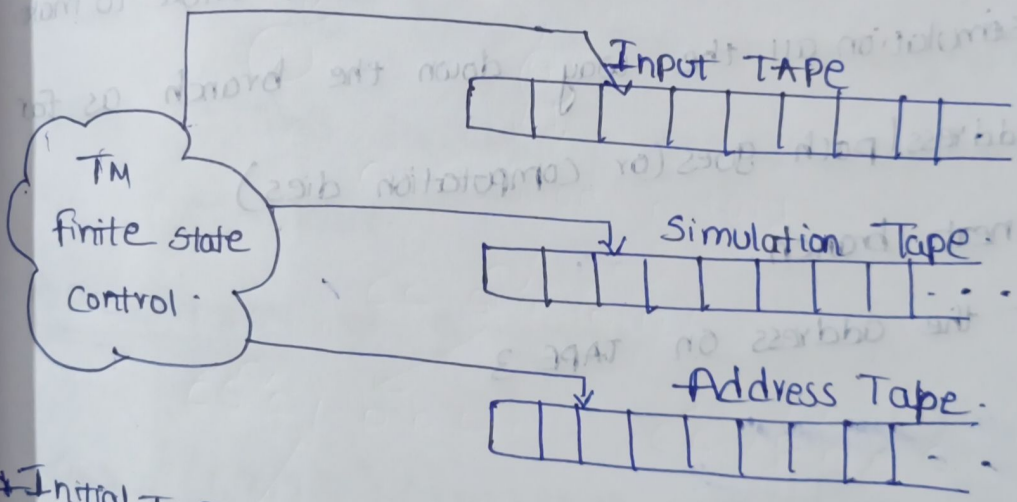
- A path to any Node is given by a number
- Search the tree looking for accept

Search Orders:-

- Depth First Search ✗ [Some paths may infinite, so this is not fi
- Breadth first Search ✓

To examine a node:-

* perform the entire computation from scratch.
* path numbers Tells which of the many non deterministic choices to make.



* Initial Tape can never be modified.
* Sim. Tape can be used like the to tape of Deterministic TM to perform the Simulation
* Address Tape used to control the BFS; Tells which choices to mate during a simulation.

Algorithm:-

Initially: Tape 1 Contains Input

Tape 2 and Tape 3 are empty.

- Coppy Tape 1 to Tape 2
- Run the Simulation
- Use Tape 2 as "The Tape"
- When choices occur (i.e. when non deterministic branch Points are encountered) Consult TAPE 3
- TAPE 3 Contains a path. each number tells which choice to make
- Run the simulation all the way down the branch as far as the address/path goes (or computation dies)
- Try the next branch.
- Increment the address on TAPE 3
- Repeat.

If accept is ever encountered, Halt and Accept

If all branches Reject or die out then Halt and Reject

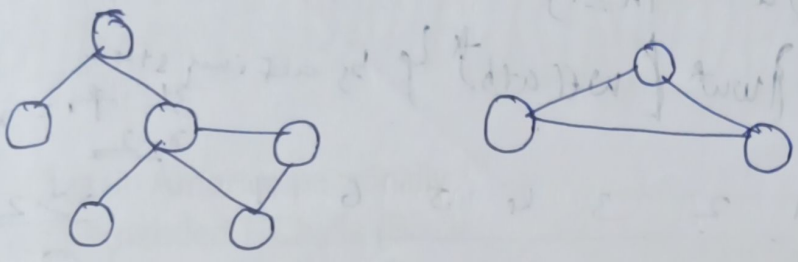
T.M as Problem Solvers:-

* Any arbitrary problem can be expressed as language
* Any instance of the problem is encoded into a string

The string is in language => Yes

The string is not in language => NO.

example:- Is this graph [undirected] connected?



step-1:- we must encode the problem into language.

$A = \{ \langle G \rangle \mid G$ is a connected graph$\}$

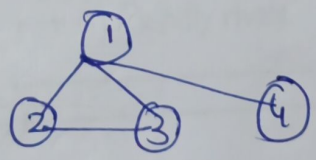
we would like to find a TM to decide this language

ACCEPT = "yes", (connected graph)

Reject = "NO", (This is not connected graph")

LOOP = This problem is decidable, our TM will always halt.

Representation of a graph:-



$\langle G \rangle = (1,2,3,4)\ ((1,2),(2,3),(1,3)$
node $(1,4))$
edge.

$\Sigma = \{ (,), ,, 1, 2, 3, 4, \ldots \ldots 0 \}$

(	1	,	2	,	3	,	4	.....	)	)	⊔