

**Department of Computer Science and Engineering
S R Engineering College**

II B.Tech II Sem

Lecture Notes - Theory of Computation

(2018-19)

PROPERTIES OF BINARY OPERATIONS:-

Postulate 1 :

Closure :

$$a, b \in A, a * b \in A, \forall a, \forall b.$$

Postulate 2 :

Associative :

$$a, b, c \in A, a * (b * c) = (a * b) * c, \\ \forall a, \forall b, \forall c.$$

Postulate 3 :

Identity element :

$$a * e = e * a = a, \forall a \in A, \text{ where } e \text{ is} \\ \text{identity element}$$

Postulate 4 :

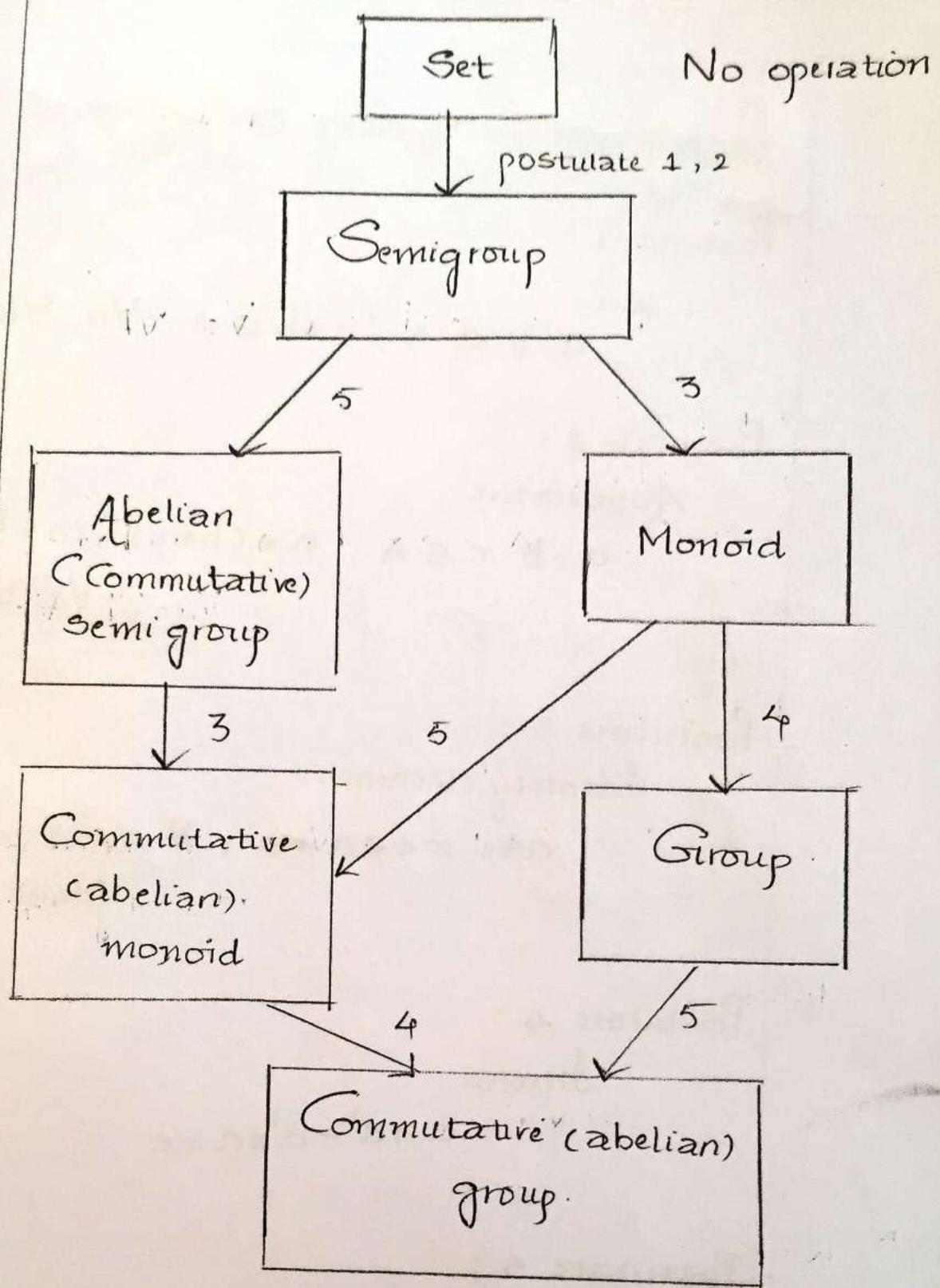
Inverse :

$$a * a' = a' * a = e$$

Postulate 5 :

Commutative :

$$a * b = b * a \quad \forall a, \forall b \in A.$$



Let $S = \{1, 2, 3, 4, \dots\}$ and binary operation
is subtraction i.e., $\langle S, - \rangle$

Postulate 1:

Closure

$$1 - 2 = -1 \notin S.$$

The given set does not satisfy P1.
so, 'S' does not have binary operation i.e.,
Subtraction

The powerset 2^A of $A (A \neq \emptyset)$ with union is a
Commutative monoid

Let,

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

$$2^S = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$$

operation is $\langle 2^S, \cup \rangle$

P1: closure.

It satisfies P1.

P2: Associative.

It satisfies P2

P3: Identity

$$\{\} \cup \emptyset = \{\}$$



Identity element

P4: Inverse

$$A \cup A' = \emptyset$$

It does not satisfy P4

P5: Commutative

It satisfies Commutative

So, it is called Commutative monoid.

$\langle \mathbb{Z} \times \rangle$

It satisfies P1, P2, P3, & P5 so, it is
Commutative monoid.

$\langle R \times \rangle$

It satisfies P1, P2, P3, P4 & P5 so, it is
called as Commutative group.

SET WITH TWO BINARY OPERATIONS:

let us consider '2' binary operations $*$ &
here,

P1 - P5 $\rightarrow *$ and

P6, P7, P8, P10 $\rightarrow o$

P9: P4

Inverse

$$a * a' = a' * a = e$$

$$a * a' = a' * a = e'$$

P11:

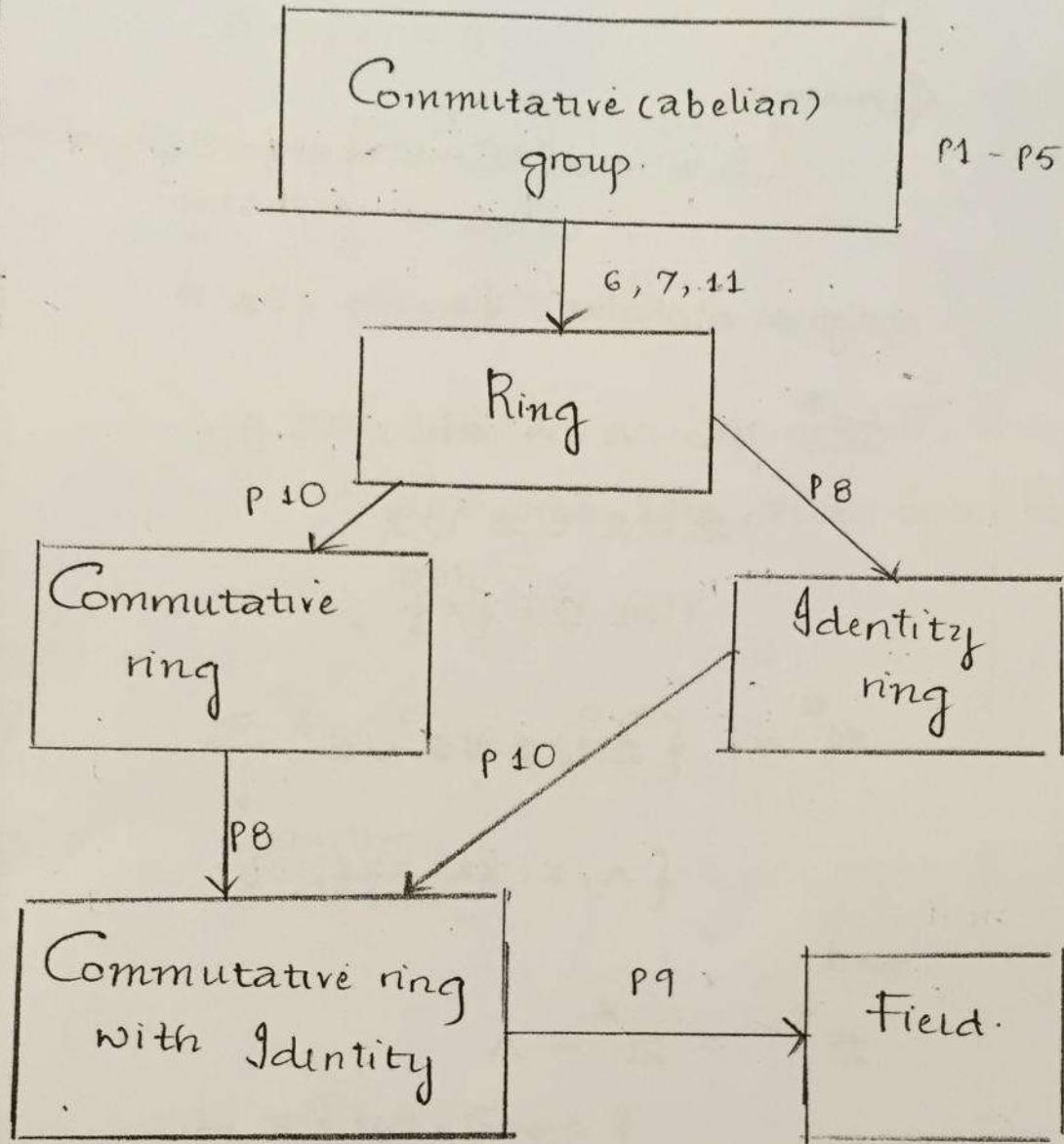
Distributive

$$a * (b * c) = (a * b) * (a * c)$$

$\forall a, \forall b, \forall c, \in A$

for eg, $\langle \mathbb{Z}, +, * \rangle$

(u)



$\langle \mathbb{Z}, +, * \rangle$

P 11 \rightarrow eg,

$$2 * (3 + 5) = (2 * 3) + (2 * 5)$$

$$2 * 8 = 6 + 10$$

$$16 = 16$$

It satisfies

so, It is Commutative ring with Identity.

ALPHABET:

Collection of Similar objects Symbols
is called as Alphabet.

english alphabet = {a, ..., z, A, ..., Z}

$$\Sigma^* = \{a, a, \wedge, abc, acz, \dots\} \\ = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3$$

Here $\Sigma = \{x\}$

$$\Sigma^* = \{\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \dots\} \\ = \{\wedge, x, xx, xxx, \dots\}$$

and

$$\Sigma^+ = \Sigma^* - \wedge$$

CONCATENATION :

let x, y be 2 variables and

$$z = xy$$

$$x = 001$$

$$y = 10$$

then

$$\begin{cases} xy = 00110 \\ yx = 10001 \end{cases}$$

Concatenation

for eg,

(1) $\Sigma = \{a, b\}$

$$\Sigma^* = \{\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots \dots \}$$

Kleen's closure

$$= \{a, b, \lambda, ab, ba, aa, bb, aaa, aab, aab, aba, baa, abb, bab, bba, bbb, \dots \dots \}$$

$$\Sigma^+ = \Sigma^* - \{\lambda\}$$

positive closure

(2)

$$L_1 = \{\text{red}, \text{Green}\}$$

$$L_2 = \{\text{pen}, \text{Ink}\}$$

$$L_1 \cup L_2 = (L_1 + L_2)$$

$$= \{\text{red}, \text{Green}, \text{pen}, \text{Ink}\}$$

$$L_1 L_2 = \{ \text{red pen}, \text{red Ink}, \text{Green pen}, \text{Green Ink} \}$$

↓
concatenation

CONCATENATION PROPERTIES :

P1 : Associative

$x, y, z \in \Sigma^*$

$$x(yz) = (xy)z$$

let $x = ab, y = pq, z = cd$

$$ab(pqcd) = (abpq)cd$$

$$abpqcd = abpqcd$$

P2 :

For concatenation the Identity element is 'λ'

$\forall, x \in \Sigma^*$

$$\lambda x = x \lambda = x$$

P3 : left Concatenation and right Cancellation

LC :

$$ap = aq$$

by left cancellation,

$$P = q$$

$$a = hai$$

$$P = xx xx$$

$$q = yyyy$$

$$xxxx = yyyy$$

RC :

$$pa = qa$$

$$P = Q$$

P4 : length.

$\forall x, \forall y \text{ in } \Sigma^*$

$$|xy| = |x| + |y|$$

where $|xy|, |x|, |y|$ are the lengths of the strings xy, x & y respectively.

P5 : Transpose

$$\begin{aligned} (xa)^T &= a(x)^T \\ &= y(abc)^T \\ &= yc(ab)^T \\ &= ycb(a)^T \\ &= ycba \end{aligned}$$

→ Even length palindrome is obtained by the concatenation and transpose of a strings

LEVIS THEOREM :

Here $ab = xy \text{ in } \Sigma^*$

case (i) :

$a = xz$ and $y = zb$ if

$$|a| > |x|$$

Case (ii) :

$$a = x \text{ and } b = y \text{ if}$$

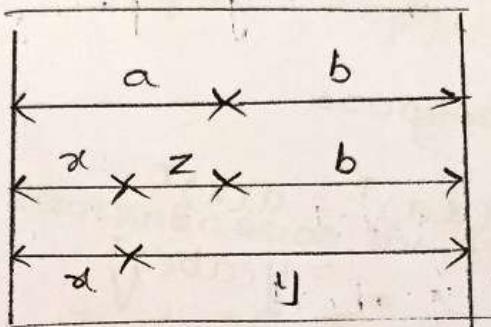
$$|a| = |x|$$

case (iii) :

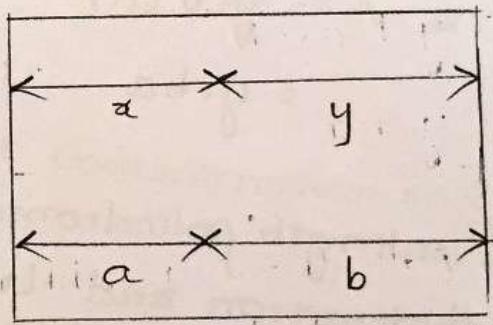
$$x = az \text{ and } b = zy \text{ if}$$

$$|a| < |z|$$

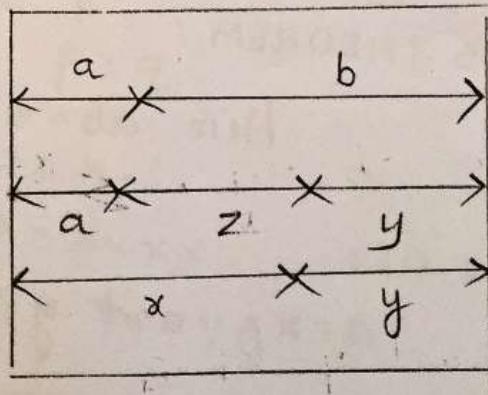
case (i) :



case (ii) :



case (iii) :



PREFIX :

prefix is the leading symbols of a string
for eg,

prefix of abc is
 λ, a, ab, abc

SUFFIX :

Suffix is the trailing of a substring of a string

for eg,

Suffix of abc is
 λ, c, bc, abc .

TERMINAL :

It is symbol or string which cannot be split is called terminal.

It is used to generate the strings

eg, a, b, c, d.

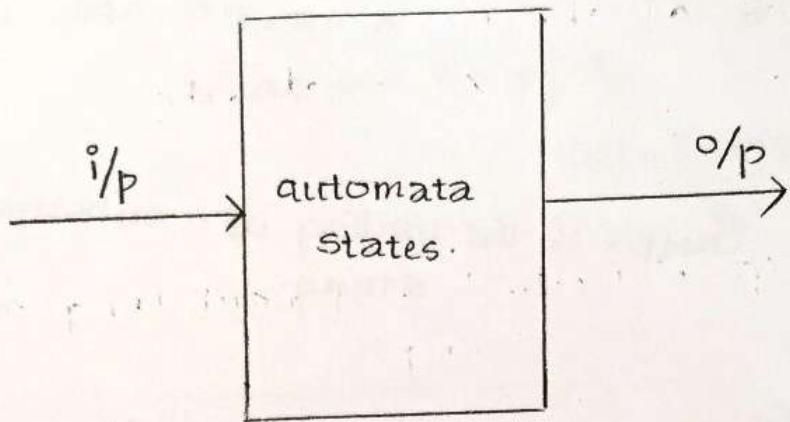
NON - TERMINAL :

It is a symbol or string which cannot be split is non-terminal

It is used to generate the strings

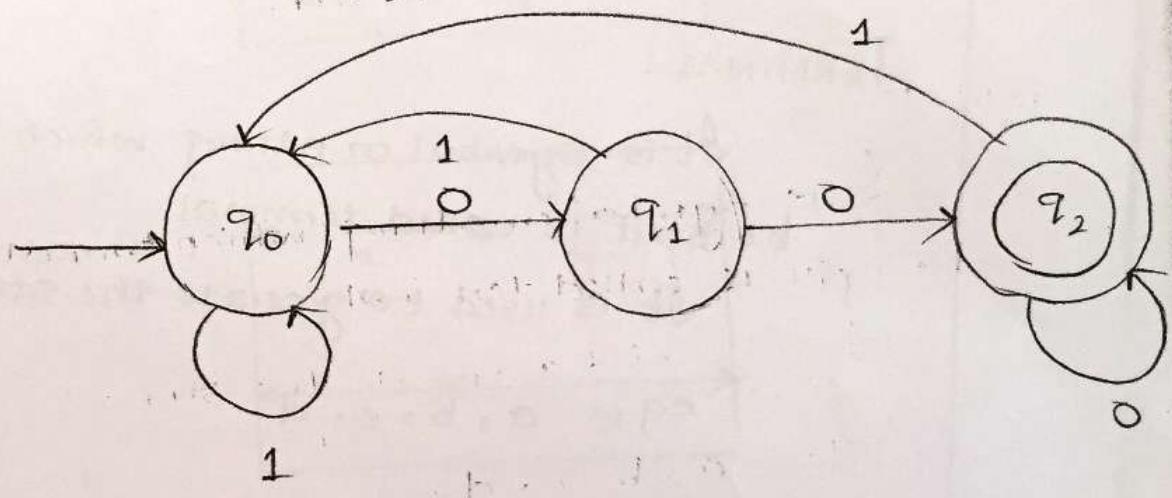
eg, abc, ab, xyz

FINITE AUTOMATA :



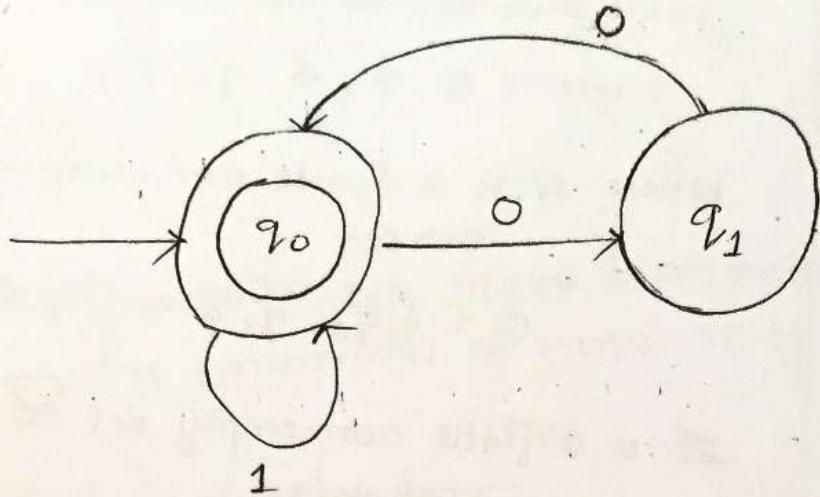
Examples:

(1) ending with 00



- Input is represented by Σ
- In a finite automata the input is finite
- The states in the above diagram are q_0, q_1, q_2, \dots
- Output relation depends on input and state
- final state 'F' is the set of all finite states

(2) even number of zeros



$\rightarrow 0000$

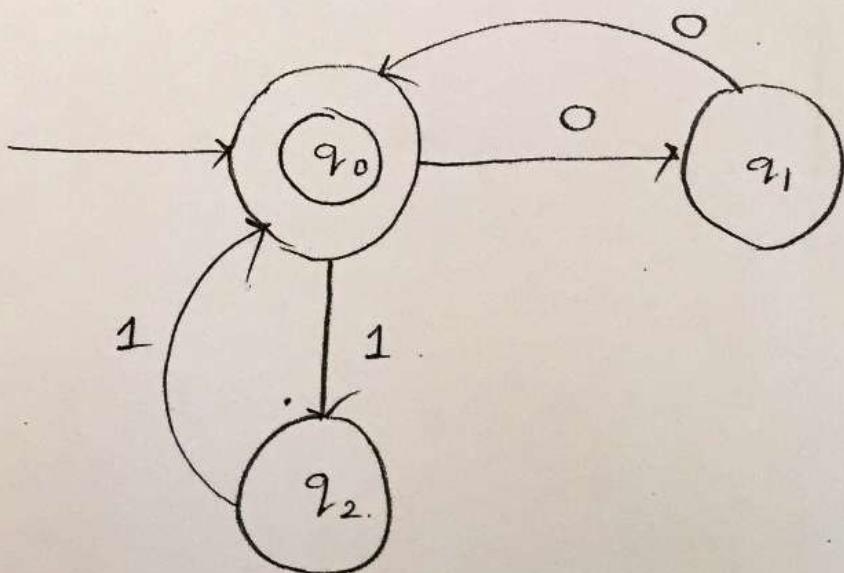
$$(q_0, 0000) \rightarrow (q_1, 000) \rightarrow (q_0, 00) \rightarrow (q_1, 0) \rightarrow q_0$$

$\rightarrow 001$

$$(q_0, 001) \rightarrow (q_1, 01) \rightarrow (q_0, 1) \rightarrow q_0$$

(3) Even no. of 1's & 0's.

$\rightarrow \wedge, 00, 11$



finite automachine can be represented by 5-tuple $(Q, \Sigma, \delta, q_0, F)$

where Q is a finite non-empty set of states

$$Q = \{q_0, q_1\}$$

Σ is a finite non-empty set of inputs called input alphabet

δ is a function which maps $Q \times \Sigma$ into Q and is usually called

TRANSITION SYSTEM

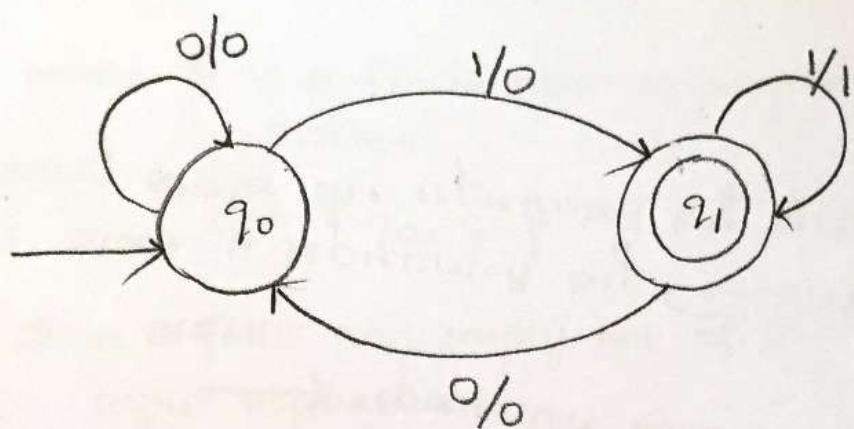
Transition system is finite labelled directed graph.

In the above graph the initial state can be represented by

→ represented towards to initial state

final state is represented by concentric circles.
remaining states are represented by circles.

eg,



On the q_0 state input 1 is applied the next state is q_1 , and the output is 0

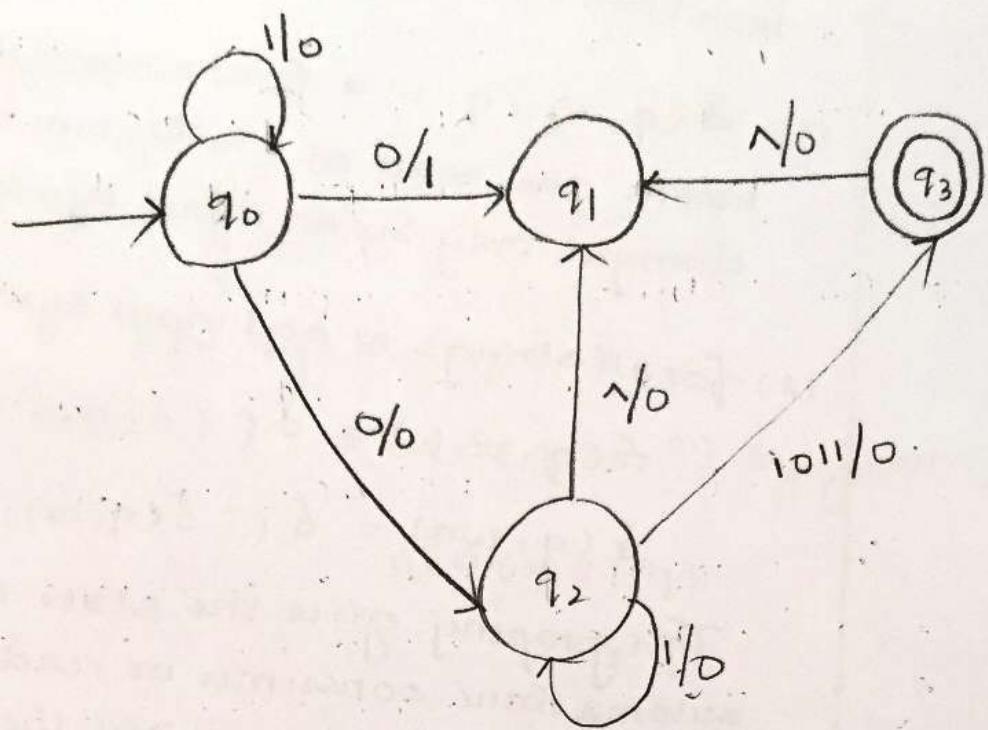
→ A transition system is a 5-tuple $(Q, \Sigma, \delta, Q_0, F)$

Where Q_0 is a set of all the initial states

→ A transition system accepts a string w in Σ^* if

- (i) There exist a path which originates from some initial state, goes along the arrows and terminates at some final state and
- (ii) The path value obtained by concatenation of all edge labels of the path is equal to w

Describe the transition system



This transition system has 2 initial states

$$\text{i.e., } Q_0 = \{q_0, q_1\}$$

It has 4 states i.e., $Q = \{q_0, q_1, q_2, q_3\}$

It has 1 final state i.e., $F = \{q_3\}$

Find the acceptability of 101011 and 111010

for 101011

The path is $q_0 - q_2 - q_3$

1 0 1011

by concatenation

101011

so, it is accepted

111010 is not accepted by the transition system.

PROPERTIES OF TRANSITION FUNCTION:

(1) $\delta(q, \lambda) = q$ in a finite automachine this means the state of the system can be changed only by an input symbol

(2) - for all strings w and input symbols A a

$$\delta(q, a, w) = \delta((\delta(q, a)), w)$$

$$\delta(q, wa) = \delta((\delta(q, w)), a)$$

This property gives the state after the automachine consumes or reads the first symbol of a string aw and the state after the automachine consumes a prefix of the string wa

ACCEPTABILITY OF A STRING BY FINITE AUTO MACHINE:

A string α is accepted by a finite automach

$$M = (Q, \Sigma, \delta, q_0, F) \text{ if}$$

$\delta(q_0, \alpha) = q$ if $q \in F$ this is basically acceptability of a string by the final state

NOTE :

A final state is also called as an accepting state

Prove $\delta(q, xy) = \delta(\delta(q, x), y)$ where x, y are the strings

Step 1:

Let

$$|y| = 1$$

$$y = a$$

$$\begin{aligned}\delta(q, xy) &= \delta(q, xa) \\ &= \delta(\delta(q, x), a) \text{ by property 2} \\ &= \delta(\delta(q, x), y).\end{aligned}$$

Step 2:

$$\delta(q, xy) = \delta(\delta(q, x_1), y) \text{ for } |y| = n$$

Step 3:

$$|y| = n+1$$

let $y = y_1 a$, where $|y_1| = n$.

$$\text{let } x_1 = xy_1$$

$$\delta(q, x_1 a) = \delta(\delta(q, x_1), a) \text{ by p-2.}$$

$$= \delta(\delta(q, xy_1), a)$$

$$= \delta(\delta(\delta(q, x), y_1), a) \text{ by}$$

Induction hypothesis

— (1)

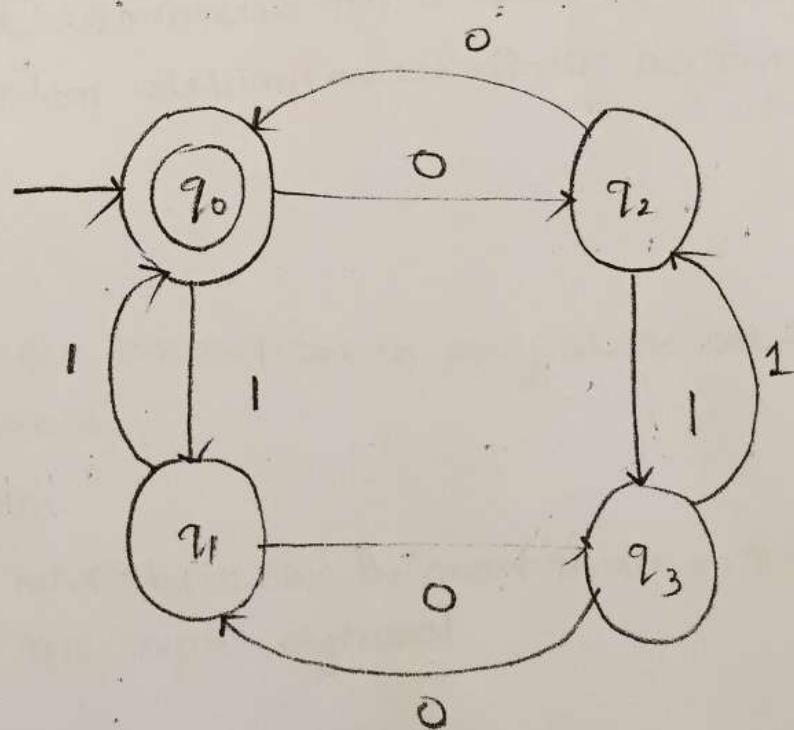
$$\begin{aligned}\text{R.H.S} &= \delta(\delta(q, x), y) \\ &= \delta(\delta(q, x), y_1 a)\end{aligned}$$

$$= \delta(\delta(\delta(q_1, x), y_1), a) \quad (2)$$

R.H.S = L.H.S

— hence proved.

	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2



find the acceptability of a string

110101

(2)

$$\begin{aligned}
 \delta(q_0, \downarrow \overset{\downarrow}{110101}) &= \delta(q_1, \downarrow \overset{\downarrow}{10101}) \\
 &= \delta(q_0, \downarrow \overset{\downarrow}{0101}) \\
 &= \delta(q_2, \downarrow \overset{\downarrow}{101}) \\
 &= \delta(q_3, \downarrow \overset{\downarrow}{01}) \\
 &= \delta(q_1, \downarrow \overset{\downarrow}{1}) \\
 &= \delta(q_0, \wedge) \\
 &= q_0
 \end{aligned}$$

$$q_0 \in F$$

so, given string is accepted

Finite automachine is 2-types.

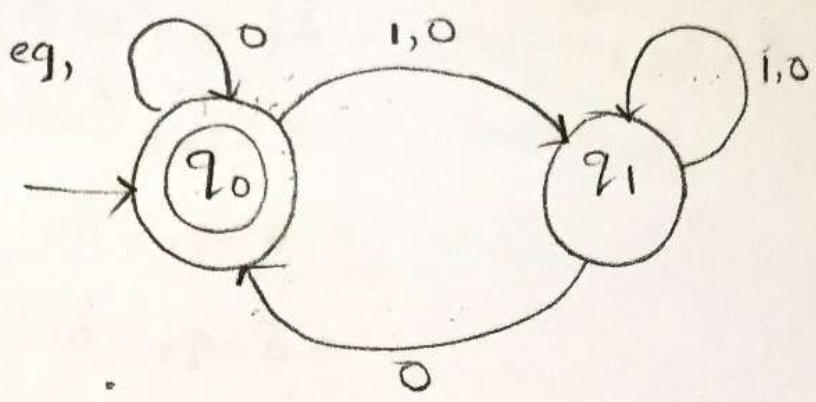
- Deterministic finite automachine (DFA)
- Non-deterministic finite automachine (NFA)

DFA :

unique transition in any state on an input symbol.

NFA :

In NFA there can be more than one transition on an input symbol.

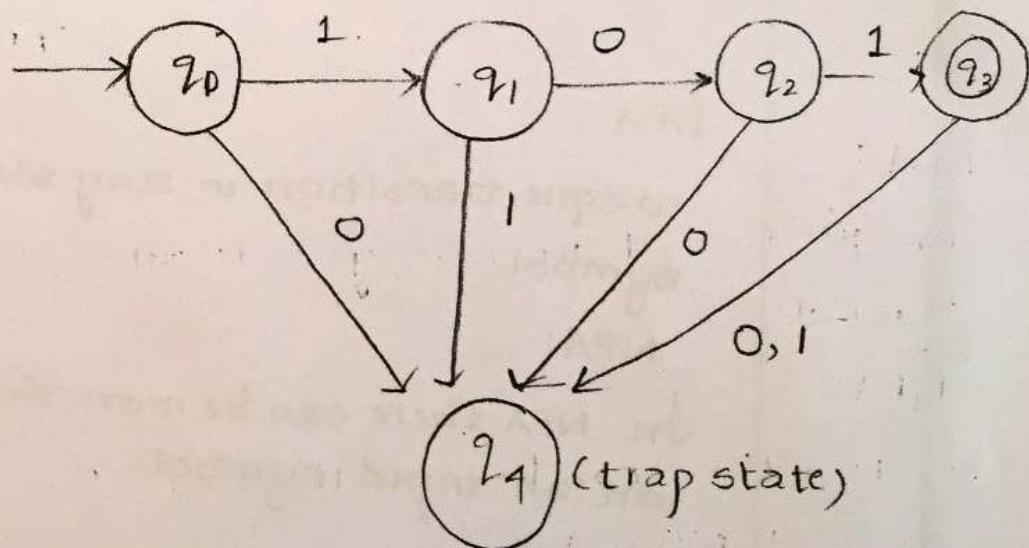


DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

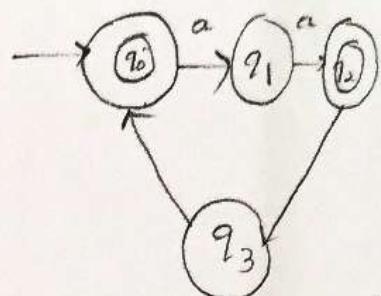
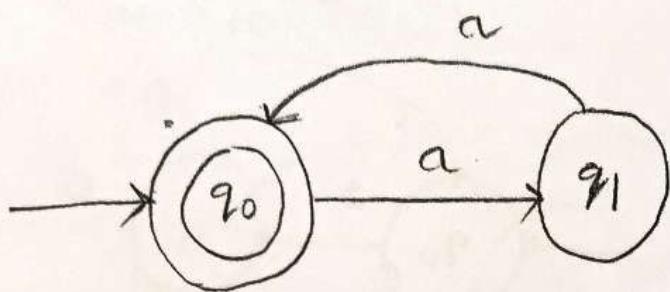
DFA can be used as a finite acceptor because its job is to accept certain input strings and rejects others.

It is also called language recogniser because it recognises whether the input strings are in the language or not.

Design DFA which accepts only input 101 over the set $\{0, 1\}$



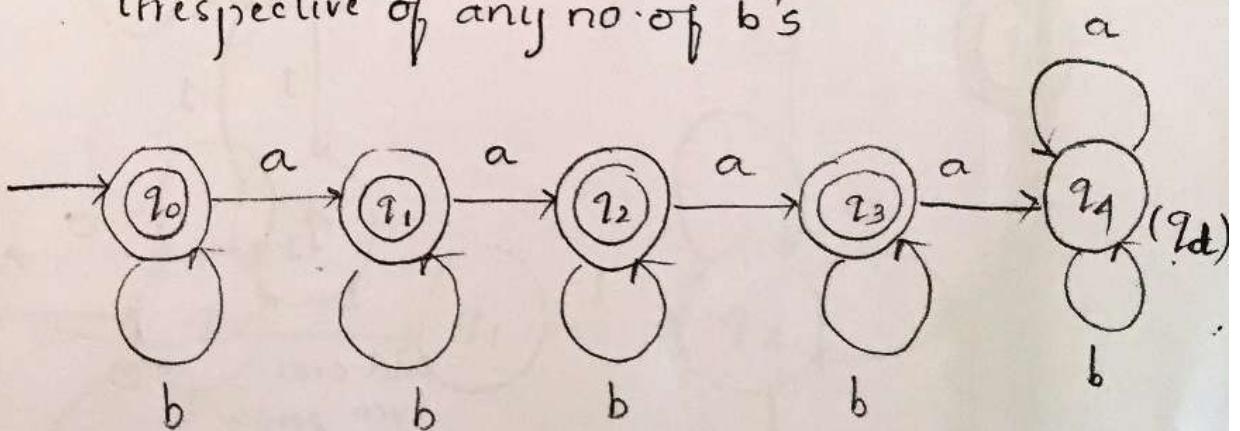
Design a DFA that accepts an even number of a's
over $\Sigma = \{a\}$



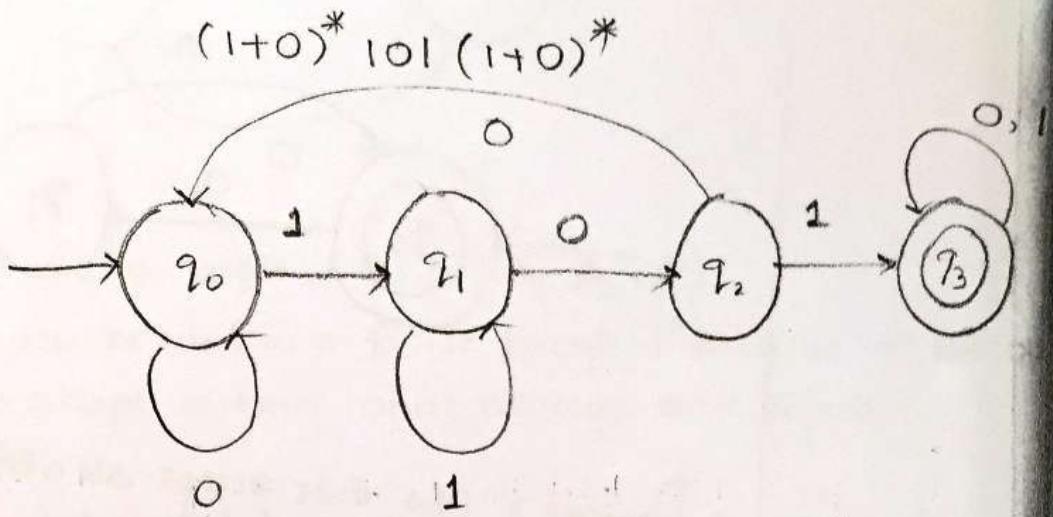
Design a DFA that accept all strings with atmost
3 a's over $\Sigma = \{a, b\}$

- 0 a's \rightarrow accepted state
- 1 a's \rightarrow accepted state
- 2 a's \rightarrow accepted state
- 3 a's \rightarrow accepted state
- 4 or more a's \rightarrow rejected state.

irrespective of any no. of b's

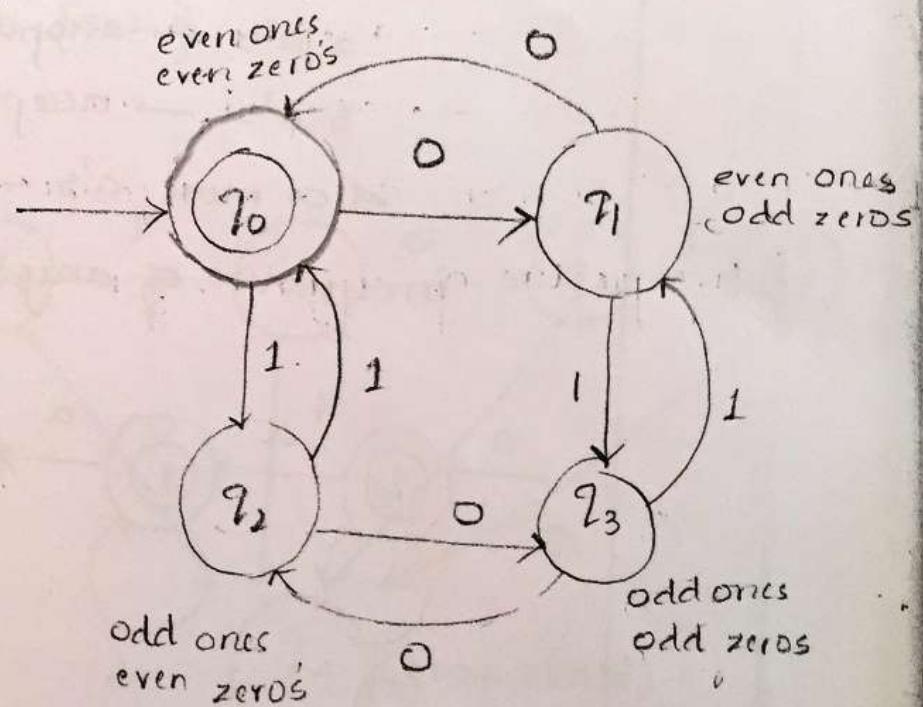


Design a DFA that contains 101 as a substring
in all strings over $\Sigma = \{0, 1\}$



Design a DFA - for the string must have 101 as a substring and contains odd number of 1's

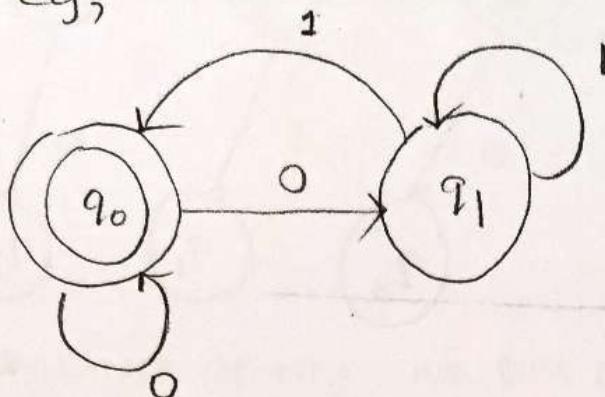
Design a DFA for even number of 1's & even number of 0's



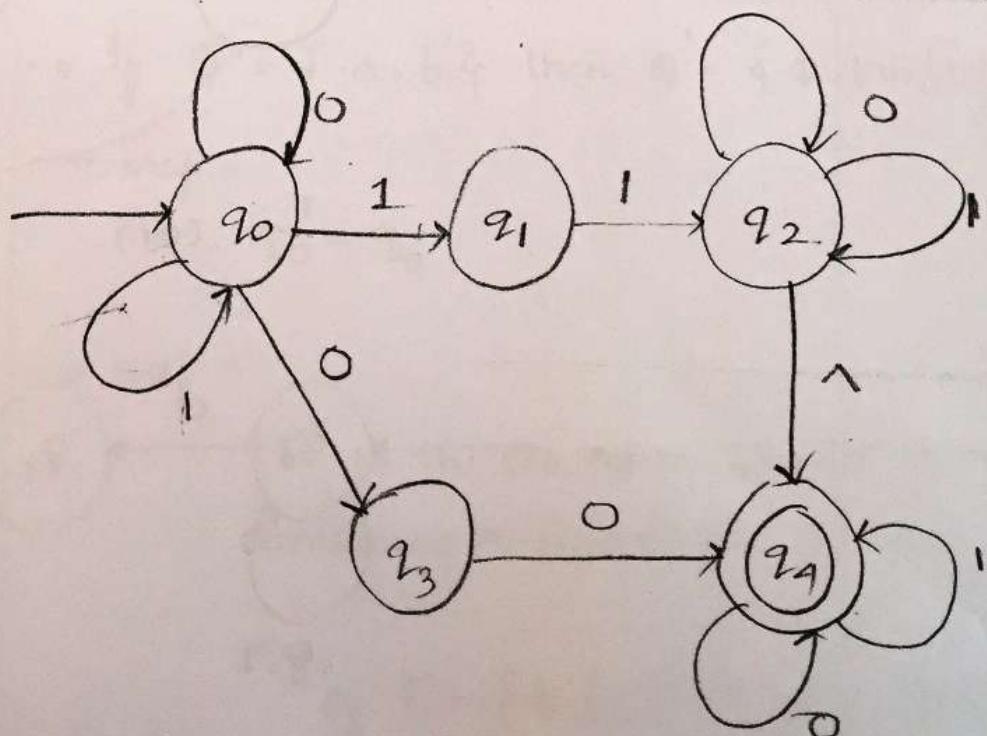
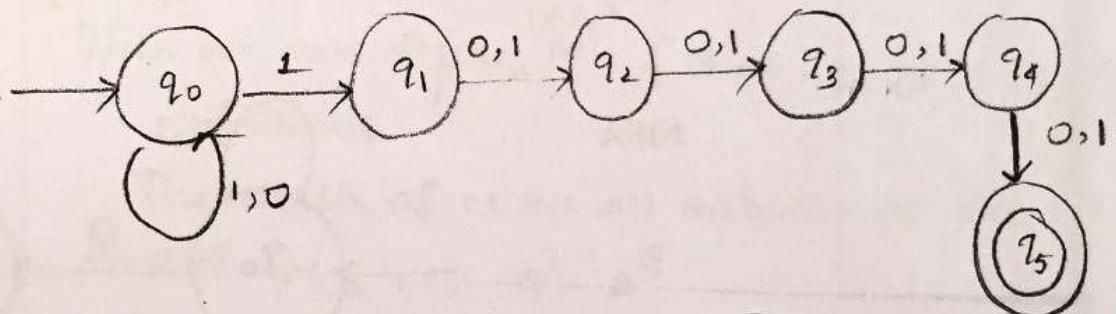
NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$
where δ is a transition function

$$\delta: Q \times \Sigma \rightarrow 2^{Q^*}$$

e.g.,

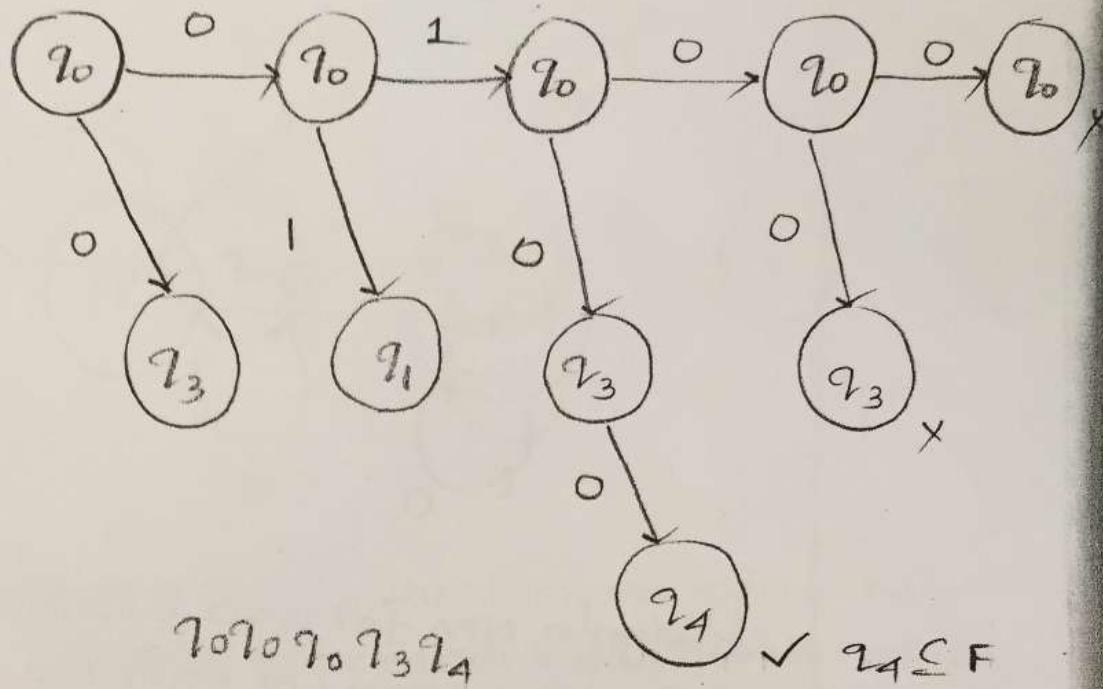


Design a NFA for a set of strings such that the 5th symbol from the right end is '1'.



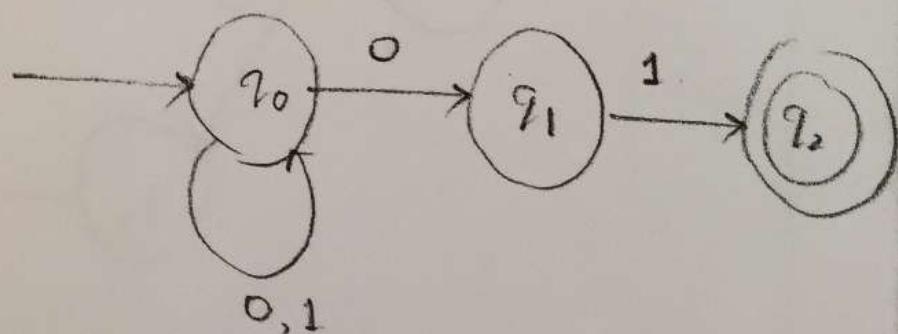
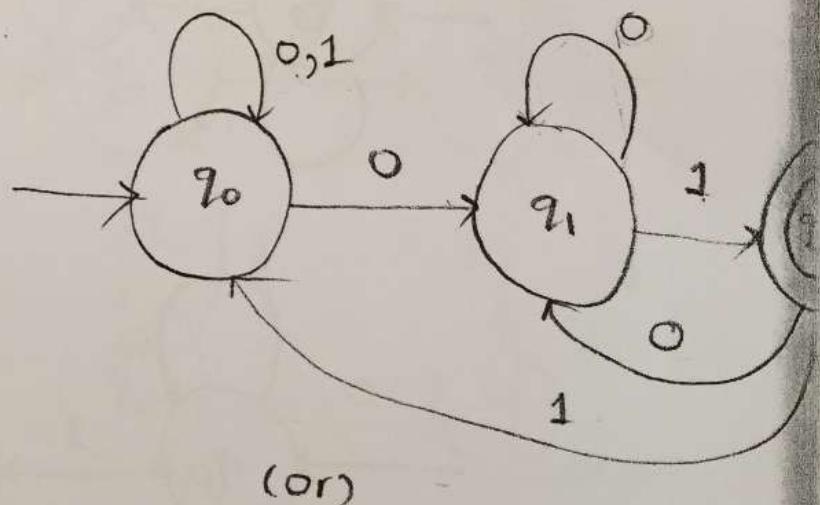
Find the acceptability of 0100

$$\delta(q_0, 0100)$$

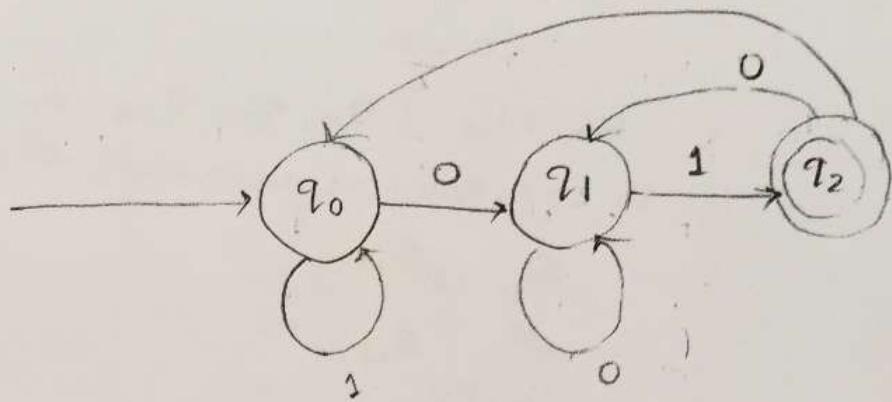


Design a NFA and DFA accepting all strings ending with 01 over $\Sigma = \{0, 1\}$

NFA:



DFA :



EQUIVALENCE OF NFA AND DFA :

Let 'L' be a set accepted by a NFA then there exist a DFA that accepts L

Let $M = (Q, \Sigma, q_0, \delta, F)$ be an NFA accepting L

Then we can define $M^{\text{DFA}} = \{Q', \Sigma', q_0', \delta', F'\}$ as follows

The states of M' are all subsets of the set of states M, i.e., $Q' = 2^Q$

\rightarrow If $Q = \{a, b\}$ then $Q' = \{\emptyset, [a], [b], [ab]\}$

\rightarrow Step 2:

$$q_0' = q_0$$

\rightarrow Step 3:

F' is the set of all states in Q' containing a final state of 'M'

e.g., If $F = \{b\}$ then $F' = \{[b], [ab]\}$

Converting NFA (M_N) to DFA (M_D) — subset construction:

Let $M_N = \{Q_N, \Sigma_N, \delta_N, q_{0N}, F_N\}$ be given NFA
to construct an equivalent DFA M_D as follows

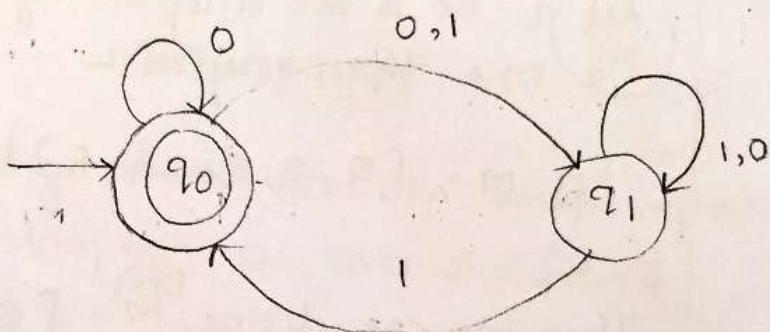
$$Q_D = 2^{Q_N}$$

$$\Sigma_D = \Sigma_N$$

$$q_{0D} = q_{0N}$$

$F_D = \text{set of all states of } Q_D \text{ that contains at least one final state of } F_N$

Convert the following NFA to DFA



$$\text{Here } Q = \{q_0, q_1\}$$

and

$$M = \{Q, \Sigma, \delta, q_0, F\}$$

for DFA:

$$M' = \{Q', \Sigma', \delta', q_0', F'\}$$

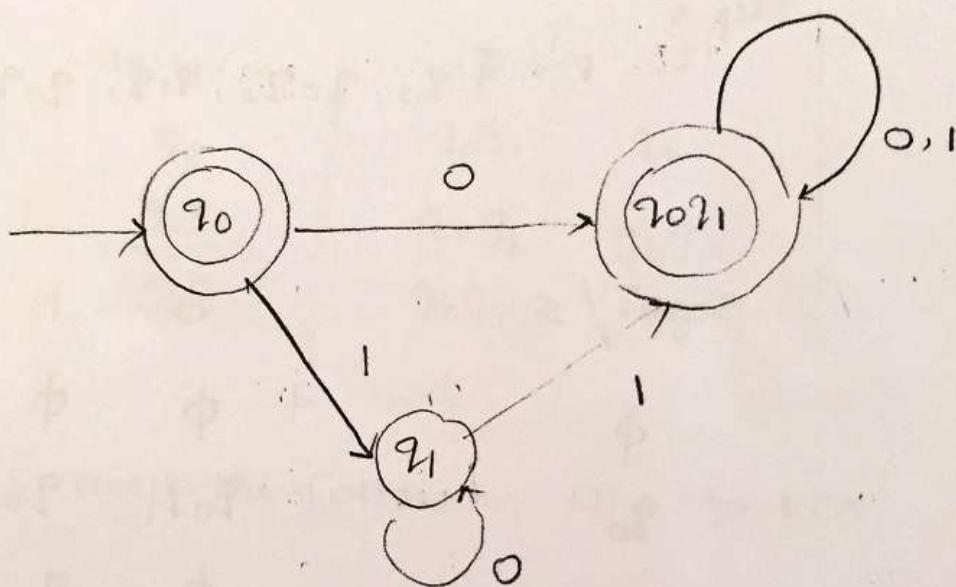
$$\text{Step 1: } Q' = \{\emptyset, q_0, q_1, q_0q_1\}$$

$$\text{Step 2: } q_0' = q_0$$

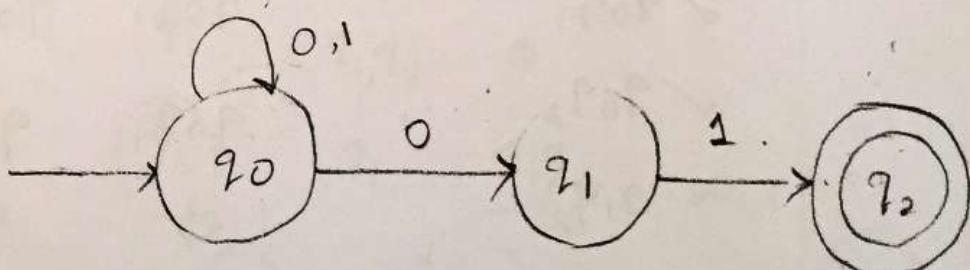
$$\text{Step 3: } F' = \{q_0, q_0q_1\}$$

CONVERSION TO DFA :

states / ϵ	0	1
\emptyset	\emptyset	\emptyset
q_0	<u>$q_0 q_1$</u>	<u>q_1</u>
q_1	<u>q_1</u>	<u>$q_0 q_1$</u>
$q_0 q_1$	$q_0 q_1$	$q_0 q_1$



Convert the following NFA to DFA



$$\text{Hence } Q = \{q_0, q_1, q_2\}$$

and

$$M = \{Q, \Sigma, \delta, q_0, F\}$$

for DFA :

$$M^1 = \{ Q^1, \Sigma^1, \delta^1, q_0^1, F^1 \}.$$

Step 1 :

$$Q^1 = \{ \emptyset, q_0, q_1, q_2, q_0q_1, q_0q_2, q_1q_2, q_0q_1q_2 \}$$

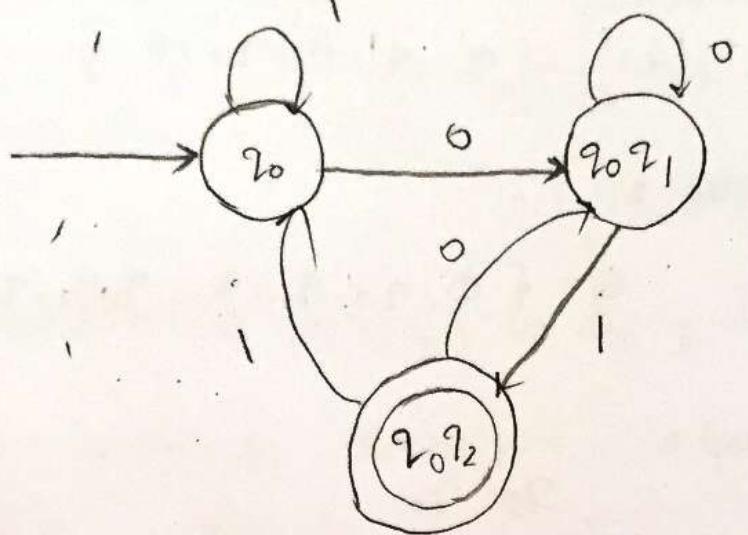
Step 2 :

$$q_0^1 = q_0$$

Step 3 :

$$F^1 = \{ q_2, q_0q_2, q_1q_2, q_0q_1q_2 \}$$

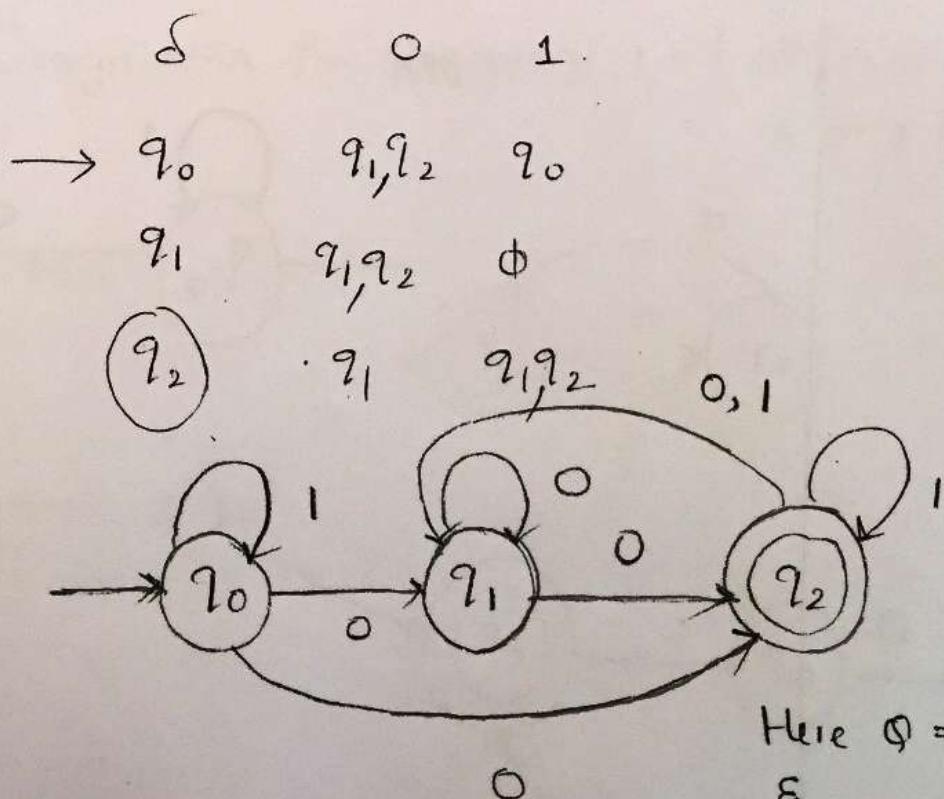
Status / Σ	0	1
\emptyset	\emptyset	\emptyset
q_0	q_0q_1	q_0
$\times q_1$	\emptyset	q_2
$\times q_2$	\emptyset	\emptyset
$\checkmark q_0q_1$	q_0q_1	q_0q_2
$\checkmark q_0q_2$	q_0q_1	q_0
$\times q_1q_2$	\emptyset	q_2
$\times q_0q_1q_2$	q_0q_1	q_0q_2



Alternate method:-

states/ ϵ	0	1
q_0	q_0q_1	q_0
$\underline{q_0q_1}$	q_0q_1	q_0q_2
q_0q_2	q_0q_1	q_0

Convert the following NFA to DFA.



$$Hence Q = \{q_0, q_1, q_2\}$$

$$\Sigma = \{\varnothing, \epsilon, \delta, q_0, F\}$$

for DFA:

$$M' = \{ Q', \leq', \delta', q_0', F' \}$$

step 1:

$$Q' = \{ \emptyset, q_0, q_1, q_2, q_0q_1, q_0q_2, q_1q_2, q_0q_1q_2 \}$$

step 2:

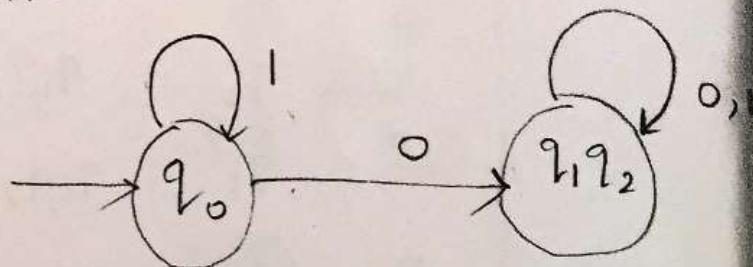
$$q_0' = q_0$$

step 3:

$$F' = \{ q_2, q_0q_2, q_1q_2, q_0, q_1q_2 \}$$

status / \leq	0	1
q_0	q_1q_2	q_0
q_1q_2	q_1q_2	q_1q_2

DFA:

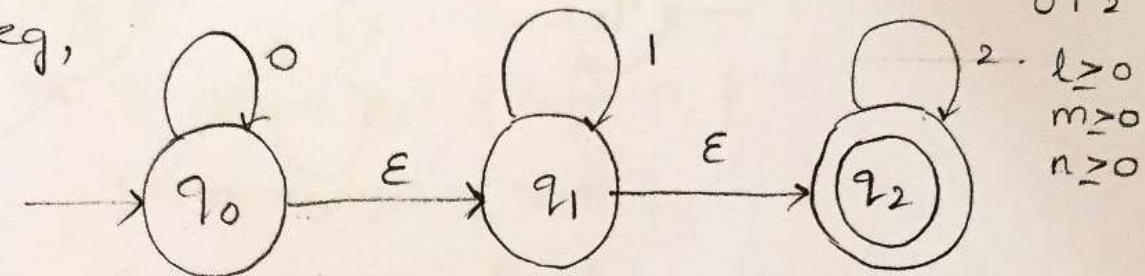


$$\delta_D((q_1, q_2, q_3), a) = \delta_N(q_1, a) \cup \delta_N(q_2, a) \\ \cup \delta_N(q_3, a) \\ = \{p_1, p_2, p_3\}$$

add start $[p_1, p_2, p_3]$ to Q_D if it is not there.

NFA WITH ϵ TRANSITION:

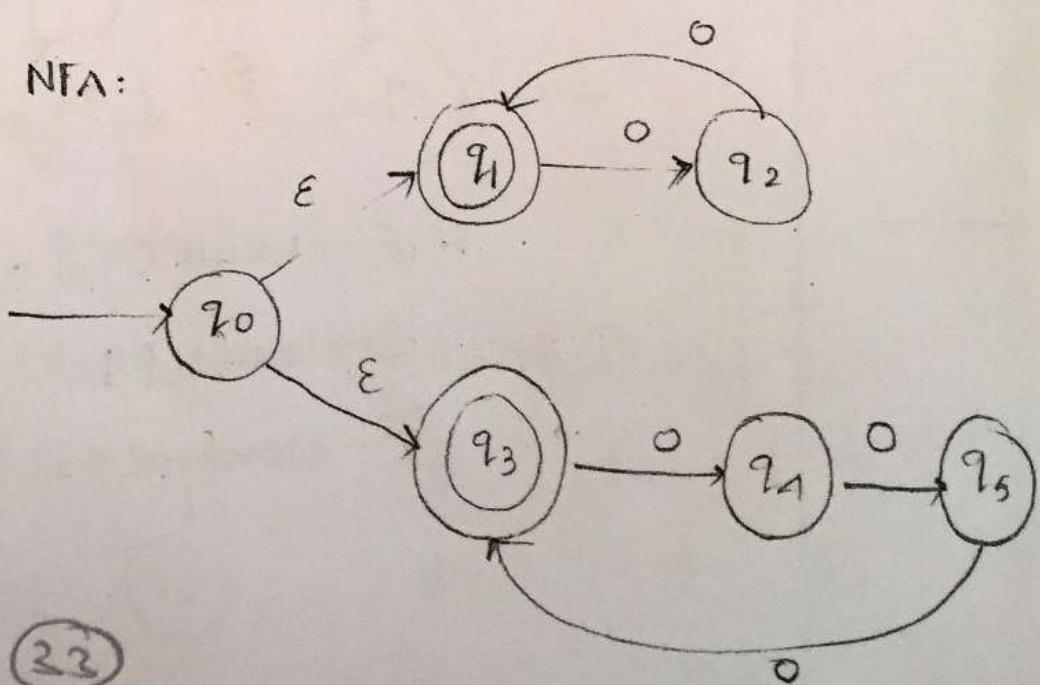
e.g,



We can extend an NFA by introducing a ' ϵ '-moves that allows us to make a transition on the empty string. There would be an edge labelled ' ϵ ' between '2' states, which allows the transition from one state to another even without receiving an input symbol.

Design NFA - For language $L = \{0^k / k \text{ is multiple of } 2 \text{ or } 3\}$

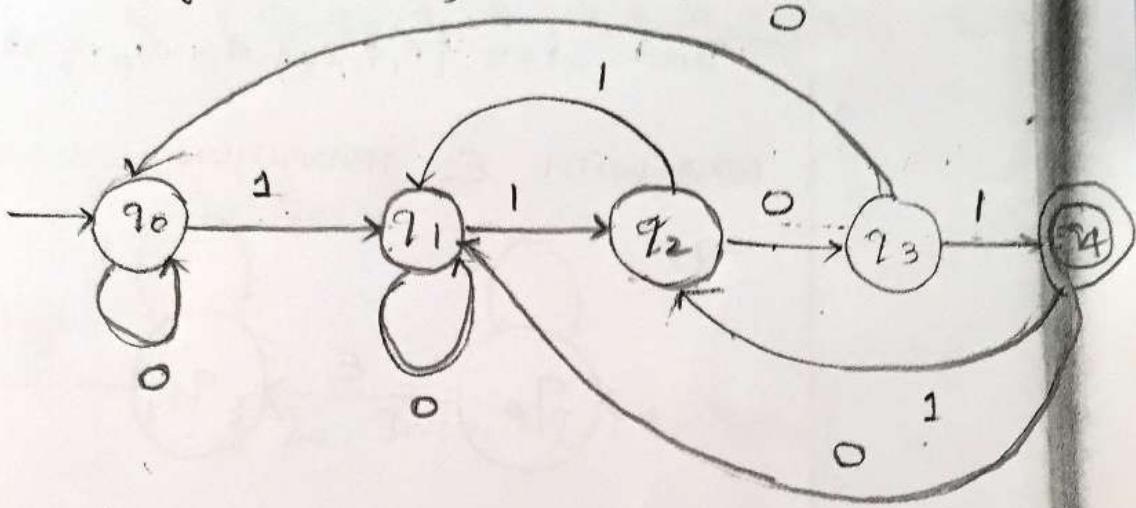
NFA:



$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

$$F = \{q_1, q_3\}$$

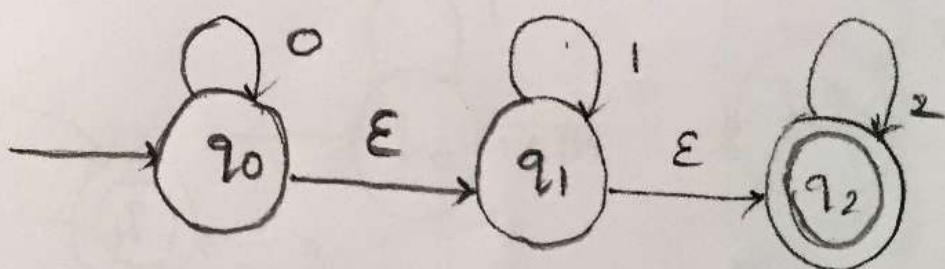
Design DFA ending with 101 & odd number of one.



ϵ -CLOSURE:

ϵ -closure of a state is simply the set of all states we can reach by following transitions from the given state that are labeled ϵ . This can be expressed as $\hat{\epsilon}(q)$ or ϵ -closure(q).

for the below example



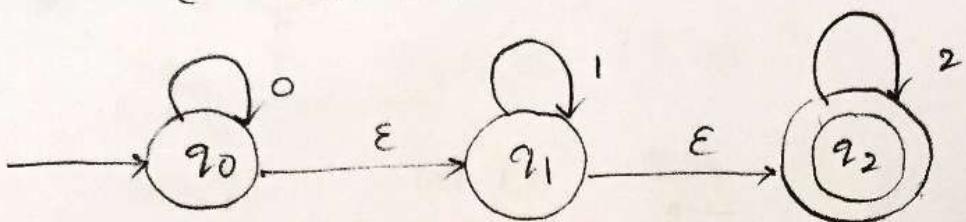
$$\epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$$

$$\epsilon\text{-closure}(q_1) = \{q_1, q_2\}.$$

$$\epsilon\text{-closure}(q_2) = \{q_2\}.$$

ELIMINATING 'ε' TRANSITIONS:

Converting NFA with ε transition to NFA without ε transition:



Step 1:

$$\text{ε-closure}(q_0) = \{q_0, q_1, q_2\}$$

$$\text{ε-closure}(q_1) = \{q_1, q_2\}$$

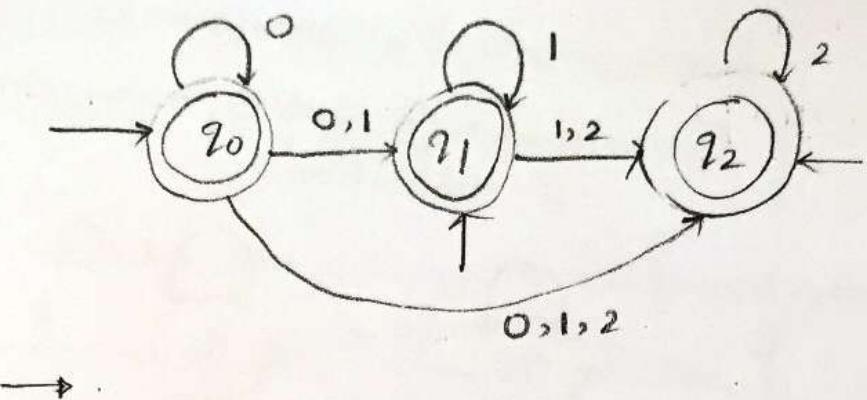
$$\text{ε-closure}(q_2) = \{q_2\}$$

Step 2:

States / ε	0	1	2	ε
→ q ₀	q ₀	∅	∅	q ₁
q ₁	∅	q ₁	∅	q ₂
q ₂	∅	∅	q ₂	∅

Step 3:

States / ε	0	1	2
→ q ₀	{q ₀ , q ₁ , q ₂ }	{q ₁ , q ₂ }	q ₂
q ₁	∅	{q ₁ , q ₂ }	q ₂
q ₂	∅	∅	q ₂



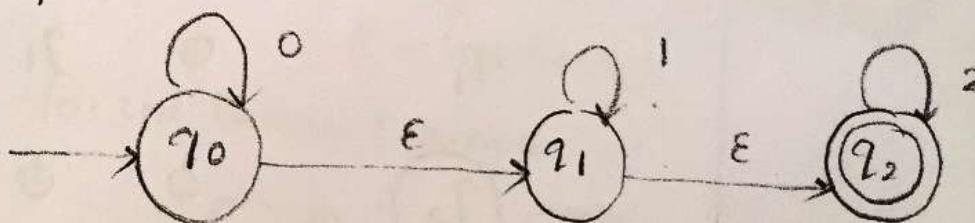
for each state compute ϵ closure on each input symbol $a \in E$ if the ϵ -closure of a state contains final state then make that state as final. ϵ closure of initial state are the initial states of the initial states.

CONVERTING NFA WITH ' ϵ ' TRANSITIONS TO DF.

Step 1 :

Compute ' ϵ ' closure for the current state, resulting in a set of states (init state)

Step 2 :



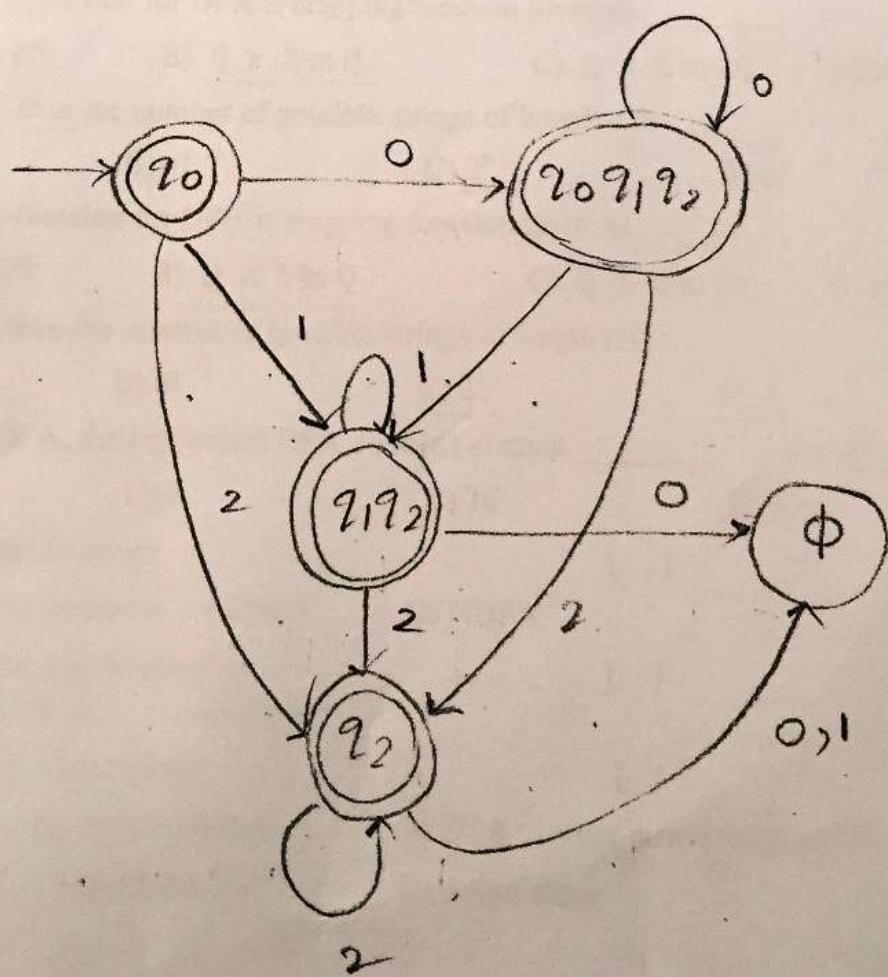
states/ ϵ	0	1	2	ϵ
q_0	q_0	\emptyset	\emptyset	q_2
q_1	\emptyset	q_1	\emptyset	q_2
q_2	\emptyset	\emptyset	q_2	\emptyset

step 3:

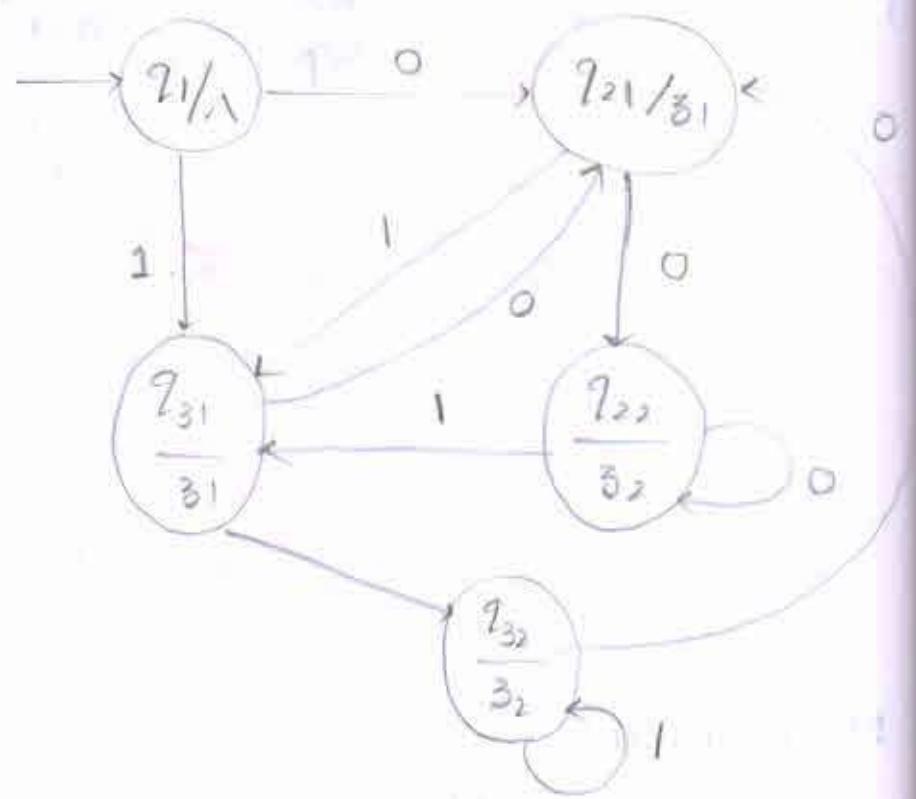
status/ ϵ	0	1	2
q_0	$[q_0 q_1 q_2]$	$[q_1 q_2]$	q_2
$q_0 q_1 q_2$	$q_0 q_1 q_2$	$q_1 q_2$	q_2
$q_1 q_2$	\emptyset	$q_1 q_2$	q_2
q_2	\emptyset	\emptyset	q_2

step 4:

Make a state as an accepting state if it includes any final states in the NFA



of



MINIMISATION OF FINITE AUTOMATA:

Two states q_1 & q_2 are equivalent if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states or both them are non-final states for all $x \in \Sigma$ it means from 1 to infinite.

As it is difficult to construct $\delta(q_1, x)$ $\delta(q_2, x)$ for all $x \in \Sigma^*$ (there may be infinite no. of strings).

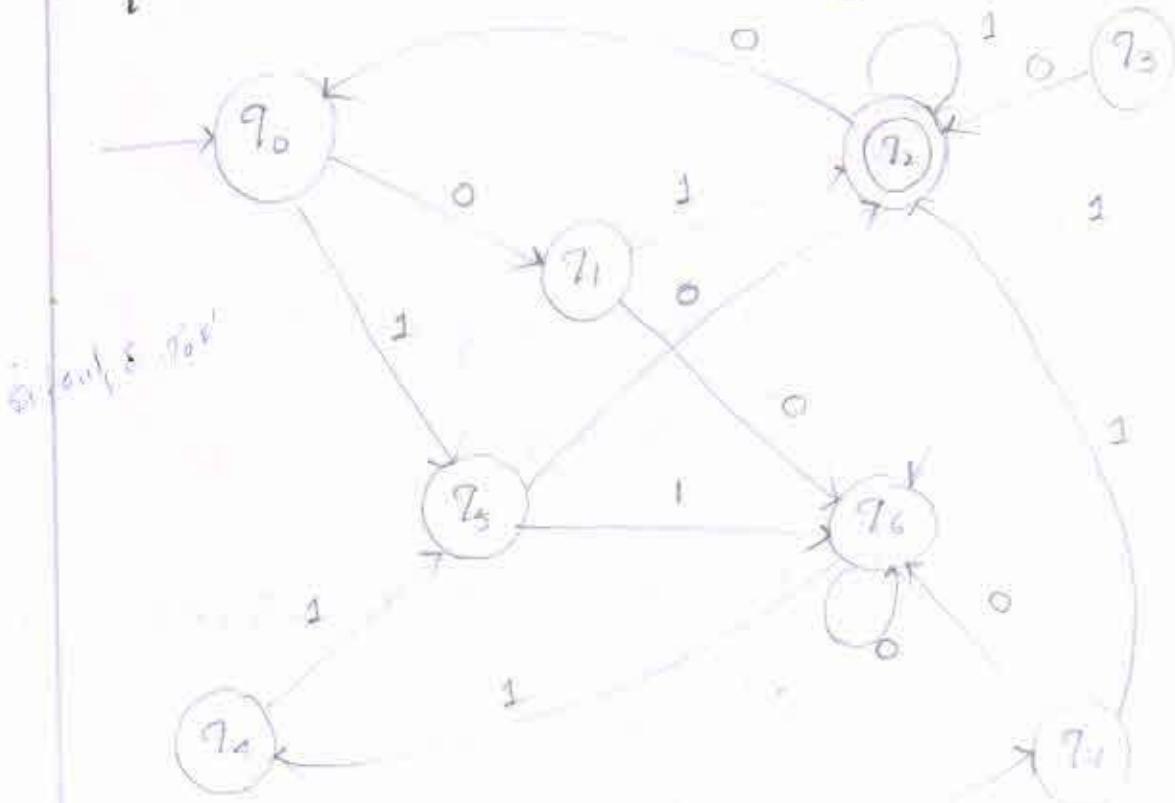
Two states q_1 and q_2 are k -equivalent if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final or both are non-final states for all strings x of length k or less.

NOTE: By default all the final states are 0-equivalent and all non-final states are 0-equivalent.

PROPERTY 1

The relations would have defined i.e., equivalence and k -equivalence are equivalence relations i.e., they are reflexive, symmetric and transitive.

Construct a minimum state automaton equivalent to finite automaton given below:



states / ϵ

	0	1
$\rightarrow q_0$	q_1	q_5
q_1	q_6	q_2
q_2	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_0	q_4
q_7	q_2	q_2

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

$$\Pi_0 = \left\{ \{q_2\}, \{q_0, q_1, q_3, q_4, q_5, q_6, q_7\} \right\}$$

$$Q_1^0 = q_2$$

$$Q_2^0 = Q - Q_1^0$$

Step 2 Π_1

$$Q_1^1 = \{q_2\}$$

$$Q_2^1 = \{q_0, q_1, q_3, q_5, q_7\}$$

$$Q_3^1 = \{q_4, q_6\}$$

$$\Pi_1 = \left\{ \{q_2\}, \{q_0, q_1, q_3, q_5, q_7\}, \{q_4, q_6\} \right\}$$

$$Q_1^2 = \{q_2\}$$

$$Q_2^2 = \{q_1, q_7\}$$

$$Q_3^2 = \{q_3, q_5\}$$

$$Q_4^2 = \{q_0, q_4\}$$

$$Q_5^2 = \{q_6\}$$

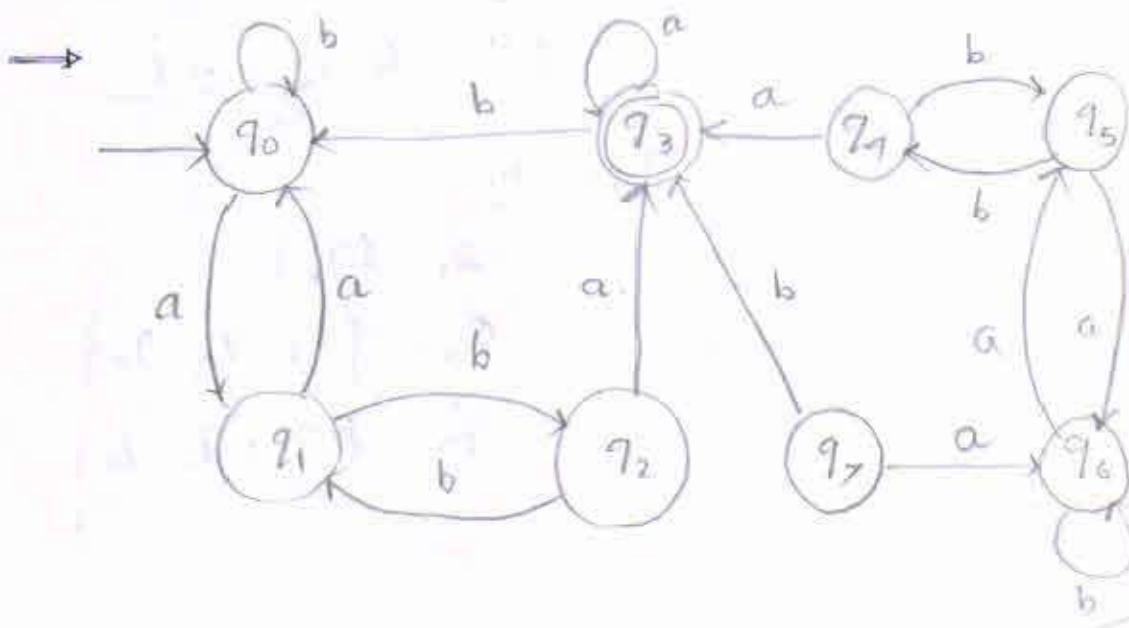
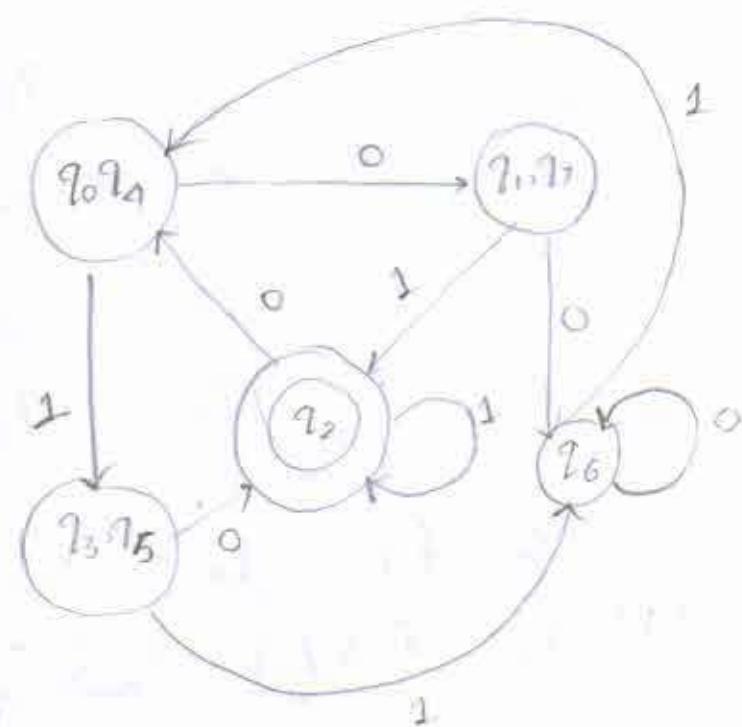
$$\Pi_2 = \left\{ \{q_2\}, \{q_1, q_7\}, \{q_3, q_5\}, \{q_0, q_4\}, \{q_6\} \right\}$$

$$M' = \{q^1, \{q_0, q_1\}, \delta^1, q_0^1, F^1\}$$

$$Q' = \{q_2, [q_0, q_4], q_6, [q_1, q_3], [q_3, q_5]\}$$

$$q_0^1 = [q_0, q_4]$$

$$F^1 = \{q_2\}$$



States	input	
	<u>a</u>	<u>b</u>
$\rightarrow q_0$	q_1	q_0
q_1	q_0	q_2
q_2	q_3	q_1
q_3	q_3	q_0
q_4	q_3	q_5
q_5	q_6	q_4
q_6	q_5	q_6
q_7	q_6	q_3

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

Step 1: $\pi_Q = \{\{q_3\}, \{\underline{q_0, q_1, q_2, q_4, q_5, q_6, q_7}\}\}$

$$Q_1^0 = q_3 \rightarrow f$$

$$Q_2^0 = Q - Q_1^0 - n \cdot f$$

Step 2: π_1

$$Q_1' = \{q_3\}$$

$$Q_2' = \{q_2, q_4, q_7\}$$

$$Q_3' = \{q_0, q_1, q_5, q_6\}$$

$$\pi_1 = \{ \{q_3\}, \{q_1, q_4, q_7\}, \{q_0, q_1, q_5, q_6\} \}$$

$$Q_1^2 = \{q_3\}$$

$$Q_2^2 = \{q_1, q_5\}$$

$$Q_3^2 = \{q_0, q_6\}$$

$$Q_4^2 = \{q_1, q_4\} \quad Q_5^2 = \{q_7\}$$

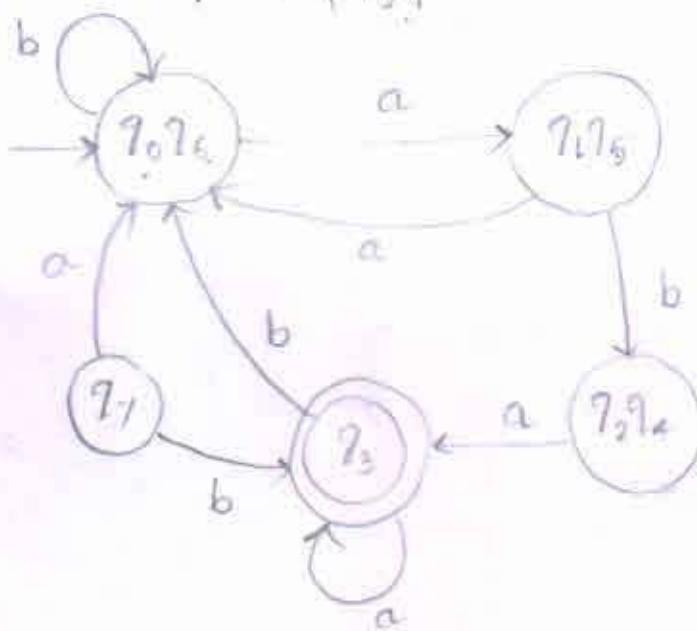
$$\pi_2 = \{ \{q_3\}, \{q_1, q_5\}, \{q_0, q_6\}, \\ \{q_2, q_4\}, \{q_7\} \}$$

$$M^1 = \{Q^1, \{a, b\}, \delta^1, q_0^1, F^1\}$$

$$Q^1 = \{q_3, [q_1, q_5], [q_0, q_6], [q_2, q_4], q_7\}$$

$$q_0^1 = [q_0, q_6]$$

$$F^1 = \{q_3\}$$



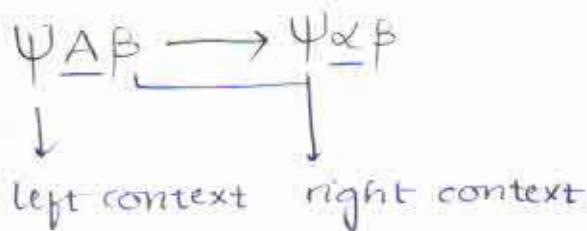
def:

If q_i, q_j are said to be k -equivalence $\forall k \in$
then q_i, q_j are said to be equivalence

→ If q_i, q_j are said to be k -equivalence for
some k then q_i, q_j are said to be $(k-1)$
equivalence

3. FORMAL LANGUAGES:

Chomsky → classification of languages



Grammar is basically a-tuple

$$G_1 = (V_N, \Sigma, P, S)$$

V_N - it is finite set collection of variables or non-terminals

Σ - it is finite set collection of terminals
here,

$$V_N \cap \Sigma = \emptyset$$

S - starting symbol where $S \in V_N$

$S \longrightarrow <\text{Noun}> <\text{verb}>$

$S \longrightarrow <\text{Noun}> <\text{verb}> <\text{adverb}>$

Noun → Priya, Raj

verb → run, ate

adverb → quickly, slowly

P - collection of productions in the form of $\alpha \rightarrow \beta$ where

$$\alpha, \beta \in V_N \cup \Sigma$$

eg, $0^n 1^n, n \geq 0$.

$S \rightarrow 0S1/\lambda$

$0S1$

$00S11$

$0^2 S_1^2$

$00^2 S_{11}^2$

$0^3 S_1^3$

Replacing S by λ we get, $0^3 \lambda$.

eq,

→ In a production of the form $\phi A \psi \rightarrow \phi \alpha$, where A is a variable (non-terminal) ϕ is called the left context, ψ is called the right context, and $\phi \alpha \psi$ is the replacement string.

→ $abAbcd \rightarrow ab\lambda Bbcd$

$ab \rightarrow$ left context

$bcd \rightarrow$ right context

here $\alpha = AB$

→ $\lambda C \Delta \rightarrow \lambda \Delta \Delta$

left context is λ

right context is λ

and $\alpha = \lambda$

→ $C \rightarrow \lambda$ (here 'C' is erased)

left context & right context is ' λ '

here $\alpha = \lambda$

type 0 \rightarrow A production without any restrictions
 type 1
 type 2
 type 3

TYPE - 1

A production of the form $\phi\lambda\psi \rightarrow \phi\alpha\psi$
 is called a type-1 production if $\alpha \neq \lambda$ (i.e., $A \neq \lambda$)
 In type-1 production erasing of λ is not permitted

eg,

(1) $a\underline{AbcD} \rightarrow a\underline{bcD}bcD$
 type-1 grammar

$$(2) \frac{\overbrace{AB}^1}{\downarrow \text{upt}} \rightarrow \frac{\overbrace{AbBc}^1}{\downarrow \text{upt}}$$

(3) $\underline{A} \rightarrow \underline{aba}$

\rightarrow A grammar is called type-1 (or)
 context sensitive (or) context dependent if
 all its productions are type-1 productions
 The production $s \rightarrow \lambda$ is allowed in type-1
 grammar but in this case λ does not
 appear on the righthand side of any
 production

The language generated by type-1
 grammar is called type-1 or context
 sensitive language

In a context sensitive grammar if
 we allow $s \rightarrow \lambda$ apart from $s \rightarrow \lambda$ all the
 other productions do not decrease the

length of the working string.

TYPE-2

It is the production of the form $A \rightarrow \alpha$
where $A \in V_N$, $\alpha \in (V_N \cup \epsilon)^*$

In other words the L.H.S has no right
and left context.

e.g.,

$$A \rightarrow a, B \rightarrow abc, A \rightarrow aa$$

A grammar is called type-2 grammar if it contains only type-2 production. It is also called a context free grammar.

A language generated by context free grammar is called a type-2 language or context-free language.

TYPE 3:

A production of the form $A \rightarrow a$ or
 $A \rightarrow aB$ where $A, B \in V_N$ and $a \in$
is called a type 3 production.

A grammar is called a type-3 (or) regular grammar if all its production are type 3 productions.

A production $s \rightarrow n$ is allowed in
type-3 grammar but in this case
 s does not appear on the RHS of any
production.

Find the highest type of number which can be applied to following grammars

(a) $S \rightarrow Aa \rightarrow \text{type } 2$

$$A \rightarrow C / Ba$$

\downarrow \downarrow
type - 3 type - 2

$$B \rightarrow abc \rightarrow \text{type } 2$$

(b) $S \rightarrow ASB / d$

$$\downarrow \qquad \downarrow$$

type 2 type 3

$$A \rightarrow AA$$

$$\downarrow$$

type 3

(c) $S \rightarrow aS / ab$

$$\downarrow \qquad \downarrow$$

type 3 type 2

REGULAR EXPRESSION & REGULAR GRAMMAR

Any terminal or element of Σ is regular expression.

for eg, a in Σ , \emptyset is also regular.

$\{a\}$ \downarrow expression

λ is regular expression

$\{^n\}$

UNION:

Union of '2' regular expressions R_1 and R_2 is a regular expression R' ($R = R_1 + R_2$)

let a' be regular expression in R_1

b' be regular expression in R_2 then

($a+b$) is also a regular expression R' having the elements { a, b }

CONCATENATION:

Concatenation of '2' regular expressions R_1 and R_2 written as R_1R_2 is also a regular expression R ($R = R_1R_2$)

let a' be regular expression in R_1

b' be regular expression in R_2

(ab) is also a regular expression R' having the elements { ab }

ITERATION (CLOSURE):

Iteration (closure) of a regular expression R' is written as R^* is also a regular expression

let a be a regular expression then

$\lambda, a, aa, aaa, \dots$ are also regular expressions

NOTE:

If L is a language represented by a regular expression R then the Kleen's closure of L is denoted as L^* and is given as

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

The positive closure of L^* is denoted as L^+

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

If R is a regular expression then $(R)^*$ is also a regular expression

Regular expression over Σ is precisely those obtain recursively by the application of the above rules once or several times.

Closure has highest precedence next highest is for concatenation and least is for union

IDENTIFIER NOTATION

notation : $I(\{d\})^*$

REGULAR SET

Any set represented by a regular expression is called a regular set

If a, b are the elements of S then the regular expressions a denote the set $\{a\}$

$a+b$ denote the set $\{a, b\}$

ab denote the set $\{ab\}$

a^* denote the set $\{\lambda, a, aa, aaa, \dots\}$

$(a+b)^*$ denote the set

$\{\lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, \dots\}$

Regular set

$\{101\}$

$\{1, a\}$

Regular expression

101

$a + a$

$\{ \epsilon, a, b, aa, bb, ba \dots \} \quad (a+b)^*$

$\{ ab, ba \} \quad ab+ba$

Describe the following set by regular expression

all string with 0's and 1's $\rightarrow (0+1)^*$

set of all strings of 0's & 1's ending with 00

$\rightarrow (0+1)^*00$

Set of all strings 0's & 1's begin with 0 and end with 1

$0(0+1)^*1$

Set of all strings having even number of 1's

$(0+11)^*$

Set of all string having odd number of 1's

$1(11)^* \text{ or } (11)^*1$

Strings of 0's and 1's with at least 2 consecutive zeros

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

All strings of 0's and 1's beginning with 1 or 0 and not having 3 consecutive zeros

$(1+01)^*$

Set of strings in which every zero immediately followed by at least 2 1's

$(1+011)^*$

Set of all strings ending with 011

$(0+1)^*011$

Identities for Regular Expressions

$$I_1 \quad \phi + R = R$$

$$I_2 \quad \phi R = R\phi = \phi$$

$$I_3 \quad \lambda R = R\lambda = R$$

$$I_4 \quad \lambda^* = \lambda \quad \& \quad \phi^* = \lambda$$

$$I_5 \quad R + R = R$$

$$I_6 \quad R^* R^* = R^*$$

$$I_7 \quad R R^* = R^* R$$

$$I_8 \quad (R^*)^* = R^*$$

$$I_9 \quad \lambda + R R^* = \lambda + R^* R = R^*$$

$$I_{10} \quad (PQ)^* = P(QP)^*$$

$$I_{11} \quad (P+Q)^* = (P^* Q^*)^* = (P^* + Q^*)^*$$

$$I_{12} \quad (P+Q)R = PR + QR \quad \& \quad R(P+Q) = RP + RQ$$

\rightarrow 2 regular expressions P & Q are equivalent
if P & Q represent the same set of strings.

ARDEN'S THEOREM:

Let P and Q be 2 regular expressions over Σ if P does not contain null(λ) then the following equation in $R = Q + RP$ has a unique solution given by $R = QP^*$

PROOF:

$$\text{Case(i)} \quad R = Q + RP$$

$$R = Q + (QP^*)P$$

$$= Q(\lambda + PP^*)$$

$$R = QP^* \quad (\text{from } T_q)$$

Case (ii) :

$$R = Q + RP$$

$$R = Q + (Q + RP)P$$

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

$$= Q + QP + (Q + RP)P^2$$

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

$$\vdots$$

$$= Q + QP + QP^2 + \dots + QP^i +$$

$$QP^{i+1}$$

$$= Q(\lambda + P + P^2 + \dots + P^i) +$$

$$RP^{i+1}$$

$$= Q(P^*) + RP^{i+1}$$

$$= Q(P^*)$$

Given a regular expression represent
the set 'L' of strings in which

Prove that the regular expression $R = \lambda +$

$$1^*(011)^*(1^*(011)^*)^*$$

$$= (1+011)^*$$

$$\text{d-H.S} = \lambda + 1^*(011)^*(1^*(011)^*)^*$$

$$\text{Let } R = 1^*(011)^*$$

$$\text{L-H.S} = \lambda + R(R)^*$$

$$= R^* \text{ by } T_q$$

$$\text{L.H.S} = (I^*(QII)^*)^*$$

$$= (I+QII)^* \text{ by } T_{II}$$

$$= R \cdot H \cdot S$$

ALGEBRA LAW FOR REGULAR EXPRESSIONS:

→ union operation on regular expressions are commutative i.e., $R+S=S+R$.

These are associative i.e., $(R+S)+T=R+(S+T)$

→ concatenation operation on regular expression are associative

$$t(st) = (ts)t$$

→ concatenation is right distributive over addition union & & left distributive over union

$$(R+S)T = RT+ST$$

$$T(R+S) = TR+TS$$

$$\rightarrow \phi^* = \lambda$$

FINITE AUTOMATA AND REGULAR EXPRESSIONS:

TRANSITION SYSTEM AND REGULAR EXPRESSIONS:

Every regular expression R can be recognised by a transition system i.e., for every string ' w ' in the set R there exist a path from

the initial state or final state with the path value 'w'

TRANSITION SYSTEM CONTAINING 'NULL-MOVES'

Suppose we want to replace a λ -move from vertex v_1 to vertex v_2 then we proceed as follows.

Step 1:

Find all edges starting from v_2

Step 2:

Duplicate all these edges starting from v_1 , without changing the edge labels

Step 3:

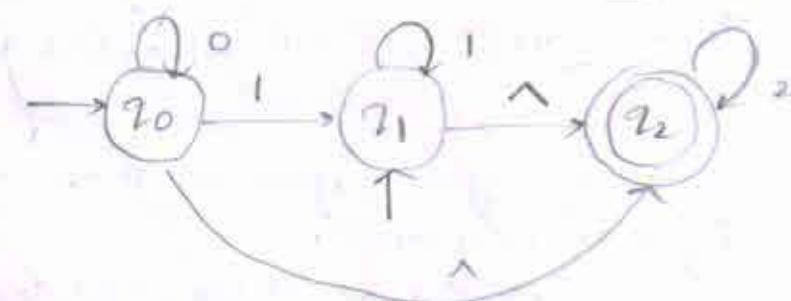
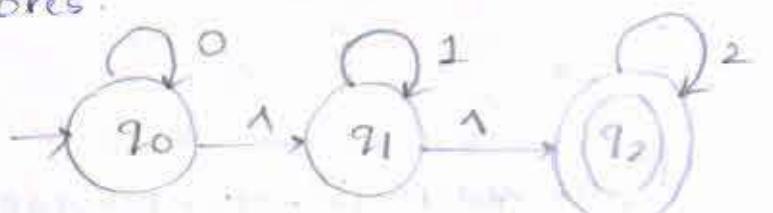
If v_1 is the initial state, make v_2 also as initial state

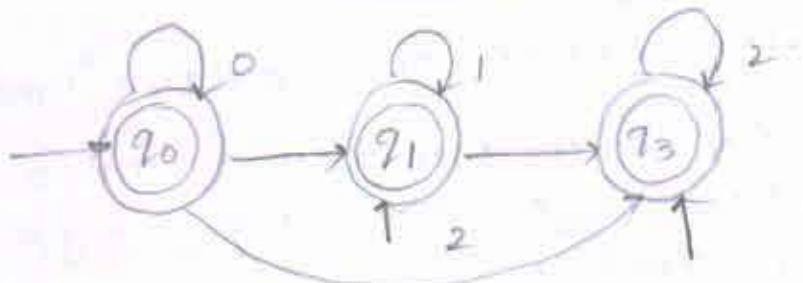
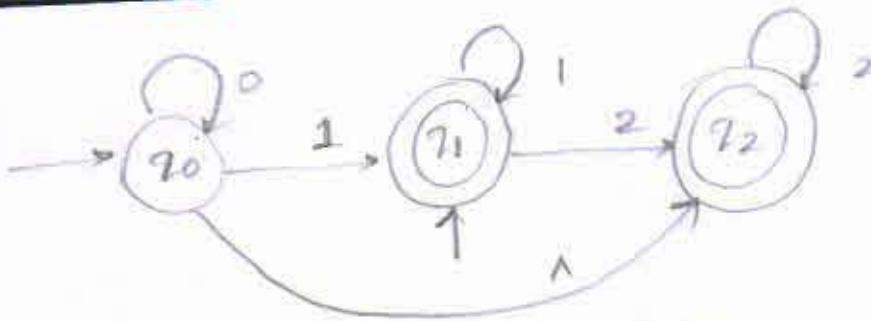
Step 4:

If v_2 is a final state make v_1 as the final state

e.g.,

Consider a finite automata with null moves & obtain an equivalent automata without null moves.

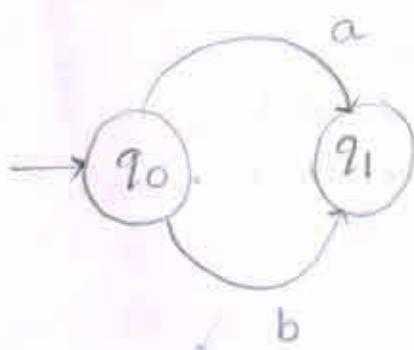




Finite automata

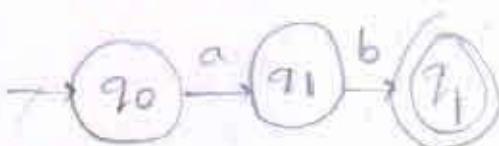
Regular set

Regular expression



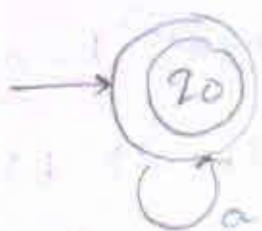
$\{a, b\}$

$a+b$



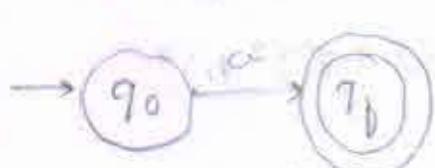
$\{ab\}$

ab



$\{1, a, aa, aaa \dots\}$

a^*



$\{a, aa, aaa \dots\}$

a^+

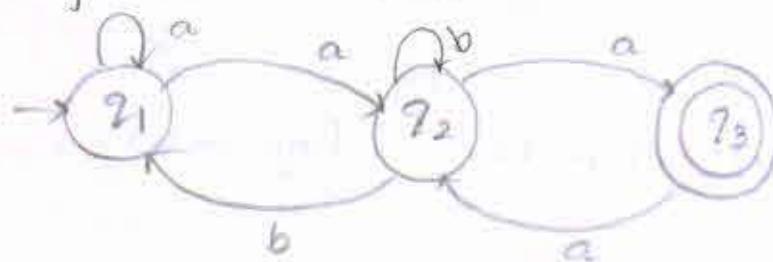


$\{1, aa, aaa \dots\} (aa)^*$



ALGEBRAIC METHOD USING ARDEN'S THEOREM

Consider the transition system and find out its equivalent regular expression using Arden's theorem.



$$q_1 = q_1 a + q_2 b + \lambda$$

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

$$q_3 = q_2 a$$

$$q_1 = q_1 a + q_2 b + q_2 aa$$

$$q_2 = q_1 a + q_2 (b + a^2)$$

$$R = Q + RP$$

$$q_2 = q_1 a (b + a^2)^* = (Q(P))^*$$

$$q_1 = q_1 a + q_1 a (b + a^2)^* b + \lambda$$

$$q_1 = \lambda + q_1 (a + a (b + a^2)^* b)$$

$$q_1 = \lambda (a + a (b + a^2)^* b)^*$$

$$q_1 = (a + a (b + a^2)^* b)^*$$

$$q_3 = q_2 a$$

$$q_3 = q_1 a (b + a^2)^*$$

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

The following method is an extension of arden's theorem. This is used to find the expression recognised by a transition system. The following assumptions are made regarding transition system.

- (1) The transition graph does not have λ moves.
- (2) It has only one initial state.
- (3) Let its vertices are v_1, v_2, \dots, v_n .
- (4) v_i is the regular expression representing the set of strings accepted by the system even though v_i is a final state.
- (5) α_{ij} denotes the regular expression representing the set of labels of edges from v_i to v_j when there is no such edge $\alpha_{ij} = \emptyset$. Consequently, we get a following set of equations in v_1 to v_n

$$v_1 = v_1\alpha_{11} + v_2\alpha_{21} + \dots + v_n\alpha_{n1} + \lambda$$

$$v_2 = v_1\alpha_{12} + v_2\alpha_{22} + \dots + v_n\alpha_{n2}$$

$$\vdots$$

$$v_n = v_1\alpha_{1n} + v_2\alpha_{2n} + \dots + v_n\alpha_{nn}$$

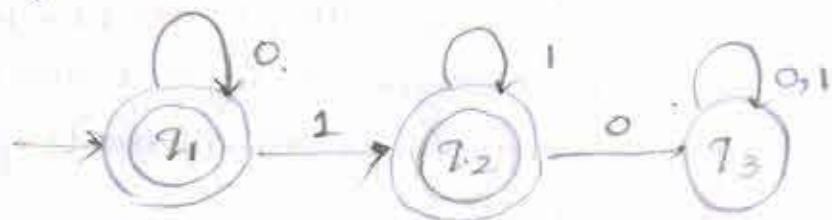
By repeatedly applying substitutions and arden's theorem we can express v_i in terms of α_{ij} 's (inputs).

For getting the set of strings recognised by the transition systems we have to take the union of all v_i 's corresponding to final states.

$$P = Q^P \cap P$$

$$P = QP^*$$

Describe in English (statements) the set accepted by finite automata whose transition diagram is,



$$q_1 = q_1 0 + A$$

$$P = QP$$

$$q_2 = q_2 1 + q_1 1$$

$$R = Q + RP$$

$$= Q(P)^*$$

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

$$= q_2 0 + q_3 (0+1)$$

$$q_1 = \lambda + q_1 0$$

$$q_1 = \lambda 0^*$$

$$q_1 = 0^* \checkmark$$

$$q_2 = q_2 1 + q_1 1$$

$$P = QP \frac{Q}{Q} = Q(P)^*$$

$$q_2 = q_1 11^* \text{ by cardenstheom}$$

$$q_2 = 0^* 11^*$$

$$q_2 = 0^* 1^*$$

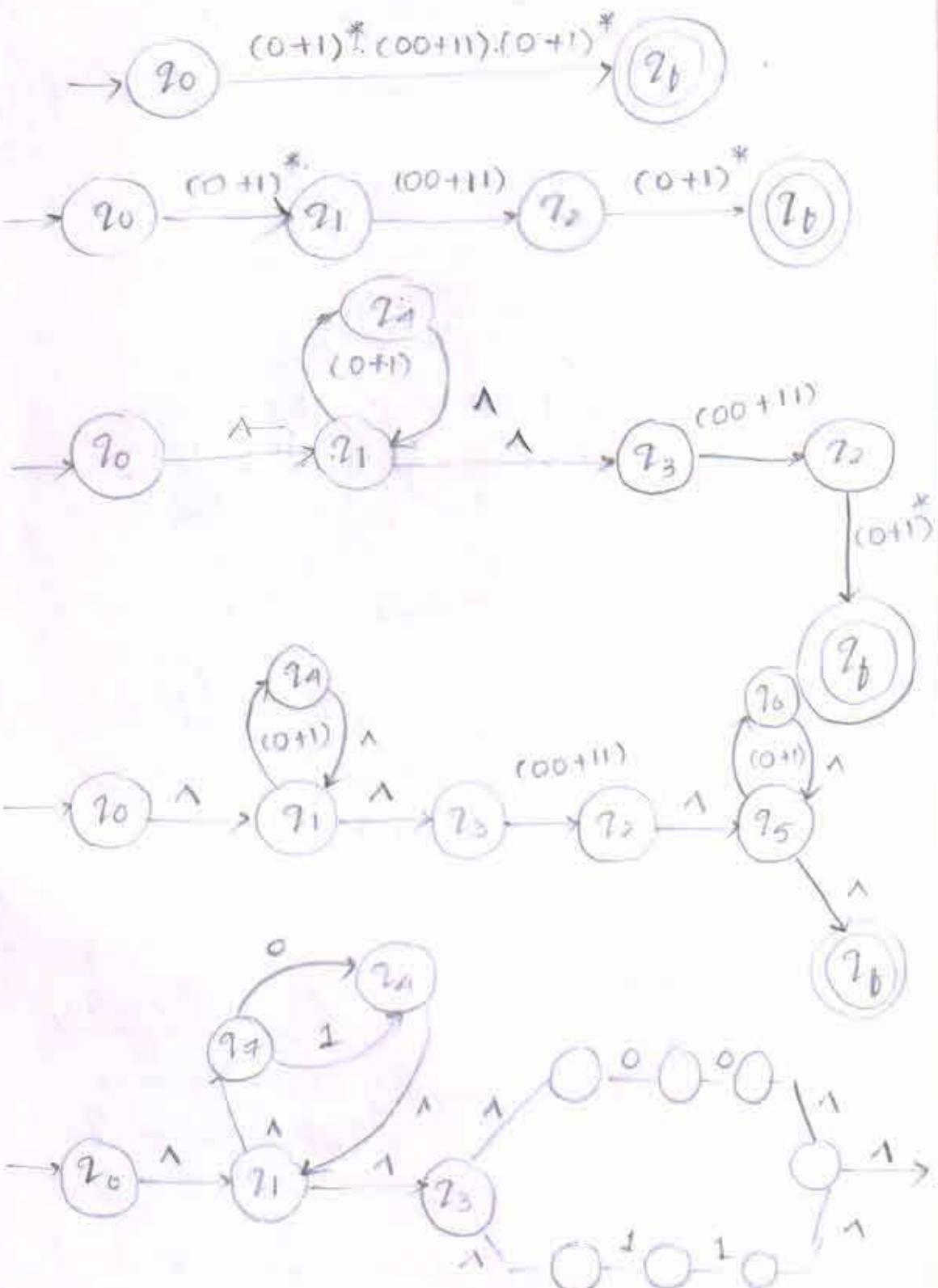
$$q_1 + q_2 = 0^* + 0^* 11^* 0$$

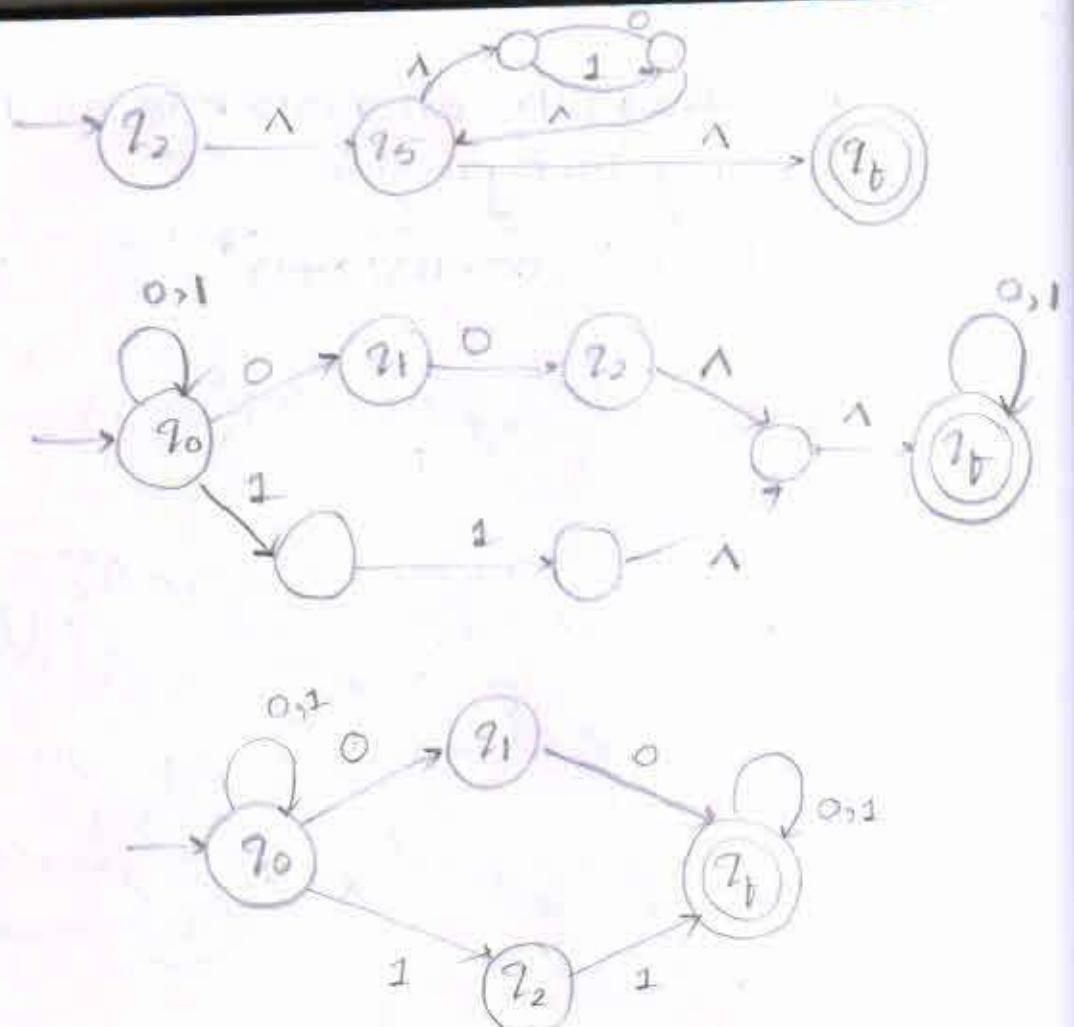
$$= 0^* (\lambda + 1^*)$$

$$= 0^* 1^*$$

Construct the finite automata equivalent to the regular expression

$$\underline{(0+1)^*(00+11)(0+1)^*}$$





NFA

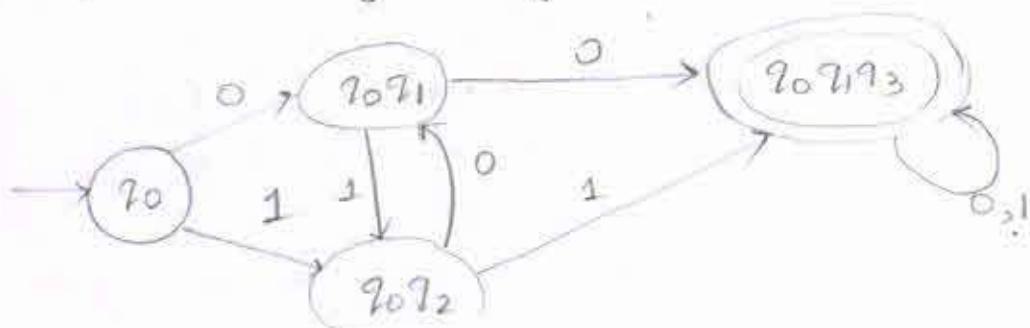
NFA table

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

NFA to DFA table

	0	1
$\rightarrow q_0$	$[q_0 q_1]$	$[q_0 q_2]$
$[q_0 q_1]$	$[q_0 q_1 q_3]$	$[q_0 q_2]$
$[q_0 q_2]$	$[q_0 q_1]$	$[q_0 q_2 q_3] \cup [q_0 q_1 q_3]$
$[q_0 q_1 q_3]$	$[q_0 q_1 q_3]$	$[q_0 q_2 q_3] \cup [q_0 q_1 q_3]$
$[q_0 q_2 q_3]$	$[q_0 q_1 q_3]$	$[q_0 q_2 q_3] \times$

Here $[q_0 q_1 q_3]$ and $[q_0 q_2 q_3]$ are final states and both have identical iotas so, we can neglect any one final state.



Languages

Type 0

Context-sensitive (Type 2)

Context-free (Type 2)

Regular (Type-3)

Automata

TM

LBA

PDA

FA

where

TM - Turing machine

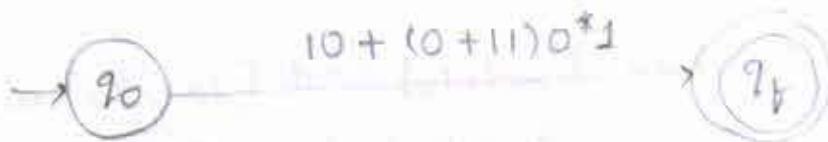
LBA - linear bounded automaton

PDA - push down automaton

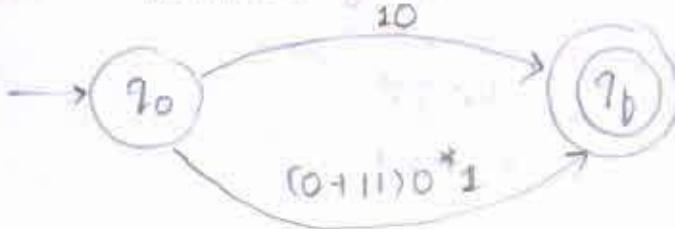
FA - Finite automata

Construct DFA with reduced states equivalent to the regular expression

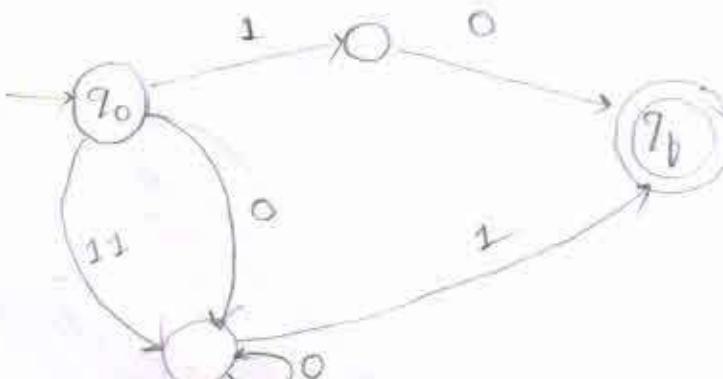
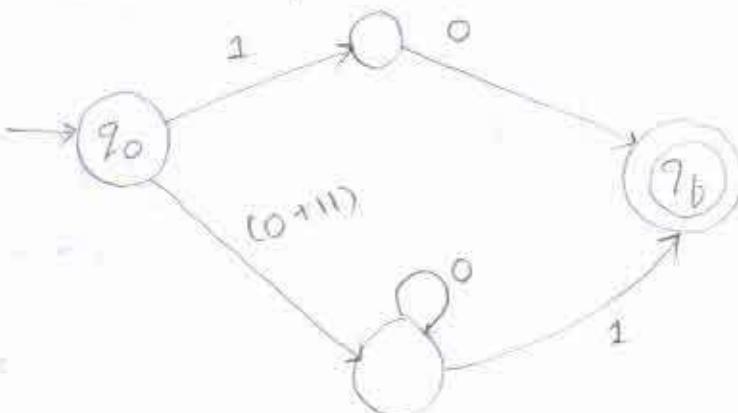
$$10 + (0 + \emptyset) 11^* 0^* 1$$

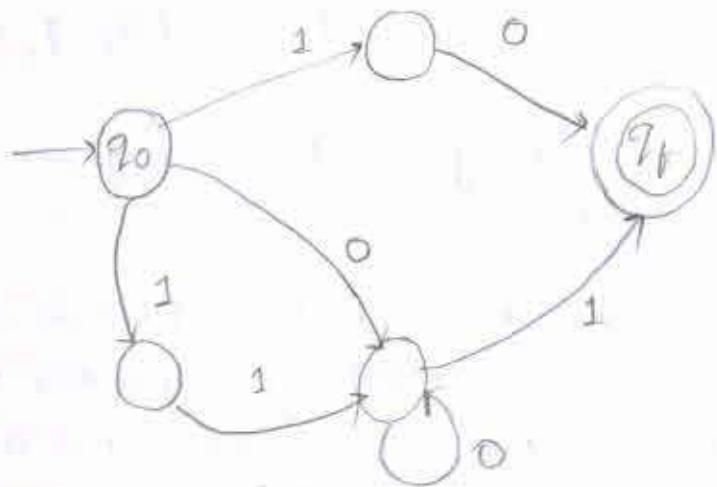


elimination of union



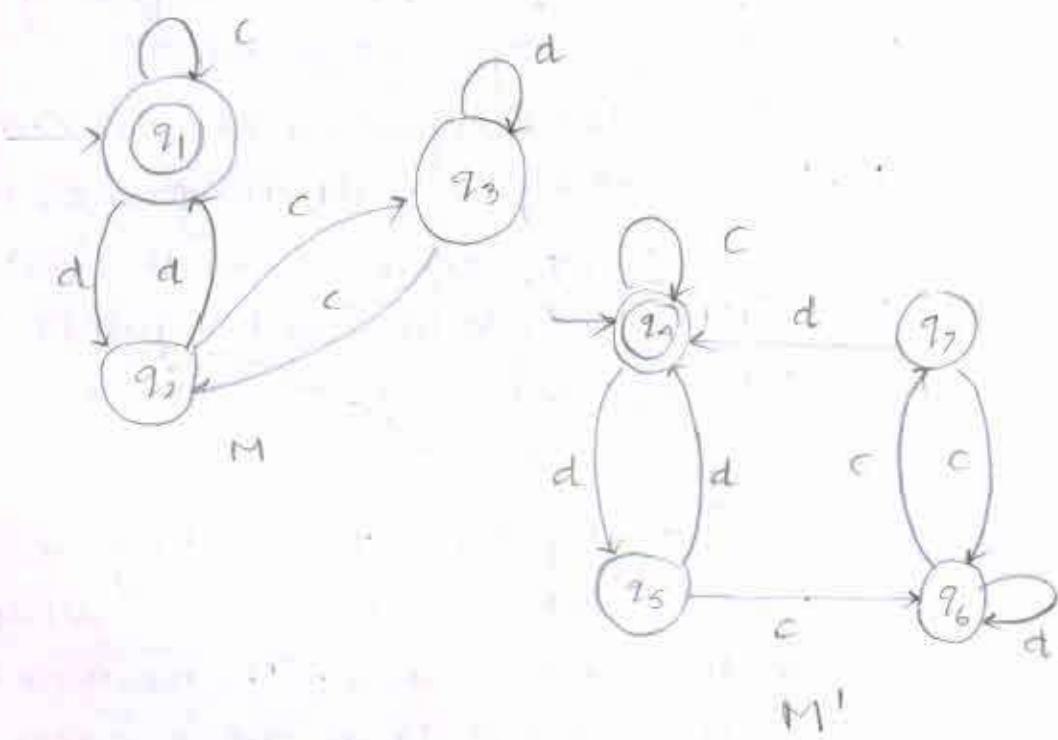
elimination of concatenation





COMPARISON METHOD OF DFA'S

Consider the following 2' DFA's M & M' over $\{0, 1\}$ and determine whether M & M' are equivalent.



(q_1, q'_1)	(q_2, q'_1)
(q_1, q'_4)	(q_2, q'_4)
(q_2, q'_5)	(q_3, q'_6)
(q_3, q'_6)	(q_1, q'_4)
(q_2, q'_7)	(q_3, q'_6)

(q_2, q_7) (q_3, q_6) (q_1, q_4)

The given two automachines are equivalent.

Let M and M' be Σ finite automata over Σ . We construct a comparison table consisting of $n+1$ columns where n is the number of input symbols. The first column consists of pairs of vertices of the form (q, q') where $q \in M$, $q' \in M'$. If (q, q') appears in some row of the first column then the corresponding entry in the a column ($a \in \Sigma$) is (q_a, q'_a) where q_a and q'_a are reachable from q and q' respectively on application of a .

The comparison table is constructed by starting with the pair of initial vertices (q_{in}, q'_{in}) of M & M' in the first column. The first elements in a subsequent column are (q_a, q'_a) where q_a & q'_a are reachable by a -paths from q_{in} & q'_{in} . We repeat the construction by considering the pairs in the second and subsequent columns which are not in the first column. The row wise construction is repeated. There are 2^n cases.

Case 1:

If we reach a pair (q, q') such that q is the final state of M & q' is the non-final state of M' or vice versa we terminate the construction and conclude that M & M' are not equivalent.

case 2:

Here the construction is terminated when no new element appears in the second & the subsequent columns which are not in the 1st column i.e., when all the elements in the 2nd and subsequent columns appear in the 1st column. In this case we conclude that $M \& M'$ are equivalent.

CLOSURE PROPERTIES OF REGULAR SETS:

→ If 'L' is a regular then L^T is also regular.

→ If 'L' is a regular set over Σ , then $\Sigma^* - L$ is also regular over Σ .

Here $M = (Q, \Sigma, \delta, q_0, F)$ accepting 'L'.

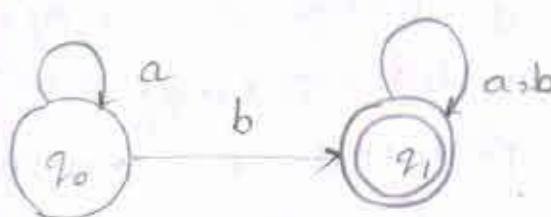
→ we construct another DFA $M' = (Q, \Sigma, \delta', q_0, F')$

by defining $F' = Q - F$ i.e., $M \& M'$ differ only in their final states.

→ If x and y are regular sets over Σ then $x \cap y$ is also regular over Σ .

→ If L and M are regular languages then $L - M$ is also regular language.

CONVERSION OF TRANSITION SYSTEM TO



GIRANIMAR :

$q_i \rightarrow aq_j \text{ if } \delta(q_i, a) = q_j \text{ with } q_j \notin F$

$q_i \rightarrow aq_j, q_i \rightarrow a^{-1}q_j \text{ if } \delta(q_i, a) \in F$

$$q_0 \rightarrow aq_0$$

$$q_0 \rightarrow bq_1$$

$$q_1 \rightarrow aq_1$$

$$q_1 \rightarrow bq_1$$

$$q_0 \rightarrow b$$

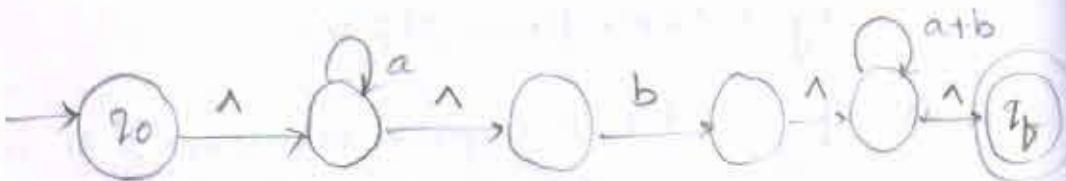
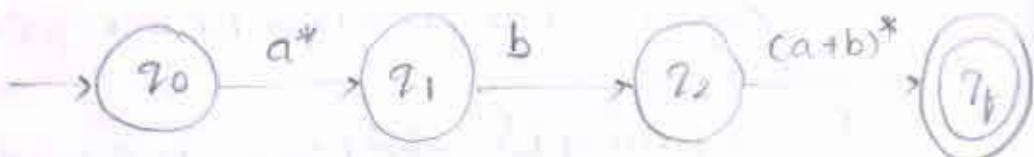
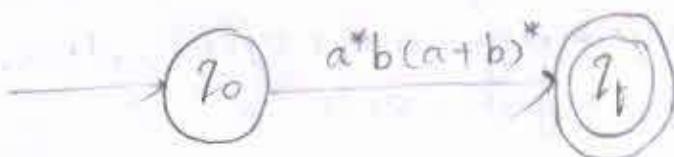
$$q_1 \rightarrow a$$

$$q_1 \rightarrow b$$

construct a regular grammar given a set represented by $a^*b(a+b)^*$

$$M = \{ Q, \Sigma, \delta, q_0, F \}$$

$$G_1 = \{ V_N, \Sigma, P, S \}$$



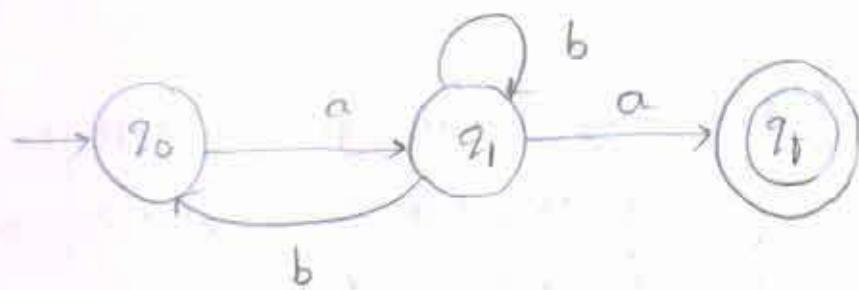
CONSTRUCTION OF TRANSITION SYSTEM FOR A GIVEN REGULAR GRAMMAR

Each production $A_i \rightarrow a A_j$ induces a transition from q_i to q_j with label 'a'

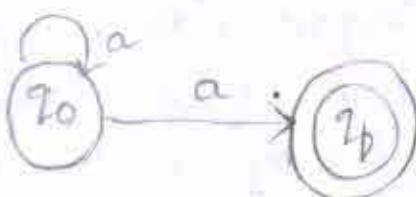
each production $A_k \rightarrow a$ induces a transition from q_k to q_f with label 'a'.

let $G_1 = (\{V_P, A_0, A_1\}, \{a, b\}, P, A_0)$ where P consists of $A_0 \rightarrow aA_1, A_1 \rightarrow bA_1, A_1 \rightarrow a, A_1 \rightarrow bA_0$
 construct a transition system accepting this grammar

$$M = \{ \{q_0, q_1, q_f\}, \{a, b\}, \delta, q_0, \{q_f\} \}$$

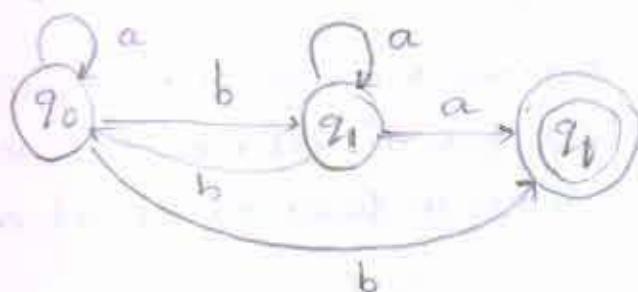


$s \rightarrow as/a$



$s \rightarrow as/bA/b$

$A \rightarrow aA/bA/a$



PUMPING LEMMA

(2)

Assume L is regular. Let n be the number of states in the corresponding finite automata.

(2)

Choose a string w such that $|w| \geq n$.
use pumping lemma to write $|w| = xyz$
then, $|xy| \leq n$ and $|y| > 0$.

(3) Find a suitable integer i such that

$xy^iz \notin L$ thus contradicts our assumption
hence L is not regular.

NOTE:

The crucial part of the procedure is to find i such that $xy^iz \notin L$. In some cases we prove $xy^iz \notin L$ by considering the length of $|xy^iz|$. In some cases we may have to use the structure of strings in L .

Show that the set $L = \{a^{i^2} \mid i \geq 1\}$ is not regular.

(1) Suppose L is regular and we get a contradiction. Let n be the no. of states in finite automata accepting L .

$$(2) \quad w = a^{n^2}$$

$$|w| = |a^{n^2}| = n^2 \geq n.$$

Let $w = xyz$

$|xy| \leq n$ and $|y| > 0$

(5)

$$w = xyz$$

$$xyz = xy^2z$$

$$|xyz| = |xyz| + |y|$$

$$= n^2 + |y|$$

$$> n^2 \text{ --- (1)}$$

- from step
from (2), $|xy| \leq n$

$$|y| \leq n$$

$$|xyz| = |xyz| + |y| = n^2 + |y| \leq n^2 + n$$

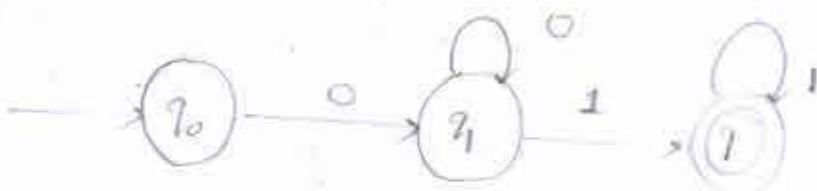
$$< n^2 + n + n + 1$$

$$< (n+1)^2 \text{ --- (2)}$$

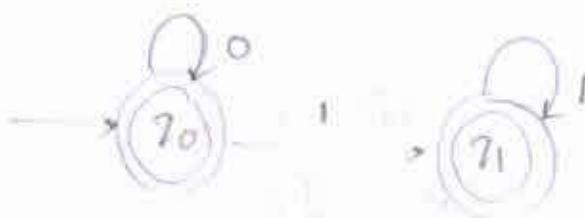
- from (1) & (2),

$n^2 < |xyz| < (n+1)^2$ Thus a contradiction so, it is not a regular grammar

$0^m 1^n$ where $m, n \geq 1$



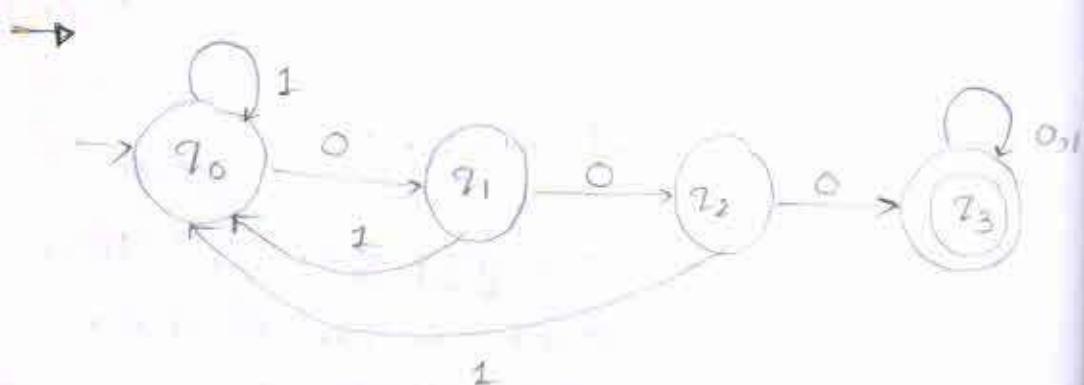
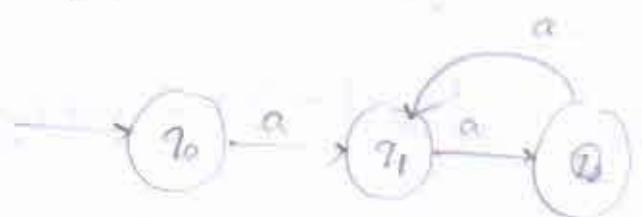
$0^m 1^n$ where $m, n \geq 0$



a^{2n} , $n \geq 0$



a^n , $n \geq 1$



Convert into regular grammar.

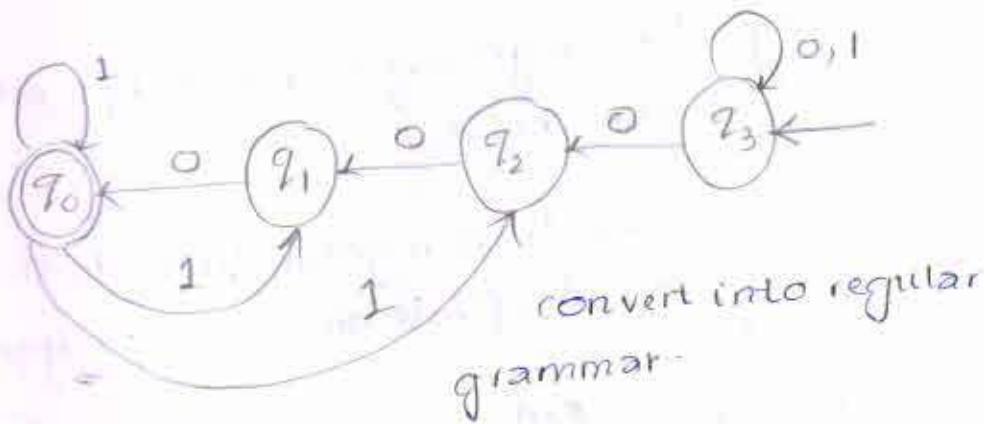
$$q_0 \rightarrow 0q_1 / 1q_0$$

$$q_1 \rightarrow 0q_2 / 1q_0$$

$$q_2 \rightarrow 0q_3 / 1q_0 / 0$$

$$q_3 \rightarrow 0q_3 / 1q_3 / 0 / 1$$

Right linear grammar



$$q_0 \rightarrow 1q_0 / 1q_1 / 1q_2 / 1$$

$$q_1 \rightarrow 0q_0 / 0$$

$$q_2 \rightarrow 0q_1$$

$$q_3 \rightarrow 0q_2 / 0q_3 / 1q_3$$

left linear grammar



$$q_0 \rightarrow q_0^2 / q_1^1 / q_2^1 / 1$$

$$q_1 \rightarrow q_0 \ 0 / 0$$

$$q_2 \rightarrow q_1 0$$

$$q_3 \rightarrow q_2 0 / q_3 0 / q_3 1$$

Find the regular expression for the following

$$a^m b^n c^p, \ m, n, p \geq 1$$



$$aa^*bb^*cc^* \rightarrow a^+b^+c^+$$

$a^m b^n c^p \quad m, n, p \geq 1$

$aa^* bb(bb)^* ccc(ccc)^*$

$a^n b a^{2m} b^2 \quad m \geq 0, n \geq 1$

$aa^* b (aa)^* b^2$

Verify a^p is a regular grammar or not while p is a prime number.

(1) Assume 'L' is regular let 'n' be the number of states in finite automata accepting L.

(2) $w = a^{pn}$

Let, $m > n$

$|w| > n$

$w = xyz, |xy| \leq n \text{ and } |y| > 0$

(3) $w = xyz$

Let $i = m+1$

$$\begin{aligned}|ay^i z| &= |ayz| + |y^{i-1}| \\ &= m + (i-1)|y|\end{aligned}$$

$$\begin{aligned}|ay^i z| &= m + m|y| \\ &= m(1 + |y|) \text{ not prime}\end{aligned}$$

def:

If q_i, q_j are said to be κ -equivalence & $\kappa \geq 0$
then q_i, q_j are said to be equivalence.

→ If q_i, q_j are said to be κ -equivalence for
some κ then q_i, q_j are said to be $(\kappa-1)$
equivalence.

$$\Psi \underline{A} \underline{\beta} \longrightarrow \Psi \underline{\alpha} \underline{\beta}$$

↓
left context right context

Grammar is basically a - tuple

$$G_1 = (V_N, \Sigma, P, S)$$

V_N — it is finite set collection of
variables or non-terminals

Σ — it is finite set collection of terminals
— here,

$$V_N \cap \Sigma = \emptyset$$

S — starting symbol where $S \in V_N$

$$S \longrightarrow <\text{Noun}> <\text{verb}>$$

$$S \longrightarrow <\text{Noun}> <\text{verb}> <\text{adverb}>$$

$$\text{Noun} \rightarrow P_{\text{Noun}}, R_{\text{Noun}}$$

$$\text{verb} \rightarrow V_{\text{verb}}, A_{\text{verb}}$$

adverb → quickly, slowly

Chomsky → classification of languages

3. FORMAL LANGUAGES :

P — collection of productions in
the form of $\alpha \rightarrow \beta$ where

$$\alpha, \beta \in V_N^* \cup \Sigma^*$$

eg, $0^n 1^n, n \geq 0$

$s \rightarrow os_1 / \lambda$

os_1

$00s_11$

$00^2 s_11^2$

$0^3 s_13$

Replacing s by λ we get, $0^3 1^3$.

eg, In a production of the form $\phi A \psi \rightarrow \phi \alpha \psi$

where A is a variable (non-terminal) ϕ is called

the left context, ψ is called the right context, and $\phi \alpha \psi$ is the replacement string.

$\rightarrow abAbed \rightarrow ab\lambda b ed$

$ab \rightarrow \text{left context}$

$bcd \rightarrow \text{right context}$

here $\alpha = AB$.

$\rightarrow A \underline{C} \Delta, A \underline{\Delta} \underline{\Delta}$

left context is A

right context is Δ

and $\alpha = \lambda$

$\rightarrow c \rightarrow \lambda$ (here ' c ' is erased)

left context & right context is ' λ '
here $\alpha = \lambda$

type 0 → A production without any restrictions
type 1
type 2
type 3

Type - 1

A production of the form $\phi A \psi \rightarrow \phi \alpha \psi$ is called a type-1 production if $\alpha \neq \lambda$ (i.e., $\alpha \neq \lambda$) In type-1 production erasing of λ is not permitted.

eg,

(1) $a \underline{A} bcd \rightarrow a \underline{bcd} bcd$

type 2 grammar

(2) $\overset{L}{\overbrace{AB}} \rightarrow \overset{L}{\overbrace{AbBc}}$
 $\text{left} \quad \text{left}$

(3) $\underline{A} \rightarrow \underline{abA}$

→ A grammar is called type-1 (or) context sensitive (or) context dependent if all its productions are type-1 productions.

The production $s \rightarrow \lambda$ is allowed in type-1 grammar but in this case ' s ' does not appear on the righthand side of any production.

The language generated by type-1 grammar is called type-1 or context sensitive language.

In a context sensitive grammar 'S' we allows $\rightarrow \lambda$ apart from $s \rightarrow \lambda$ all the other productions don't derive λ (2)

length of the working string.

TYPE - 2

It is the production of the form $A \rightarrow \alpha$ where $\alpha \in V_N^*$, $\alpha \in (V_N \cup \Sigma)^*$. In other words the LHS has no right and left context.

e.g.,

$$A \rightarrow a, B \rightarrow abc, A \rightarrow aa$$

A grammar called type-2 grammar if it contains only type-2 production. It is also called a context-free grammar.

A language generated by context-free grammar is called a type-2 language or context-free language.

TYPE 3:

A production of the form $A \rightarrow \alpha$ (or) $A \rightarrow \alpha B$ where $A, B \in V_N$ and $\alpha \in \Sigma^*$ is called a type 3 production.

A grammar is called a type-3 (or)

regular grammar if all its productions are type 3 productions.

A production $s \rightarrow \lambda$ is allowed in type-3 grammar but in this case ' s ' does not appear on the RHS of any production.

Find the highest type of numbers which can be applied to following grammars

(a) $S \rightarrow Aa \rightarrow \text{type 2}$

$$\begin{array}{l} A \rightarrow c \mid Ba \\ \downarrow \\ \text{type 3} \end{array} \quad \begin{array}{l} A \rightarrow c \mid Ba \\ \downarrow \\ \text{type 2} \end{array}$$

$$B \rightarrow abc \rightarrow \text{type 2}$$

(b) $S \rightarrow ASB \mid d$

$$\begin{array}{c} \downarrow \\ \text{type 2} \end{array} \quad \begin{array}{c} \downarrow \\ \text{type 3} \end{array}$$

$$A \rightarrow aA$$

$$\begin{array}{c} \downarrow \\ \text{type 3} \end{array}$$

(c) $S \rightarrow aS \mid ab$

$$\begin{array}{c} \downarrow \\ \text{type 3} \end{array} \quad \begin{array}{c} \downarrow \\ \text{type 2} \end{array}$$

REGULAR EXPRESSION & REGULAR GRAMMAR

Any-terminal element of Σ is regular expression.

for eg, $a \in \Sigma$, \emptyset is also regular expression. Σ is regular expression.

UNION

Union of \geq regular expressions R_1 and R_2 is a regular expression R' ($R = R_1 + R_2$)

Let a' be regular expression in R_1

b' be regular expression in R_2 then

($a+b$) is also a regular expression R' having the elements of a, b by

CONCATENATION:

Concatenation of \geq regular expressions R_1 and R_2 written as R_1R_2 is also a regular expression R ($R = R_1R_2$)

Let a' be regular expression in R_1

b' be regular expression in R_2

(ab) is also a regular expression R' having the elements of ab

ITERATION (CLOSURE):

Iteration closure of a regular expression R' is written as R^* is also a regular expression.

Let a be a regular expression then

$\lambda, a, aa, aaa \dots$ are also regular expressions

NOTE:

If L is a language represented by the regular expression R then the Kleen's closure of L is denoted as L^* and is given as

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

The positive closure of L^* is denoted as L^+

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

If R is a regular expression then $(R)^*$ is also a regular expression

Regular expression over Σ is precisely those obtain recursively by the application of the above rules once or several times.

Closure has highest Precedence next highest is for concatenation and last is for union

IDENTIFIER NOTATION:

$$\text{notation: } L(x/y/z)^*$$

REGULAR SET

Any set represented by a regular expression is called a regular set

If a, b are the elements of Σ then the regular expressions a denote the set $\{a\}$

$a+b$ denote the set $\{a, b\}$

ab denote the set $\{ab\}$

a^* denote the set of $\lambda, a, aa, aaa, \dots$

$(ab)^*$ denote the set

$\{ \lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, \dots \}$

Regular set Regular expression

of 1013

§ n, a §

At a

④

$\{ \epsilon, ab, ba, bb, ba \dots \} \quad (a+b)^*$

$\{ ab, ba \} \quad ab + ba$

Describe the following set by regular expression
all string with 0's and 1's - $(0+1)^*$
set of all strings of 0's & 1's ending with 00
 $\rightarrow (0+1)^*00$

Set of all strings 0's & 1's begin with 0 and
end with 1

$0(0+1)^*1$

Set of all strings having even number of 1's

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

Set of all string having odd number of 1's

$1(11)^* \text{ or } (11)^*1$

Strings of 0's and 1's with at least 2' consecutive zeros

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

Austings of 0's and 1's beginning with
1 or 0 and not having 2' consecutive zeros

$(1+01)^*$

Set of strings in which zero immediate
tely followed by at least 2 1's

$(1+01)^*$

Set of all strings ending with 011

$(0+1)^*011$

Identities for regular expressions

$$I_1 \quad \phi + R = R$$

$$I_2 \quad \phi R = R\phi = \phi$$

$$I_3 \quad RR = R\Lambda = R$$

$$I_4 \quad \Lambda^* = \Lambda \quad \& \quad Q^* = \Lambda$$

$$I_5 \quad R + R = R$$

$$I_6 \quad R^* R^* = R^*$$

$$I_7 \quad RR^* = R^*R$$

$$I_8 \quad (R^*)^* = R^*$$

$$I_9 \quad \Lambda + RR^* = \Lambda + R^*R = R^*$$

$$I_{10} \quad (PQ)^* = P(QP)^*$$

$$I_{11} \quad (P+Q)^* = (P^*Q^*)^* = (P^*Q^*)^*$$

$$I_{12} \quad (P+Q)R = PR + QR \quad \& \quad R(P+Q) = RP + RQ$$

\rightarrow 2' regular expressions P & Q are equivalent
if P & Q represent the same set of strings

REDEIN'S THEOREM:

Let P and Q be 2' regular expressions
over Σ if P does not contain nulla then
the following equation in $R = Q + RP$ has a
unique solution given by $R = QP^*$

PROOF:

$$\text{Case 1: } R = Q + RP \\ R = Q + (Q + RP)^*P \\ = Q(\Lambda + PP^*)$$

$$R = QP^* \quad (\text{from } T_1)$$

case (ii)

$$R = Q + RP$$

$$\begin{aligned} R &= Q + (Q + RP)P \\ &= Q + QP + RP^2 \\ &= Q + QP + (Q + RP)P^2 \\ &= Q + QP + QP^2 + RP^3 \end{aligned}$$

$$\begin{aligned} LHS &= (1^*(011)^*)^* \\ &= ((1+011)^*)^* \quad \text{by } T_1 \\ &= R : H S \end{aligned}$$

ALGEBRA LAW FOR REGULAR EXPRESSIONS:

- union operation on regular expressions are commutative i.e., $R+S = S+R$.
- There are associative i.e., $(R+S)+T = R+(S+T)$

→ concatenation operation on regular expression are associative

$$(st) = (sts)t$$

$$\begin{aligned} &= Q(C^* + P + P^2 + \dots + P^i + \dots + QP^i) \\ &= Q(C^* + P + P^2 + \dots + P^i)^+ \\ &= Q(P^*)^+ + RP^{i+1} \\ &= Q(P^*)^+ \end{aligned}$$

→ concatenation right distributive over addition union & left distributive over union

$$(R+S)T = RT+ST$$

$$T(R+S) = TR+TS$$

Given a regular expression represent the set 'L' of strings in which

Prove that the regular expression $R = \Lambda +$

$$\begin{aligned} &1^*(011)^* (1^*(011)^*)^* \\ &= (1+011)^* \end{aligned}$$

FINITE AUTOMATA AND REGULAR EXPRESSIONS:
TRANSITION SYSTEM AND REGULAR EXPRESSIONS:

Every regular expression R can be recognised by a transition system i.e., for every string w , in the set R , there exist a path from \emptyset

The initial state - final state with the path value
'w'

TRANSITION SYSTEM CONTAINING 'NULL' MOVES.

Suppose we want to replace a λ -move from vertex v_1 to vertex v_2 . Then we proceed as follows

Step 1 :

Find all edges starting from v_2

Step 2

Duplicate all these edges starting from v_1 ,
without changing the edge labels

Step 3:

If v_1 is the initial state, make v_2 also as

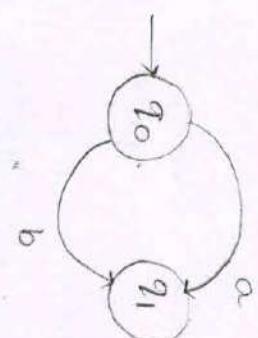
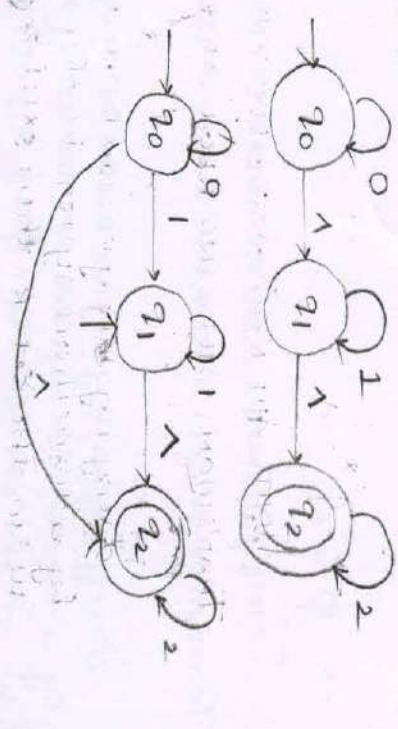
initial state

Step 4:

If v_2 is a final state make v_1 as the final state

e.g,

Consider a finite automata with null moves &
obtain a equivalent automata without null moves



Finite automata

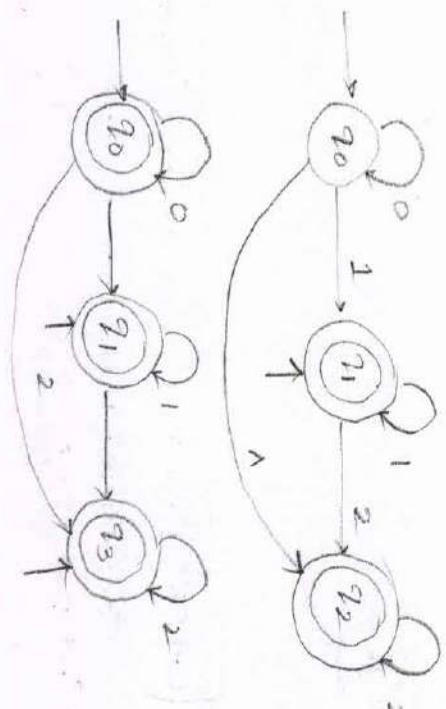
Regular set

Regular expression

$a+b$

$\{a, b\}$

$a+b$



$\{ab, bab\}$

$a+b$

$a+b$

$a+b$

$\{a, aa, aaa, \dots\}$

a^*

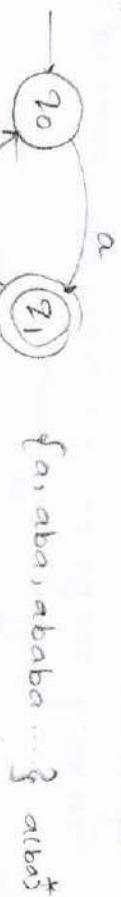
$\{a, aa, aaa, \dots\}$

a^*

$\{a, aa, aaa, \dots\}$

$(aa)^*$

7



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

b.

ALGEBRAIC METHOD USING ARDEN'S THEOREM

Consider the transition system and find out its equivalent regular expression using Arden's theorem.



$$\begin{aligned}q_1 &= q_1a + q_2b + \lambda \\q_2 &= q_1a + q_2b + q_3a \\q_3 &= q_2a\end{aligned}$$

$$q_2 = q_1a + q_2b + q_2aa$$

$$q_2 = q_1a + q_2(b+a^2)$$

$$q_2 = Q + RP$$

$$q_2 = q_1a(b+a^2)^* = Q(P)^*$$

$$q_1 = q_1a + q_2a(b+a^2)^*b + \lambda$$

$$q_1 = \lambda + q_1(a + a(b+a^2)^*b)$$

$$q_1 = a(a + a(b+a^2)^*b)^*$$

$$q_3 = q_2a$$

$$q_3 = q_1a(b+a^2)^*$$

The following method is an extension of Arden's theorem. This is used to find the expression recognised by a transition system.

The following assumptions are made regarding transition system

- (1) The transition graph does not have λ moves.
- (2) It has only one initial state.
- (3) Let its vertices are v_1, v_2, \dots, v_n

(4) v_i^* is the regular expression representing the set of strings accepted by the system even though v_i is a final state

(5) If α_{ij} denotes the regular expression representing the set of labels of edges from v_i to v_j when there is no such edge $\alpha_{ij}^* = \emptyset$ consequently, we get a following set of equations in v_i to v_n

$$v_1 = v_1\alpha_{11} + v_2\alpha_{21} + \dots + v_n\alpha_{n1} + \lambda$$

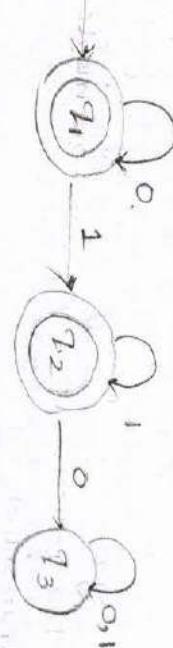
$$v_2 = v_1\alpha_{12} + v_2\alpha_{22} + \dots + v_n\alpha_{n2}$$

$$v_n = v_1\alpha_{1n} + v_2\alpha_{2n} + \dots + v_n\alpha_{nn}$$

By repeatedly applying substitution and Arden's theorem we can express v_i^* in terms of α_{ij} 's (inputs).

For getting the set of strings recognised by the transition systems we have to take the union of all v_i^* 's corresponding to final states (8)

Describe in English (statements) the set accepted by finite automata whose transition diagram is,



$$q_1 = q_1^0 + \lambda$$

$$q_2 = \frac{q_1^0}{R} + q_1\lambda$$

$$q_3 =$$

$$= q_2^0 + q_3^0 + q_3\lambda$$

$$= q_2^0 + q_3(\lambda + \lambda)$$

$$q_1 = \lambda + q_1\lambda$$

$$q_1 = \lambda\lambda^*$$

$$q_1 = \lambda^*$$

$$q_2 = q_2^0 + q_2\lambda$$

$$q_2 = \frac{q_1^0}{R} + q_1\lambda$$

by Arden's theorem

$$q_2 = \lambda^* q_1^*$$

$$\text{and } q_3 =$$

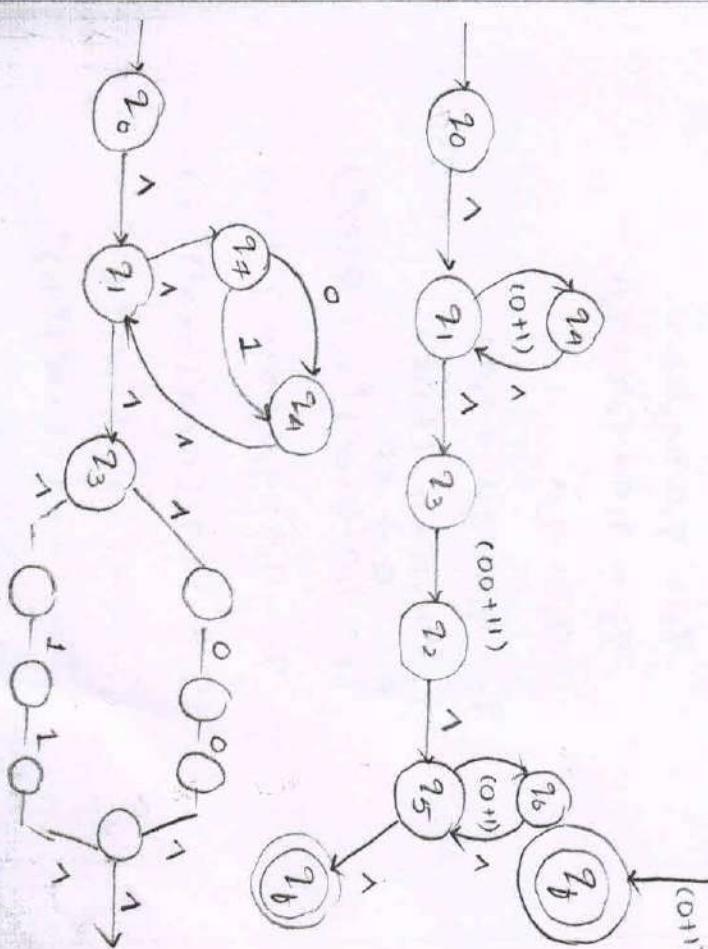
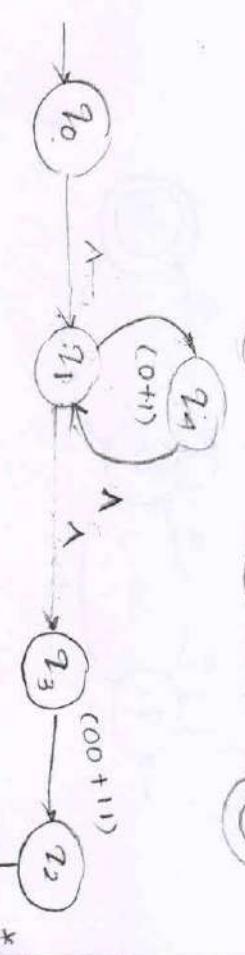
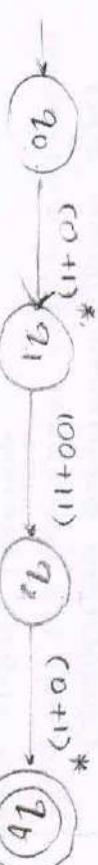
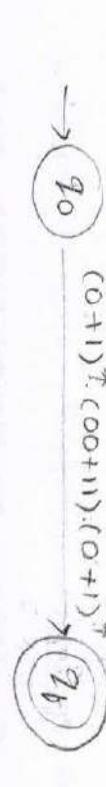
$$q_1 + q_2 = \lambda^* + \lambda^*\lambda^*$$

$$= \lambda^* (\lambda + \lambda^*)$$

So L(A) = $\lambda^* (\lambda + \lambda^*)$
or L(A) = $\lambda^* \cup \lambda^*\lambda^*$
or L(A) = $\lambda^* \cup \lambda^* \cup \lambda^*\lambda^*$
or L(A) = $\lambda^* \cup \lambda^*\lambda^* \cup \lambda^*\lambda^*\lambda^*$

Construct the finite automata equivalent to the regular expression

$$(0+1)^* (00+11)(0+1)^*$$



where

TM - Turing machine

LBA - linear bounded automaton

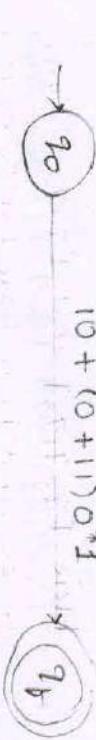
PDA - push down automaton

FA - finite automata

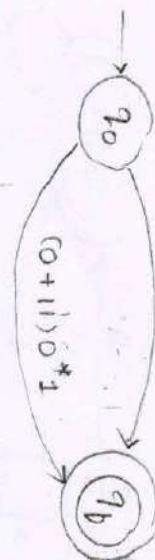
Construct DFA with reduced states equivalent

to the regular expression

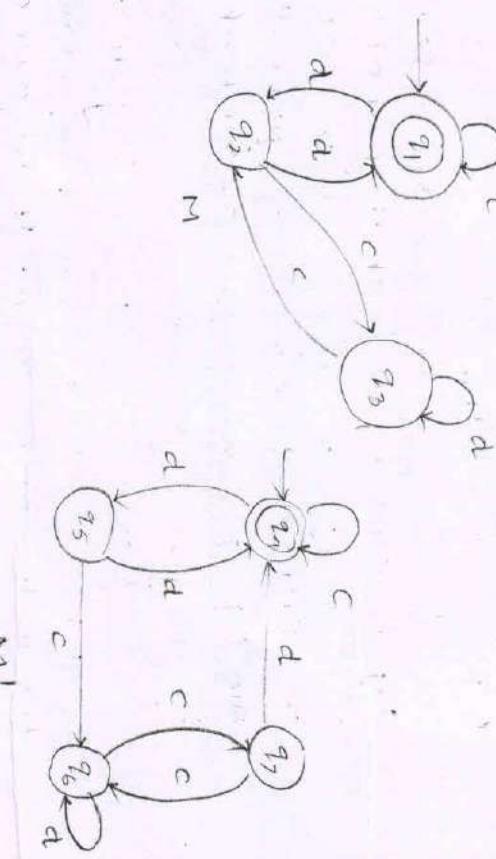
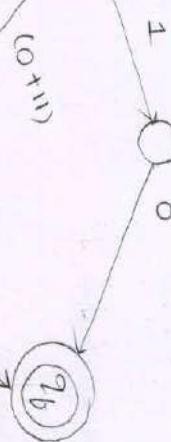
$$10 + (0 + \emptyset_{11}) 0^* 1$$



elimination of union

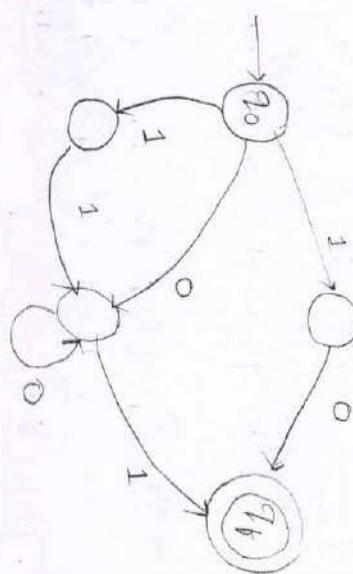


elimination of concatenation



COMPARISON METHOD OF DFAs:

Consider the following 2' DFAs M & M' over $\{0, 1\}$ and determine whether M & M' are equivalent



(q_1, q_4)	(q_2, q_5)
(q_1, q_4)	(q_2, q_5)
(q_1, q_4)	(q_2, q_5)
(q_3, q_6)	(q_3, q_6)

(q_2, q_7)

(q_3, q_6)

(q_1, q_4)

The given two automata's are equivalent.

Let M and M' be Σ' finite automata over Σ . we construct a comparison table consisting of $n+1$ columns where n is the number of input symbols. The first column consists of pairs of vertices of the form $\{q_0, q'\}$ where $q \in M$, $q' \in M'$. If (q, q') appears in some row of the first column then the corresponding entry in the a column ($a \in \Sigma$) is (q_a, q'_a) where q_a and q'_a are reachable from q and q' respectively on application of a .

The comparison table is constructed by the starting or with the pair of initial vertices (q_{in}, q'_{in}) of M & M' in the first column. The first elements in a subsequent column are (q_a, q'_a) where q_a & q'_a are reachable by a paths from q_{in} & q'_{in} we repeat the construction by considering the pairs in the second and subsequent columns which are not in the first column. The row wise construction is repeated there are 2 cases

case 1:

If reach a pair (q_a, q'_a) such that q is the final state of M & q'_a is the non-final state of M' or vice versa we terminate the construction and conclude that $M \neq M'$ are not equivalent

case 2:

Here the construction is terminated when no new element appears in the second & the subsequent columns which are not in the 1st column i.e., when all the elements in the 2nd and subsequent columns appear in the 1st column. In this case we conclude that $M \equiv M'$ are equivalent

CLOSURE PROPERTIES OF REGULAR SETS:

→ If L' is a regular then L^T is also regular
→ If L' is a regular set over Σ' , then $\Sigma - L$ is also regular over Σ'

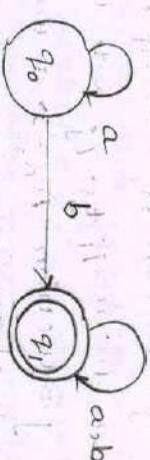
Here $M = (Q, \Sigma, \delta, q_0, F)$ accepting L

→ we construct a another DFA $M' = (Q, \Sigma, \delta, q_0, F')$

by defining $F' = Q - F$ i.e., M & M' differ only in their final states.

→ If x and y are regular sets over Σ then $x \cap y$ is also regular over Σ'
→ If L and M are regular language then $L - M$ is also regular language.

CONVERSION OF TRANSITION SYSTEM TO GRAMMAR.



$q_i \rightarrow aq_j$ if $\delta(q_i, a) = q_j$ with $q_j \notin F$
 $q_i \rightarrow aq_j, q_j \rightarrow a$ if $\delta(q_i, a) \in F$

$$q_0 \rightarrow a q_0$$

$$q_0 \rightarrow b q_1$$

$$q_1 \rightarrow a q_1$$

$$q_1 \rightarrow b q_1$$

$$q_0 \rightarrow b$$

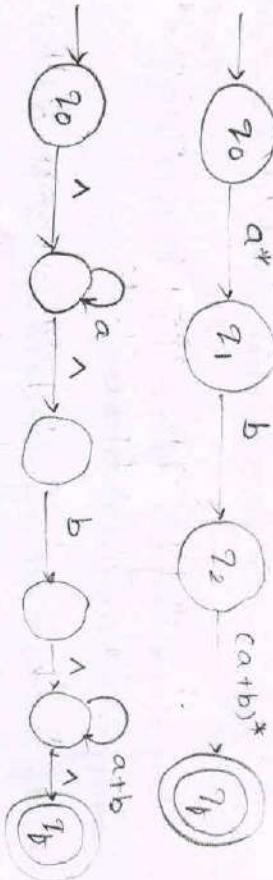
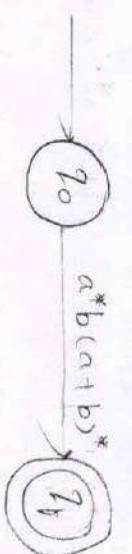
$$q_1 \rightarrow a$$

$$q_1 \rightarrow b$$

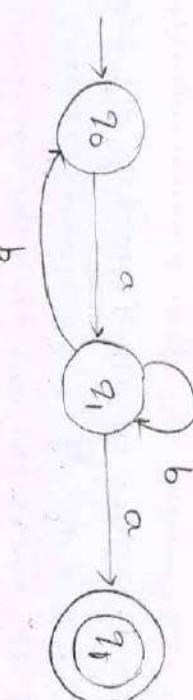
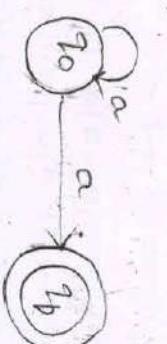
construct a regular grammar generated by
represented by $a^*b(a+b)^*$

$$M = \{ Q, \Sigma, \delta, \delta_0, F \}$$

$$G_1 = \{ V_N, \Sigma, P, S \}$$

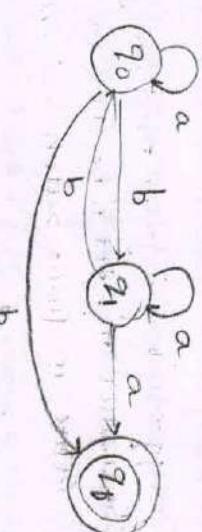


$$S \rightarrow as/a$$



$$M = \{ \{ q_0, q_1, q_b \}, \{ a, b \}, \delta, \delta_0, \{ q_b \} \}$$

Let $G_1 = (\{ \varphi_0, \Lambda_0, \Lambda_1, S, \{ a, b \}, P, \Lambda_0 \})$ where P
consists of $\Lambda_0 \rightarrow a\Lambda_1$, $\Lambda_1 \rightarrow b\Lambda_1$, $\Lambda_1 \rightarrow a$, $\Lambda_1 \rightarrow b\Lambda_0$
construct a transition system accepting this
grammar.



CONSTRUCTION OF TRANSITION SYSTEM FOR A
GIVEN REGULAR GRAMMAR:

Each production $A_i \rightarrow a A_j$ induces a
transition from q_i to q_j with label 'a'
each production $A_k \rightarrow a$ induces a
transition from q_k to q_b with label 'a'.

$$S \rightarrow as/ba/b$$

$$A \rightarrow a A / b A / a$$

PUMPING LEMMA

(1)

Assume L is regular. Let n be the number of states in the corresponding finite automata.

(2)

Choose a string ' w ' such that $|w| \geq n$. Use pumping lemma to write $|w| = xyz$ where $|xy| \leq n$ and $|y| > 0$.

(3) Find a suitable integer v such that

$\alpha y^v z \notin L$ thus contradicts our assumption hence L is not regular.

NOTE:

The crucial part of the procedure is to find ' v ' such that $\alpha y^v z \notin L$. In some cases we prove $\alpha y^v z \notin L$ by considering the length of $|ay^v z|$. In some cases we may have to use the structure of strings in L .

Show that the set $L = \{ \alpha^{i^2} \mid i \geq 1 \}$ is not regular.

(1) Suppose L is regular and we get a contradiction. Let 'n' be the no. of states in finite automata accepting L .

(2)

$$w = \alpha^{n^2}$$

$$|w| = |\alpha^{n^2}| = n^2 \geq n$$

$$\text{Let } w = xyz$$

$$|xy| \leq n \text{ and } |y| > 0$$

$$w = xyz$$

$$xyz = xyz$$

$$|ay^2z| = |ayz| + |y|$$

$$= n^2 + |y|$$

$$> n^2$$

From (2), $|yz| \leq n$.

$$|y| \leq n$$

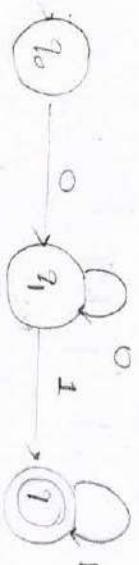
$$|ay^2z| = |ayz| + |y| = n^2 + |y| \leq n^2 + n$$

$$\leq n^2 + n + n^2 \\ \leq (n+1)^2 - c_2$$

From (1) & (2),

$n^2 < |ay^2z| < (n+1)^2$ This is a contradiction so, it is not a regular grammar.

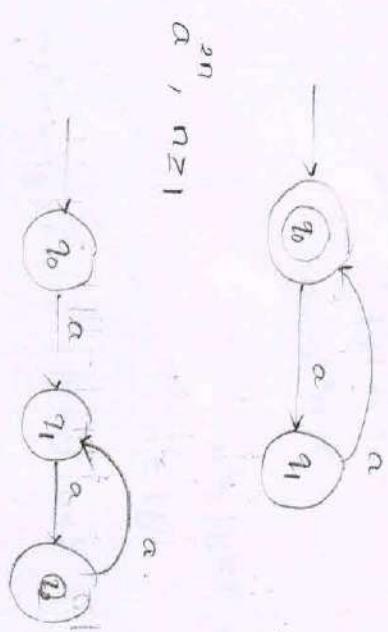
$o^m 1^n$ where $m, n \geq 1$



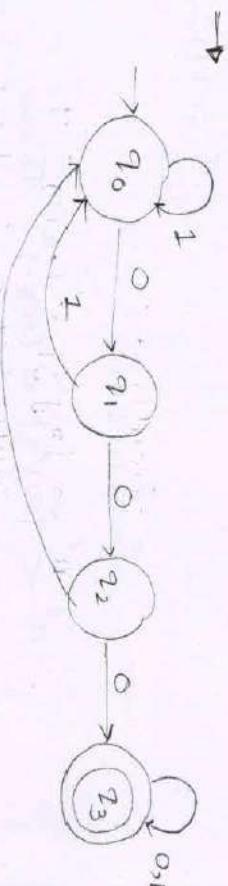
$o^m 1^n$ where $m, n \geq 0$



α^{2n} , $n \geq 0$



α^{2n} , $n \geq 1$



Convert into regular grammar.

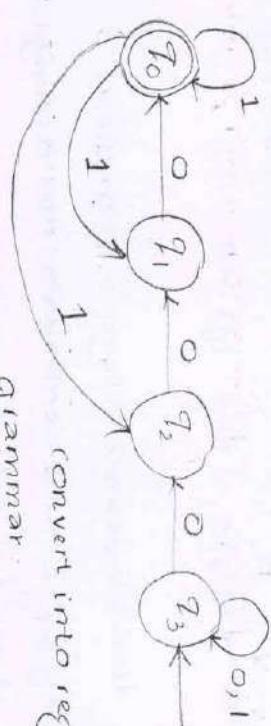
$$q_0 \rightarrow oq_1 / 1q_0$$

$$q_1 \rightarrow oq_2 / 1q_0$$

$$q_2 \rightarrow oq_3 / 1q_0 / o$$

$$q_3 \rightarrow oq_3 / 1q_3 / o / 1$$

Right linear grammar.



convert into regular

grammar

$$q_0 \rightarrow 1q_0 / 1q_1 / 1q_2 / 1$$

$$q_1 \rightarrow oq_0 / o$$

$$q_2 \rightarrow oq_1$$

$$q_3 \rightarrow oq_2 / oq_3 / 1q_3$$

↓
left linear
grammar

$$q_0 \rightarrow q_0\alpha / q_1\alpha / q_2\alpha / 1$$

$$q_1 \rightarrow q_0 o / o$$

$$q_2 \rightarrow q_1 o$$

$$q_3 \rightarrow q_2 o / q_3 o / q_3 1$$

Find the regular expression P_0 , the following

$$\alpha^m b^n c^p, m, n, p \geq 1$$

↓

$$\alpha^* b b^* c c^* \rightarrow \alpha^* b^+ c^+$$

(15)

$a^m b^{2n} p \quad m, n, p \geq 1$

$aa^* bb(bb)^* ccc(ccc)^*$

$a^n b a^{2m} b^2 \quad m \geq 0, n \geq 1$

$aa^* b (aa)^* b^2$

Verify if a^p is a regular grammar or not where p is a prime number.

(1) Assume 'L' is regular let 'n' be the number of states in finite automata accepting L.

(2) $w = a^p n$

Let, $m \geq n$.

$|w| \geq n$

$w = a^p b^q, |a| \leq n$ and $|b| > 0$

(3)

$w = a^p b^q$

Let $i = m+1$

$$|a^p b^q| = |a^p b^i| + |b^{i-1}|$$

$$= m + (i - 1) |b|$$

$$|a^p b^q| = m + m |b|$$

$= m(m + p) \text{ not prime}$

CONTEXT FREE GRAMMAR:
obtain the context-free grammar for signed integers.

$S \rightarrow \langle \text{sign} \rangle \langle \text{integer} \rangle$
 $\text{sign} \rightarrow + / -$

$\text{integer} \rightarrow \langle \text{digit} \rangle \langle \text{integer} \rangle / \langle \text{digit} \rangle$
 $\text{digit} \rightarrow 0 / 1 / \dots / 9$

$G_1 = (V_N, \Sigma, P, S)$

$V_N = \{S, \text{sign}, \text{integer}, \text{digit}\}$

$S \rightarrow \langle \text{sign} \rangle \langle \text{integer} \rangle$

$\text{sign} \rightarrow +$
 $\text{integer} \rightarrow \langle \text{digit} \rangle \langle \text{integer} \rangle$

$\rightarrow + \langle \text{integer} \rangle$
 $\rightarrow - \langle \text{integer} \rangle$

$\rightarrow 1 \langle \text{digit} \rangle \langle \text{integer} \rangle$
 $\rightarrow 12 \langle \text{integer} \rangle$

$\rightarrow 125 \langle \text{integer} \rangle$
 $\rightarrow 125 \langle \text{digit} \rangle$

$\rightarrow 1250$

$* N = (0, 1) \rightarrow \langle \text{sign} \rangle \langle \text{integer} \rangle$

Define context-free grammar for palindrome over $\{a, b\}$.

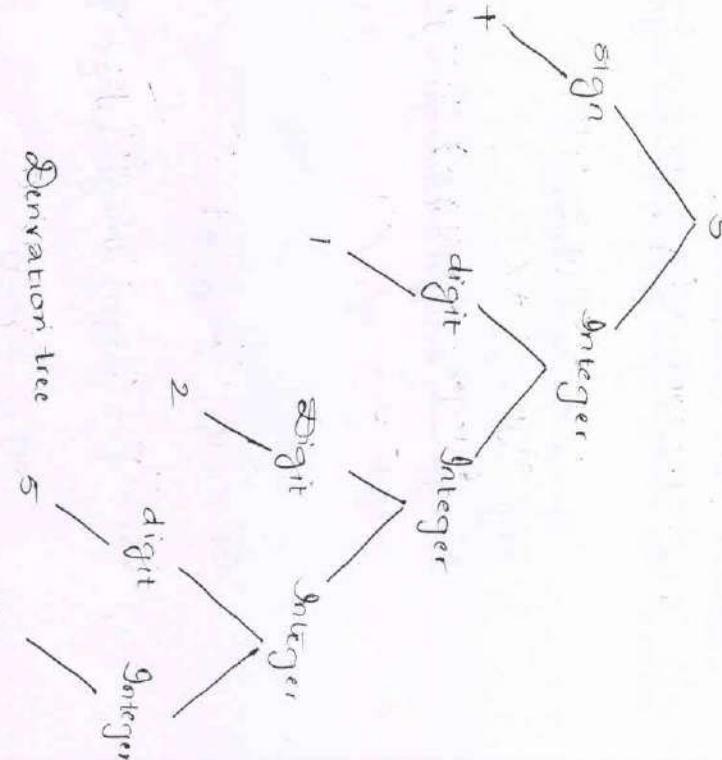
$a, a, b, aa, bb, aaa \dots$

Palindrome - initial and final must be same
in the middle it should contain
Palindrome.

Productions:

$$\begin{aligned} S &\rightarrow a S a \\ S &\rightarrow b S b \\ S &\rightarrow \lambda / a / b. \end{aligned}$$

Derivation tree



TERMINAL: α

non-terminal: S

Productions: $S \rightarrow aS$

$S \rightarrow \lambda$

$S \rightarrow a/\lambda$

$\lambda, a, a, aas, \underline{aaas}$

$\lambda, a, \underline{aa}, \underline{aaa}$

λ, a, aa, aaa

Here,

$$L(G_1) = \alpha^*$$

Obtain context-free grammar for equal no. of a 's and equal no. of b 's.

Productions:

$$S \rightarrow s a s b s / s b s a s / \lambda$$

Derive context free grammar for different number of a 's and b 's

$abb, baa, bba, abab, ababa,$

$abbaaa, bbaab, aba, bab, a, b.$

$$S \rightarrow aSbb/abb$$

$$x \rightarrow sasbs/sbsas/\lambda$$

$$S \rightarrow u/v \rightarrow |a| < |b|$$

$$|a| > |b|$$

$$U \rightarrow xau/xax$$

$$V \rightarrow xbv/xbx$$

$$X \rightarrow xaxbx/xbxa/\lambda$$

Give the context free grammar for

$$(01 + 11)^* (01)^*$$

$$(01 + 11)^* (01)^*$$

$S \rightarrow AB$

$$A \rightarrow ca/\lambda$$

$$C \rightarrow 0n_1 1n_2 \dots n_k$$

$$B \rightarrow DB/\lambda$$

$$D \rightarrow 0n_1 1n_2 \dots n_k$$

Derive the context free grammar for $a^n b^n, n \geq 1$

Derive the Context free grammar, the first and last symbol should be different over $\Sigma = \{0, 1\}$

$$S \rightarrow 0S1/1S0/10/01/$$

$$S \rightarrow 01/10/0A1/1A0$$

$$A \rightarrow 0A1/1A0/01/1$$

DERIVATION TREE:

A derivation tree is also called parse tree for a context free grammar (V_N, Σ, P, S) is a tree satisfying the following

In a tree, every vertex has a label which is a variable or terminal or null

The root has the label S' .

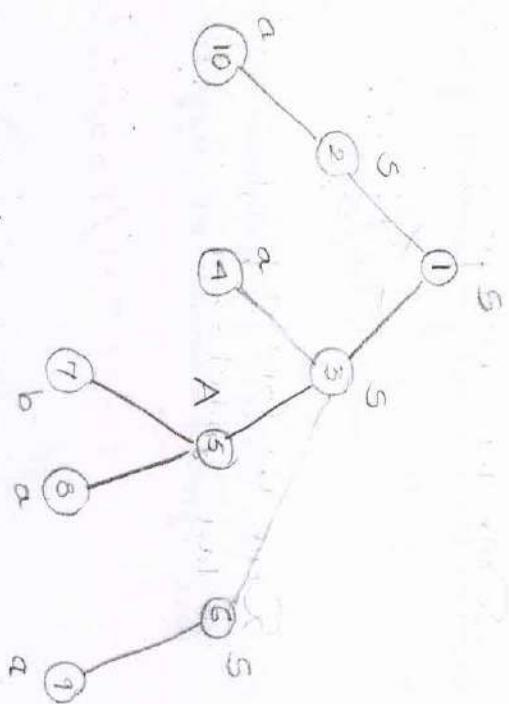
Internal terminal vertex is the non-terminal or variable.

The $\{n_1, n_2, \dots, n_k\}$ are written with the labels x_1, x_2, \dots, x_k are the sons of the vertex n with label α ,

then $\alpha \rightarrow x_1, x_2, \dots, x_k$ is a production in P .

A vertex n is a leaf if its label is $\alpha \in \Sigma$ or null.

$S \rightarrow SS \rightarrow aS \rightarrow aaAs \rightarrow aabaa$



Productions:

$$S \rightarrow SS$$

$$S \rightarrow a$$

$$S \rightarrow aAs$$

$$A \rightarrow ba$$

ORDERING OF LEAVES FROM LEFT:

The yield of a derivation tree is the concatenation of the labels of the leaves without repetition in the left-to-right ordering.

NOTE

If we draw the sons of the every vertex in the left-to-right ordering we get the left-to-right ordering of the leaves, by reading the leaves in the anticlockwise direction.

A subtree of a derivation tree 'T' is a tree
 (1) whose root is vertex \vee of 'T' i.e., $\vee(T)$
 (2) whose vertices are the descendants of \vee
 together with their labels
 (3) whose edges are those connecting the descendents of \vee

Let $G_1 = (V_N, \Sigma, P, S)$ be a context free grammar, then $S \rightarrow \alpha$ if and only if there is a derivation tree for ' G_1 ' which yield ' α '.

e.g., $S \rightarrow aab \times$

cannot be formed using the before 4 productions

SENTENTIAL FORM

Yield is also known as *sentential form*.

Consider \mathcal{G}_1 whose productions are

$$S \rightarrow aAs/a$$

$$A \rightarrow SbN/ss/ba$$

Show that $S \rightarrow abbaa$ and Consider a derivation tree whose yield is $abbaa$.

$$\begin{array}{l} S \rightarrow aAs/a \\ A \rightarrow sba/ss/ba \end{array}$$

$S \Rightarrow aabbaa$

LEFT MOST DERIVATION

$S \rightarrow aAs$ A is replaced by aS

$$S \rightarrow a \underline{S} b A S$$

$$S \rightarrow a\overline{abb}a$$

represented by ba s by a

\downarrow
babaa

(2) $S \rightarrow abaa$

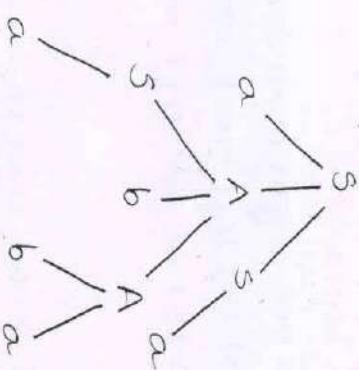
$s \rightarrow aAs -$

But in derivation(3) no such conditions are satisfied.

DEEPMOST DERIVATIONS.

The derivation $A \rightarrow w$ is left-most derivation if we apply a production to the left most variable at every step.

DERIVATION TREE



String → abbaa

In derivation (1) whenever we replace a variable by using a production, there is no variable to the left of.

In derivation (2) there are no variables to the right of α .

But in derivation(3) no such conditions are

DEEHOST DERIVATIONS

The derivation

derivation if we apply a production to the left most variable at every step.

RIGHT MOST DERIVATION:

A derivation $A \Rightarrow^* w$ is right most derivation, if we apply a production to right most variable.

In previous example (1) is a left most and (2) is right most derivation.

But equation (3) is neither left most nor right most.

In 2nd step of equation (3) the right most variable s is not replaced.

so equation (3) is not right most derivation.

In third step of (3) the left most variable is not replaced so (3) is not left most derivation.

THEOREM:

If $A \Rightarrow^* w$ is in G₁ then there is a left most derivation of w .

Even derivation tree of w induces a left most derivation of w once we get some derivation of w it is easy to get a left most derivation w in the following way. From the derivation tree for w at even level consider the projections for variables at that level, taken in the left-to-right ordering.

The left most derivation is obtained by applying productions in this order.

NOTE:

A derivation tree may contain more than one derivation

Let G_1 be a grammar to $S \rightarrow OB/1A$

$A \rightarrow oB/oS/1AA$, $B \rightarrow o/S/OBB$ for the string $oo11010101$

Find a) leftmost b) right most c) derivation tree

LEFT MOST DERIVATION:

$S \rightarrow OB/1A$, $A \rightarrow o/oS/1AA$, $B \rightarrow 1/1S/OBB$

$S \rightarrow \underline{OB}$

\downarrow

$o\underline{oB}B$

\downarrow

$oo1\underline{SB}$

\downarrow

$oo11\underline{AB}$

\downarrow

$oo110\underline{S}B$

\downarrow

$oo1101\underline{A}B$

\downarrow

$oo11010\underline{B}$

RIGHT MOST DERIVATION:

$S \rightarrow \underline{OB}$

\downarrow

$o\underline{oB}B$

\downarrow

$oo1\underline{S}B$

\downarrow

$oo11\underline{AB}$

\downarrow

$oo110\underline{S}B$

\downarrow

$oo1101\underline{A}B$

\downarrow

$oo11010\underline{B}$

\downarrow

$oo110101$

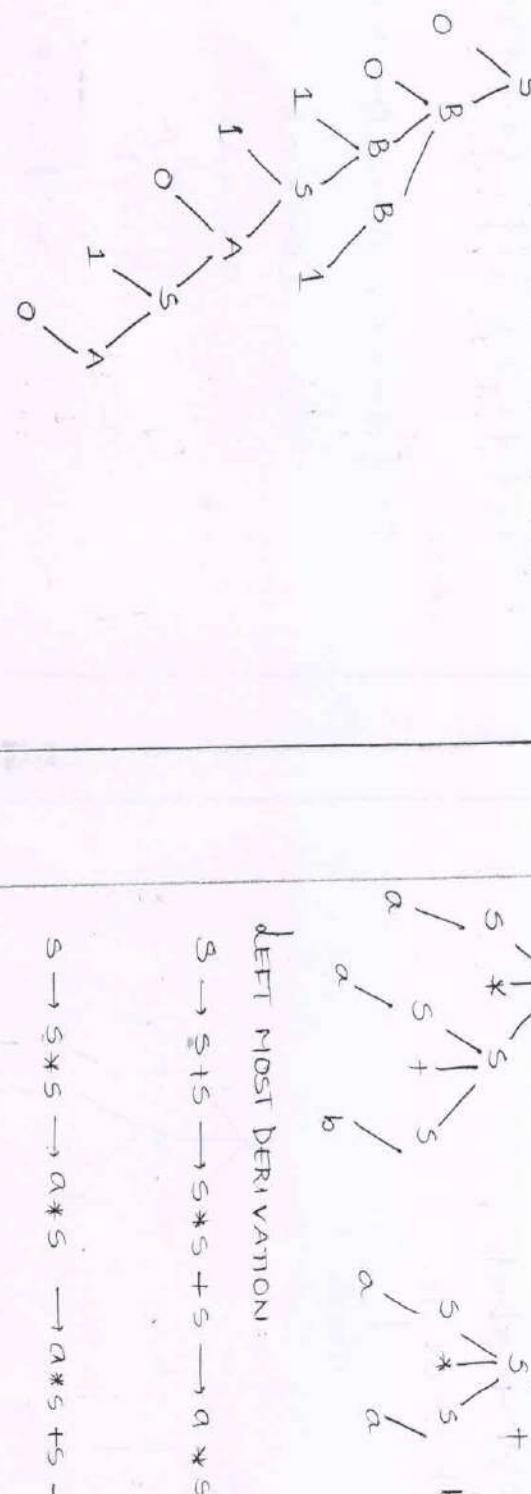
derivation tree for ϕ

eg., $G_1 = (\{s_3\}, \{a, b\} +, *, \{j_1, p, s\})$

where P consists of

$$s \rightarrow s + s / s * s / a / b / a * a + b$$

DERIVATION TREE.



DEFT MOST DERIVATION

$$S \rightarrow S + S \rightarrow S * S + S \rightarrow a * S + S \rightarrow a * a + S$$

$$\alpha * \alpha + b$$

$$S \rightarrow S * S \rightarrow a * S \rightarrow a * a + S \rightarrow a * a + b$$

Here we are getting 2 different derivations or 2 different derivation trees so, $a * a + b$ is ambiguous

A context free grammar is ambiguous if there exists some $w \in L(G)$ which is ambiguous.

So that $s \rightarrow sbs/a$ is ambiguous.

$$s \rightarrow \underline{s}bs \rightarrow \underline{a}bs \rightarrow aba$$

'2' derivation tree
so, LCA is "minimally"

(i) 'c' does not derive any terminal string

(ii) 'E' and 'c' do not appear in any sentential form

(3) $E \rightarrow A$ is a null production

(4) $B \rightarrow C$ simply replaces B by C .

(5) we give the construction to eliminate

(i) variables not deriving terminal strings

(ii) symbols not appearing in any sentential form

(iii) null production

(iv) productions of the form $A \rightarrow B$

CONSTRUCTION OF REDUCED GRAMMAR

THEOREM 1:

If G_1 is a context free grammar such that $L(G_1) \neq \emptyset$ we can find an equivalent grammar G_1' such that each variable in G_1' derives some terminal string. Let $G_1 = (V_N, \Sigma, P, S)$ we define $G_1' = (V_N', \Sigma', P', S)$ as follows:

Step 1. Construction of V_N'

We define V_N' is a subset of V_N i.e., $V_N' \subseteq V_N$ by recursion.

$$W_1 = \{ A \in V_N \mid \text{there exist a production } A \rightarrow \alpha \text{ where } \alpha \in (\Sigma \cup W_1)^*\}$$

$$A \rightarrow \alpha \text{ where } \alpha \in (\Sigma \cup W_1)^*$$

If $W_1 = \emptyset$ some variable will remain after the application of any production and so, $L(G_1) = \emptyset$

$$W_{i+1} = W_i \cup \{ A \in V_N \mid \text{there exists a some production } A \rightarrow \alpha \text{ with } \alpha \in (\Sigma \cup W_i)^*\}$$

so, by the definition of W_i , W_i is a subset of W_{i+1} i.e., $W_i \subseteq W_{i+1} \subseteq V_N$. V_i as V_N has only finite number of variables $W_N = W_{N+1}$ for some $N \leq |V_N|$

$$\therefore W_N = W_{N+j} \text{ for } j \geq 1. \text{ So, here we define } V_N' = W_N$$

step 2 :

Construction of P'

$$P' = \{ A \rightarrow \alpha \mid A \in V_N', \alpha \in (\Sigma \cup V_N')^*\}$$

$$W = G_1' = (V_N', \Sigma, P', S)$$

Let $G_1 = (V_N, \Sigma, P, S)$ be given by the productions $S \rightarrow AB$, $A \rightarrow \alpha$, $B \rightarrow b$, $B \rightarrow C$, $E \rightarrow e$ find G_1' such that every variable in G_1' derive some terminal string.

Construction of V_N'

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

Since $A \rightarrow \alpha$, $B \rightarrow b$, $E \rightarrow c$ are productions with a terminal string on R.H.S

$$W_2 = W_1 \cup \{ A_1 / A_1 \in V_N, A_1 \rightarrow \alpha, \alpha \in (\Sigma, A, B, E)^*\}$$

$$W_2 = W_1 \cup S$$

$$W_2 = \{ A, B, E, S \} \text{ from the production } S \rightarrow ABA$$

$$W_3 = W_2 \cup \{ A_i | A_i \in V_N, A_i \rightarrow \alpha \}$$

$$\alpha \in (\infty, A, B, E, S)^*$$

(a) Construction of W_i for $i \geq 1$
 (in initially $W_1 = \{S\}$)
 (ii) $W_{i+1} = W_i \cup \{x \in V_N \cup \infty \mid \exists \text{ a production } A \rightarrow \alpha \text{ with } A \in W_i \text{ and } \alpha$

$$W_3 = W_2 \cup \emptyset$$

$$\therefore V_N' = W_2 = \{A, B, E, S\}$$

Construction of P' .

$$P' = \{ A_i \rightarrow \alpha \mid A_i \in V_N', \alpha \in (V_N' \cup \infty)^* \}$$

$$= \{ S \rightarrow AB, A \rightarrow a, B \rightarrow b, E \rightarrow c \}$$

here,

$$G_1' = \{ A, B, E, S \}, \{ a, b, c \}, P', S$$

THEOREM 2:

for every CFG $G_1 = (V_N, \Sigma, P, S)$ we can construct an equivalent grammar $G_1' = (V_N', \Sigma', P', S')$ such that every symbol in $V_N \cup \Sigma$ appears in some sentential form (i.e., for every $x' \in V_N' \cup \Sigma'$ there exist α such that $x' \Rightarrow^* \alpha$ and α is a symbol in the string α)

we can construct $G_1' = (V_N', \Sigma', P', S)$ as follows:

so, here $S \in W_1$ and $S \rightarrow AB$.

$$W_2 = \{S\}$$

$$W_2 = \{S, A, B\}$$

containing the symbol $x\}$.

we may note that $W_i \subseteq V_N \cup \Sigma$ & $W_i \subseteq W_{i+1}$ as we have only finite number of elements in $V_N \cup \Sigma$, $W_k = W_{k+1}$ for some k - this means

$$W_k = W_{k+j} \quad \forall j \geq 0$$

(b) Construction of V_N' , Σ' and P' .

$$\text{we define } V_N' = V_N \cap W_k$$

$$\Sigma' = \Sigma \cap W_k$$

$$P' = \{ A \rightarrow \alpha \mid A \in W_k \}$$

Eq, Consider $G_1 = (S, A, B, E, S), \{a, b, c\}, P, S$

where P consists of $S \rightarrow AB, A \rightarrow a, B \rightarrow b, E \rightarrow c$

$$W_1 = \{S\}$$

$W_2 = W_1 \cup \{x \mid x \in V_N \cup \Sigma \mid \exists \text{ a production } A_i \rightarrow \alpha \text{ with } A_i \in W_1 \text{ and }$

$\alpha \text{ containing } x\}$

$W_3 = W_2 \cup \{ x \in V_N^0 \mid \exists \text{ a production}$

$A_1 \rightarrow \alpha \text{ with } A_1 \in W_2 \text{ and }$

$$\{ s, A, B \}$$

$\alpha \text{ containing } x \}$

$$W_3 = \{ s, A, B \} \cup \{ a, b \}$$

$$= \{ s, A, B, a, b \}$$

$$W_4 = W_3 \cup \{ x \in V_N^0 \mid \exists \text{ a production}$$

$A_1 \rightarrow \alpha \text{ with } A_1 \in W_3 \text{ and }$

$\alpha \text{ containing } x \}$

$$= \{ s, A, B, a, b \} \cup \phi$$

$$= \{ s, A, B, a, b \}$$

$$= W_3$$

$$W_K = W_{K+1} = \{ s, A, B, a, b \}$$

(b) construction of V_N^1, P^1 and S^1

$$V_N^1 = W_K \cap V_N$$

$$= V_N \cap W_3$$

$$= V_N \cap \{ s, A, B, a, b \}$$

$$= \{ s, A, B \} \cap \{ s, A, B, a, b \}$$

$$= \{ s, A, B \}$$

$\leq^1 = \leq_{\text{NFA}}$

$$= \{ a, b \} \cap \{ s, A, B, a, b \}$$

$$= \{ a, b \} \cap \{ s, A, B, a, b \}$$

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

$$P^1 = \{ A_1 \rightarrow a \mid A \in W_K \}$$

$$= \{ A_1 \rightarrow a \mid A \in W_3 \}$$

$$= \{ S \rightarrow AB, A \rightarrow a, B \rightarrow b \}$$

the required grammar $G_1^1 = (V_N^1, \Sigma^1, P^1, S)$

THEOREM 3:

for every CFG G^1 there exist a reduced grammar which is equivalent to G^1 .

Proof:

we construct the reduced grammar in 2'

steps

Step 1: we construct a grammar G_1 equivalent to the grammar G^1 so that every variable in G_1 derives for some terminal string (Theorem 1).

Step 2: we construct a grammar G_1^* (V_N^1, Σ^1, P^1)

equivalent to G_1 , so that every symbol in G_1^* appears in some sentential form of G_1^* which is equivalent to G_1 and hence to G^1 . G_1^* is the required reduced grammar.

By step even symbol α in G_1' appears in some sentential form, say $\alpha\beta$.
By step even symbol in $\alpha\beta$ derives some terminal string

$$S \rightarrow \alpha\beta \Rightarrow w \in \Sigma^* ; i.e., G_1' \text{ is reduced}$$

NOTE:

To get a reduced grammar, we must first apply theorem 1 and then theorem 2. If you apply theorem 2 first and then theorem 1 we may not get a reduced grammar.

find a reduced grammar equivalent to the grammar G_1 whose productions are

$$S \rightarrow AB/cA$$

$$B \rightarrow BC/AB$$

$$A \rightarrow a$$

$$C \rightarrow aB/b$$

$$G_1 = (S, \{A, B, C\}, \{a, b\}, P, S)$$

Step 1:

$$w_1 = A/a \rightarrow w_1, w \in \Sigma^*$$

$w_1 = \{A, c\}$ from the productions

$$A \rightarrow a, C \rightarrow b$$

$$w_2 = w_1 \cup \{a \mid a \in (\Sigma^* \cup \epsilon), A_1 \rightarrow a\}$$

$$w_2 = \{A, C\} \cup \{a\} = \{S, A, C\}$$

from the production $S \rightarrow CA$

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

$$= w_2 = w_K = w_{K+1}$$

$$V_N^1 = w_K = w_2 = \{S, A, C\}$$

$$P' = \{A_1 \rightarrow \alpha \mid A_1 \in V_N^1, \alpha \in (\Sigma \cup V_N^1)^*\}$$

$$= \{S \rightarrow ca, A \rightarrow a, C \rightarrow b\}$$

$$G_1' = (S, \{A, C\}, \{a, b\}, P', S)$$

Step 2:

$$W_1 = \{S\}$$

$$W_2 = W_1 \cup \{x \mid x \in (V_N^1 \cup \epsilon), A_1 \rightarrow a \text{ new, and } a \text{ containing symbol } x\}$$

$$W_2 = SW_1 \cup \{a\}$$

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

containing symbol $x \in$

$$W_3 = W_2 \cup \{x \mid x \in (V_N^1 \cup \epsilon), A_1 \rightarrow a \text{ new, and } a \text{ containing symbol } x\}$$

$$= \{S, C, A, a, b\}$$

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

$$W_3 = \{S, A, C\} \cup \{A_1 \rightarrow a \mid a \in (\Sigma^*, S, A, C)\}$$

$$= \{S, A, C\} \cup \{\phi\}$$

$$W_K = W_3 - \{s, c, A, a, b\}$$

$$V_N^1 = V_N \cap W_K$$

$$= \{s, A, c\} \cap \{s, A, c, a, b\}$$

$$= \{s, A, c\}$$

$$Z' = Z \cap W_K$$

$$= \{a, b\} \cap \{s, A, c, a, b\}$$

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

$$P^1 = \{A_1 \rightarrow \alpha / \alpha \in W_K\}$$

$$= \{s \rightarrow cA, A \rightarrow \alpha, c \rightarrow b\}$$

$$G_1^1 = (\{s, A, c\}, \{a, b\}, P^1, S)$$

Construct a reduced grammar equivalent to the grammars,

$$\begin{aligned} S &\rightarrow aAa, \\ A &\rightarrow sb/bcc/Daa \\ C &\rightarrow abb/BDD \\ E &\rightarrow aC \end{aligned}$$

$$D \rightarrow aDA$$

in step 4.

$$\underline{W_1 = \{c\}}$$

$$W_2 = N_0 \{A_1 / A_1 \in V_N, A_1 \rightarrow \alpha\}$$

$$\alpha \in \{a, b\}^*$$

from the productions

$$A \rightarrow bCC, E \rightarrow aC$$

$$W_2 = C \cup \{A_1, E\}$$

$$= \{A_1, C, E\}$$

$$W_3 = W_2 \cup \{A_1 / A_1 \in V_N, A_1 \rightarrow \alpha, \alpha \in (A_1, C, E, a, b)^*\}$$

from the productions $s \rightarrow aAa$

$$W_3 = \{A_1, C, E\} \cup \{s\} = \{s, A, C, E\}$$

$$W_4 = W_3 \cup \{A_1 / A_1 \in V_N, A_1 \rightarrow \alpha\}$$

$$= \{s, A, C, E\} = W_3 = W_K$$

from the production

$$A \rightarrow sb$$

$$W_4 = \{s, A, C, E\} \cup \{A\}$$

$$W_4 = \{s, A, C, E\} = W_3 = W_K$$

$$V_N^1 = W_3 = W_K = \{s, A, C, E\}$$

$$P^1 = \{s \rightarrow aAa, A \rightarrow bCC, C \rightarrow abb, E \rightarrow aC\}$$

$$\underline{\text{Hence Step 4:}}$$

$$G_1^1 = \{ \{s, A, C, E\}, \{a, b\}, P^1, S \}$$

STEP 2:

$$S' = S \cap W_2$$

$$= \{a, b\} \cap \{s, a, c, a, b\}$$

$$W_1 = \{s\}$$

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

$$W_2 = W_1 \cup \{x / x \in V_N \cup \epsilon, A_1 \rightarrow \alpha$$

$\Delta \in W_1 \& \alpha$ containing
symbol $x \notin$

$$W_2 = \{s\} \cup \{a, \alpha\}$$

$$= \{s, a, \alpha\}$$

$$W_3 = W_2 \cup \{x / x \in V_N \cup \epsilon, A_1 \rightarrow \alpha,$$

$A_1 \in W_2 \& \alpha$ containing
symbol x .

$$W_4 = W_3 \cup \{s, A_1 \alpha\} \cup \{b, c\}$$

$$= \{s, A_1 \alpha, a, b, c\}$$

$$W_4 = W_3 \cup \{x / x \in V_N \cup \epsilon, A_1 \rightarrow \alpha,$$

$A_1 \in W_3 \& \alpha$ containing
symbol x .

we find the nullable variables recursively
we find the nullable variables recursively

$$(i) W_1 = \{A \in V_N / A \rightarrow \alpha \text{ is in } P\}$$

$$(ii) W_{i+1} = W_i \cup \{A \in V_N / A \rightarrow \alpha \rightarrow \beta \in W_i^*\}$$

by definition of W_i , $W_i \subseteq W_{i+1} \forall i$ as V_N is
finite $W_{k+1} = W_k$ for some $k \leq |V_N|$ also, $W_{k+j} = W_k$

$$W_1 = V_N \cap W_3$$

$$= \{s, A_1 \alpha, a, b\}$$

$$= \{s, A_1 \alpha\}$$

Step 3: Construction of the set of nullable variables

A variable X in a context free grammar
is nullable if $\lambda \rightarrow X$

THEOREM:

If $G_1 = (V_N, \Sigma, P, S)$ is a CFG then we
can find $CFG G_1$ having no null productions
such that $L(G_1) = L(G_1) - \{\lambda\}$
we construct $G_1 = (V_N, \Sigma, P^1, S)$ as follows:

Step 4:

Construction of the set of nullable variables

we find the nullable variables recursively

$$\{s, A_1 \alpha, a, b\} \cup \{a, b\}$$

$$= \{s, A_1 \alpha, a, b\} = W_3 = W_k.$$

$$W_1 = V_N \cap W_3$$

$$= \{s, A_1 \alpha, a, b\}$$

$$= \{s, A_1 \alpha\}$$

let $w = W_k$ is the set of all nullable
variables

Step 2

Construction of P'

(i) Any production whose R.H.S does not have any nullable variable is included in P'

(ii) If $A \rightarrow x_1 x_2 \dots x_k$ is in P , the productions of form $A \rightarrow \alpha_1 \alpha_2 \dots \alpha_k$ included in P' where $\alpha_i = x_i$ if

$x_i \notin w$

$\alpha_i = x_i / \lambda$ if $x_i \in w$ and $\alpha_1, \alpha_2, \dots, \alpha_k \neq \lambda$

Actually it gives several productions in P' so, the productions are obtained either by not erasing any nullable variable on R.H.S of $A \rightarrow x_1 x_2 \dots x_k$ or by erasing some or all nullable variables provided some symbol appears on the R.H.S after erasing now $G_1 = (V_N, S, P^1, \delta)$ has no null productions.

$S \rightarrow \alpha S / AB, A \rightarrow \lambda, B \rightarrow \lambda, D \rightarrow b$ find

$G_1 = (V_N, S, P^1, \delta) - \{\lambda\}$.

Sup 1:

$w_1 = \{A, B\} \cup \{A_1 \in V_N, A_1 \rightarrow \lambda\} \subseteq V_N$

$w_2 = \{A_1 B, S\} \mid A_1 \in V_N, A_1 \rightarrow \alpha_1, \alpha_2 \in w^*$

$w_3 = w_2 - w_1$

Step 2:

Construction of P'

(i) $D \rightarrow b$

(ii) $S \rightarrow as$

$S \rightarrow AB, S \rightarrow A, S \rightarrow B$

$S \rightarrow \lambda, A \rightarrow B, B \rightarrow C, C \rightarrow \lambda, D \rightarrow a, D \rightarrow aA$

Step 1

$w_1 = \{C\}$

$w_2 = \{C, B\} \cup \{x / x \rightarrow \alpha, \alpha \in w_1^*\}$

$w_3 = \{C, B\} \cup \{x / x \rightarrow \alpha, \alpha \in w_2^*\}$

$w_4 = \{C, B, A\} \cup \{x / x \rightarrow \alpha, \alpha \in w_3^*\}$

$w_4 = \{S, A, B, C\} \cup \emptyset$

$w_4 = \{S, A, B, C\} = w_3 = w_K = w$

Step 2: construction of P'

(i) $D \rightarrow a$

(ii) $D \rightarrow A E$

\downarrow

$\alpha_1 \alpha_2$

$D \rightarrow A E$

$D \rightarrow E$

$P_2: S \rightarrow AB$

\downarrow

$S \rightarrow A$

\downarrow

$S \rightarrow B$

COROLLARY 1 :

There exists an algorithm to decide whether, $\lambda \in L(G)$ for a given G

$$C \in G$$

Proof:

$\lambda \in L(G)$, iff $S \in W$ i.e., S is nullable
the construction given in theorem 4 is recursive and terminates in finite number
of steps. (Actually in atmost $|V_N|$ steps
so, the required Algorithm is as follows.)

- Construct W
- Test whether $S \in W$

COROLLARY 2 :

If $G_1 = (V_{N_1}, \Sigma, P_1, S)$ is a CFG we
can find an equivalent CFG $G_1' = (V_{N_1}', \Sigma, P_1', S_1)$
without λ productions except $S_1 \rightarrow \lambda$ when
 λ is in $L(G_1)$. If $S_1 \rightarrow \lambda$ is in P_1' , S_1 does not
appear on the RHS of any production in P_1 .

Proof:

By corollary 1, we can decide whether

λ is in $L(G_1)$

Case(i) If λ is not in $L(G_1)$

G_1' obtained by using theorem 4 is
a required equivalent grammar.

Case(ii)

If λ is in $L(G_1)$, Construct $G_1' = (V_{N_1}', \Sigma, P_1', S)$
using theorem 4 $L(G_1') = L(G_1) - \{\lambda\}$

$$\text{Define } G_1' = (V_{N_1} \cup \{S_1\}, \Sigma, P_1', S_1), \text{ where}$$

$$P_1' = P_1 \cup \{S_1 \rightarrow S, S_1 \rightarrow \lambda\}$$

S_1 does not appear on the RHS of any
production in P_1 , and so G_1' is the required
grammar with $L(G_1') = L(G_1)$.

DEFINITION OF UNIT PRODUCTION:

def: A unit production (or a chain rule) In CFG G_1'
is a production of the form $\lambda \rightarrow B$, where $\lambda \notin B$ are
variables in G_1' .

THEOREM: If G_1 is a CFG we can find a CFG G_1' which
has no null productions & unit productions
such that $L(G_1) = L(G_1')$

Proof: We can apply Corollary 2 of theorem 4
to grammar G_1 to get a grammar $G_1' = (V_{N_1}, \Sigma,
P_1, S)$ without null productions such that
 $L(G_1') = L(G_1)$

Let λ be any variable in V_{N_1}

Step 1:

Construction of the set of variables
derivable from λ :
define $W_1(\lambda)$ recursively as follows:
 $\phi \cup \{ \lambda \}$

$$\omega_0(A) = \{ A \}$$

$$\omega_{i+1}(A) = \omega_i(A) \cup \{ B / B \in V_N, C \rightarrow B \text{ is in } P \text{ with } C \in \omega_i(A) \}$$

by definition of $\omega_i(A)$, $\omega_i(A) \subseteq \omega_{i+1}(A)$
as V_N is finite, $\omega_{k+j}(A) = \omega_k(A)$ for some $k \leq |V_N|$
so, $\omega_{k+j}(A) = \omega_{k+1}(A) \forall j \geq 0$. let $w(A) = \omega_k(A)$
then $w(A)$ is the set of all variable derivable
from A .

Step 2:

Construction of A -productions in G_1

- (1) The non-unit production in G_1 ,
(2) $A \rightarrow \alpha$ whenever $B \rightarrow \alpha$ is in G_1 with $B \in w(A)$
and $\alpha \notin V_N$ (actually α covered 1 as
 $A \in w(A)$) now we define $G_1 = (V_N, \Sigma, R, S)$
where R_1 is constructed using step 2, $\forall A \in V_N$.

Ex:

let G_1 be $S \rightarrow AB, A \rightarrow a, B \rightarrow C/b, C \rightarrow D,$

$D \rightarrow E, E \rightarrow a$: eliminate unit

Productions and get an equivalent

grammar

Step 1:

$$w(B) = \{ B \}$$

$$w(S) = \{ S \}$$

$$\omega_1(C) = \omega_0(C) + \{ x / x \in V_N, y \rightarrow x \text{ is in } P, \\ \text{and } w(C) = w_0(C) \neq \emptyset \text{ and } y \in \omega_0(C) \}$$

$$= \{ S \} \cup \emptyset$$

$$\omega_1(S) = \{ S \}$$

$$= \omega_0(S)$$

$$w(A)$$

$$w_0(A) = \{ A \}$$

$$w(A)$$

$$w_1(A) = w_0(A) + \{ x / x \in V_N, y \rightarrow x \text{ is in } P, \\ y \in w_0(A) \}$$

$$y \in \omega_0(A)$$

$$\omega_1(A) = \{ A \} + \emptyset$$

$$\omega_1(A) = \{ A \}$$

$$= w_0(A)$$

$$w(A) = \{ A \}$$

$$w(B)$$

$$w_0(B) = \{ B \}$$

$$w_1(B) = w_0(B) + \{ x / x \in V_N, y \rightarrow x \text{ is in } P, \\ y \in w_0(B) \}$$

$$= \{ B \} + \{ C \}$$

$$= \{ B, C \} \cup \{ a \}$$

$$w_2(B) = w_1(B) + \{ D \}$$

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

$$\omega_3(B) = \{B, C, D\} \cup \{E\}$$

$$= \{B, C, D, E\}$$

$$\omega_4(B) = \{B, C, D, E\} \cup \phi$$

$$= \{B, C, D, E\}$$

$$= \omega_3(B)$$

$$\omega(B) = \{B, C, D, E\}$$

$\omega(CC)$

$$\omega_0(CC) = \{C\}$$

$$\omega_1(CC) = \{C\} \cup \{D\} = \{C, D\}$$

$$\omega_2(CC) = \{C, D\} \cup \{E\}$$

$$= \{C, D, E\}$$

$$\omega_3(CC) = \omega_2(CC) \cup \phi$$

$$= \{C, D, E\}$$

$$= \omega_1(CC)$$

$$\omega(CC) = \{C, D, E\}$$

$\omega(D)$

$$\omega_0(D) = \{D\}$$

$$\omega_1(D) = \{D\} \cup \{F\}$$

$$= \{D, F\}$$

$$\omega_2(D) = \{D, E\} \cup \phi$$

$$= \omega_1(D)$$

$$\omega(D) = \{D, E\}$$

$\omega(E)$

$$\omega_0(E) = \{E\}$$

$$\omega_1(E) = \omega_0(E) \cup \phi$$

$$= \{E\}$$

$$\omega(E) = \{E\}$$

Step 2:

Construction of S in G_1

$$S \rightarrow AB$$

$$A \text{ in } G_1$$

$$A \rightarrow \alpha$$

$$B \text{ in } G_1$$

$$B \rightarrow b$$

$$C \text{ in } G_1$$

$$C \rightarrow \alpha$$

$$D \text{ in } G_1$$

$$D \rightarrow \alpha$$

$$E \text{ in } G_1$$

$$E \rightarrow \alpha$$

Productions in G_1 are $\alpha \rightarrow AB$

$$\alpha \rightarrow \alpha$$

$$B \rightarrow b/\alpha$$

$$C \rightarrow \alpha$$

$$D \rightarrow \alpha$$

$$E \rightarrow \alpha$$

COROLLARY:

If G_1 is CFG we can construct an equivalent grammar G'_1 which is reduced & has no null productions or unit productions.

proof:

we can construct G'_1 in the following way,

Step 1:

Eliminate null productions to get G_1' ,
(Theorem 4 or Corollary 2) of this theorem.

Step 2:

Eliminate unit productions in G_1' to get G_2 ,
(theorem 5)

Step 3:

Construct a reduced grammar G'_1 equivalent
to G_2 (theorem 3)
 G'_1 is the required grammar equivalent to
 G_1 .

NOTE:

we have to apply the constructions only
in the order given in the Corollary of theorem 5
to simplify grammars. If we change the
order we may not get the grammar in the
most simplified form.

To reduce a CFG we have to follow the
below theorems in given order.

- 1) Elimination of null productions
- 2) elimination of unit productions

3) elimination of variables which are does not derive
any terminal strings.

- 4) elimination of symbols which are not in any
sentential form (terminal/variable).

Normal forms to the CFG:
CHOMSKY NORMAL FORM:

In the Chomsky normal form (CNF),
we have restriction on the length of RHS and
the nature of symbols in the RHS of production

def:

A CFG G_1 is in CNF if every production is
of the form $A \rightarrow a$ (or) $A \rightarrow BC$ and $S \rightarrow A$
is in G_1 if $A \in L(G_1)$. When null is in L(G), we
assume that S does not appear on the RHS
of any production.
for ex, consider S whose productions are

$$S \rightarrow AB/\Lambda, A \rightarrow a, B \rightarrow b \text{ then } G_1 \text{ is in CNF}$$

Remark:

for a grammar in CNF the derivation
tree has the following property:
every node has at most 2 descendants
either 2 internal vertices (variables) or 2
terminal symbols.

When a grammar is in CNF, some of the proofs
and constructions are simple.

Reduction to chomsky normal form

Now we develop a method of constructing

2. *Gymnomyces* C. T. Gilg

Let us first consider an example

Let G be $S \rightarrow ABC/aC, A \rightarrow a, B \rightarrow b, C \rightarrow c$

except $S \rightarrow aC / ABC$ all the productions are in the form required for CNF. The terminals

$S \rightarrow aC$ can be replaced by a new variable D by adding a new production $D \rightarrow a$, the effect of applying $S \rightarrow aC$ can be achieved by $S \rightarrow DC$ and $D \rightarrow a$. $S \rightarrow ABC$ is not in

—the required form, and hence this production can be replaced by $S \rightarrow \Lambda E$ and $E \rightarrow BC$.

Thus an equivalent grammar is $S \rightarrow AE / DC$,
 $E \rightarrow BC$, $A \rightarrow a$, $B \rightarrow b$, $C \rightarrow c$, $D \rightarrow a$

The techniques applied in this study are used in

The following theorem (reduction to CNF)

for every CFG_1 , there is an equivalent

grammar G_1 in CNF

proof: Construction of grammar in CNF

Step 1: Define the unit conditions.

we apply theorem 4 to eliminate null & elimination of null

production, we then apply theorem 5 to eliminate the resulting grammar by the chain (unit) productions

Elimination of ε we define $G_1 = (V_N^1, \varepsilon, p_1, s)$ where p_1 and V_N^1 are constructed as follows:

are constructed as follows:

(i) All the productions in P of the form $A \rightarrow \alpha$ or $A \rightarrow BC$ are included in P_1 , all the variables in V_N are included in V_1 .

(iii) Consider $A \rightarrow x_1 x_2 \dots x_n$ with some terminals on R.H.S if x_i is a terminal say a_i add a new variable C_{a_i} to V_N' and $C_{a_i} \rightarrow a_i \rightarrow p_i$

In production $A \rightarrow x_1 x_2 \dots x_n$ every terminal on R.H.S is replaced by the corresponding new variable and the variables on the R.H.S are retained. The resulting production is added to P_1 . Thus we get $\tilde{G}_{11} = (V_N', Z, P_1, S)$

Step 3: Restricting the no. of variables on R.H.S
for any production in P, the R.H.S consists of either a single terminal or (λ in $S \rightarrow A$) two or more variables. we define $G_{12} = (V_N'', \in, P_2, S)$.

as follows:

- i) All productions in P_1 are added to P_2 if they are in the required form. All the variables in V_1 are added to V_2 .

(ii) Consider $A \rightarrow A_1 \wedge A_2 \dots \wedge A_m$ where $m \geq 5$.
we introduce new productions $A \rightarrow A_1 G_1$

$c_1 \rightarrow \lambda_2 c_2 \dots c_{m-2} \rightarrow$
new variables c_1, c_2, c_{m-2}

These are added to P_2'' and V_N''' respectively. Thus we get G_2 in CNF.

Reduce the following grammar G_1 to CNF

$$G_1 \text{ is } S \rightarrow aAD, A \rightarrow \underline{c_1B} / bAB, \underline{B \rightarrow b}, D \rightarrow d$$

In a given grammar there are no null productions and unit productions

Step 2

$$(a) \quad B \rightarrow b, D \rightarrow d$$

(b) $S \rightarrow aAD$ is replaced by $S \rightarrow c_aAD$, $c_a \rightarrow a$

$A \rightarrow aB$ is replaced by $A \rightarrow c_aB$.

$A \rightarrow bAB$ is replaced by $A \rightarrow c_bAB$,

$$c_b \rightarrow b..$$

$$V_N'''(S, A, B, D, c_a, c_b)$$

Step 3:

$$B \rightarrow b, D \rightarrow d$$

$$A \rightarrow c_aB, c_a \rightarrow a, c_b \rightarrow b$$

$$A \rightarrow c_bC_2, C_2 \rightarrow AB$$

$$V_N''' = (S, A, B, D, c_a, c_b, C_1, C_2).$$

$$A \rightarrow aA, B \rightarrow bB$$

$$A \rightarrow aB$$

$$\text{eg, } A \rightarrow AC_aBC_bDE$$

$$c_1 = c_aBC_bDE$$

$$c_2 = \underline{BC_bDE}$$

$$c_3 = c_bDE$$

$$c_4 = DE$$

$$S \rightarrow aAb \cancel{B}, A \rightarrow a \cancel{A/a}, B \rightarrow b \cancel{B/a}$$

Step 2:

$$(a) \quad A \rightarrow a, B \rightarrow a$$

(b) $S \rightarrow aAbB$ is replaced by $S \rightarrow c_aAC_B$

$$c_a \rightarrow a, c_b \rightarrow b$$

$A \rightarrow aA$ is replaced by $A \rightarrow c_aA$
 $B \rightarrow bB$ is replaced by $B \rightarrow c_bB$

$$V_N''' = (\pi, A, B).$$

Step 3:

$$A \rightarrow c_aA, B \rightarrow c_bB$$

$$S \rightarrow c_a \cancel{c_b}AB, c_a \cancel{AC_B} \rightarrow AB$$

$$A \rightarrow c_bC_2, C_2 \rightarrow AB$$

$$V_N''' = (S, A, B, D, c_a, c_b, C_1, C_2).$$

$$A \rightarrow aA, B \rightarrow bB$$

$$A \rightarrow aB$$

$$A \rightarrow aB$$

GIREBACH NORMAL FORM:

def:

A_{CFG} is in GNF if every production is of the form $A \rightarrow \alpha\alpha$ where $\alpha \in V_N^*$ and $\alpha \in \Sigma$ (α may be λ) and $S \rightarrow A$ is in G' if $A \in L(G)$ when

$\Lambda \in L(G)$ we assume that 'S' does not appear on the R.H.S of any production. For eg,

G_1 given by $S \rightarrow aAB/\lambda$, $A \rightarrow bC$, $B \rightarrow b$,

$C \rightarrow c$ is in GNF.

lemma 1

let $G_1 = (V_N, \Sigma, P, S)$ be a CFG, let $\Lambda \rightarrow B\beta$ be an A -production in P . Let the B -productions be $B \rightarrow \beta_1/\beta_2 \dots / \beta_s$ define $P_1 = (P - \{\Lambda \rightarrow B\beta\})$

$$\cup \{A \rightarrow \beta_i \mid 1 \leq i \leq s\}$$

then $G_1 = (V_N, \Sigma, P_1, S)$ is a CFG equivalent to G_1 .

Note:

lemma 2 is useful for deleting a variable 'B' appearing as the first symbol on the R.H.S of A -productions, provided no B -productions has B as the first symbol on R.H.S.

The construction given in lemma 2 is

simple. To eliminate B in $A \rightarrow B\beta$ is simply replaced B by the right hand side of every

B -production

for eg, we can replace $A \rightarrow Bab$ by $A \rightarrow aAab$, $A \rightarrow bBab$, $A \rightarrow aaab$, $A \rightarrow ABab$ when the B -productions are aA , bB , aaB , AB .

If $\Lambda \rightarrow \alpha_1\alpha_2/\alpha_3 \dots / \alpha_n/\beta_1/\beta_2 \dots / \beta_s$ then

$$\Lambda \rightarrow \beta_1/\beta_2 \dots / \beta_s$$

$$\Lambda \rightarrow \beta_1z_1/\beta_2z_2$$

$$\Lambda \rightarrow \alpha_1/\alpha_2$$

$$\Lambda \rightarrow \alpha_1z_1/\alpha_2z_2$$

lemma 2:

let $G_1 = (V_N, \Sigma, P, S)$ be CFG, let the set of Λ -productions be $\Lambda \rightarrow A\alpha_1/\alpha_2 \dots / \alpha_n/\beta_1/\beta_2 \dots / \beta_s$ (β_i don't start with Λ)

Let Σ be a new variable. Let $G_1 = (V_N \cup \{\Sigma\}, \Sigma, P_1, S)$ where P_1 is defined as follows,

- (1) The set of Λ -productions in P_1 are $\Lambda \rightarrow \beta_1/\beta_2 \dots / \beta_s$.
- (2) If $\Lambda \rightarrow A\alpha_1/\alpha_2 \dots / \alpha_n$ is an A -production in P then $\Lambda \rightarrow A\alpha_1\Sigma/\alpha_2\Sigma \dots / \alpha_n\Sigma/\beta_1\Sigma \dots / \beta_s\Sigma$

The set of Σ -productions in P_1 are

$$\Sigma \rightarrow \alpha_1\Sigma/\alpha_2\Sigma \dots / \alpha_n\Sigma$$

(iii) The productions for the other variables are as in P

then G_1 is a CFG₁ and equivalent to G_1 .

Apply lemma 2 to the following A-productions in a CFG₁ in G_1 .

$$A \rightarrow aBD / bDB / c / AB / AD.$$

$$(i) A \rightarrow aBD / bDB / c$$

$$A \rightarrow aBDz / bDBz / cz$$

$$(ii) z \rightarrow B / D$$

$$z \rightarrow Bz / Dz$$

CONVERSION OF CNF TO GNF :

Step 1:

Eliminate null & unit productions (then convert it into CNF which are not in required form).

Convert the variables as A_1, A_2, \dots, A_n with $\{S, Z, P\}$. Now G_1 has $\{A_1, A_2, \dots, A_n, S, Z, P, A_1\}$.

Step 2:

Convert the productions of the form $A_i \rightarrow A_j \delta$ where $i > j$ into $A_i \rightarrow A_j \delta'$ by applying lemma 2 iteratively.

Step 3: $S \rightarrow A_1 A_2 \dots A_n S \rightarrow A_1 A_2 \dots A_n$

Apply the lemma 2 for the productions

$$\text{of the form } A_i \rightarrow A_i \delta^*$$

Step 4:

Find out the other productions which are not in required form and apply previous steps.

Step 5:

Replace the left most symbol (variable) in RHS of a variable Z_i .

Step 1:

Construct a grammar in GNF equivalent to the grammar $S \rightarrow AA/a, A \rightarrow SS/b \rightarrow$

Step 1:

The given grammar has no null & unit productions so, it is in CNF.

Step 2: Rename the variables as S as A_1 and A as A_2 , so,

A_1

$$A_1 \rightarrow (A_2 A_2) / b$$

CNF

Step 2:

$A_1 \rightarrow A_2 A_2 / a$ & $A_2 \rightarrow b$ are required form

Consider $A_2 \rightarrow A_1 A_1$

by applying lemma 1 we get,

$$A_2 \rightarrow A_1 A_1$$

$$A_2 \rightarrow \underline{\underline{A_2}} A_2 / a A_1$$

Step 3:

$$\text{Applying lemma 2, } A_2 \rightarrow \underline{\underline{A_2}} A_2 A_1$$

$$(i) A_2 \rightarrow aA_1, \quad A_2 \rightarrow aA_1z_2 \xrightarrow{(i)} A_2 \rightarrow b.$$

$$(ii) z_2 \rightarrow A_2A_1; z_2 \rightarrow A_2A_1(z_2)$$

Step 4:

The production $A_1 \rightarrow a$ is in the required form

and $A_1 \rightarrow A_2A_1$ is not in the required form so, apply lemma 1

$$A_1 \rightarrow aA_1A_2 / aA_1z_2 A_2 \xrightarrow{(2)}$$

$$A_1 \rightarrow a$$

Step 5:

Consider the production $z_2 \rightarrow A_2A_1 / A_2A_1z_2$ to apply lemma 1,

$$\begin{aligned} z_2 &\rightarrow aA_1A_1 / aA_1z_2 A_1 \\ z_2 &\rightarrow aA_1A_1z_2 / aA_1z_2 A_1 \end{aligned} \quad \xrightarrow{(3)}$$

Convert the grammar to GNF $B \rightarrow a, C \rightarrow a$

$$S \rightarrow A^B, A \rightarrow \overset{a}{\textcircled{C}}, \underset{B \rightarrow \textcircled{C}}{\textcircled{B} \rightarrow \textcircled{D}}, \underset{C \rightarrow \textcircled{D}}{\textcircled{C} \rightarrow \textcircled{E}}, \underset{D \rightarrow \textcircled{E}}{\textcircled{D} \rightarrow E}, E \rightarrow a$$

Convert the grammar to GNF by applying lemma 1,

$$S \rightarrow \underline{AB}, A \rightarrow \underline{BSB}, A \rightarrow \underline{BB}, B \rightarrow \underline{AAB}$$

$$B \rightarrow a, A \rightarrow b$$

Convert the grammar to GNF
 $S \rightarrow AB, A \rightarrow BS / b, B \rightarrow SA / a.$

Step 1:

Given grammar does not have unit & null productions and it is in CNF

Replace the variables, S, A, B as A_1, A_2, A_3 respectively.

The productions are

$$A_1 \rightarrow A_2A_3$$

$$A_2 \rightarrow A_3A_1 / b$$

$$A_3 \rightarrow A_1A_2 / a$$

Step 2:

The production $A_1 \rightarrow A_2A_3$ are in required form
 The production $A_2 \rightarrow A_3A_1 / b$ & $A_3 \rightarrow a$ is in required form but the production $A_3 \rightarrow A_1A_2$ is not.

by applying lemma 1,

$$A_3 \rightarrow A_2A_3A_2 \quad (\text{not})$$

by applying lemma 1,

$$A_3 \rightarrow A_3A_1A_3A_2$$

$$A_3 \rightarrow A_3 b A_3 A_2$$

$$A_3 \rightarrow A_3 A_2 A_3 A_2 \leftarrow \text{not}$$

Step 3:

Apply lemma 2 for the production $A_3 \rightarrow A_3 A_1 A_3 A_2$

$$A_3 \rightarrow b A_3 A_2$$

$$A_3 \rightarrow b A_3 A_2 z_3 \quad (1)$$

$$z_3 \rightarrow A_1 A_3 A_2$$

$$z_3 \rightarrow A_1 A_3 A_2 z_3$$

Step 4:

$A_2 \rightarrow b$ is in the required form but $A_2 \rightarrow A_3 A_1$ is not the GNF

$$A_2 \rightarrow A_3 A_1$$

Applying lemma 1,

$A_2 \rightarrow b A_3 A_2 A_1 / b A_3 A_2 z_3 A_1$ and

$$A_2 \rightarrow b \quad (2)$$

$A_1 \rightarrow A_2 A_3$ is not in GNF

Applying lemma 1,

$$A_1 \rightarrow b A_3 A_2 A_1 A_3$$

$$z_3 \rightarrow b A_3 A_2 z_3 A_1 A_3 \rightarrow (3)$$

Step 5 \cap \cup \leftarrow \rightarrow

$$z_3 \rightarrow b A_3 A_2 A_1 A_3 A_2$$

$$z_3 \rightarrow b A_3 A_2 z_3 A_1 A_3 A_2$$

$$z_3 \rightarrow b A_3 A_2$$

$$\begin{cases} z_3 \rightarrow b A_3 A_2 A_1 A_3 A_2 z_3 \\ z_3 \rightarrow b A_3 A_2 z_3 A_1 A_3 A_2 z_3 \end{cases}$$

Properties of CFL languages

1) CFL's are closed under union.

e.g.,

Let L_1 & L_2 are CFL's and s_1 and s_2 are start symbols for the languages L_1 and L_2 respectively.

$\therefore S \rightarrow s_1 / s_2$ is also a CFL

2) CFL's are closed under Concatenation

$$S \rightarrow s_1 s_2$$

3) CFL's are closed under closure.

$$S \rightarrow s_1^*$$

$$S \rightarrow s_1 s / \wedge$$

4) CFL's are closed under reversal.

5) CFL's are closed under cyclic shift

e.g.,
 $S \rightarrow abcb$ by cyclic shift $S \rightarrow bacab$.

is also CFL.

$a\beta\gamma\delta\rightarrow\alpha\beta\gamma$ by cyclic shift $S \rightarrow \beta\alpha\beta\gamma$ is also CFL

also CFL
since $\alpha\beta\gamma\delta$ is also CFL
and $\beta\alpha\beta\gamma$ is also CFL

- 6) CFL's are not closed under intersection.

⁷⁾ Cf. 's are not closed under difference.

8) CFL's are not closed under Complement

APPLICATIONS OF CFG:

- 1) Grammars are useful in specifying syntax of programming languages. They are mainly used in design of programming languages.

2) They are also used in natural language processing.

3) Tamil poetry called Venpa is described by a CFG.

4) CFG's are used in speech recognition also in processing the spoken word.

5) The expressive power of CFG is too limited to adequately capture all natural language phenomena. Therefore extensions of CFG are of interest for Computational linguistics.

LMD & RMD represents different parse trees

NDE, RND represents same parse tree.

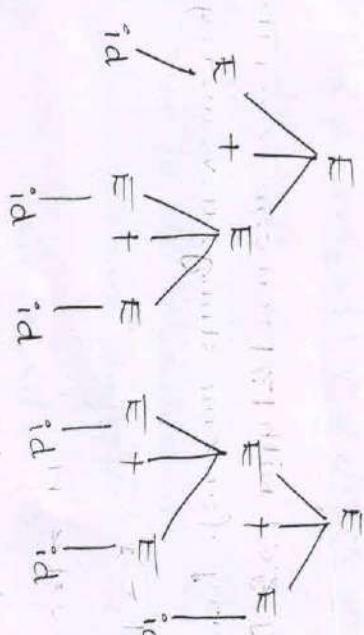
More than a Parsee
tree for a string

$$E \rightarrow id / E+E / E * E / E - E$$

卷之三

$$E \rightarrow id + E + E$$

$$E \rightarrow id + id + id$$



The string id + id + id has '2' derivation
so the string Did + id + id is ambiguous

2) A ~~leg~~ ^{leg} ~~leg~~ ^{leg}

THREE **CRIMES**

Unique LMD/RMD

PUMPING LEMMA FOR CFG_i

Let L be a CFG, then we can find a natural number n such that

$\rightarrow a^n b^n$ is a CFG, i.e., $S \rightarrow a^n b^n$.

$\rightarrow a^n b^n \rightarrow$ not a CFG

$S \rightarrow a^k b^k$

wu^k

$\rightarrow w \in (a, b)^*$

$S \rightarrow a^k a^k / b^k b^k / a^k b^k / a^k b^k$

It is a CFG.

(1) Every $z \in L$ with $|z| \geq n$ can be written as

~~uvwy~~ for some strings u, v, w, y .

(2) $|vz| \geq 1$

(3) $|vwz| \geq n$

(4) $uv^k wz^k \in L \quad \forall k \geq 0$.

We use the Pumping lemma to show that a language L is not a CFL. We assume that L is a CFL. By applying the Pumping lemma we get a contradiction.

The procedure can be followed by using the following steps:

Step 1

Assume L is CF let n' be the natural number obtained by using the Pumping lemma

Step 2

choose $z \in L$ so that $|z| \geq n'$ while $z = uvwyz$ using the Pumping lemma

Step 3

Find a suitable k so that $uv^k wz^k \notin L$. This is a contradiction and so, L is not CFG.

Show that $L = \{a^p / p \text{ is a prime}\}$ is not a CFG.

Step 1

Suppose $L = L(G)$ is a CFG let n' be the natural number obtained by using the Pumping lemma

Step 2

Let p be a prime number $\geq n'$

then $z = a^p \in L$

we write $z = uvwyz$ then

the condition of pumping lemma is satisfied
Step 3: Let $k \geq 0$ then the string
 $u^k v^k w^k y^k = u^k v^k y^k = u^k w^k y = e$

$|uvwy| = q$ ($\because q$ is prime)

assume $|vz| = n$

then apply assume $k \geq q$

then $uv^{k+1}wz^qy$

$$|uv^{k+1}wz^qy| = |uvwy| + |v^qz^q|$$

$$= q + qn$$

$$= q(1+n)$$

This is not a prime number so, $uv^{k+1}wz^qy \notin L$. This is a contradiction so, L is not a CFL.

DECISION ALGORITHMS:

Algorithm 1: for determining if a given CFL is empty

After application of theorem 1, if set S is empty then the given CFL is not empty otherwise the CFL is empty

Algorithm 2:

Algorithm for determining if a CFL is infinite.

Construct a non redundant CFG G_1 in CNF.
 $L = \{g_1^3\}$, $L = \{g_1^3\}$ is infinite

We draw a directed graph whose vertices are variables in grammar $A \rightarrow B$ is a production
 $(A \rightarrow B) \rightarrow C \rightarrow D \rightarrow \dots$

It is a finite iff the directed graph has no cycles.

Algorithm 3:

Algorithm for deciding whether a regular language L is empty

Construct a deterministic FA M accepting L initial state q_0 as below

we find the states which are reachable from q_0 by applying a single input symbol. These states are arranged as a row under columns corresponding to every input symbol.

The construction is repeated for every state appearing in an earlier row.

The construction terminates in a finite no. of steps. If a final state appears in this table, column (state heading) then L is non-empty otherwise L is empty.

Algorithm 4:

Algorithm for deciding whether a regular language L is infinite.

Construct a deterministic FA M (transition diagram) accepting L . L is infinite iff it has a cycle.

UNIT - IV

PUSHDOWN AUTOMATA

A pushdown automata consists of 7-tuples

- (i) a finite non-empty set of states denoted by Q
- (ii) a finite non-empty set of input symbols denoted by Σ
- (iii) a finite non-empty set of pushdown symbol denoted by Γ
- (iv) a special state called the initial state denoted by q_0
- (v) a special push down symbol called -the initial symbol on the pushdown store (pds) denoted by z_0
- (vi) a set of final states denoted by F ,
 $F \subseteq Q$
- (vii) a transition function δ from $Q \times (\Sigma \times \{ \lambda \})$ to the set of finite subsets of $Q \times \Gamma^*$

Symbolically, a pda is a 7 tuple namely
 $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

PDA has read only input tape, an input alphabet, a finite store control, a set of final states and an initial state as in the case of an FA

①

Example for $a^n b^n$:

transition function rules:

$$\delta(q_0, a, z_0) = (q_0, az_0)$$

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

$$\delta(q_0, b, a) = (q_1, \lambda)$$

$$\delta(q_1, b, a) = (q_1, \lambda)$$

$$\delta(q_1, \lambda, z_0) = (q_f, \lambda)$$

e.g., aaabbb

$$\delta(q_0, aaabbb, z_0) = \delta(q_0, aabbb, az_0)$$

$$= \delta(q_0, abbb, aaaz_0)$$

$$= \delta(q_0, bbb, aaaz_0)$$

$$= \delta(q_1, bb, aaaz_0)$$

$$= \delta(q_1, b, az_0)$$

$$= (q_1, \lambda, z_0)$$

$$= (q_f, \lambda).$$

Equal no. of a's and 1's.

$$S \rightarrow 1A / 0B$$

$$A \rightarrow 0 / 0S / 1AA$$

$$B \rightarrow 1 / 1S / 0BB$$

Obtain the transition rules for equal no. of a's and b's.

$$\delta(q_0, a, z_0) = (q_0, az_0)$$

① ②

$$\delta(q_0, b, z_0) = (q_0, bz_0)$$

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

$$\delta(q_0, b, b) = (q_0, bb)$$

$$\delta(q_0, a, b) = (q_0, \lambda)$$

$$\delta(q_0, b, a) = (q_0, \lambda)$$

$$\delta(q_0, \lambda, z_0) = (q_f, \lambda)$$

Design PDA for $0^n 1^{2n}$ where $n \geq 1$

let q_0 be the initial state, q_f be final state
and z_0 be bottom of the stack

$$\delta(q_0, 0, z_0) = (q_0, 0z_0)$$

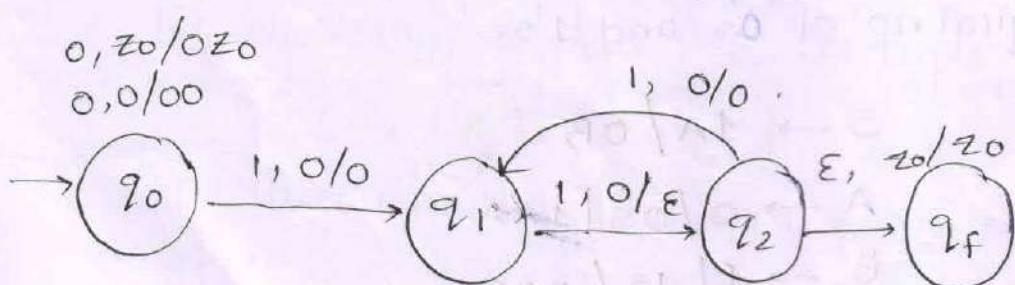
$$\delta(q_0, 0, 0) = (q_0, 00)$$

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

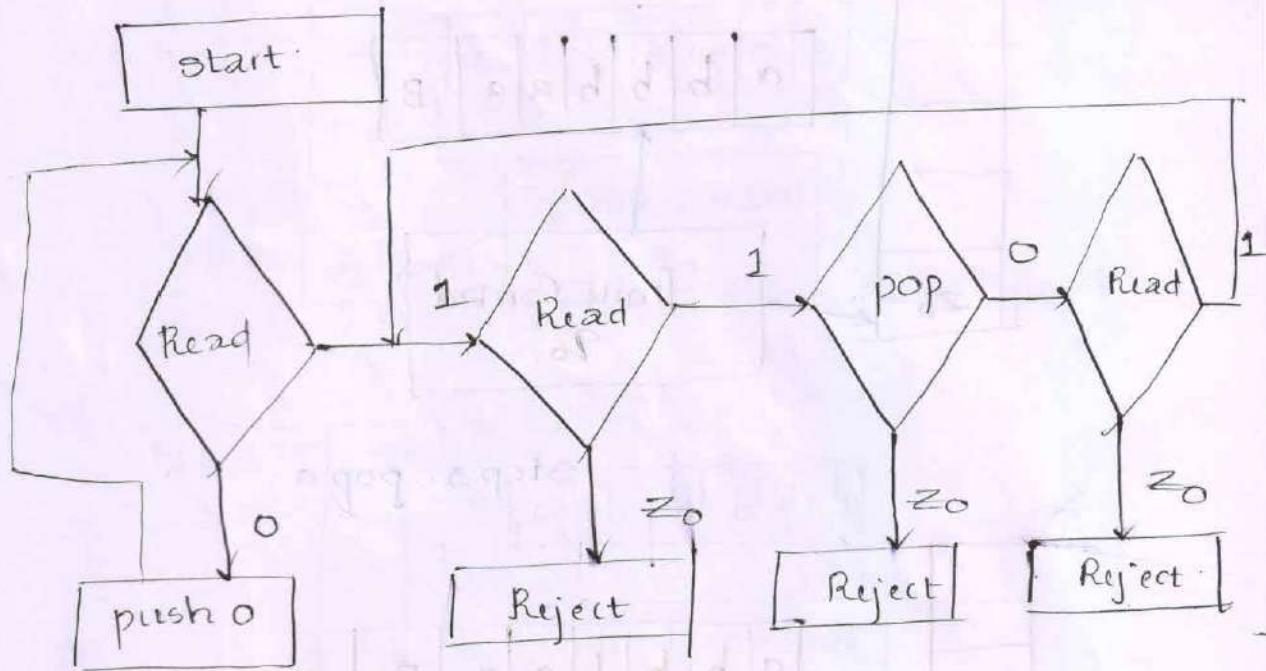
$$\delta(q_1, 1, 0) = (q_2, \epsilon)$$

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

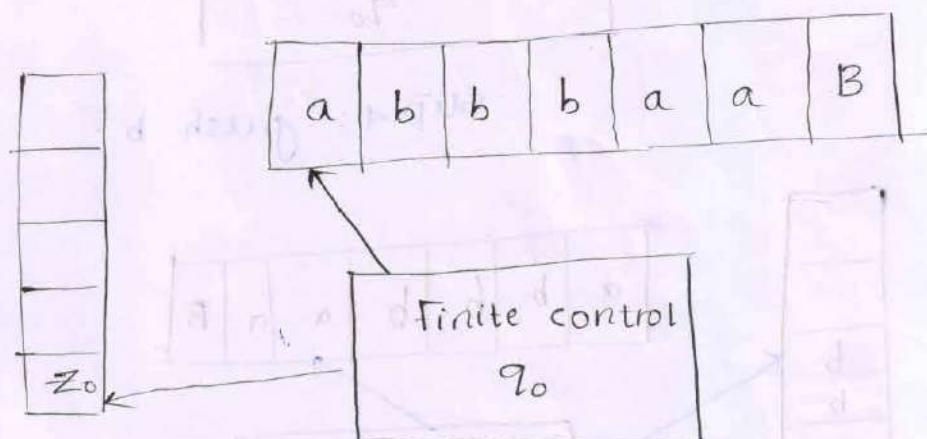
$$\delta(q_2, \epsilon, z_0) = (q_f, z_0)$$



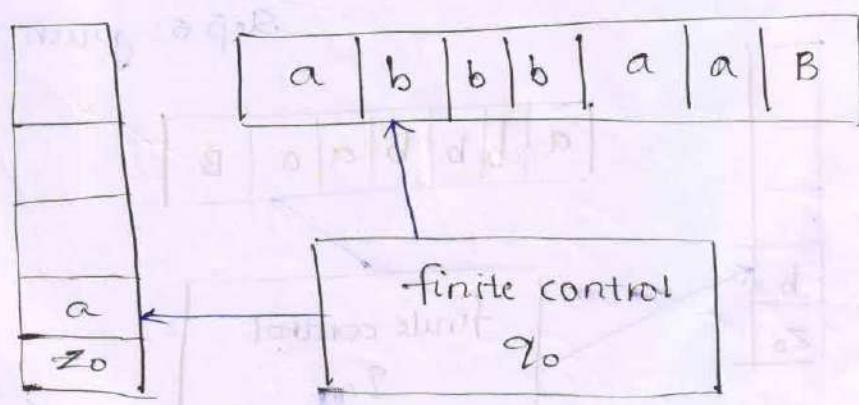
③



equal no. of a's and equal no. of b's.

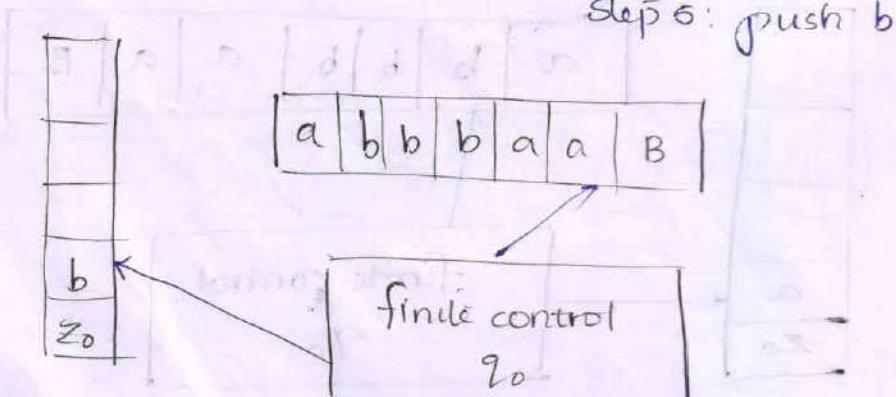
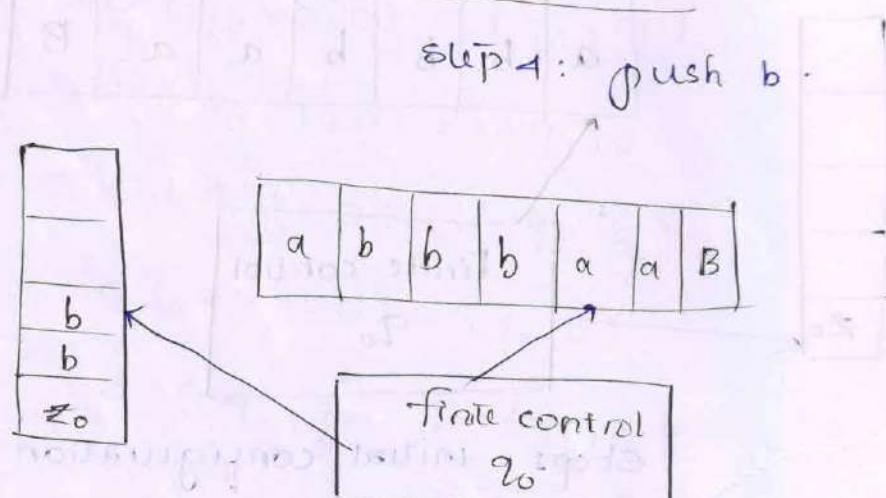
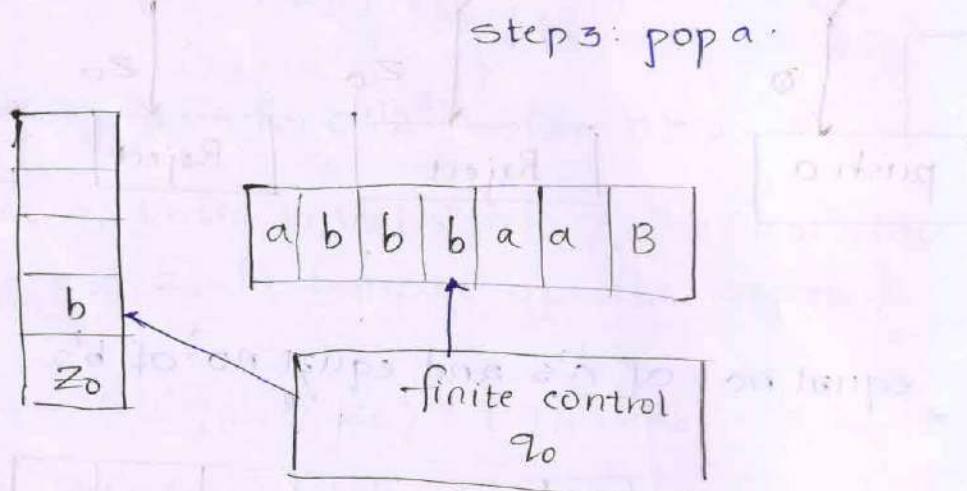
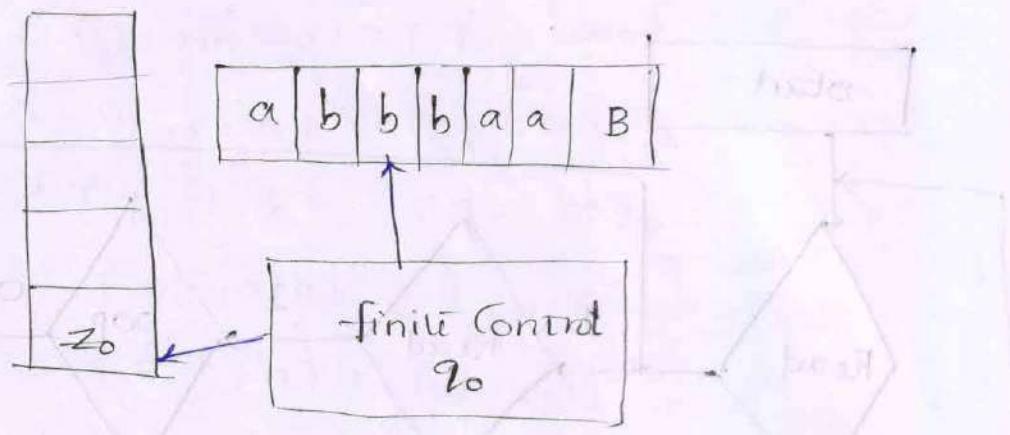


Step 1: initial configuration

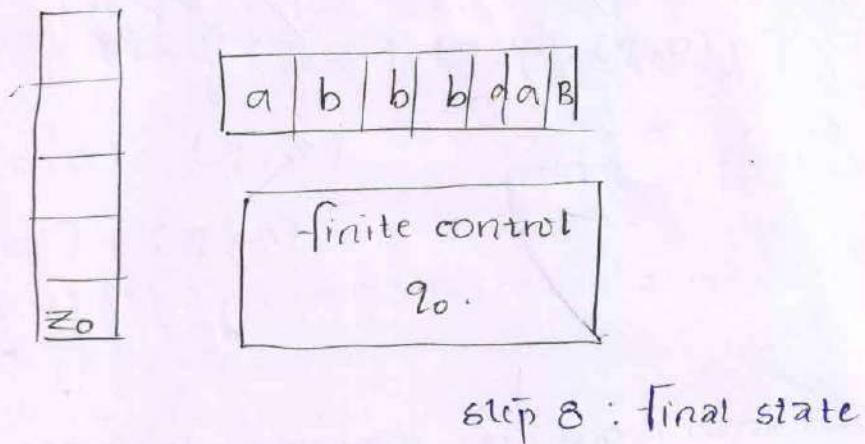
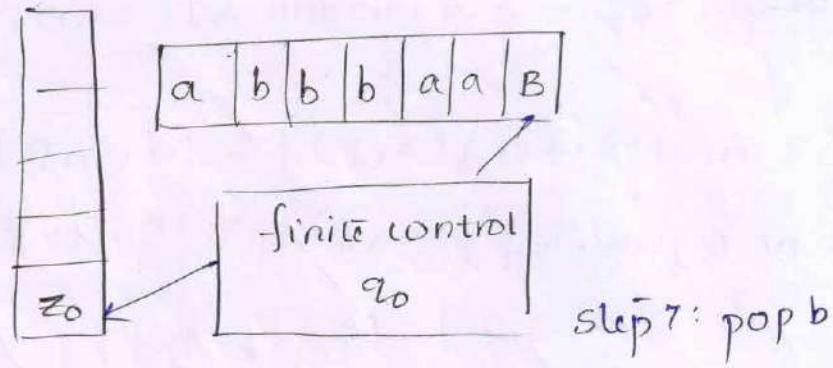


Step 2: a is pushed

2 4



④ ⑤



Convert the CFG1 to PDA the CFG1 is $S \rightarrow OB\bar{B}$, $B \rightarrow OS/IS/O$.

$$R_1: \delta(q, \lambda, A) = \{(q, \alpha) / A \rightarrow \alpha \text{ is in } P\}$$

$$R_2: \delta(q, \alpha, A) = \{(q, \lambda) / \text{for every } \alpha \in \Sigma\}$$

$$\delta(q, \lambda, S) = (q, OB\bar{B})$$

$$\delta(q, \lambda, B) = \{(q, OS), (q, IS), (q, O)\}$$

$$\delta(q, O, O) = (q, \lambda)$$

$$\delta(q, I, I) = (q, \lambda)$$

STEP 2:

Any sentential form in a Left most derivation is of the form $UA\alpha$, where $U \in \Sigma^*$, $A \in V_N$ and $\alpha \in V_N \cup \Sigma^*$

If $S \xrightarrow{*} UA\alpha$ by a left most derivation - then,

$$(q, UV, S) \xrightarrow{*} (q, V, A\alpha)$$

e.g., $S \rightarrow aABA$

$$(i) (q, aaA, S) \xrightarrow{*} (q, A, BA)$$

$$(ii) (q, aAa, S) \xrightarrow{*} (q, Aa, aba)$$

CONVERSION FROM PDA TO CFG1: If $A(Q, \Sigma, \Gamma, \delta)$,

is a PDA \Rightarrow Then there exist a CFG1 such that

$$L(G1) = N(A)$$

Step 1: $(Q, \Sigma, \Gamma, \delta) \xrightarrow{*} (P, S, P)$

Construction of $G1$ we define $G1(V_N, \Sigma, P, S)$

where $V_N = \{S\} \cup \{(q, z, q') / q, q' \in Q, z \in \Sigma\}$

If $Q = \{q_0, q_1\}$

$Z = \{z_0, z_1\}$

$$\delta(q_1, a, z) \cdot (q^1) \rightarrow \delta(q_1, z, q^1)$$

The non-terminals of V_N is $S, [q_0, z_0, q_0]$,
by $[q_0, z_0, q_1], [q_0, z_0, q_1], [q_1, z_0, q_0],$
 $[q_0 z_1, q_0], [q_0, z_1, q_1], [q_1, z_1, q_1]$.
 $[q_1, z_1, q_0]$.

The productions are

R_1 : S -productions are given by $S \rightarrow [q_0, z_0, q]$ for
every q in Q .

ex: $S \rightarrow [q_0, z_0, q_0]$

$S \rightarrow [q_0, z_0, q_1]$

R_2 : each move erasing a push down symbol given by
 $(q^1, \lambda) \in \delta(q, a, z)$ induces the production.

$[q_1, z_1, q^1] \rightarrow a$

ex: $\delta(q_0, a, z_0) = (q_1, \lambda)$

$[q_0, z_0, q_1] \rightarrow a$

R_3 : Each move not erasing a push down symbol given
by $(q_1, z_1 z_2, \dots, z_n) \in \delta(q, a, z')$
induces many productions of the
form $[q_1, z_1, q^1] \rightarrow a [q_1, z_1, q_2] [q_2, z_2, q_3]$

⑧

where each of the states q^1, q_2, \dots, q_m

can be any state in Q .

ex: $Q = \{q_0, q_1\}$

$$\Sigma = \{z_0, z\}$$

$$\delta(q_0, b, z_0) = (q_0, zz_0)$$

$$[q_0, z_0, q_0] \rightarrow b[q_0, z, q_0][q_0, z, q_0]$$

$$[q_0, z_0, q_0] \rightarrow b[q_0, z, q_1][q_1, z, q_0]$$

Construct a PDA accepting $a^n b^m a^n$ $m, n \geq 1$.

Construct the corresponding CFG accepting the same sets.

$$\delta(q_0, a, z_0) = (q_0, az_0)$$

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

$$\delta(q_0, b, a) = (q_1, a)$$

$$\delta(q_1, b, a) = (q_1, a)$$

$$\delta(q_1, a, a) = (q_2, \lambda)$$

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

$$\delta(q_2, \lambda, z_0) = (q_f, \lambda)$$

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

$$\Sigma = \{z_0\}$$

R₁: S-productions are given by $S \rightarrow [q_0, z_0, z]$
for every q in Q

$$S \rightarrow [q_0, z_0, q_0] / [q_0, z_0, q_1] / [q_0, z_0, q_2]$$

①

$$[q_0, z_0, q_f]$$

$$\delta(q_0, a, z_0) = (q_0, az_0) \text{ gives}$$

~~$[q_0, z_0, q_0] \rightarrow a[q_0, a, q_1][q_1, z_0, q_0]$~~
 $\overbrace{[q_0, z_0, q_1] \rightarrow a[q_0, a, q_2][q_2, z_0, q_1]}$

$$\delta(q_0, a, a) = (q_0, aa) \text{ gives}$$

$$[q_0, a, q_0] \rightarrow a[q_0, a, q_1][q_1, a, q_0].$$

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

$$[q_2, a, q_2] \rightarrow a$$

$$\delta(q_1, a, a) = (q_2, \wedge)$$

$$[q_1, a, q_2] \rightarrow a$$

$$\delta(q_2, \wedge$$

$$\{ab, ba\} = ?$$

$$\{ab, ab\} = ?$$

(Please) can you prove the correctness of
your solution

Prop

P₁₅

P₁₆

P₁₇

(10)

(P)

(11)

$$P_1: S \rightarrow [q_0, z_0, q_0] / [q_0, z_0, q_1] / [q_0, z_0, q_2] / [q_0, z_0, q_b]$$

$\delta(q_0, a) \cdot z_0 = (q_0, \underline{a} \underline{z_0})$ gives

$$P_2: [q_0, \underline{z_0}, q_0] \rightarrow a[q_0, a, \underline{q_0}] [q_0, \underline{z_0}, q_0]$$

$$P_3: [q_0, z_0, q_0] \rightarrow a[q_0, a, \underline{q_1}] [\underline{q_1}, z_0, q_0]$$

$$P_4: [q_0, z_0, q_0] \rightarrow a[q_0, a, q_2] [q_2, z_0, q_0]$$

$$P_5: [q_0, z_0, q_0] \rightarrow a[q_0, a, q_b] [q_b, z_0, q_0]$$

$$P_6: [q_0, z_0, q_1] \rightarrow a[q_0, a, q_0] [q_0, z_0, q_1]$$

$$P_7: [q_0, z_0, q_1] \rightarrow a[q_0, a, q_1] [q_1, z_0, q_1]$$

$$P_8: [q_0, z_0, q_1] \rightarrow a[q_0, a, q_2] [q_2, z_0, q_1]$$

$$P_9: [q_0, z_0, q_1] \rightarrow a[q_0, a, q_b] [q_b, z_0, q_1]$$

$$P_{10}: [q_0, z_0, q_2] \rightarrow a[q_0, a, q_0] [q_0, z_0, q_2]$$

$$P_{11}: [q_0, z_0, q_2] \rightarrow a[q_0, a, q_1] [q_1, z_0, q_2]$$

$$P_{12}: [q_0, z_0, q_2] \rightarrow a[q_0, a, q_2] [q_2, z_0, q_2]$$

$$P_{13}: [q_0, z_0, q_2] \rightarrow a[q_0, a, q_b] [q_b, z_0, q_2]$$

$$P_{14}: [q_0, z_0, q_b] \rightarrow a[q_0, a, q_0] [q_0, z_0, q_b]$$

$$P_{15}: [q_0, z_0, q_b] \rightarrow a[q_0, a, q_1] [q_1, z_0, q_b]$$

$$P_{16}: [q_0, z_0, q_b] \rightarrow a[q_0, a, q_2] [q_2, z_0, q_b]$$

$$P_{17}: [q_0, z_0, q_b] \rightarrow a[q_0, a, q_b] [q_b, z_0, q_b]$$

$\delta(q_0, b, a) = (q_1, a)$ gives.

$$P_1 : [q_0, \alpha, q_0] = b[q_1, \alpha, q_0]$$

$$P_2 : [q_0, a, q_1] = b[q_1, a, q_1]$$

$$P_3: [q_0, a, q_2] = b[q_1, a, q_2]$$

$$P_1: [q_0, a, q_f] = b[q_1, a, q_f]$$

Turing Machine

Turing machine is represented by 3-types

- (1) ID (Instantaneous description) using move relations
- (2) Transition table
- (3) Transition diagram (Transition grammar).

ID

Moves in turing machine

$$\delta(q_i, x_i) = (p_i, y, L)$$

$x_1 x_2 \dots \dots x_{i-1} q x_i \dots \dots x_n$

$$\delta(q_i, x_i) = (p_i, y, L)$$

$x_1 x_2 \dots p x_{i-1}, y \dots \dots x_n$

$$\delta(q_i, x_i) = (p, y, R)$$

$x_1 x_2 \dots \dots x_{i-1} q x_i \dots \dots x_n$

$x_1 x_2 \dots \dots x_{i-1} y p \dots \dots x_n$

TRANSITION TABLE :

Present state	b	0	1
$\rightarrow q_1$	$1L q_2$	$0R q_1$	
q_2	$bR q_3$	$0L q_2$	$1L q_2$
q_3		$bR q_4$	$bR q_5$
q_4	$0R q_5$	$0R q_4$	$1R q_4$
q_5	$0L q_2$		

(13)

(14)

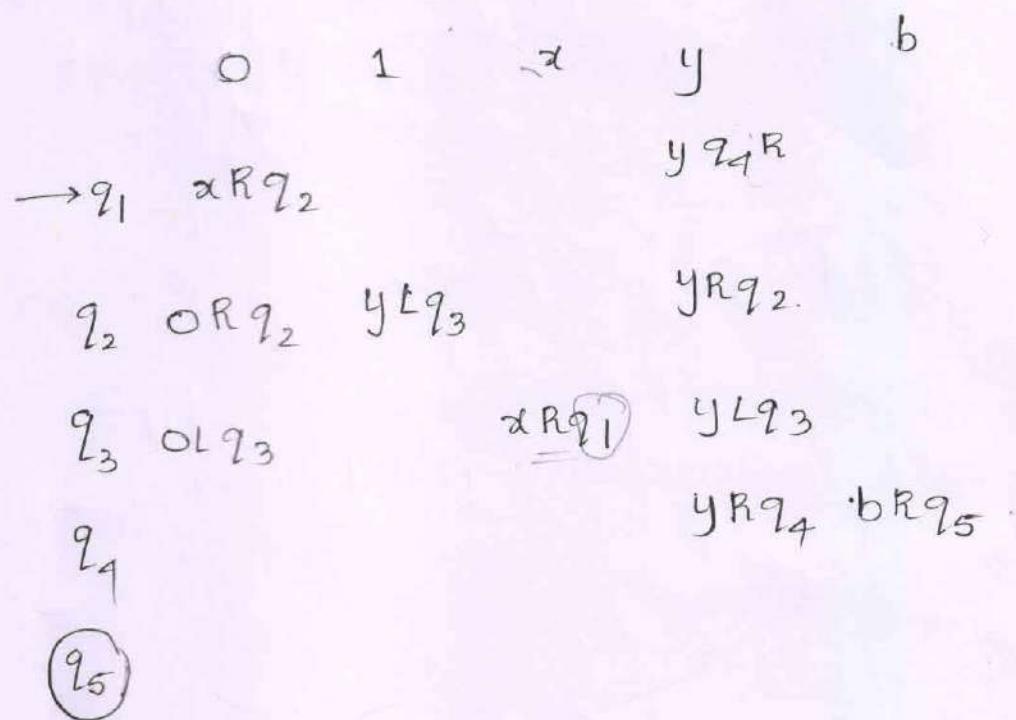
standard print

NOTE:

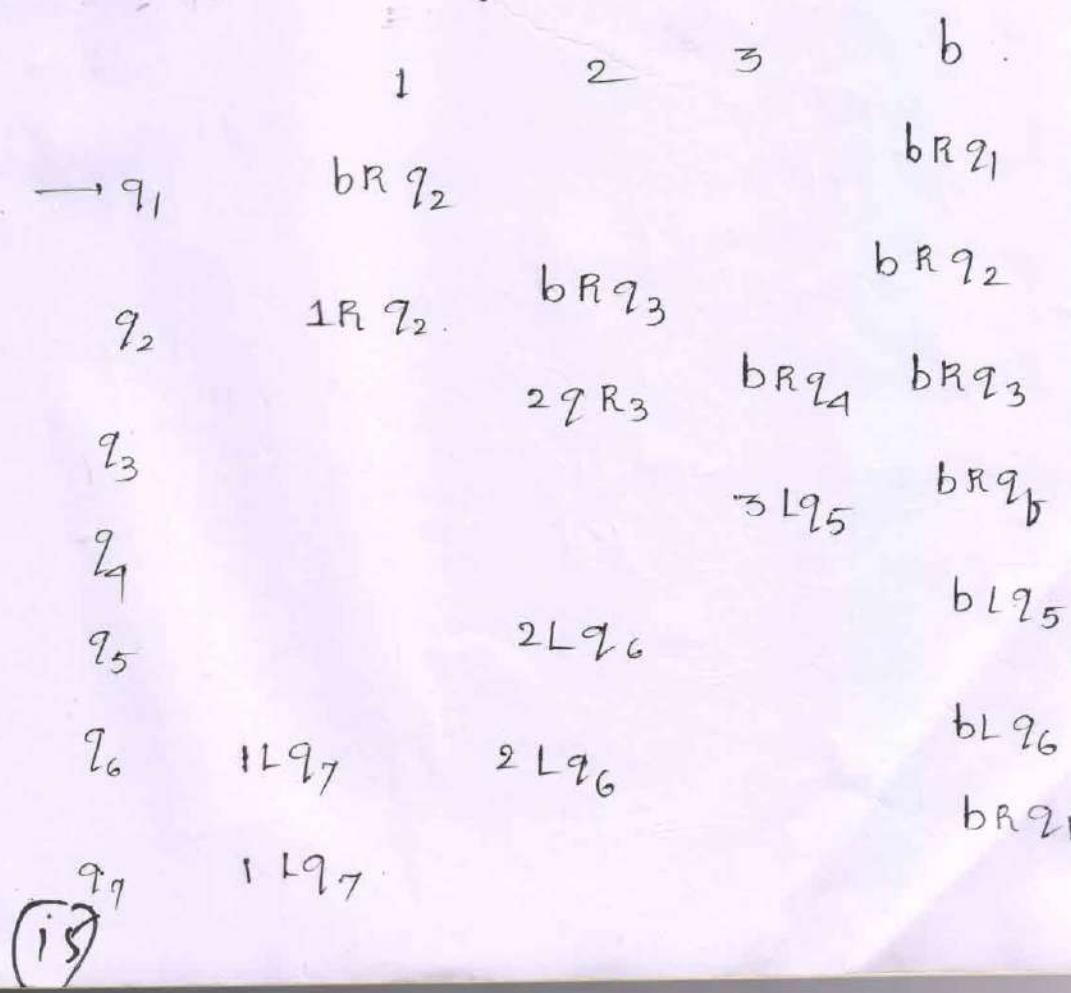
If $\delta(q, a) = (\delta_1, \alpha, \beta)$ we write α, β, δ under the a column and in the q row. So, if we get α, β, δ in the table it means α is written in the current cell (a is replaced by α), β gives the movement of the head (L or R) and δ denotes the new state into which the turing machine enters.

Consider above TM and draw the Computation sequence of the input string 00.

Design turing machine for $0^n 1^n$.



Design a Turing machine $1^m 0^n 3^k$.



Computable function

Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithm. They are used to discuss computability without referring to any concrete model of computation such as Turing machines or register machines.

According to the Church–Turing thesis, computable functions are exactly the functions that can be calculated using a mechanical calculation device given unlimited amounts of time and storage space.

Each computable function f takes a fixed, finite number of natural numbers as arguments. A function which is defined for all possible arguments is called **total**. If a computable function is total, it is called a **total computable function** or **total recursive function**.

The basic characteristic of a computable function is that there must be a finite procedure (an algorithm) telling how to compute the function. The models of computation listed above give different interpretations of what a procedure is and how it is used, but these interpretations share many properties.

Recursive and Recursively Enumerable Languages

Remember that there are *three* possible outcomes of executing a Turing machine over a given input. The Turing machine may

- Halt and accept the input;
- Halt and reject the input; or
- Never halt.

A language is *recursive* if there exists a Turing machine that accepts every string of the language and rejects every string (over the same alphabet) that is not in the language.

Note that, if a language L is recursive, then its complement $-L$ must also be recursive. (Why?)

A language is *recursively enumerable* if there exists a Turing machine that accepts every string of the language, and does not accept strings that are not in the language. (Strings that are not in the language may be rejected or may cause the Turing machine to go into an infinite loop.)

Recursively enumerable languages

Recursive languages

Clearly, every recursive language is also recursively enumerable. It is not obvious whether every recursively enumerable language is also recursive.

Closure Properties of Recursive Languages

- **Union:** If L1 and L2 are two recursive languages, their union $L1 \cup L2$ will also be recursive because if TM halts for L1 and halts for L2, it will also halt for $L1 \cup L2$.
- **Concatenation:** If L1 and L2 are two recursive languages, their concatenation $L1 \cdot L2$ will also be recursive. For Example:
 - $L1 = \{a^n b^n c^n | n \geq 0\}$
 - $L2 = \{d^m e^m f^m | m \geq 0\}$
 - $L3 = L1 \cdot L2$
 - $= \{a^n b^n c^n d^m e^m f^m | m \geq 0 \text{ and } n \geq 0\}$ is also recursive.

L1 says n no. of a's followed by n no. of b's followed by n no. of c's. L2 says m no. of d's followed by m no. of e's followed by m no. of f's. Their concatenation first matches no. of a's, b's and c's and then matches no. of d's, e's and f's. So it can be decided by TM.

- **Kleene Closure:** If L1 is recursive, its kleene closure $L1^*$ will also be recursive. For Example:

$$L1 = \{a^n b^n c^n | n \geq 0\}$$

$$L1^* = \{a^n b^n c^n | n \geq 0\}^* \text{ is also recursive.}$$

- **Intersection and complement:** If L1 and L2 are two recursive languages, their intersection $L1 \cap L2$ will also be recursive. For Example:

- $L1 = \{a^n b^n c^n d^m | n \geq 0 \text{ and } m \geq 0\}$
- $L2 = \{a^n b^n c^n d^n | n \geq 0\}$
- $L3 = L1 \cap L2$
- $= \{a^n b^n c^n d^n | n \geq 0\}$ will be recursive.

L1 says n no. of a's followed by n no. of b's followed by n no. of c's and then any no. of d's. L2 says any no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. Their intersection says n no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. So it can be decided by turing machine, hence recursive. Similarly, complement of recursive language L1 which is $\Sigma^* - L1$, will also be recursive.

RE - Recursive Enumerable REC- Recursive Language

Note: As opposed to REC languages, RE languages are not closed under complementation which means complement of RE language need not be RE.

The Church - Turing Thesis

Intuitive notion of an algorithm: a sequence of steps to solve a problem.

Questions: What is the meaning of "solve" and "problem"?

Answers:

Problem: This is a mapping. Can be represented as a function, or as a set membership "yes/no" question.

To solve a problem: To find a Turing machine that computes the function or answers the question.

Church-Turing Thesis: Any Turing machine that halts on all inputs corresponds to an algorithm, and any algorithm can be represented by a Turing machine.

This is the formal definition of an algorithm. This is not a theorem - only a hypothesis.

In computability theory, the **Church-Turing thesis** (also known as the **Church-Turing conjecture**, **Church's thesis**, **Church's conjecture**, and **Turing's thesis**) is a combined hypothesis ("thesis") about the nature of functions whose values are effectively calculable; i.e. computable. In simple terms, it states that "everything computable is computable by a Turing machine."

Counter machine

A **counter machine** is an abstract machine used in formal logic and theoretical computer science to model computation. It is the most primitive of the four types of register machines. A counter machine comprises a set of one or more unbounded *registers*, each of which can hold a single non-negative integer, and a list of (usually sequential) arithmetic and control instructions for the machine to follow.

The primitive model register machine is, in effect, a multitape 2-symbol Post-Turing machine with its behavior restricted so its tapes act like simple "counters".

By the time of Melzak, Lambek, and Minsky the notion of a "computer program" produced a different type of simple machine with many left-ended tapes cut from a Post-Turing tape. In all cases the models permit only two tape symbols { mark, blank }.^[3]

Some versions represent the positive integers as only a strings/stack of marks allowed in a "register" (i.e. left-ended tape), and a blank tape represented by the count "0". Minsky eliminated the PRINT instruction at the expense of providing his model with a mandatory single mark at the left-end of each tape.^[3]

In this model the single-ended tapes-as-registers are thought of as "counters", their instructions restricted to only two (or three if the TEST/DECREMENT instruction is atomized). Two common instruction sets are the following:

(1): { INC (r), DEC (r), JZ (r,z) }, i.e.

{ INCrement contents of register #r; DECrement contents of register #r; IF contents of #r=Zero THEN Jump-to Instruction #z }

(2): { CLR (r); INC (r); JE (r_i, r_j, z) }, i.e.

{ CLeaR contents of register r; INCrement contents of r; compare contents of r_i to r_j and if Equal then Jump to instruction z }

Although his model is more complicated than this simple description, the Melzak "pebble" model extended this notion of "counter" to permit multi-pebble adds and subtracts.

Basic features

For a given counter machine model the instruction set is tiny—from just one to six or seven instructions. Most models contain a few arithmetic operations and at least one conditional operation (if *condition* is true, then jump). Three *base models*, each using three instructions, are drawn from the following collection. (The abbreviations are arbitrary.)

- CLR (r): CLeaR register r. (Set r to zero.)
- INC (r): INCrement the contents of register r.
- DEC (r): DECrement the contents of register r.
- CPY (r_j, r_k): CoPY the contents of register r_j to register r_k leaving the contents of r_j intact.
- JZ (r, z): IF register r contains Zero THEN Jump to instruction z ELSE continue in sequence.
- JE (r_j, r_k, z): IF the contents of register r_j Equals the contents of register r_k THEN Jump to instruction z ELSE continue in sequence.

In addition, a machine usually has a HALT instruction, which stops the machine (normally after the result has been computed).

Using the instructions mentioned above, various authors have discussed certain counter machines:

- set 1: { INC (r), DEC (r), JZ (r, z) }, (Minsky (1961, 1967), Lambek (1961))
- set 2: { CLR (r), INC (r), JE (r_j, r_k, z) }, (Ershov (1958), Peter (1958) as interpreted by Shepherdson-Sturgis (1964); Minsky (1967); Schönhage (1980))
- set 3: { INC (r), CPY (r_j, r_k), JE (r_j, r_k, z) }, (Elgot-Robinson (1964), Minsky (1967))

The three counter machine base models have the same computational power since the instructions of one model can be derived from those of another. All are equivalent to the computational power of Turing machines (but only if Gödel numbers are used to encode data in the register or registers; otherwise their power is equivalent to the primitive recursive functions). Due to their unary processing style, counter machines are typically exponentially slower than comparable Turing machines.

Universal Turing Machines

Turing machines are abstract computing devices. Each Turing machine represents a particular algorithm. Hence we can think of Turing machines as being "hard-wired".

Is there a programmable Turing machine that can solve any problem solved by a "hard-wired" Turing machine?

The answer is "yes", the programmable Turing machine is called "universal Turing machine".

Basic Idea:

The Universal TM will take as input a description of a standard TM and an input w in the alphabet of the standard TM, and will halt if and only if the standard TM halts on w .

Theory Of Computations (TOC)

UNIT – V

Syllabus:

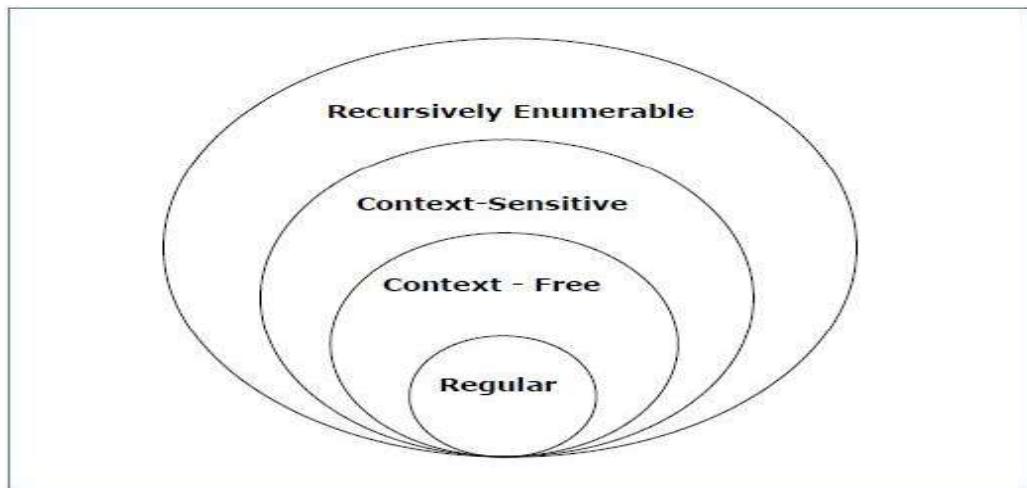
Computability Theory: Chomsky hierarchy of languages, Linear Bounded Automata and Context Sensitive Language, LR(0) grammar, Decidability of problems, Universal Turing Machine, Undecidability of Posts Correspondence Problem, Turing Reducibility, Definition of P and NP problems.

Chomsky hierarchy of languages:

According to Noam Chomsky, there are four types of grammars – Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other –

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Take a look at the following illustration. It shows the scope of each type of grammar –



Theory Of Computations (TOC)

Grammar Type	Production Rules	Language Accepted	Automata	Closed Under
Type-3 (Regular Grammar)	$A \rightarrow a$ or $A \rightarrow aB$ where $A, B \in N$ (non terminal) and $a \in T$ (Terminal)	Regular	Finite Automata	Union, Intersection, Complementation, Concatenation, Kleene Closure
Type-2 (Context Free Grammar)	$A \rightarrow p$ where $A \in N$ and $p \in (T \cup N)^*$	Context Free	Push Down Automata	Union, Concatenation, Kleene Closure
Type-1 (Context Sensitive Grammar)	$\alpha \rightarrow \beta$ where $\alpha, \beta \in (T \cup N)^*$ and $\text{len}(\alpha) \leq \text{len}(\beta)$ and α should contain atleast 1 non terminal.	Context Sensitive	Linear Bound Automata	Union, Intersection, Complementation, Concatenation, Kleene Closure
Type-0 (Recursive Enumerable)	$\alpha \rightarrow \beta$ where $\alpha, \beta \in (T \cup N)^*$ and α contains atleast 1 non-terminal	Recursive Enumerable	Turing Machine	Union, Intersection, Concatenation, Kleene Closure

Type - 3 Grammar:

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form $X \rightarrow a$ or $X \rightarrow aY$

where $X, Y \in N$ (Non terminal)

and $a \in T$ (Terminal)

The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule.

Example

$$\begin{aligned} X &\rightarrow \epsilon \\ X &\rightarrow a \mid aY \\ Y &\rightarrow b \end{aligned}$$

Theory Of Computations (TOC)

Type - 2 Grammar:

Type-2 grammars generate context-free languages.

The productions must be in the form $A \rightarrow \gamma$

where $A \in N$ (Non terminal)

and $\gamma \in (T \cup N)^*$ (String of terminals and non-terminals).

These languages generated by these grammars are recognized by a non-deterministic pushdown automaton.

Example

$$S \rightarrow Xa$$

$$X \rightarrow a$$

$$X \rightarrow aX$$

$$X \rightarrow abc$$

$$X \rightarrow \epsilon$$

Type - 1 Grammar:

Type-1 grammars **generate context-sensitive languages.**

The productions must be in the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where $A \in N$ (Non-terminal)

and $\alpha, \beta, \gamma \in (T \cup N)^*$ (Strings of terminals and non-terminals)

The strings α and β may be empty, but γ must be non-empty.

The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

$$AB \rightarrow AbBc$$

$$A \rightarrow bcA$$

$$B \rightarrow b$$

Theory Of Computations (TOC)

Type - 0 Grammar:

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phrase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of $\alpha \rightarrow \beta$ where α is a string of terminals and nonterminals with at least one non-terminal and α cannot be null. β is a string of terminals and non-terminals.

Example

$$\begin{aligned} S &\rightarrow ACaB \\ Bc &\rightarrow acB \\ CB &\rightarrow DB \\ aD &\rightarrow Db \end{aligned}$$

Linear Bounded Automata:

Definition

Linear Bounded Automata is a single tape Turing Machine with two special tape symbols call them left marker $<$ and right marker $>$.

The transitions should satisfy these conditions:

- It should not replace the marker symbols by any other symbol.
- It should not write on cells beyond the marker symbols.

Thus the initial configuration will be:

$< q_0 a_1 a_2 a_3 a_4 a_5 \dots a_n >$

Formal Definition:

Formally Linear Bounded Automata is a non-deterministic Turing Machine, $M = (Q, P, \Gamma, \delta, F, q_0, t, r)$

- Q is set of all states
- P is set of all terminals
- Γ is set of all tape alphabets $P \subset \Gamma$
- δ is set of transitions
- F is blank symbol
- q_0 is the initial state
- $<$ is left marker and $>$ is right marker
- t is accept state
- r is reject state

Theory Of Computations (TOC)

Context Sensitive Languages:

- The Context sensitive languages are the languages which are accepted by linear bounded automata. These types of languages are defined by context Sensitive Grammar. In this grammar more than one terminal or non terminal symbol may appear on the left hand side of the production rule. Along with it, the context sensitive grammar follows following rules:
- The number of symbols on the left hand side must not exceed number of symbols on the right hand side.
- The rule of the form $A \rightarrow \epsilon$ is not allowed unless A is a start symbol. It does not occur on the right hand side of any rule.

The classic example of context sensitive language is $L = \{ a^n b^n c^n \mid n \geq 1 \}$

- If G is a Context Sensitive Grammar then
 $L(G) = \{ w \mid w \in \Sigma^* \text{ and } S \Rightarrow^+ G w \}$
- CSG for $L = \{ a^n b^n c^n \mid n \geq 1 \}$
 - $N : \{S, B\}$ and $P = \{a, b, c\}$
 - $P : S \rightarrow aSBc \mid abc \quad cB \rightarrow Bc \quad bB \rightarrow bb$
- Derivation of aabbcc :
 $S \Rightarrow aSBc \Rightarrow aabcBc \Rightarrow aabBcc \Rightarrow aabbcc$

Grammar: The Context Sensitive Grammar can be written as

$S \rightarrow aBC$

$S \rightarrow SABC$

$CA \rightarrow AC$

$BA \rightarrow AB$

$CB \rightarrow BC$

$aA \rightarrow aa$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

Now to derive the string aabbcc we will start from starting symbol:

S rule $S \rightarrow SABC$

SABC rule $S \rightarrow aBC$

aBCABC rule $CA \rightarrow AC$

aBACBC rule $CB \rightarrow BC$

aBABCC rule $BA \rightarrow AB$

aABBCC rule $aA \rightarrow aa$

aaBBCC rule $aB \rightarrow ab$

Theory Of Computations (TOC)

aab <u>B</u> CC	rule bB → bb
aabb <u>C</u> C	rule bC → bc
aabb <u>c</u> C	rule cC → cc
aabbcc	

NOTE: The language $a^n b^n c^n$ where $n \geq 1$ is represented by context sensitive grammar but it can not be represented by context free grammar.

Every context sensitive language can be represented by LBA.

Closure Properties

Context Sensitive Languages are closed under

- Union
- Concatenation
- Reversal
- Kleene Star
- Intersection

All of the above except Intersection can be proved by modifying the grammar.

Proof of Intersection needs a machine model for CSG

LR-Grammar:

- Left-to-right scan of the input producing a rightmost derivation in reverse order
- Simply:
 - L stands for Left-to-right
 - R stands for rightmost derivation in reverse order

LR-Items

- An item (for a given CFG)
 - A production with a dot anywhere in the right side (including the beginning and end)
 - In the event of an ϵ -production: $B \rightarrow \epsilon$
 - $B \rightarrow \cdot$ is an item

Example: Items

- Given our example grammar:
 - $S' \rightarrow Sc, S \rightarrow SA|A, A \rightarrow aSb|ab$
 - The items for the grammar are:

$S' \rightarrow \cdot Sc, S' \rightarrow S \cdot c, S' \rightarrow S c \cdot$

$S \rightarrow \cdot SA, S \rightarrow S \cdot A, S \rightarrow SA \cdot, S \rightarrow \cdot A, S \rightarrow A \cdot$

Theory Of Computations (TOC)

$A \Rightarrow \cdot aSb$, $A \Rightarrow a \cdot Sb$, $A \Rightarrow aS \cdot b$, $A \Rightarrow aSb \cdot$, $A \Rightarrow \cdot ab$, $A \Rightarrow a \cdot b$, $A \Rightarrow ab \cdot$

Some Notation

- $*\Rightarrow$ = 1 or more steps in a derivation
- $*\Rightarrow_{rm}$ = rightmost derivation
- \Rightarrow_{rm} = single step in rightmost derivation

More terms

- **Handle**

- A substring which matches the right-hand side of a production and represents 1 step in the derivation
- Or more formally:
 - (of a right-sentential form γ for CFG G)
 - Is a substring β such that:
 - $S * \Rightarrow_{rm} \delta\beta w$
 - $\delta\beta w = \gamma$
- If the grammar is unambiguous:
 - There are no useless symbols
 - The rightmost derivation (in right-sentential form) and the handle are unique

Example

- Given our example grammar:
 - $S' \Rightarrow Sc$, $S \Rightarrow SA|A$, $A \Rightarrow aSb|ab$
- An example right-most derivation:
 - $S' \Rightarrow Sc \Rightarrow SAc \Rightarrow SaSbc$
- Therefore we can say that: $SaSbc$ is in right-sentential form
 - The handle is aSb
- Viable Prefix
 - (of a right-sentential form for γ)
 - Is any prefix of γ ending no farther right than the right end of a handle of γ .
- Complete item
- An item where the dot is the rightmost symbol

Example

- Given our example grammar:
 - $S' \Rightarrow Sc$, $S \Rightarrow SA|A$, $A \Rightarrow aSb|ab$
- The right-sentential form abc :
 - $S' * \Rightarrow_{rm} Ac \Rightarrow abc$
- Valid prefixes:
 - $A \rightarrow ab \cdot$ for prefix ab

Theory Of Computations (TOC)

- $A \rightarrow a \cdot b$ for prefix a
- $A \rightarrow \cdot ab$ for prefix ϵ
- $A \rightarrow ab \cdot$ is a complete item, $\therefore A c$ is the right-sentential form for abc

LR(0)

- Left-to-right scan of the input producing a rightmost derivation with a look-ahead (on the input) of 0 symbols
- It is a restricted type of CFG
- 1st in the family of LR-grammars
- LR(0) grammars define exactly the DCFLs having the prefix property

Definition of LR(0) Grammar

- G is an LR(0) grammar if
 - The start symbol does not appear on the right side of any productions
 - \forall prefixes γ of G where $A \rightarrow \alpha \cdot$ is a complete item, then it is unique
 - i.e., there are no other complete items (and there are no items with a terminal to the right of the dot) that are valid for γ

Facts we now know:

- Every LR(0) grammar generates a DCFL
- Every DCFL with the prefix property has a LR(0) grammar
- Every language with LR(0) grammar have the prefix property
- L is DCFL if L has a LR(0) grammar

Example Grammar

1. $S \rightarrow E\$$
2. $E \rightarrow E+(E)$
3. $E \rightarrow id$

The LR(0) items (simply place a dot at point in every production)

1. $S \rightarrow \cdot E\$$
2. $S \rightarrow E \cdot \$$
3. $S \rightarrow E \$ \cdot$
4. $E \rightarrow \cdot E+(E)$
5. $E \rightarrow E \cdot +(E)$
6. $E \rightarrow E+ \cdot (E)$
7. $E \rightarrow E+(\cdot E)$
8. $E \rightarrow E+(E \cdot)$
9. $E \rightarrow E+(E) \cdot$
10. $E \rightarrow \cdot id$
11. $E \rightarrow id \cdot$

Theory Of Computations (TOC)

Creating states from Items

States are composed of **closures** constructed from items. Initially the only closure is $\{S \rightarrow \bullet E\$ \}$. Next, we construct the closure like so:

$$\text{Closure}(I) = \text{Closure}(I) \cup \{A \rightarrow \bullet \alpha \mid B \rightarrow \beta \bullet A\gamma \in I\}$$

Basically, for a non-terminal $\backslash(A)$ in $\backslash(I)$ with a \bullet before it, add all items of the form " $A \rightarrow \bullet \dots$ ".

Given our example (**Initial** = $\{S \rightarrow \bullet E\$ \}$) we create the following closure:

$$\text{Closure}(\{S \rightarrow \bullet E\$ \}) = \{S \rightarrow \bullet E\$, E \rightarrow \bullet E+(E), E \rightarrow \bullet id\}$$

to create more closures we define a "goto" function that creates new closures. Given a closure $\backslash(I)$ and a symbol $\backslash(a)$ (terminal or non-terminal):

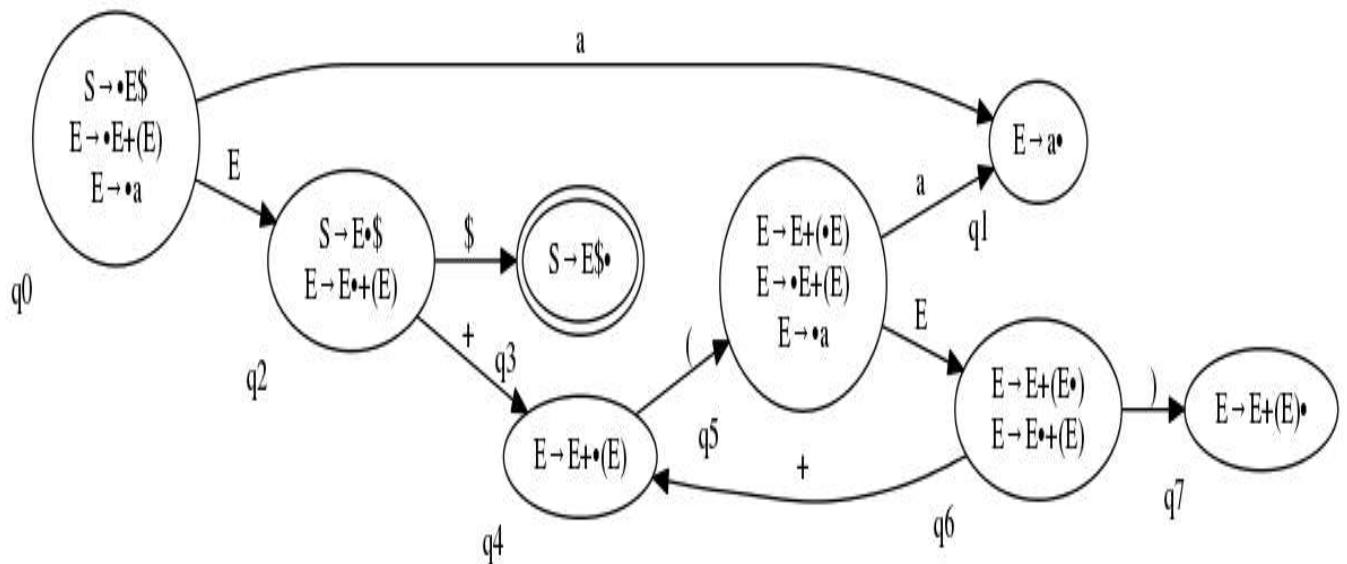
$$\text{goto}(I, a) = \{B \rightarrow \alpha a \bullet \beta \mid B \rightarrow \alpha \bullet a\beta \in I\}$$

Basically, For every item in $\backslash(I)$ that has a \bullet before $\backslash(a)$ we create a new closure by pushing the \bullet one symbol forward. For instance, given our example closure and the symbol $\backslash(E)$ we get:

$$\text{goto}(\{S \rightarrow \bullet E\$, E \rightarrow \bullet E+(E), E \rightarrow \bullet id\}, E) = \{S \rightarrow E \bullet \$, E \rightarrow E \bullet +(E)\}$$

Now, for each of these items we create a closure and for each of those closures we create all possible goto sets. We keep going until there are no more new states (items that are not part of a closure).

Lets finish building the states:



Creating the transition table

The table is index by state and symbol. We created the states already and the symbols are given by the grammar, now we need to create the action within the cells. The goto functions defines the

Theory Of Computations (TOC)

transitions between the closures. Transition from state q_1 to state q_2 given symbol $a \backslash(\backslash iff\backslash)$
goto(closure(q_1), a) = closure(q_2).

- If the • is at the end of the item, this is a reduction action.
- If the symbol is a non-terminal, the action for the transition is a go-to.
- If the symbol is a terminal, the action is a shift

Now we create the transition table:

			Actions				go-to actions
States	A	+	()	\$	S	E
0	s1						g2
1	rIII	rIII	rIII	rIII	rIII		
2		s4			s3		
3	Acc	acc	acc	acc	acc		
4			s5				
5	s1						g6
6		s4		s7			
7	rII	rII	rII	rII	rII		

Conflicts

There are two kinds of conflicts we encounter

1. Shift-reduce conflict - a state contains items that correspond to both **reduce** and **shift** actions
2. Reduce-reduce conflict - a state has 2 different items corresponding to different **reduce** actions

Indications of a conflict

Any grammar with an ϵ derivation cannot be LR(0). This is because there is no input to reduce, so at any point that derivation rule can be used to reduce (add the rule's LHS non-terminal to the stack)

Theory Of Computations (TOC)

Decidability of problems:

A problem is said to be **Decidable** if we can always construct a corresponding **algorithm** that can answer the problem correctly. We can intuitively understand Decidable problems by considering a simple example. Suppose we are asked to compute all the prime numbers in the range of 1000 to 2000. To find the **solution** of this problem, we can easily devise an algorithm that can enumerate all the prime numbers in this range.

Now talking about Decidability in terms of a Turing machine, a problem is said to be a Decidable problem if there exist a corresponding Turing machine which **halts** on every input with an answer- **yes or no**. It is also important to know that these problems are termed as **Turing Decidable** since a Turing machine always halts on every input, accepting or rejecting it.

Semi-Decidable Problems –

Semi-Decidable problems are those for which a Turing machine halts on the input accepted by it but it can either halt or loop forever on the input which is rejected by the Turing Machine. Such problems are termed as **Turing Recognisable** problems.

Examples – We will now consider few important **Decidable problems**:

- Are two **regular** languages L and M **equivalent**?

We can easily check this by using Set Difference operation.

$$L-M = \text{Null} \text{ and } M-L = \text{Null}$$

Hence $(L-M) \cup (M-L) = \text{Null}$, then L,M are equivalent.

- Membership of a CFL?

We can always find whether a string exist in a given CFL by using an algorithm based on dynamic programming.

- Emptiness of a CFL

By checking the production rules of the CFL we can easily state whether the language generates any strings or not.

Undecidable Problems –

The problems for which we can't construct an algorithm that can answer the problem correctly in a finite time are termed as Undecidable Problems. These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.

We can understand Undecidable Problems intuitively by considering **Fermat's Theorem**, a popular Undecidable Problem which states that no three positive integers a, b and c for any $n \geq 2$ can ever satisfy the equation: $a^n + b^n = c^n$.

If we feed this problem to a Turing machine to find such a solution which gives a contradiction then a Turing Machine might run forever, to find the suitable values of n, a, b and c. But we are

Theory Of Computations (TOC)

always unsure whether a contradiction exists or not and hence we term this problem as an **Undecidable Problem**.

Examples – These are few important **Undecidable Problems**:

- Whether a CFG generates all the strings or not?

As a CFG generates infinite strings ,we can't ever reach up to the last string and hence it is Undecidable.

- Whether two CFG L and M equal?

Since we cannot determine all the strings of any CFG , we can predict that two CFG are equal or not.

- Ambiguity of CFG?

There exist no algorithm which can check whether for the ambiguity of a CFL. We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.

- Is it possible to convert a given ambiguous CFG into corresponding non-ambiguous CFL?

It is also an Undecidable Problem as there doesn't exist any algorithm for the conversion of an ambiguous CFL to non-ambiguous CFL.

- Is a language Learning which is a CFL, regular?

This is an Undecidable Problem as we can not find from the production rules of the CFL whether it is regular or not.

Some more **Undecidable Problems** related to Turing machine:

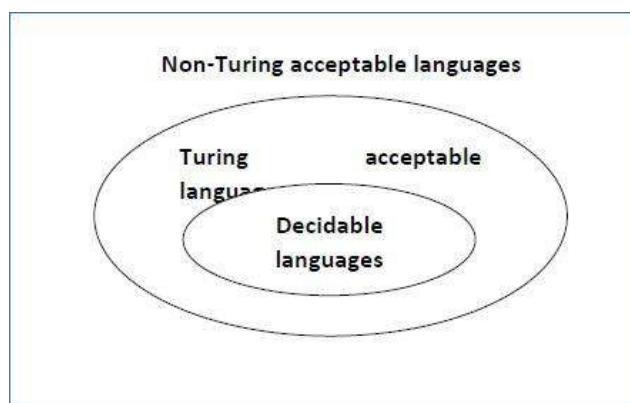
- **Membership** problem of a Turing Machine?

- **Finiteness** of a Turing Machine?

- **Emptiness** of a Turing Machine?

- Whether the language accepted by Turing Machine is regular or CFL?

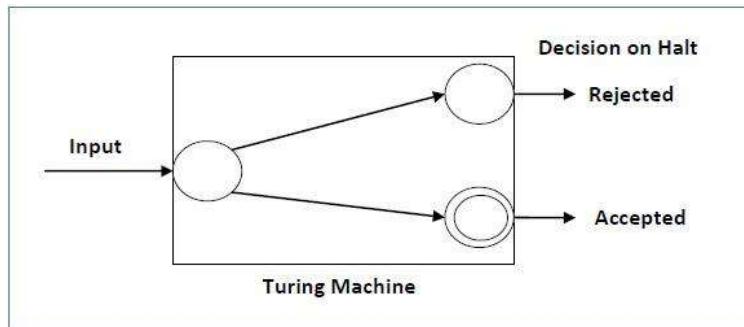
A language is called **Decidable** or **Recursive** if there is a Turing machine which accepts and halts on every input string w . Every decidable language is Turing-Acceptable.



A decision problem P is decidable if the language L of all yes instances to P is decidable.

Theory Of Computations (TOC)

For a decidable language, for each input string, the TM halts either at the accept or the reject state as depicted in the following diagram –



Example 1

Find out whether the following problem is decidable or not –

Is a number 'm' prime?

Solution

Prime numbers = {2, 3, 5, 7, 11, 13,}

Divide the number 'm' by all the numbers between '2' and ' \sqrt{m} ' starting from '2'.

If any of these numbers produce a remainder zero, then it goes to the "Rejected state", otherwise it goes to the "Accepted state". So, here the answer could be made by 'Yes' or 'No'.

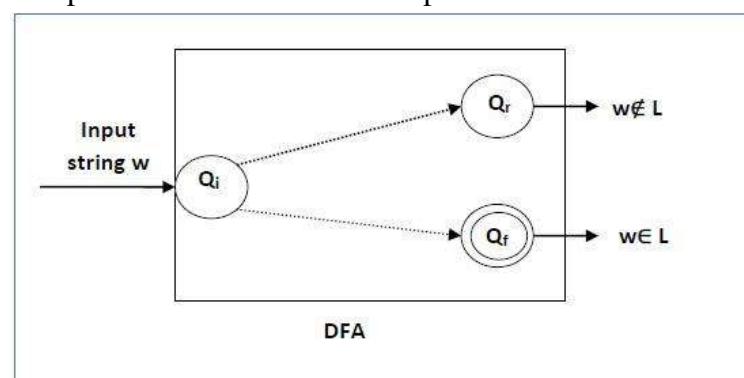
Hence, it is a decidable problem.

Example 2

Given a regular language L and string w , how can we check if $w \in L$?

Solution

Take the DFA that accepts L and check if w is accepted



Some more decidable problems are –

- Does DFA accept the empty language?
- Is $L_1 \cap L_2 = \emptyset$ for regular sets?

Note –

- If a language L is decidable, then its complement L' is also decidable
- If a language is decidable, then there is an enumerator for it.

Theory Of Computations (TOC)

Universal Turing Machines:

Turing machines are abstract computing devices. Each Turing machine represents a particular algorithm. Hence we can think of Turing machines as being "hard-wired".

Is there a programmable Turing machine that can solve any problem solved by a "hard-wired" Turing machine?

The answer is "yes", the programmable Turing machine is called "universal Turing machine".

Basic Idea:

The Universal TM will take as input a description of a standard TM and an input w in the alphabet of the standard TM, and will halt if and only if the standard TM halts on w.

Post Correspondence Problem (PCP):

The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem. The PCP problem over an alphabet Σ is stated as follows –

Given the following two lists, **M** and **N** of non-empty strings over Σ –

$$M = (x_1, x_2, x_3, \dots, x_n)$$

$$N = (y_1, y_2, y_3, \dots, y_n)$$

We can say that there is a Post Correspondence Solution, if for some i_1, i_2, \dots, i_k , where $1 \leq i_j \leq n$, the condition $x_{i1} \dots x_{ik} = y_{i1} \dots y_{ik}$ satisfies.

Example 1

Find whether the lists $M = (\text{abb}, \text{aa}, \text{aaa})$ and $N = (\text{bba}, \text{aaa}, \text{aa})$ have a Post Correspondence Solution?

Solution

	X1	X2	X3
M	abb	aa	aaa
N	bba	aaa	aa

Here,

$$x_2 x_1 x_3 = \text{'aaabbbaaa'}$$

$$\text{and } y_2 y_1 y_3 = \text{'aaabbbaaa'}$$

We can see that

$$x_2 x_1 x_3 = y_2 y_1 y_3$$

Hence, the solution is $i = 2, j = 1, \text{ and } k = 3$.

Theory Of Computations (TOC)

Example 2

Find whether the lists $M = (ab, bab, bbaaa)$ and $N = (a, ba, bab)$ have a Post Correspondence Solution?

Solution

	X1	X2	X3
M	ab	bab	bbaaa
N	a	ba	bab

In this case, there is no solution because –

$|x_2x_1x_3| \neq |y_2y_1y_3|$ (Lengths are not same)

Hence, it can be said that this Post Correspondence Problem is **undecidable**.

Turing Reducibility:

In computability theory, a Turing reduction from a problem A to a problem B, is a reduction which solves A, assuming the solution to B is already known (Rogers 1967, Soare 1987). It can be understood as an algorithm that could be used to solve A if it had available to it a subroutine for solving B. More formally, a Turing reduction is a function computable by an oracle machine with an oracle for B. Turing reductions can be applied to both decision problems and function problems.

If a Turing reduction of A to B exists then every algorithm for B can be used to produce an algorithm for A, by inserting the algorithm for B at each place where the oracle machine computing A queries the oracle for B. However, because the oracle machine may query the oracle a large number of times, the resulting algorithm may require more time asymptotically than either the algorithm for B or the oracle machine computing A, and may require as much space as both together.

Definition

Given two sets $A, B \subseteq N$ of natural numbers, we say A is Turing reducible to B and write $A \leq_T B$

if there is an oracle machine that computes the characteristic function of A when run with oracle B. In this case, we also say A is B-recursive and B-computable.

If there is an oracle machine that, when run with oracle B, computes a partial function with domain A, then A is said to be B-recursively enumerable and B-computably enumerable.

Theory Of Computations (TOC)

Definition of P & NP:

Definition of P:

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \{ L \mid \text{Time}(n) \leq k \text{ for some constant } k \}$$

Motivation: To define a class of problems that can be solved efficiently.

- P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing Machine.
- P roughly corresponds to the class of problems that are realistically solvable on a computer.

Definition of NP:

The term NP comes from nondeterministic polynomial time and has an alternative characterization by using nondeterministic polynomial time Turing machines.

Theorem

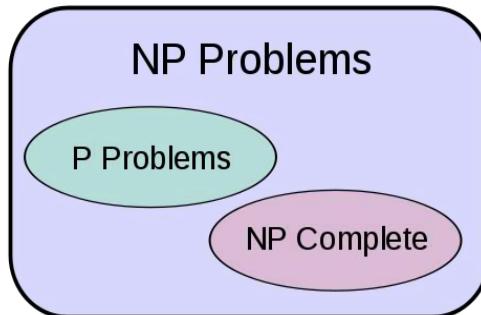
A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

Proof.

(\Rightarrow) Convert a polynomial time verifier V to an equivalent polynomial time NTM N. On input w of length n:

- Nondeterministically select string c of length at most n^k (assuming that V runs in time n^k).
- Run V on input $\langle w, c \rangle$.
- If V accepts, accept; otherwise, reject.

P vs. NP



If you spend time in or around the programming community you probably hear the term “P versus NP” rather frequently.

The Problem

P vs. NP

The P vs. NP problem asks whether every problem whose solution can be quickly verified by a computer can also be quickly solved by a computer.

Theory Of Computations (TOC)

P problems are easily solved by computers, and NP problems are not easily *solvable*, but if you present a potential solution it's easy to *verify* whether it's correct or not.

As you can see from the diagram above, all P problems are NP problems. That is, if it's easy for the computer to solve, it's easy to verify the solution. So the P vs NP problem is just asking if these two problem types are the same, or if they are different, i.e. that there are some problems that are easily verified but not easily solved.

It currently appears that $P \neq NP$, meaning we have plenty of examples of problems that we can quickly verify potential answers to, but that we can't solve quickly. Let's look at a few examples:

- A traveling salesman wants to visit 100 different cities by driving, starting and ending his trip at home. He has a limited supply of gasoline, so he can only drive a total of 10,000 kilometers. He wants to know if he can visit all of the cities without running out of gasoline. (from Wikipedia)
- A farmer wants to take 100 watermelons of different masses to the market. She needs to pack the watermelons into boxes. Each box can only hold 20 kilograms without breaking. The farmer needs to know if 10 boxes will be enough for her to carry all 100 watermelons to market.

All of these problems share a common characteristic that is the key to understanding the intrigue of P versus NP: In order to solve them you have to try all combinations.

The Solution

This is why the answer to the P vs. NP problem is so interesting to people. If anyone were able to show that P is equal to NP, it would make difficult real-world problems trivial for computers.

Summary

1. P vs. NP deals with the gap between computers being able to quickly solve problems vs. just being able to test proposed solutions for correctness.
2. As such, the P vs. NP problem is the search for a way to solve problems that require the trying of millions, billions, or trillions of combinations without actually having to try each one.
3. Solving this problem would have profound effects on computing, and therefore on our society.