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 :

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

Postulate 4 :

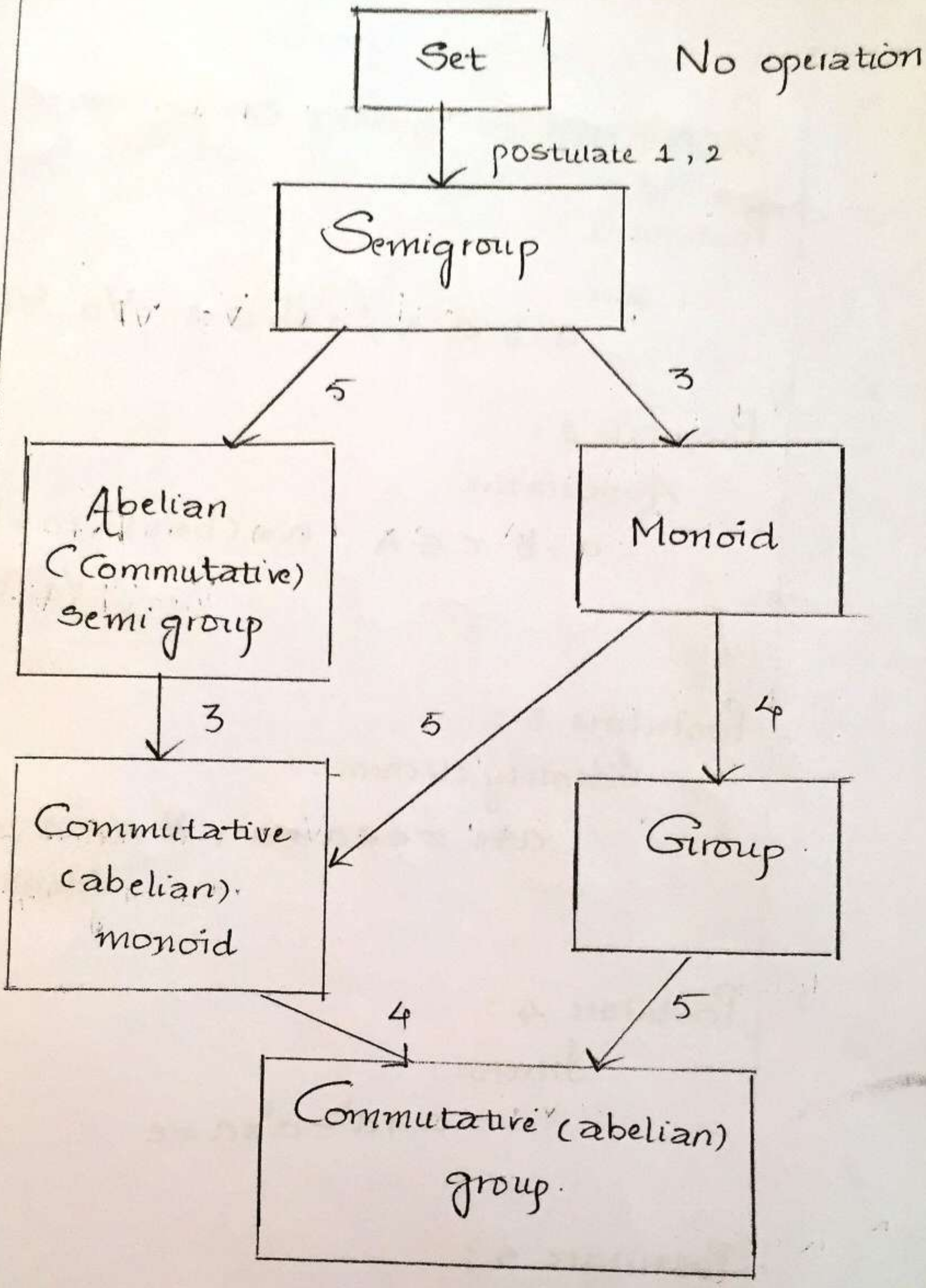
Inverse :

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

Postulate 5 :

Commutative :

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


Set
No operation
postulate 1, 2
Semigroup
5
3
Abelian (Commutative) semi group
Monoid
3
5
4
Commutative (abelian) monoid
Group
4
5
Commutative (abelian) group.

Let $S = \{1, 2, 3, 4 \ldots\ldots\}$ and binary operation is substraction 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, Substraction

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

Let,

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

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

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

P1 : closure.

It satisfies P1

P2 : Associative

It satisfies P2

P3 : Identity

$\{\} \cup \phi = \{\}$

$\downarrow$

Identity element

P4 : Inverse

$A \cup A' = \phi$

It does not satisfy P4

P5: Commutative

It satisfies Commutative

So, it is called Commutative monoid.

$< Z \times >$

It satisfies P1, P2, P3, & P5 so, it is

Commutative monoid.

$< R \times >$

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 * & o

here,

P1 - P5 $\longrightarrow$ * and

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

P9 : P4 :

Inverse

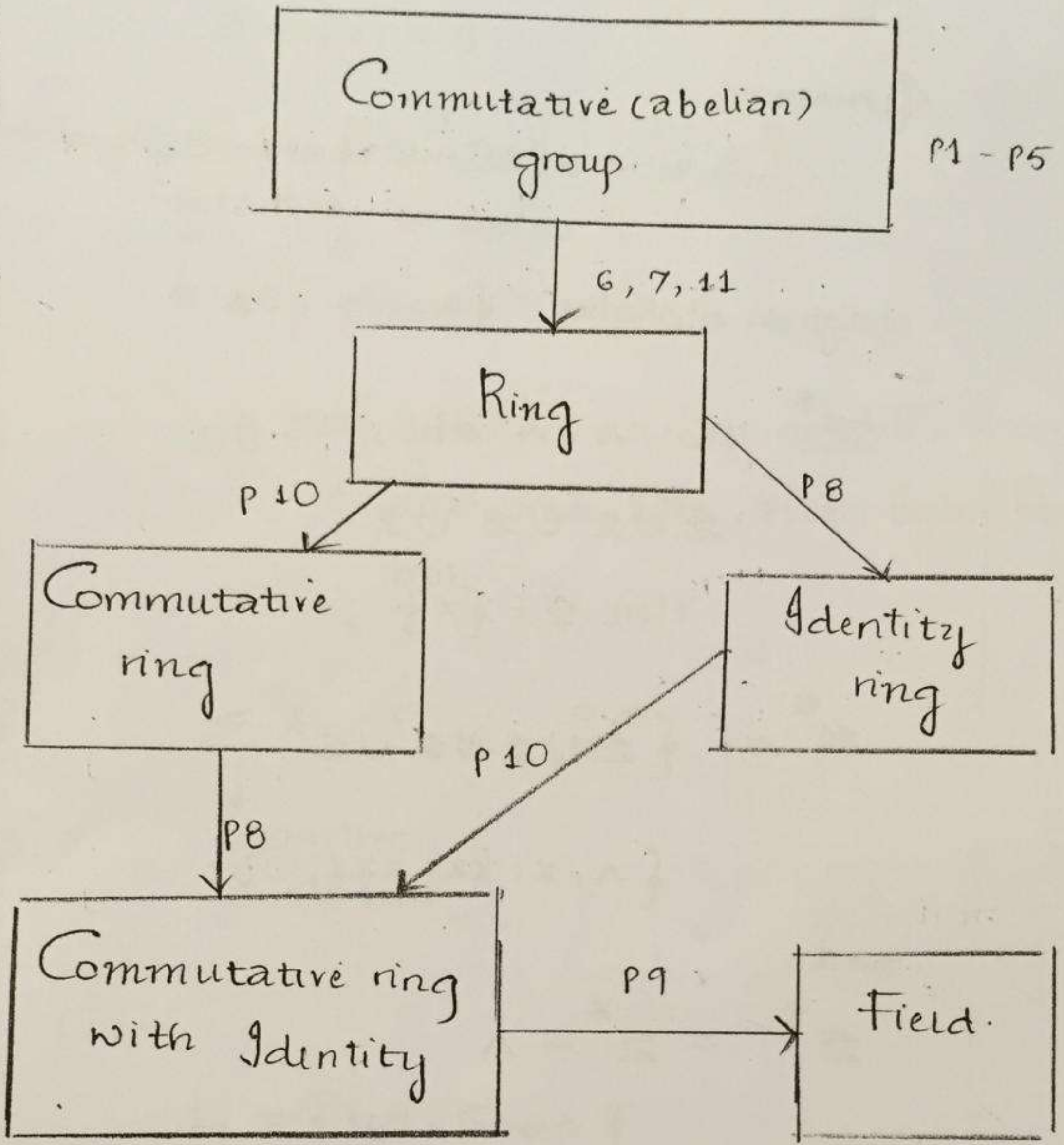
$$a * a' = a' * a = e$$
$$a \circ a' = a' \circ a = e'$$

P11 :

Distributive

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

for eg, $< Z, +, * >$.



$\langle z, +, * \rangle$

P 11 $\longrightarrow$ 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 ~~object~~ Symbols is called as Alphabet.

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

Σ^* aa, a, Λ, abc, acz, ...

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

Here $\Sigma = \{x\}$

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

$= \{\Lambda, x, xx, xxx, \ldots\ldots\ldots\}$

and

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

CONCANTENATION :

Let x, y be 2 variables and

$z = xy$

$x = 001$ &

$y = 10$

then

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

Concantenation

for eg,

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

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

↓

Kleen's closure

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

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

↓

positive closure

(2)

$L_1 = \{red, Green\}$

$L_2 = \{pen, Ink\}$

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

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

$L_1 L_2 = \{redpen, redInk, Greenpen, GreenInk\}$

↓

concantenation

CONCANTENATION PROPERTIES :

P1 : Associative

x, y, z in Σ^*

$x(yz) = (xy)z$

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

$ab(pqcd) = (abpq)cd$

$abpqcd = abpqcd$

P2 :

for cancantenation the Identity element is '$\wedge$'

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

$\wedge x = x \wedge = x$

P3 : left Concantenation and right Cancellation

LC :

$ap = aq$.

by left cancellation,

$p = q$

$a = hai$

$p = xxxx$

$q = yyyy$

$xxxx = yyyy$

RC :

$$pa = qa$$
$$p = q$$

P4 : length

$\forall x$, $\forall y$ in Σ^*

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

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

P5 : Transpose

$$(xa)^T = a(x)^T$$
$$= y(abc)^T$$
$$= yc(ab)^T$$
$$= ycb(a)^T$$
$$= ycba$$

→ Evenlength polindrome is obtained by the Concantenation and transpose of a strings

LEVIS THEOREM :

Here $ab = xy$ in Σ^*

case (i) :

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

$$|a| > |x|$$

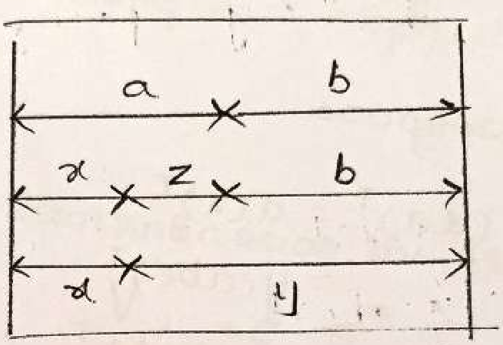
Case (ii) :

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

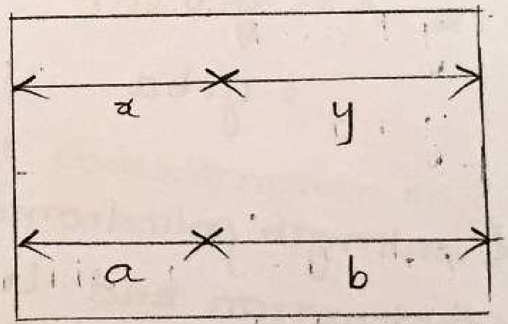
case (iii) :

$$x = az \quad \text{and} \quad b = zy \quad \text{if} \quad |a| < |x|$$

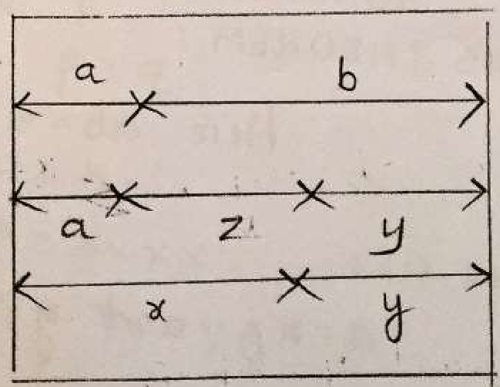
case (i):



case (ii) ;



case (iii) :



PREFIX :

prefix is the leading symbols of a string

for eg,

prefix of abc is

∧ , a , ab , abc

SUFFIX :

Suffix is the trailing of a substring of a string

for eg, .

Suffix of abc is

∧ , 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 .

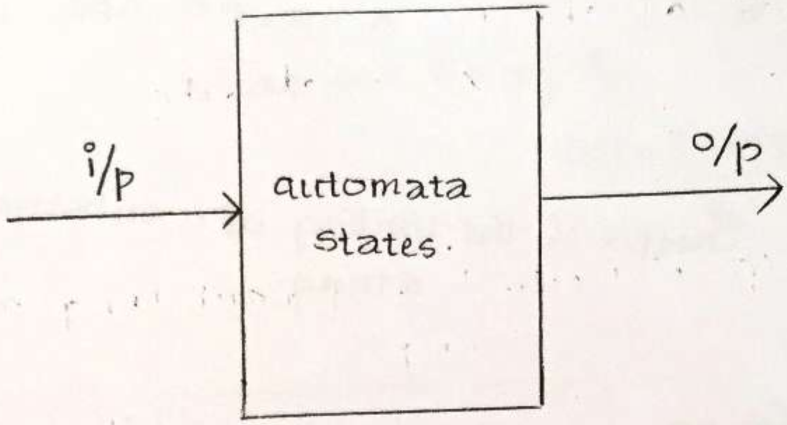
NON - TERMINAL :

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

It is used to generate the strings

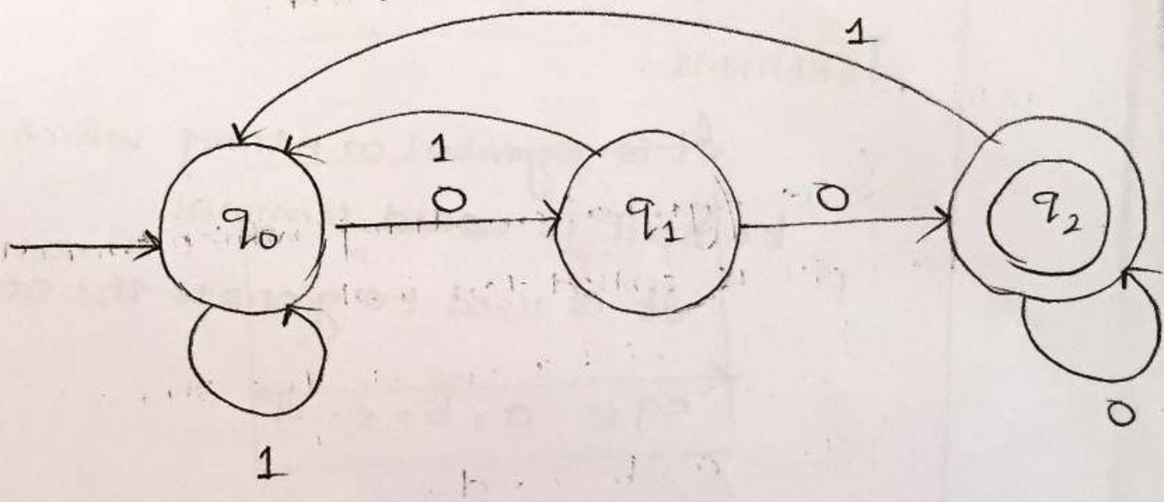
eg, abc, ab, xz

FINITE AUTOMATA :



Examples :

(1) ending with 00



→ Input is represented by Σ

→ In, finite automata the input is finite

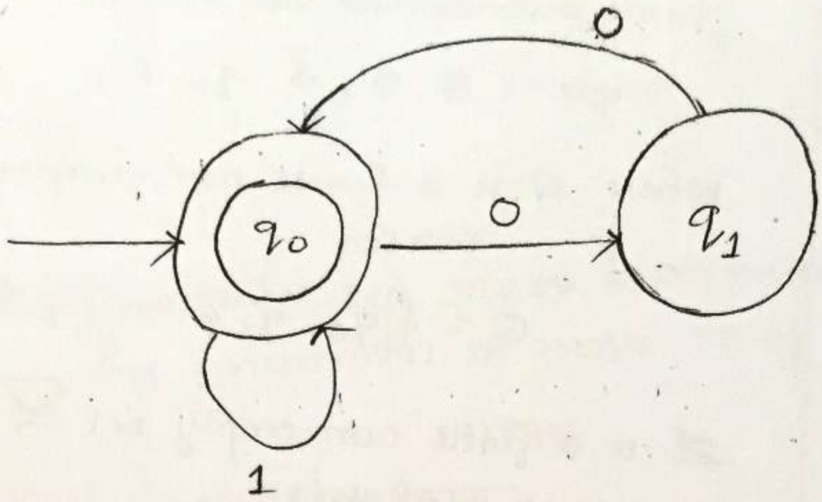
→ The states in the above diagram are

$q_0, q_1, q_2, \ldots\ldots\ldots$

→ output relation depends on input and state

→ finalstate 'F' is the set of all finitestates

(2) Even number of zeros



→ 0000

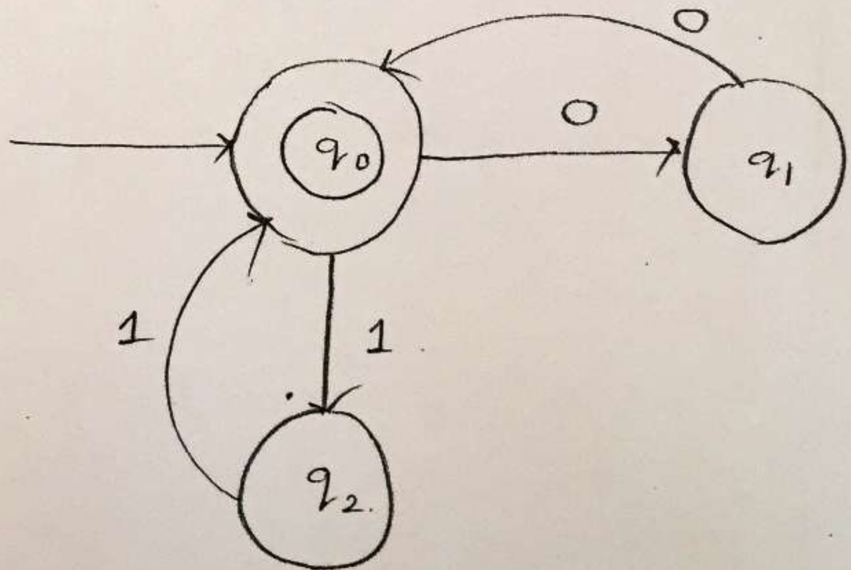
$(q_0, 0000) \rightarrow (q_1, 000) \rightarrow (q_0, 00) \rightarrow (q_1, 0)$
$\rightarrow q_0$

→ 001

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

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

→ Λ, 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 mapps $Q \times \Sigma$ into Q and is usually called ..

TRANSITION SYSTEM

Transition system is finite labelled directed graph.

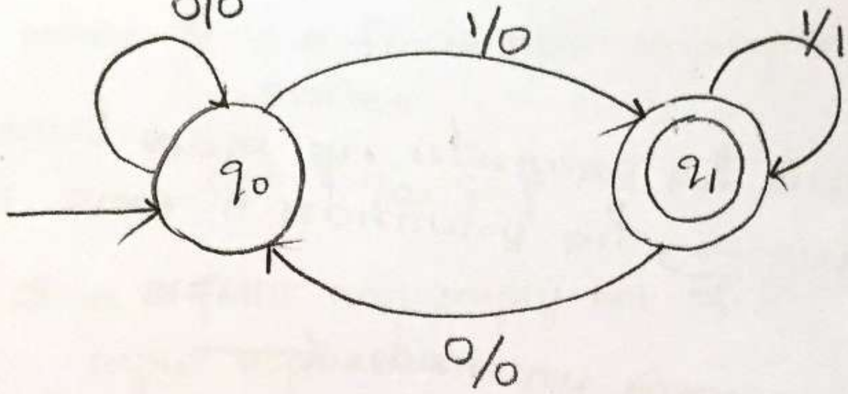
In the above graph the initial state can be represented by

⟶ 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 sysum is a 5-tuple $(Q, \Sigma, \delta, Q_0, F)$

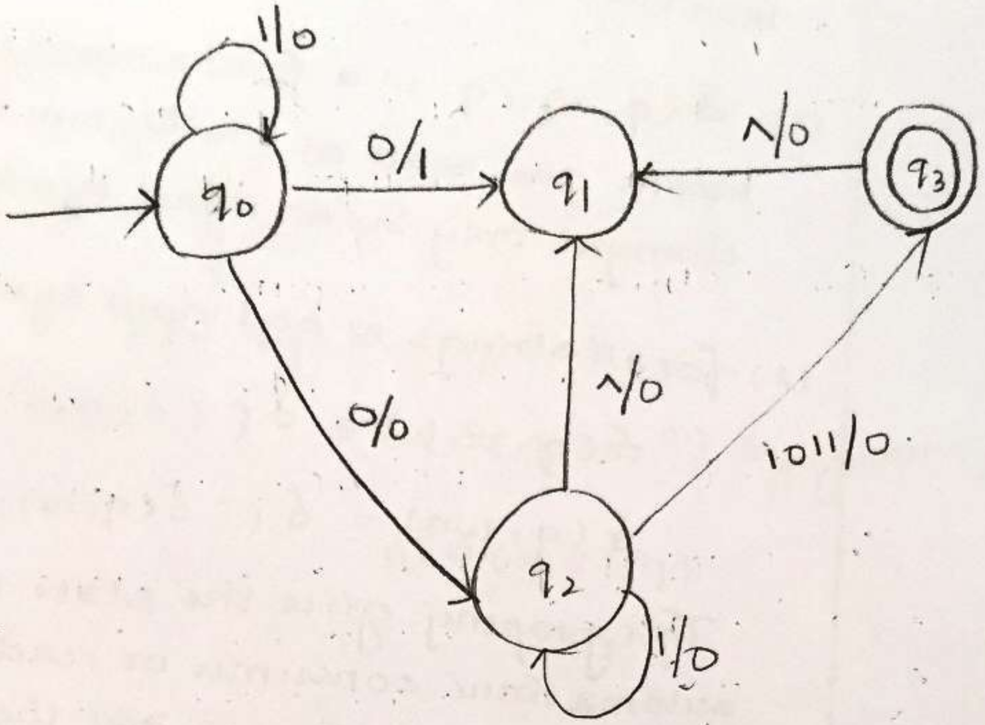
Wher Q_0 is a set of all the initial states

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

(i) There exist a path which originales from some initial stale, goes along the arrows and terminalis at some final state and

(ii) The path value obtained by concanlinatio of all edge labels of the path is equal to w

Describe the transition system.



This transition system has '2' initial states

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 concantenation

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 $\forall$ a

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

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

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 x is accepted by a finite automach

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

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

NOTE:

A final state is also called as an acceptin state

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

Step 1:

let

$|y| = 1$

$y = a$

$\delta(q, xy) = \delta(q, xa)$

$= \delta(\delta(q, x), a)$ by property 2

$= \delta(\delta(q, x), y)$.

step 2:

$\delta(q, xy) = \delta(\delta(q, x), y)$ for $|y| = n$

step 3:

$|y| = n+1$

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

let $x_1 = xy_1$

$\delta(q, x_1 a) = \delta(\delta(q, x_1), a)$ by P-2.

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

$= \delta(\delta(\delta(q, x), y_1), a)$ by Induction hypothesis ——— (1)

R.H.S $= \delta(\delta(q, x), y)$

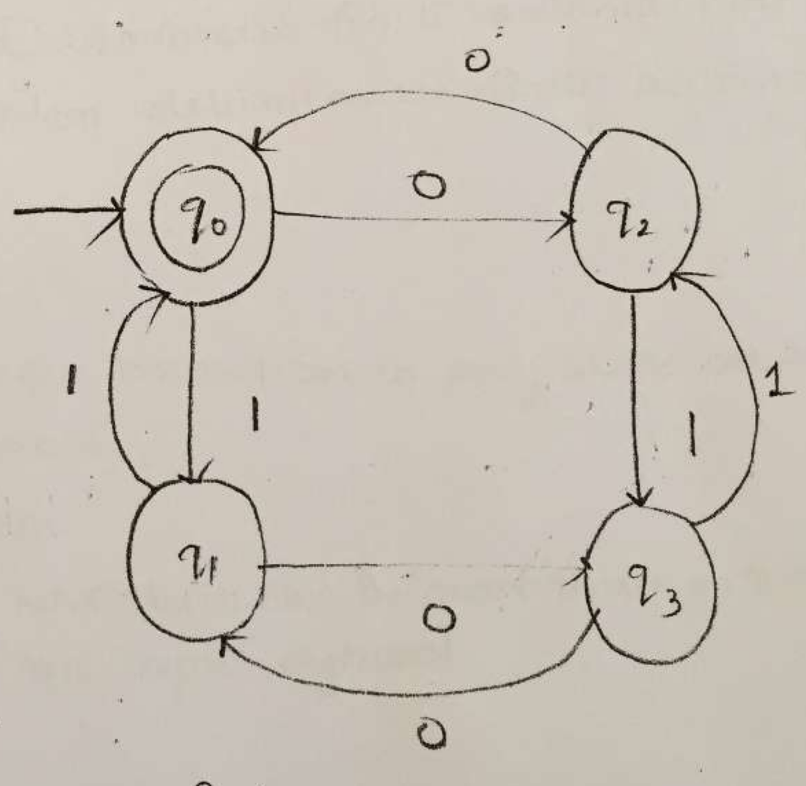
$= \delta(\delta(q, x), y_1 a)$

$= \delta(\delta(\delta(q, x), y_1), a)$ —— (2)

R.H.S = L.H.S

—— Hence proved.

	0	1
$\rightarrow q_0$	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.

$$\delta(q_0, \overset{\downarrow}{1}10101) = \delta(q_1, \overset{\downarrow}{1}0101)$$
$$= \delta(q_0, \overset{\downarrow}{0}101)$$
$$= \delta(q_2, \overset{\downarrow}{1}01)$$
$$= \delta(q_3, \overset{\downarrow}{0}1)$$
$$= \delta(q_1, \overset{\downarrow}{1})$$
$$= \delta(q_0, \wedge)$$
$$= q_0$$
$$q_0 \in F$$

so, given string is accepted

Finite automachine is 2 - types.

- Deteministic finite automachine (DFA)
- Non-deteministic finite automachine (NFA)

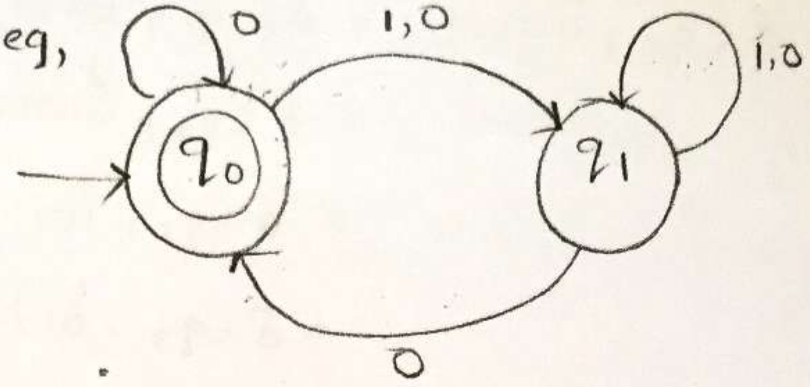
DFA :

unique transition in any state on an Input symbol.

NFA:

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

eg,

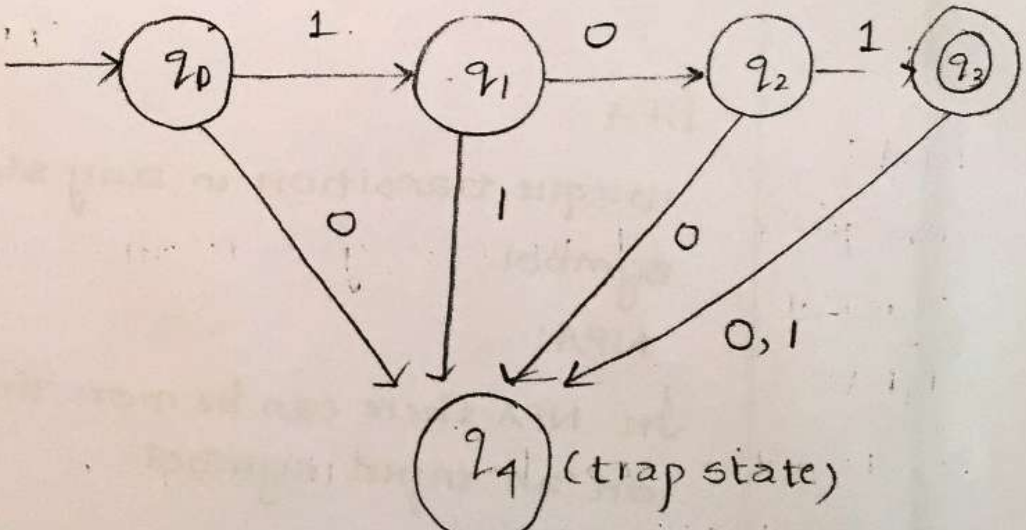


DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

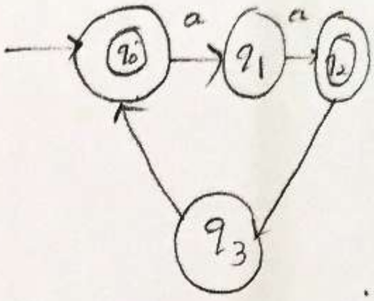
DFA can be used as a finite acceptor because its job is to accept certain input strings and rejects others.

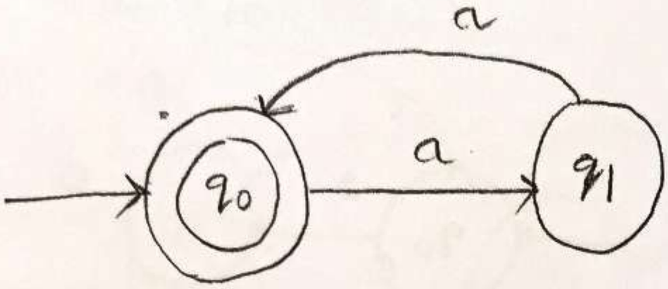
It is also called language recogniser because it recognises whether the input strings are in the language or not.

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



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

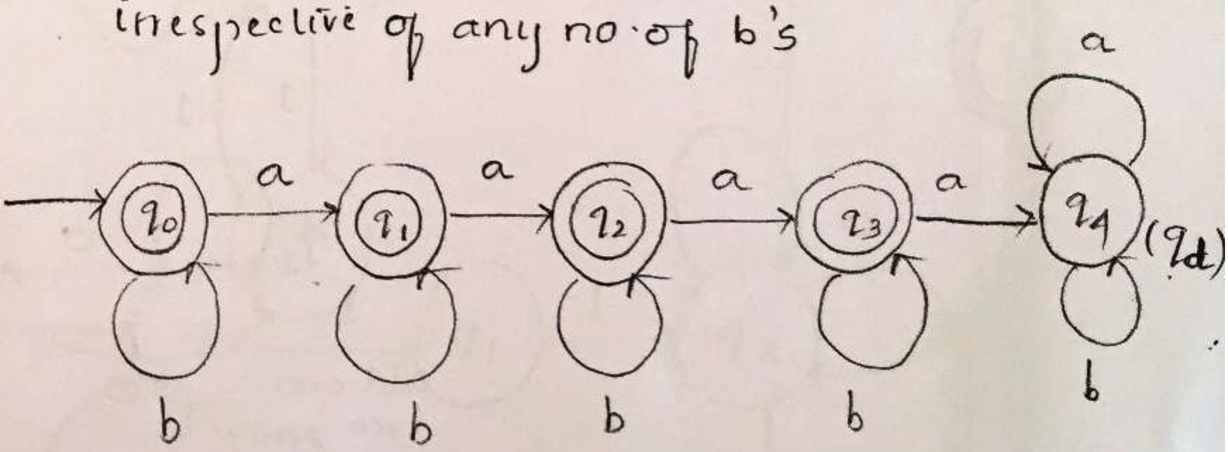




Design a DFA that accept all strings with atmost 3 a's over $\Sigma = \{a, b\}$

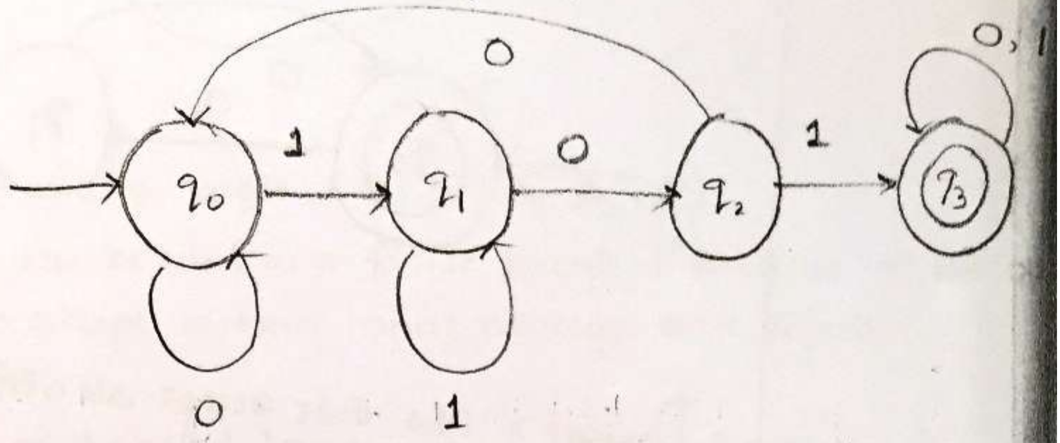
0 a's → accepted state
1 a's → accepted state
2 a's → accepted state
3 a's → accepted state
4 or more a's → rejected state.

irrespective of any no. of b's



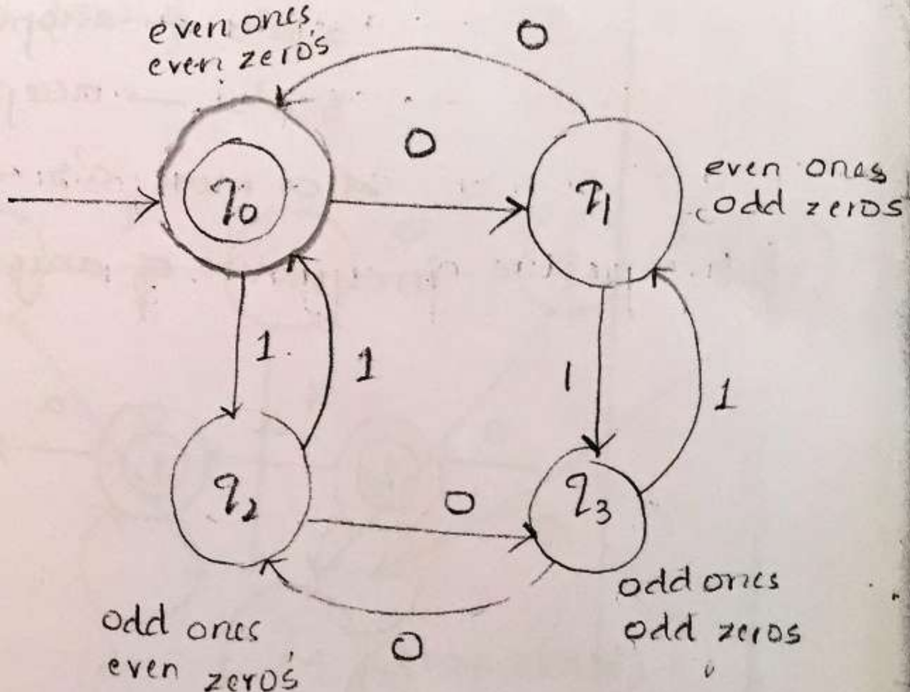
Design a DFA that contains 101 as a substring in all strings over $\Sigma = \{0,1\}$

$(1+0)^* 101 (1+0)^*$



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

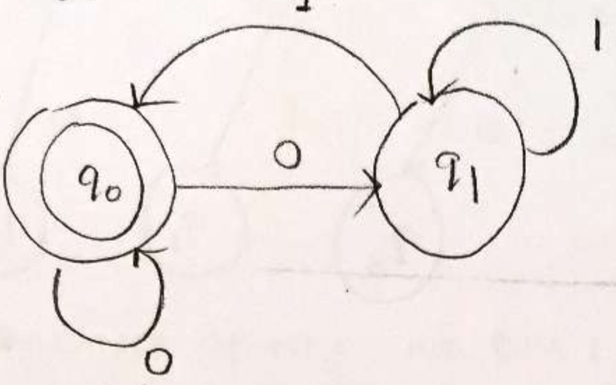
Design a DFA for even number of 1's & even numb of 0's.



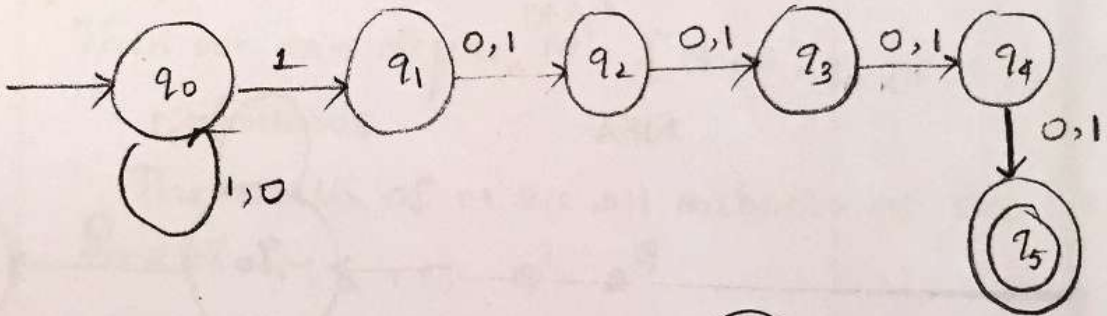
NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$
where δ is a transition function

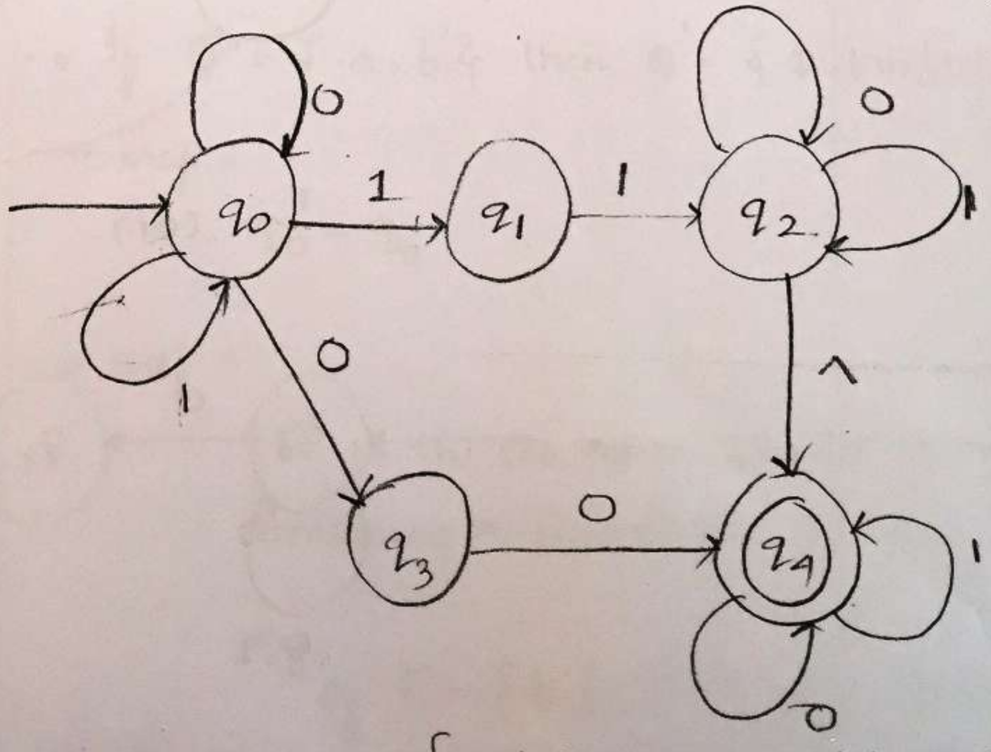
$$\delta : Q \times \Sigma \rightarrow 2^{Q}.$$

eg,



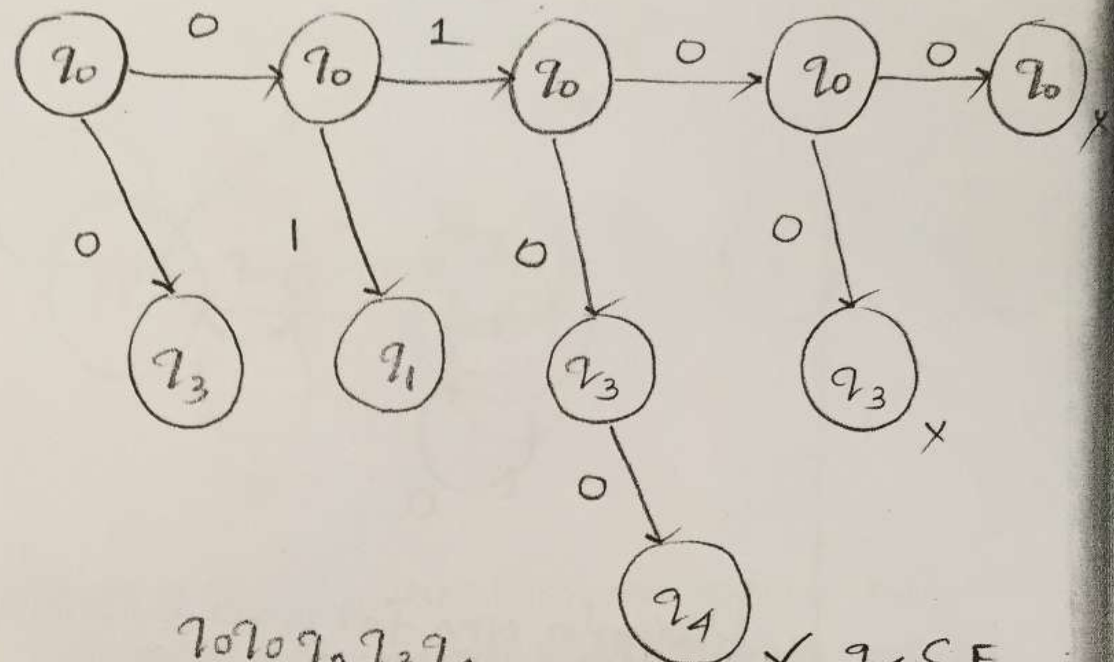
Design a NFA for a set of strings such that the 5th symbol from the right end is '1'.





find the acceptability of 0100

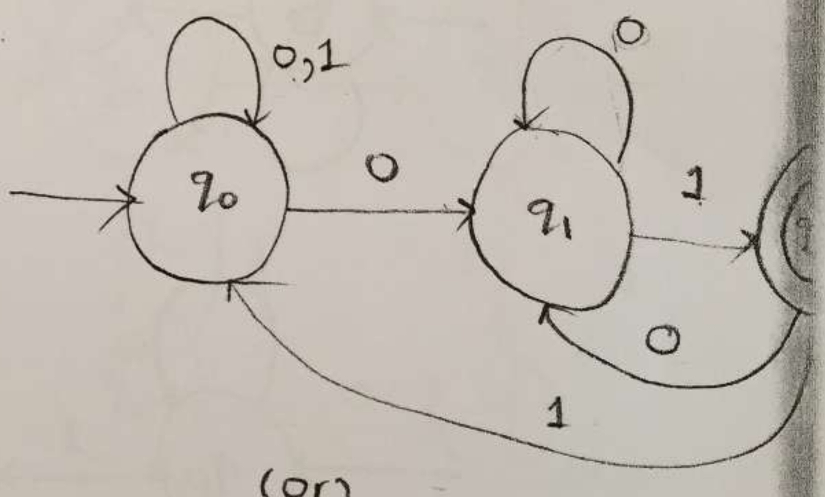
$\delta(q_0, 0100)$



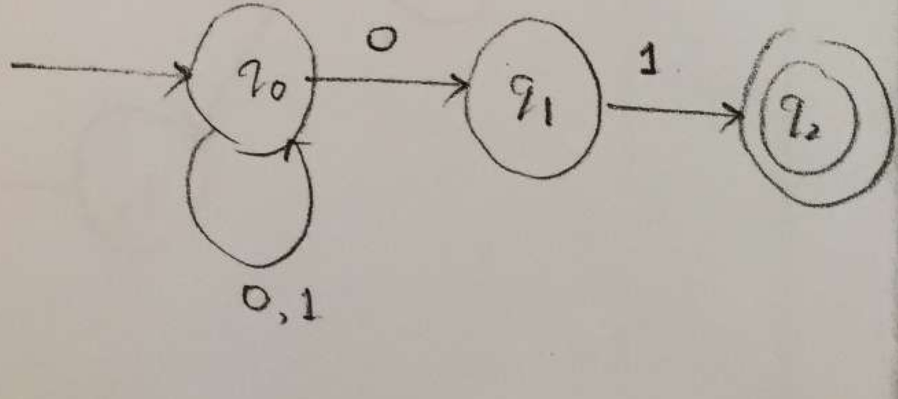
$q_0 q_0 q_0 q_3 q_4$ ✓ $q_4 \subseteq F$

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

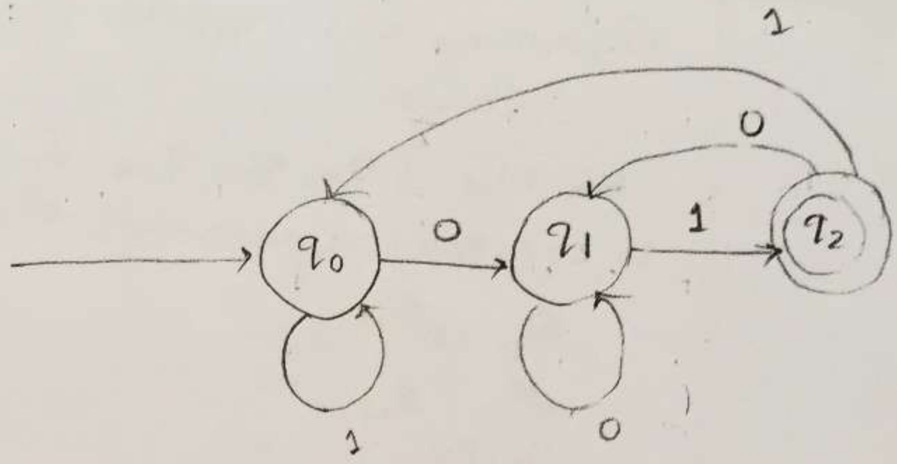
NFA :



(or)



DFA :



EQUIVIVALANCE 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 DFA $M' = \{Q', \Sigma', q_0', \delta', F'\}$ as follows

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

→ If $Q = \{a, b\}$ then $Q' = \{\phi, [a], [b], [ab]\}$

→ step 2:

$q_0' = q_0$

→ step 3:

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

eg, If $F = \{b\}$ then $F' = \{[b], [ab]\}$

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

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

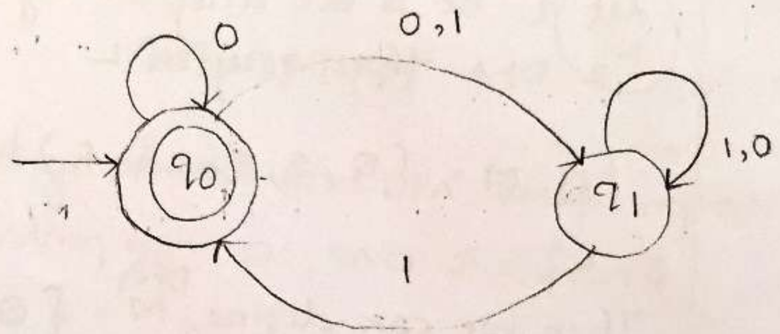
$Q_D = 2^{Q_N}$

$\Sigma_D = \Sigma_N$

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

F_D = set of all states of Q_D that contains atleast one final states of F_N.

Convert the following NFA to DFA



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

and

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

for DFA:

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

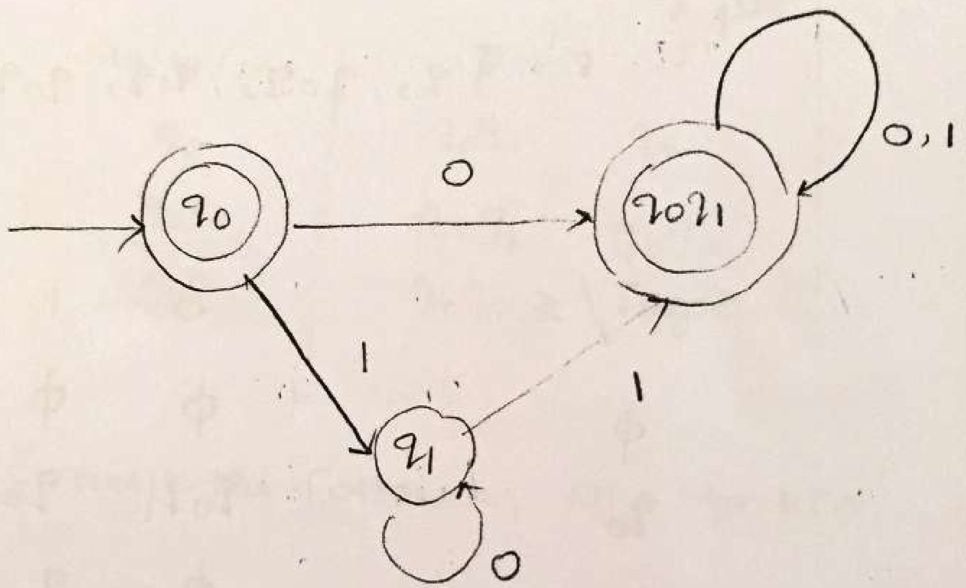
step 1 : $Q' = \{\phi; q_0, q_1, q_0q_1\}$

Step 2 : $q_0' = q_0$

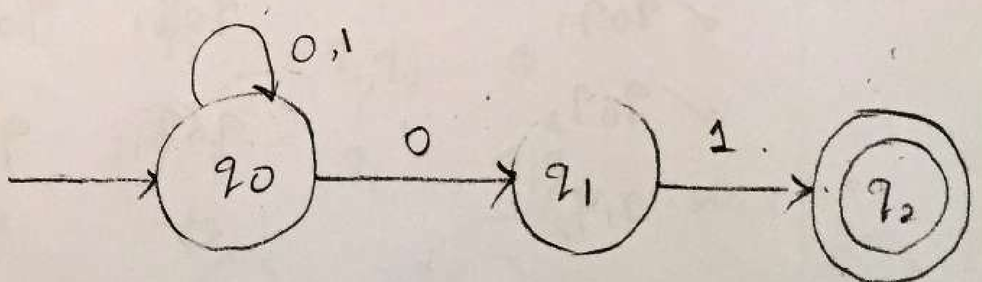
Step 3 : $F' = \{q_0, q_0q_1\}$

CONVERSION TO DFA :

states / Σ	0	1
ϕ	ϕ	ϕ
q_0	q_0q_1	q_1
q_1	q_1	q_0q_1
q_0q_1	q_0q_1	q_0q_1



Convert the following NFA to DFA



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

and

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

for DFA :

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

step 1 :

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

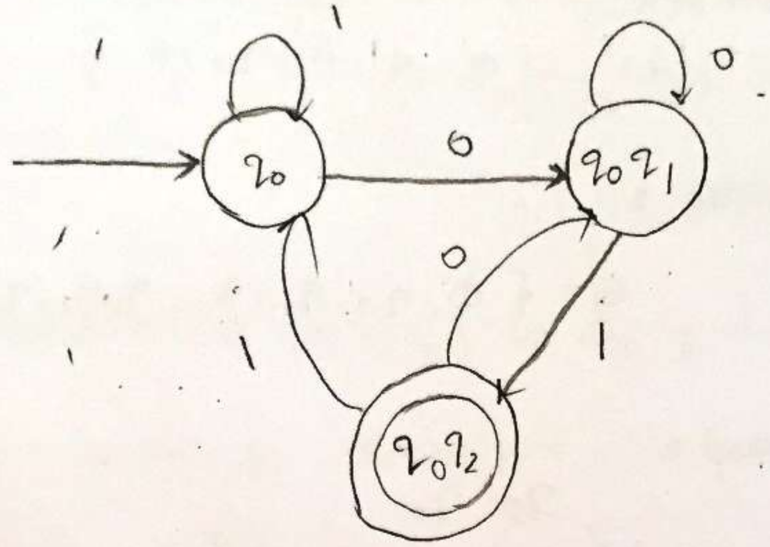
step 2 :

$$q_0' = q_0$$

step 3 :

$$F' = \{ q_2, q_0q_2, q_1q_2, q_0q_1q_2 \}$$

States / Σ	0	1
ϕ	ϕ	ϕ
q_0	q_0q_1	q_0
× q_1	ϕ	q_2
× q_2	ϕ	ϕ
✓ q_0q_1	q_0q_1	q_0q_2
✓ q_0q_2	q_0q_1	q_0
× q_1q_2	ϕ	q_2
× $q_0q_1q_2$	q_0q_1	q_0q_2

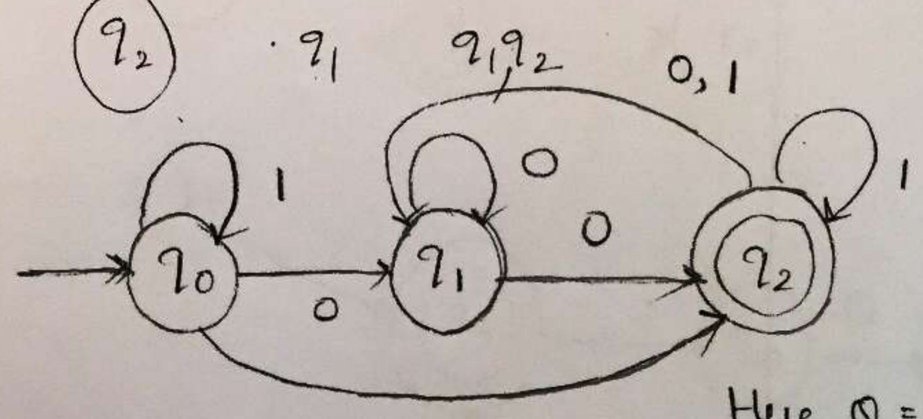


Alternate method :-

States/Σ	0	1
q_0	q_0q_1	q_0
q_0q_1	q_0q_1	q_0q_2
q_0q_2	q_0q_1	q_0

Convert the following NFA to DFA.

δ	0	1
→ q_0	q_1, q_2	q_0
q_1	q_1, q_2	ϕ
q_2	q_1	q_1, q_2



Here $Q = \{q_0, q_1, q_2\}$ &

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

for DFA:

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

Step 1:

$$Q' = \{ \phi, q_0, q_1, q_2, q_0q_1, q_0q_2, q_1q_2, q_0q_1$$

Step 2:

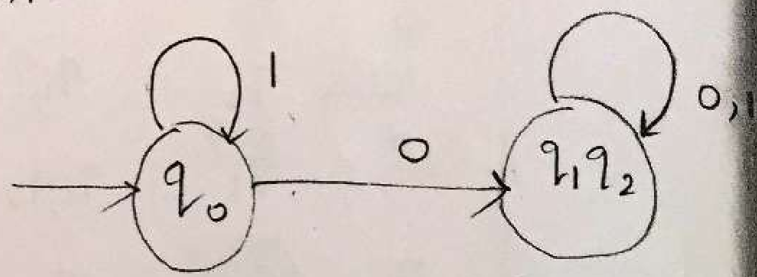
$$q_0' = q_0$$

Step 3:

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

states/Σ	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 state $[p_1 p_2 p_3]$ to Q_D if it is not there.

NFA WITH ε TRANSITION:

$0^* 1^* 2^*$

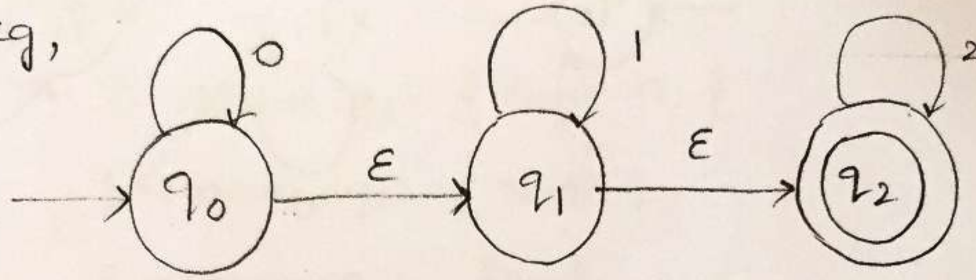
$0^l 1^m 2^n$

$l \geq 0$

$m \geq 0$

$n \geq 0$

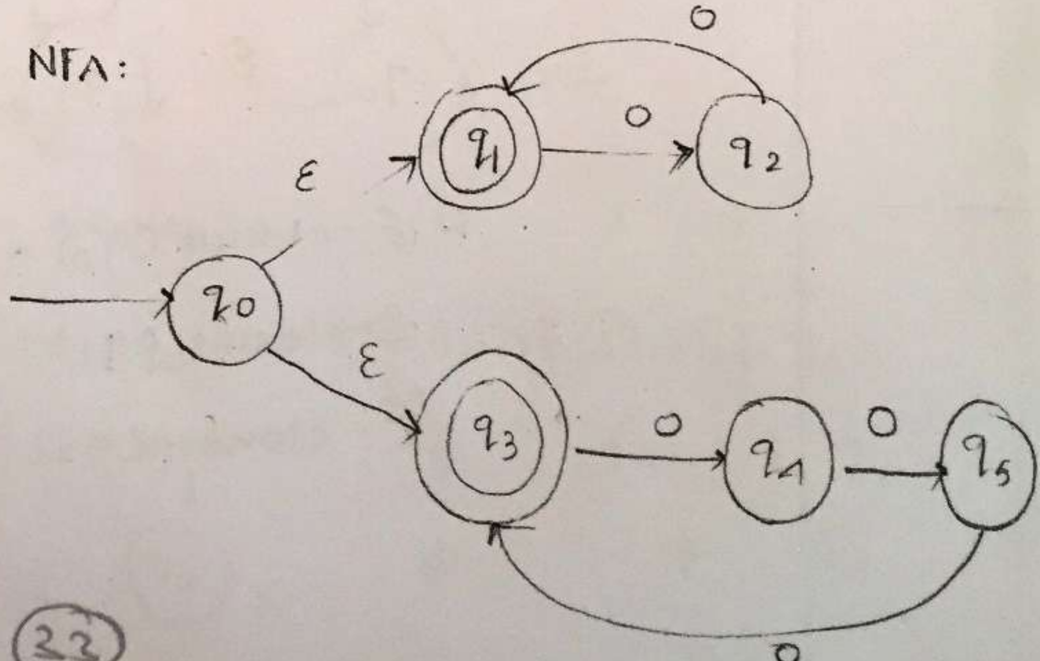
eg,



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 $L = \{0^k \,/\, k$ is multiple of 2 or 3 $\}$

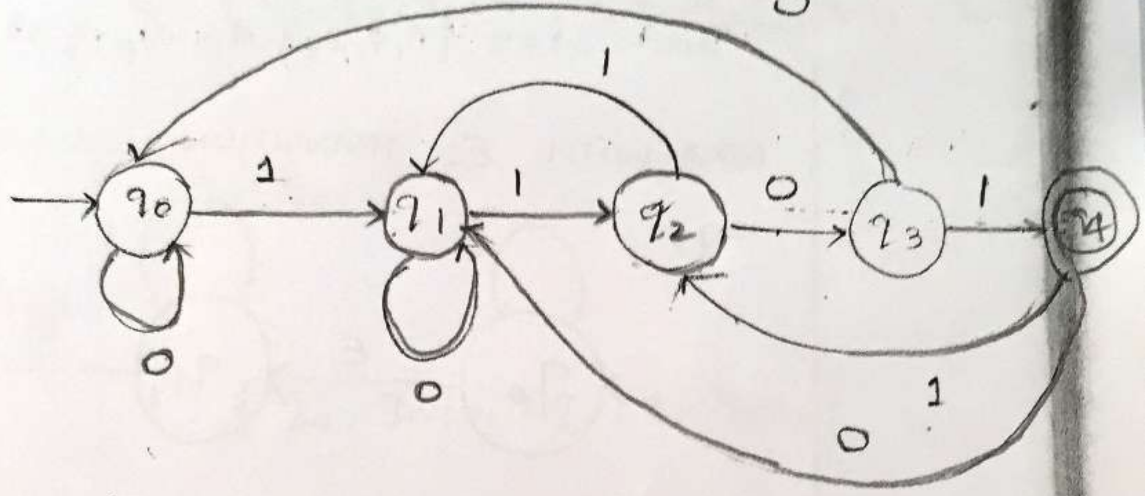
NFA:



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

$F = \{q_1, q_3\}$

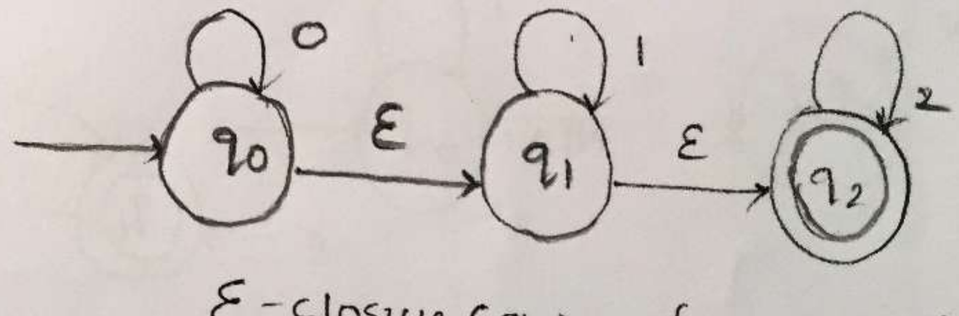
Design DFA ending with 101 & odd number of ones



'ε' CLOSURE :

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

for the below example



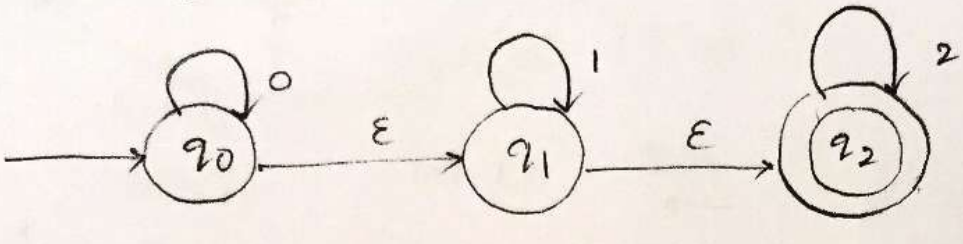
ε-closure $(q_0) = \{q_0, q_1, q_2\}$.

ε-closure $(q_1) = \{q_1, q_2\}$.

ε-closure $(q_2) = \{q_2\}$.

ELIMINATING 'ε' TRANSITIONS:

Converting NFA with ε transition to NFA without ε transition:



Step 1:

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

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

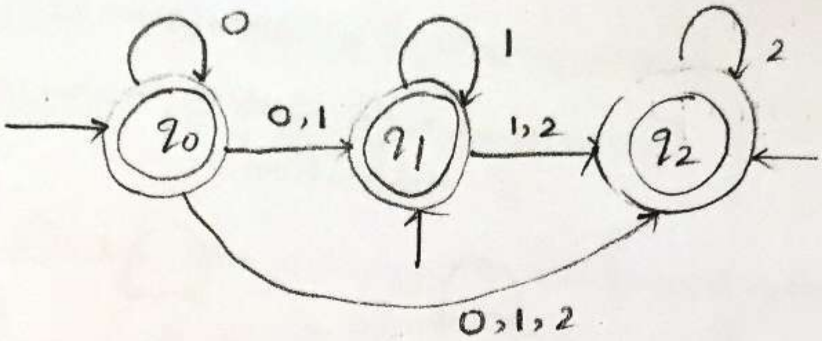
$\varepsilon\text{-closure}(q_2) = \{q_2\}$.

Step 2:

States/Σ	0	1	2	ε
$\rightarrow q_0$	q_0	ϕ	ϕ	q_1
q_1	ϕ	q_1	ϕ	q_2
q_2	ϕ	ϕ	q_2	ϕ

Step 3:

States/Σ	0	1	2
$\rightarrow q_0$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	q_2
q_1	ϕ	$\{q_1, q_2\}$	q_2
q_2	ϕ	ϕ	q_2



→ .

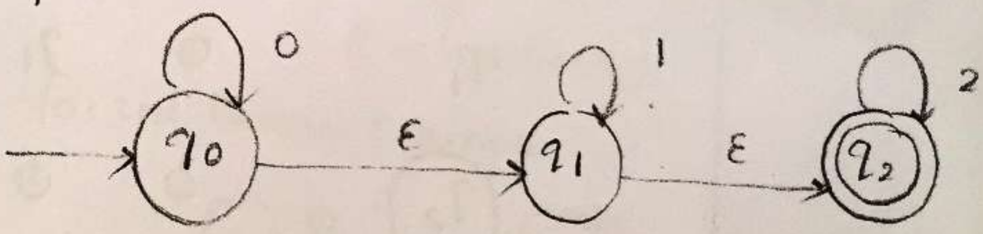
for each state compute ε closure on each input symbol a ∈ Σ if the ε-closure of a state contains final state then make that state as final. ε closure of initial state are the initial states of the initial states.

CONVERTING NFA WITH 'ε' TRANSITIONS TO DFA

Step 1 :

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

Step 2 :



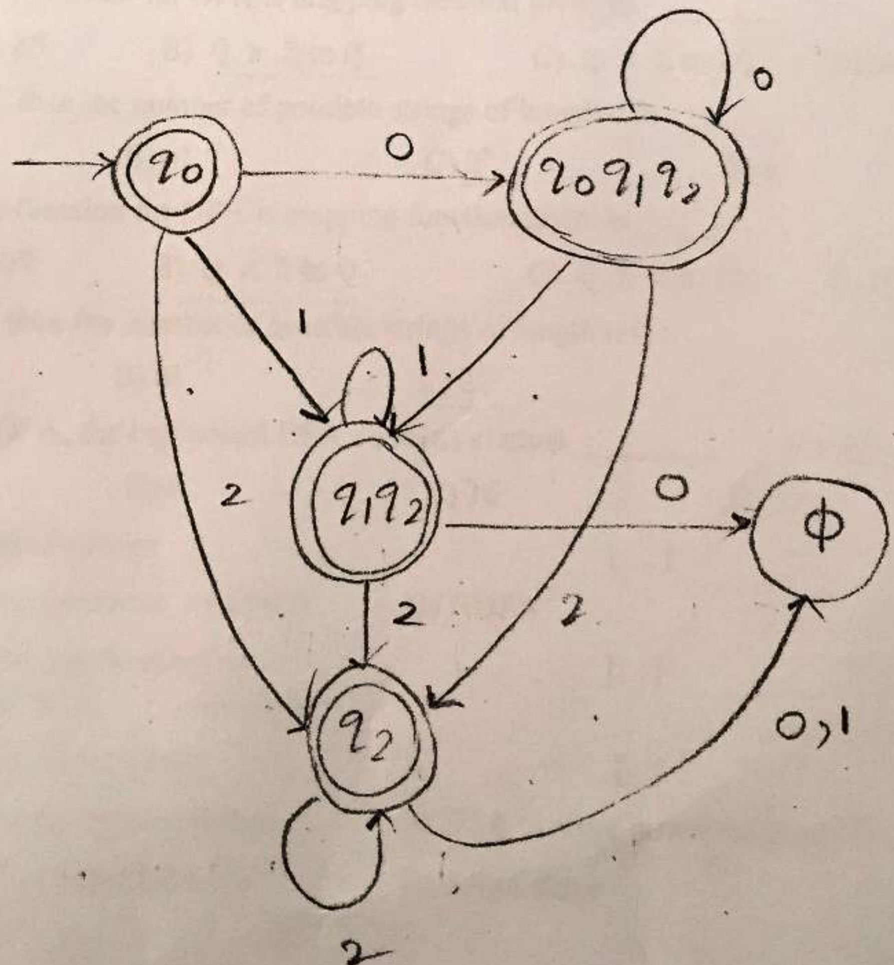
states/Σ	0	1	2	ε
q_0	q_0	ϕ	ϕ	q_1
q_1	ϕ	q_1	ϕ	q_2
q_2	ϕ	ϕ	q_2	ϕ

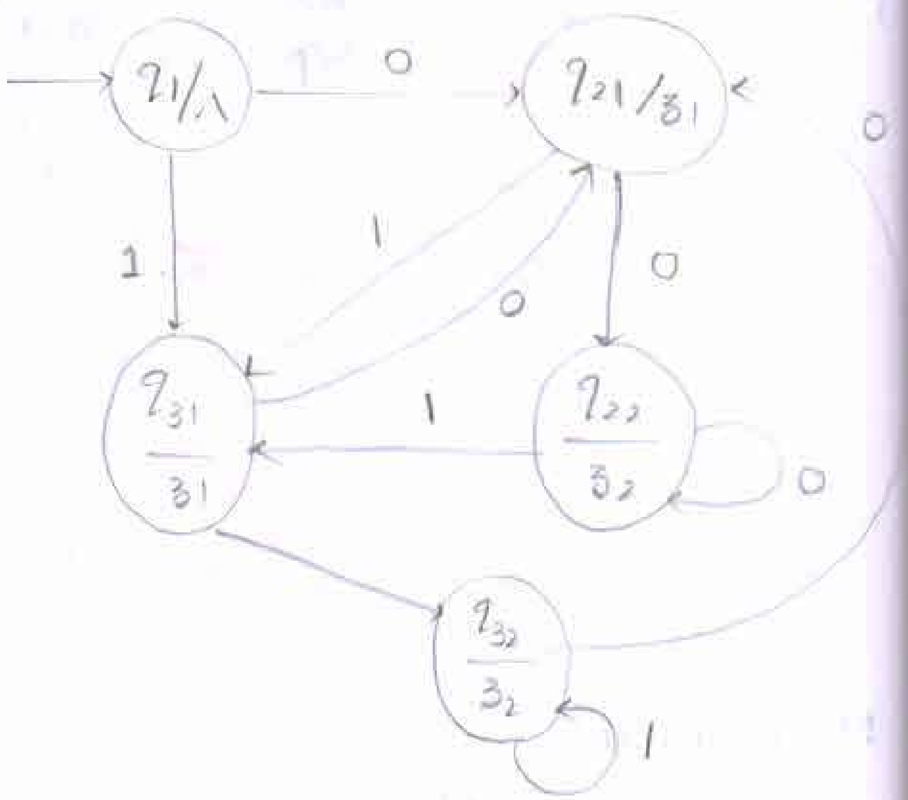
step 3 :

states/Σ	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
$q_1 q_2$	ϕ	$q_1 q_2$	q_2
q_2	ϕ	ϕ	q_2

step 4 :

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





MINIMISATION OF FINITE AUTOMATA :

Two states q_1 & q_2 are equivalent if bo[th] $\delta(q_1, x)$ 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 $\delta(q_1, x)$ [and] $\delta(q_2, x)$ for all $x \in \Sigma^*$ (there may be inf[inite] no. of strings).

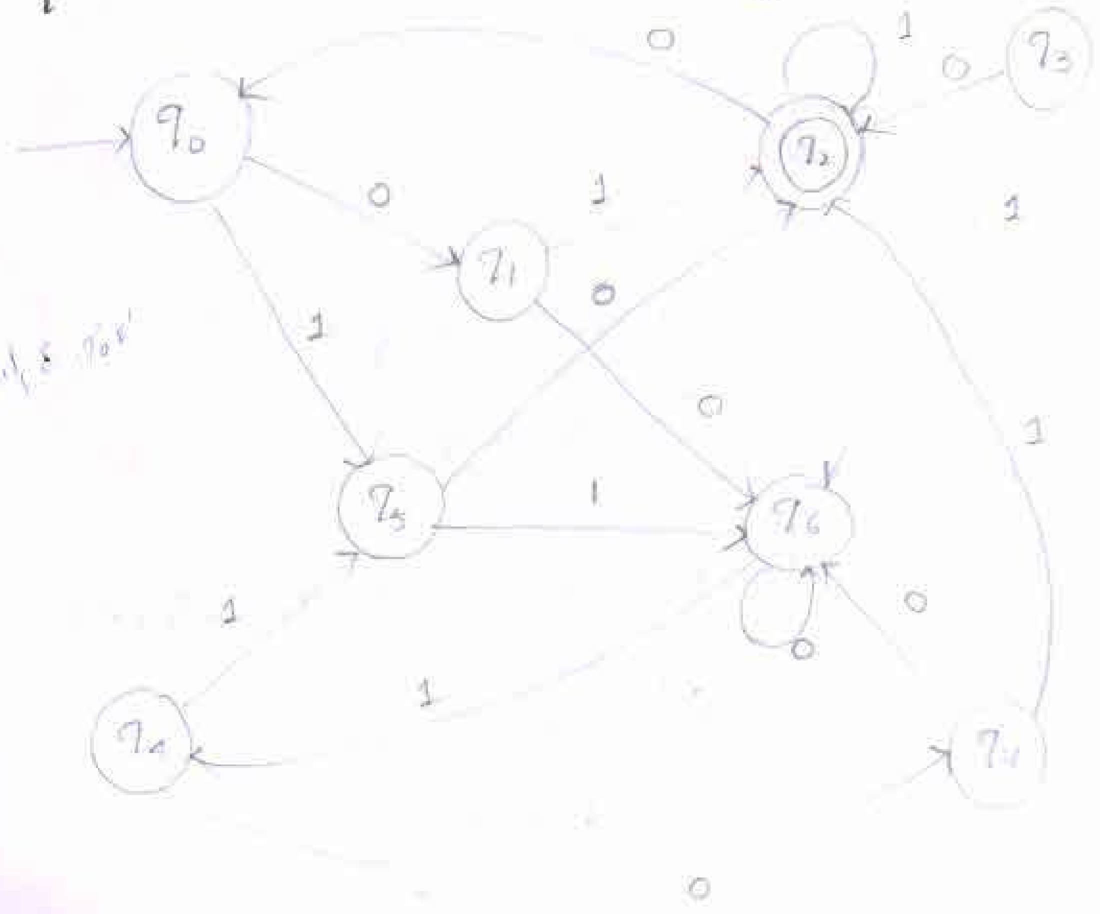
Two states q_1 and q_2 are K-equivalent (K ≥ 0) if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final [states] or both are non-final states for all strin[gs] x of length K or less.

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, equivalance and k - equivalence are equivalence relations i.e, they are reflexive, symmetric and transitive.

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



States/Σ	0	1
→ q_0	q_1	q_5
q_1	q_6	q_2
q_2	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

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

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

$Q_1^0 = q_2$

$Q_2^0 = Q - Q_1^0$

Step 2 Π_1

$Q_1^1 = \{q_2\}$

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

$Q_3^1 = \{q_0, q_4, q_6\}$

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

$Q_1^2 = \{q_2\}$

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

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

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

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

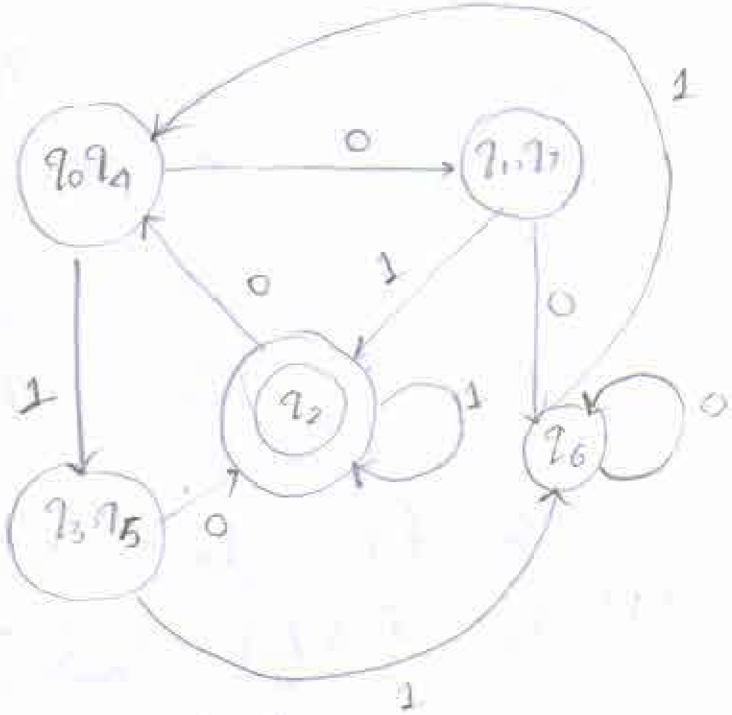
$\Pi_2 = \{\{q_2\}, \{q_1, q_7\}, \{q_3, q_5\},$
$\{q_0, q_4\}, \{q_6\}\}$

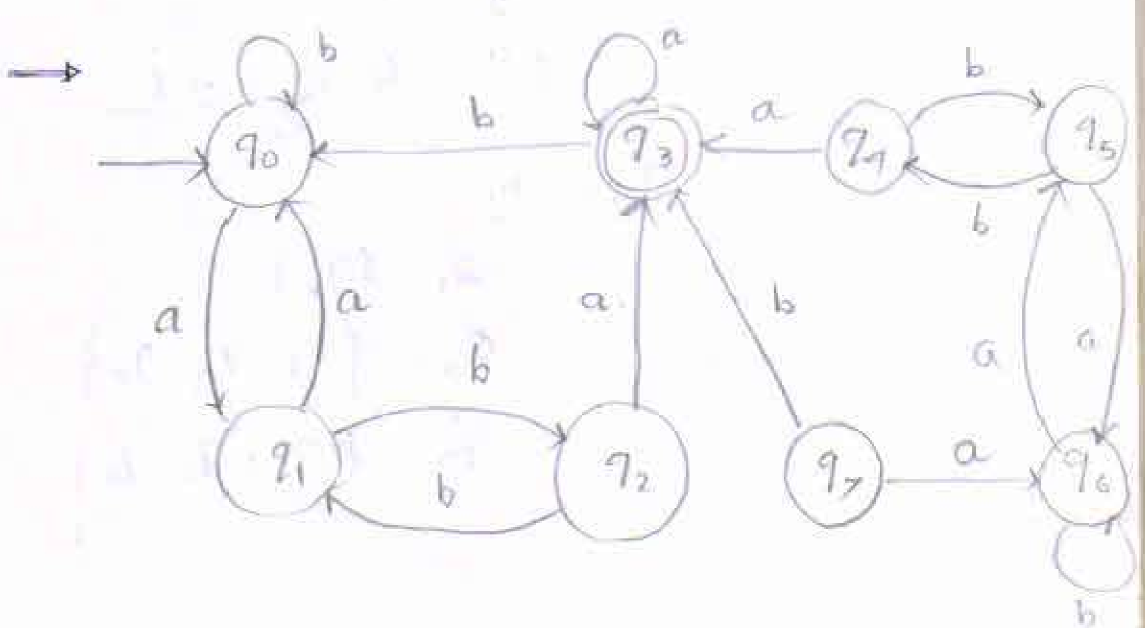
$M' = \{ Q', \{0,1\}, \delta', q_0', F' \}$

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

$q_0' = [q_0, q_4]$

$F' = \{ q_2 \}$.





States	input a	input b
$\rightarrow q_0$	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: $\Pi_Q = \{\{q_3\}, \{q_0, q_1, q_2, q_4, q_5, q_6, q_7\}\}$

$Q_1^0 = q_3 \rightarrow f$

$Q_2^0 = Q - Q_1^0 \rightarrow n.f$

Step 2: Π_1

$Q_1' = \{q_3\}$

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

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

$3,1 \quad 3,5 \quad 6,3 \qquad 6,4 \quad 5,6$

$\pi_1 = \{ \{q_3\}, \{q_2, q_4, q_7\}, \{q_0, q_1, q_5, q_6\} \}$

$1,10 \quad 0,12$

$Q_1^2 = \{q_3\}$

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

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

$Q_4^2 = \{q_2, q_4\} \qquad Q_5^2 = \{q_7\}$

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

$0,12 \quad 6,4 \qquad 1,0 \quad 5,6$

$\{q_2, q_4\}, \{q_7\} \}$

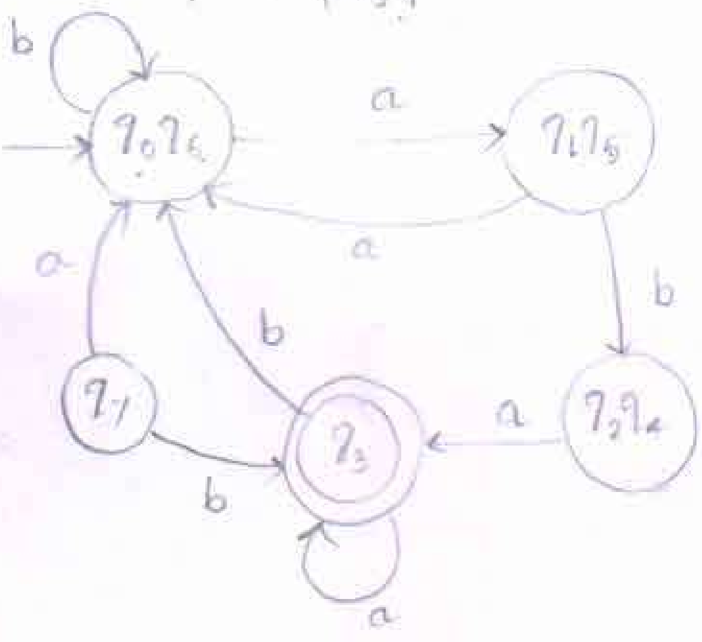
$3,1 \quad 3,5$

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

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

$q_0' = [q_0, q_6]$

$F' = \{q_3\}$



def:

If q_i, q_j are said to be K-equivalance $\forall K \geq$
then q_i, q_j are said to be equivalance

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

3. FORMAL LANGUAGES:

Chomsky → classification of languages

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

left context   right context

Grammar is basically 4 - tuple

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

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

Σ - it is finite set collection of terminals

here,

$$V_N \cap \Sigma = \phi$$

S - starting symbol where $S \in V_N$

S ⟶ <Noun><verb>

S ⟶ <Noun><verb><adverb>

Noun → Priya, Raj

verb → ran, ate

adverb → quickly, slowly

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

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

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

$S \rightarrow 0S1 / \wedge$

$0S1$

$00S11$

0^2S1^2

00^2S11^2

0^3S1^3

replacing s by $\wedge$ 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

→ $abAbcd \longrightarrow abABbcd$

ab → left context

bcd → right context

here $\alpha = AB$.

→ $AC\wedge \rightarrow A\wedge\wedge$

left context is A

right context is $\wedge$

and $\alpha = \wedge$

→ $C \rightarrow \wedge$ (here 'c' is erased)

leftcontext & right context is '$\wedge$'

here $\alpha = \wedge$

type 0 → A production without any restrictions

type 1

type 2

type 3

TYPE - 1

A production of the form $\phi A \psi \to \phi \alpha \psi$ 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) $a\underline{A}bcD \to a\underline{bcD}bcD$

type 1 grammar

(2) $\underline{A}B \longrightarrow \underline{A}bBc$

$\downarrow$ left $\qquad$ $\downarrow$ left

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

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

The production $S \to \Lambda$ 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 type1 or context sensitive language

In a context sensitive grammar G, we allow $S \to \Lambda$ apart from $S \to \Lambda$ all the other productions donot decrease the length of the working string.

TYPE-2

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

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

eg,

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

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 $A \to a$ or $A \to aB$ 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 $S \to \Lambda$ 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 number which can be applied to following grammars

(a) $S \rightarrow Aa \rightarrow$ type 2

$A \rightarrow c \mid Ba$

$c \downarrow$ type-3, $Ba \downarrow$ type-2

$B \rightarrow abc \rightarrow$ type 2.

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

$ASB \downarrow$ type 2, $d \downarrow$ type 3

$A \rightarrow aA$

$\downarrow$ type 3

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

$aS \downarrow$ type 3, $ab \downarrow$ type 2

REGULAR EXPRESSION & REGULAR GRAMMAR

Any terminal or element of Σ is regular expression

for eg, a in Σ $\downarrow$ $\{a\}$, ϕ is also regular expression ↑ empty set

$\wedge$ is regular expression $\downarrow$ $\{\wedge\}$

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}

CONCANTENATION:

Concantenation of '2' regular expre-ssions 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

Λ, a, aa, aaa ...... are also regu[lar] expressions

NOTE:

If 'L' is a language represented by th[e] 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 precedance next highest is for concantenation and least is for union

IDENTIFIER NOTATION

notation : $l(l/d/s)^*$

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 {Λ, a, aa, aaa ......}

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

{Λ, a, b, aa, ab, ba, bb, aaa, aab, aba, ...}

Regular set	Regular expression
{101}	101
{Λ, a}	Λ+a

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

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

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

→ $(0+1)^*00$

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

$1(11)^*$ or $(11)^*1$

Strings of 0's and 1's with atleast '2' connective zeros

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

All strings of 0's and 1's begining with 1 or 0 and not having '2' consecutive zeros

$(1+01)^*$

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

$(1+011)^*$

set of all string's ending with 011

$(0+1)^*011$

Identities for regular expressions

$(I)_1$ $\phi + R = R$

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

I_3 $\Lambda R = R\Lambda = R$.

I_4 $\Lambda^* = \Lambda$ & $\phi^* = \Lambda$.

I_5 $R + R = R$

I_6 $R^* R^* = R^*$

I_7 $RR^* = R^* R$

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

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

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

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

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

→ '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:

Case(i) $R = Q + RP$

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

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

$R = QP^*$ (from I_q)

Case (ii) :

$R = Q + RP$

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

$= Q + QP + RP^2$

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

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

⋮

$= Q + QP + QP^2 + \cdots + QP^i + QP^{i+1}$

$= Q(\Lambda + P + P^2 + \cdots P^i) + RP^{i+1}$

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

$= Q(P^*)$

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

Proove that the regular expression $R = \Lambda + 1^*(011)^*(1^*(011)^*)^* = (1+011)^*$

$L.H.S = \Lambda + 1^*(011)^*(1^*(011)^*)^*$

let $R = 1^*(011)^*$

$L.H.S = \Lambda + R(R)^*$

$= R^* \text{ by } I_9$

$L \cdot H \cdot S = (1^*(011)^*)^*$

$= (1+011)^* \text{ by } I_{11}$

$= R \cdot H \cdot S$

ALGEBRA LAW FOR REGULAR EXPRESSIONS:

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

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

→ concantenation operation on regular expression are associative

$$r(st) = (rs)t$$

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

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

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

→ $\phi^* = \wedge$

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 to 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 :

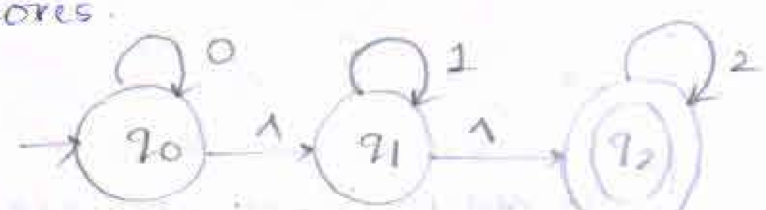
If v_1 is the initial state, make v_2 also as initial state

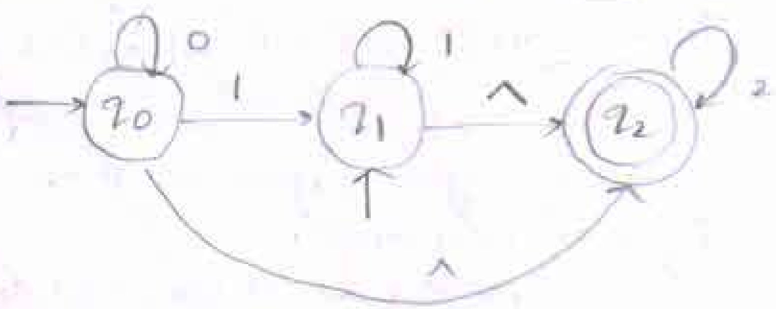
Step 4 :

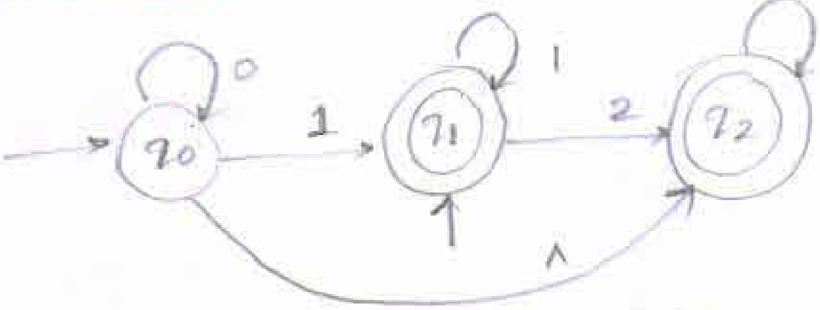
If v_2 is a final state make v_1 as the final state

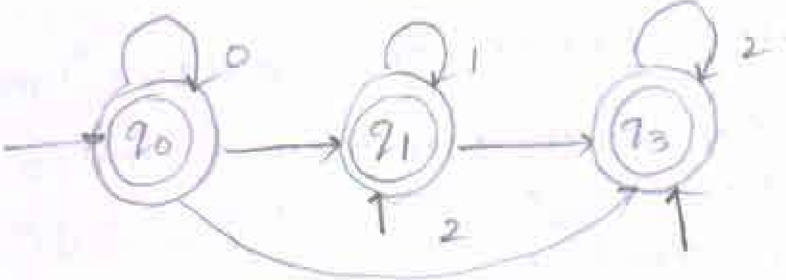
eg,

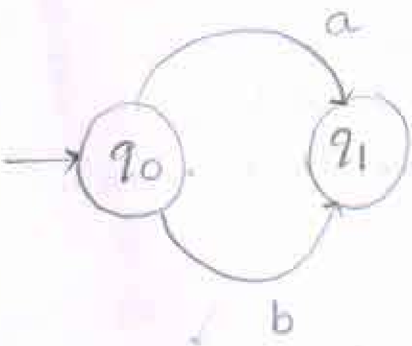
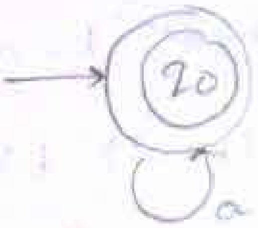
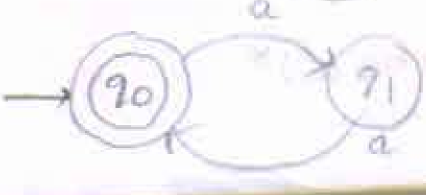
Consider a finite automata will null moves & obtain a equivalent automata without null moves.

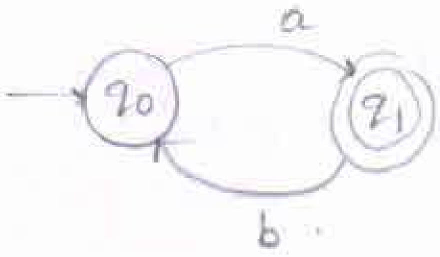








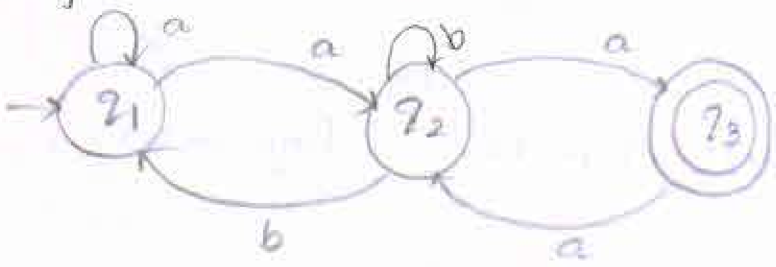
Finite automata	Regular set	Regular expression
	$\{a, b\}$	$a+b$
	$\{ab\}$	ab
	$\{\Lambda, a, aa, aaa \ldots\}$	a^*
	$\{a, aa, aaa \ldots\}$	a^+
	$\{\Lambda, aa, aaa \ldots\}$	$(aa)^*$



$\{a, aba, ababa \ldots\}$ $a(ba)^*$

Algebric method using Arden's Theorem

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



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

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

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

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

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

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

$$q_3 = q_2 a$$

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

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

The following method is an extension of arden's theorem. This is used to find the expression recognised by a transition system. The following assumption's 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 $V_1, V_2, \ldots V_n$
(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) A 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 $\alpha_{ij} = \phi$ consequently, we get a following set of equations in V_1 to V_n

$$V_1 = V_1\alpha_{11} + V_2\alpha_{21} + \cdots + V_n\alpha_{n1} + \Lambda$$

$$V_2 = V_1\alpha_{12} + V_2\alpha_{22} + \cdots + V_n\alpha_{n2}$$

$$\vdots$$

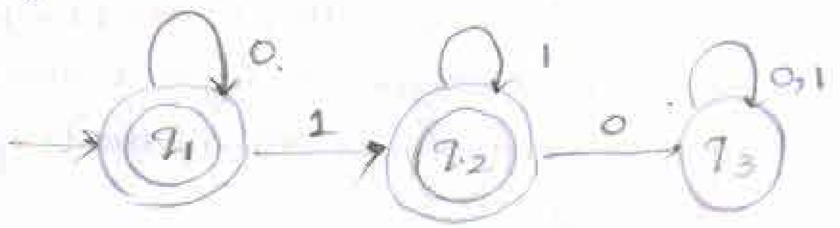
$$V_n = V_1\alpha_{1n} + V_2\alpha_{2n} + \cdots + V_n\alpha_{nn}$$

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

$R = Q + RP$
$R = QP^*$

Describe in english (statements) the set acc by finite automata whose transition diagram is,



$$q_1 = q_1 0 + \Lambda$$
$$R = RP \quad Q$$
$$q_2 = q_2 1 + q_1 1$$

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

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

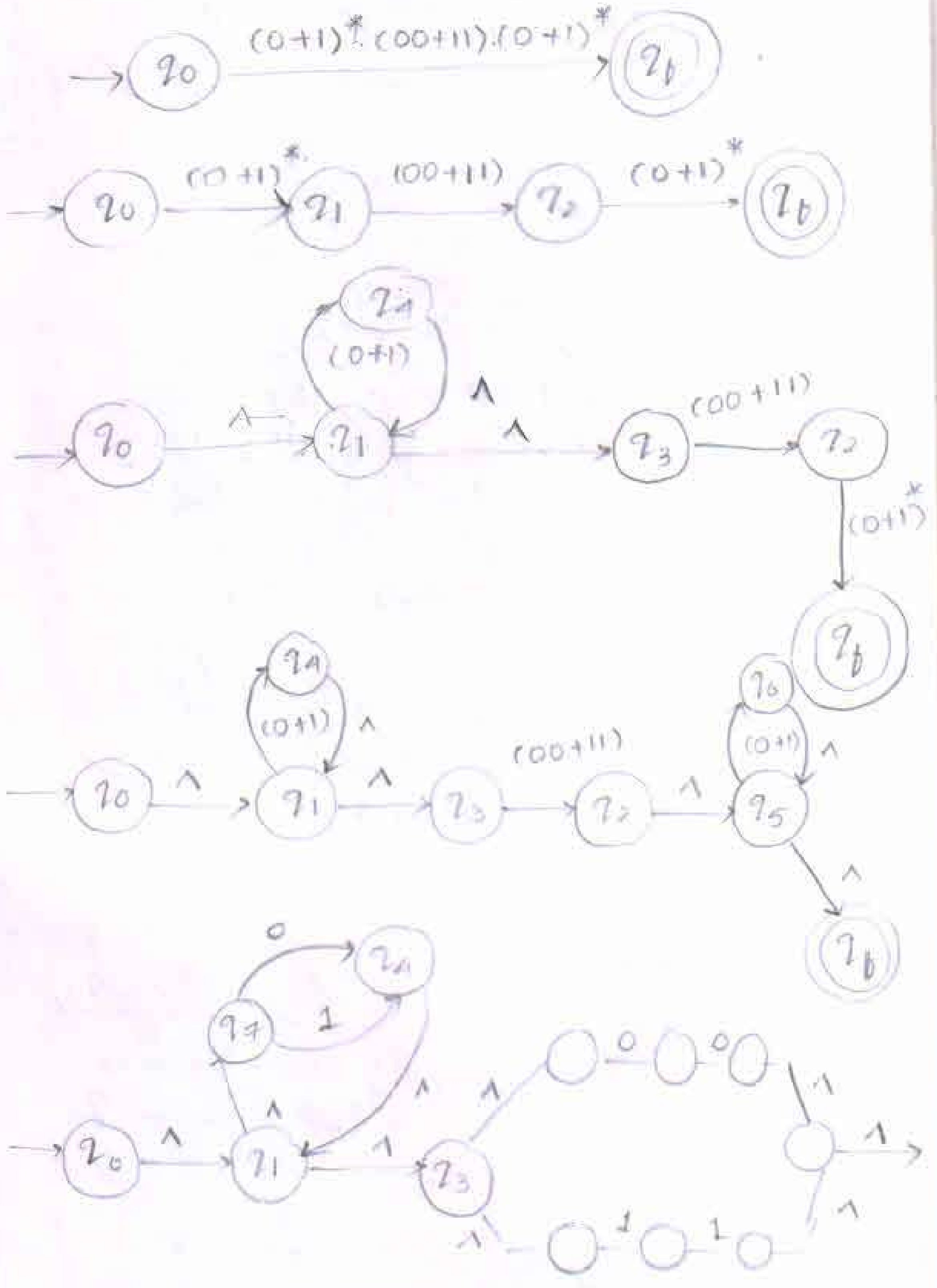
$$q_1 = \Lambda + q_1 0$$
$$q_1 = \Lambda 0^*$$
$$q_1 = 0^* \checkmark$$

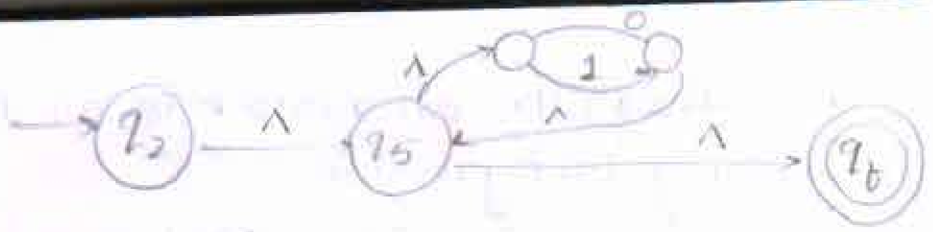
$$q_2 = q_2 1 + \underline{q_1 1}$$
$$R \quad RP \quad Q \qquad = Q(P)^*$$
$$q_2 = q_1 1 1^* \text{ by ardens theorem}$$
$$q_2 = 0^* 1 1^*$$
$$q_2 = 0^* 1^*$$

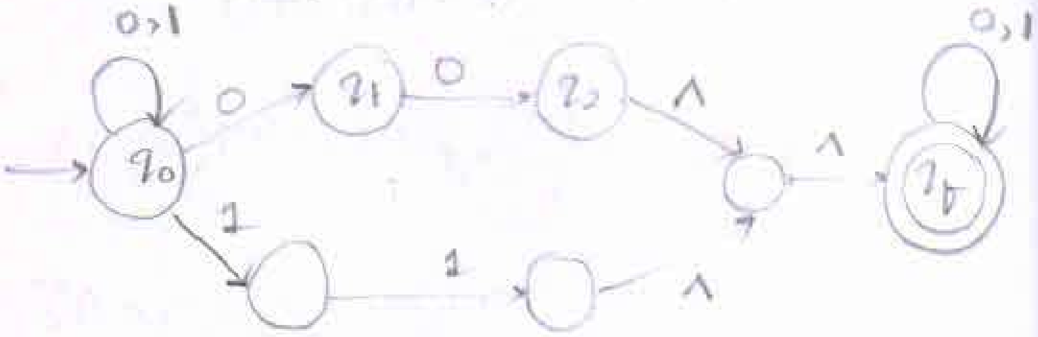
$$q_1 + q_2 = 0^* + 0^* 1^* \quad 1^* 0$$
$$= 0^* (\Lambda + 1^*)$$
$$= 0^* 1^*$$

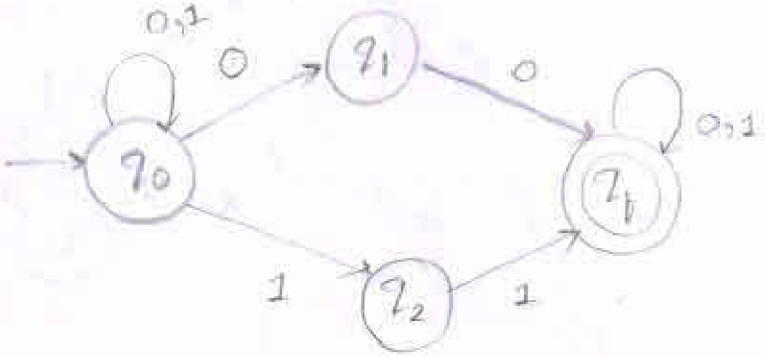
Construct the finite automata equivalent to the regular expression

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









NFA

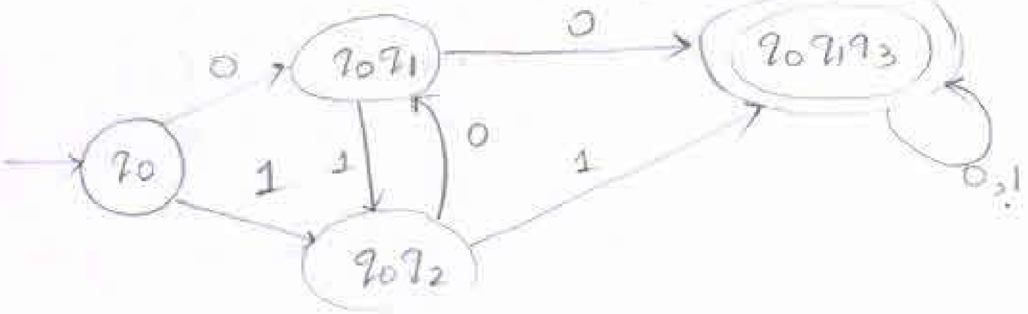
NFA table

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

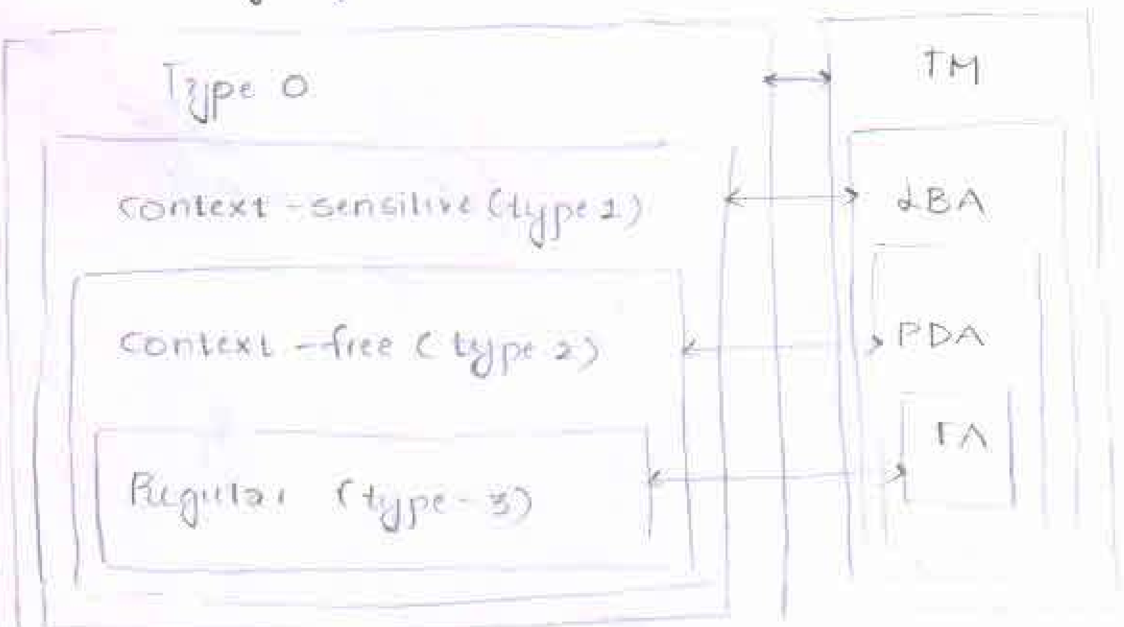
NFA to DFA table

	0	1
$\rightarrow q_0$	$[q_0 q_1]$	$[q_0 q_2]$
$[q_0 q_1]$	$[q_0 q_1 q_3]$	$[q_0 q_2]$
$[q_0 q_2]$	$[q_0 q_1]$	$[q_0 q_2 q_3]$ ($[q_0 q_1 q_3]$)
$[q_0 q_1 q_3]$	$[q_0 q_1 q_3]$	$[q_0 q_2 q_3]$ ($[q_0 q_1 q_3]$)
$[q_0 q_2 q_3]$	$[q_0 q_1 q_3]$	$[q_0 q_2 q_3]$ ×

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



Languages | Automata



where

TM - Turing machine

LBA - linear bounded automaton

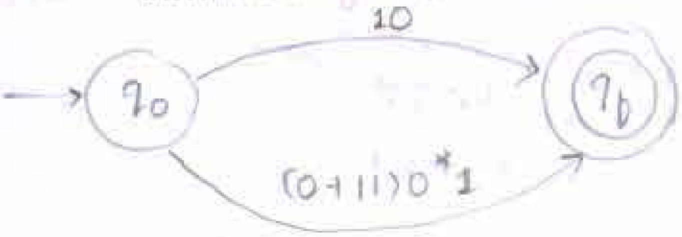
PDA - push down automaton

FA - Finite automata

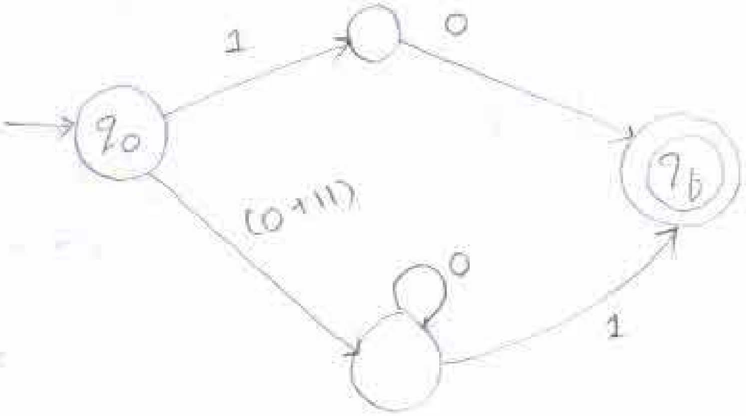
Construct DFA with reduced states equiva
of to the regular expression

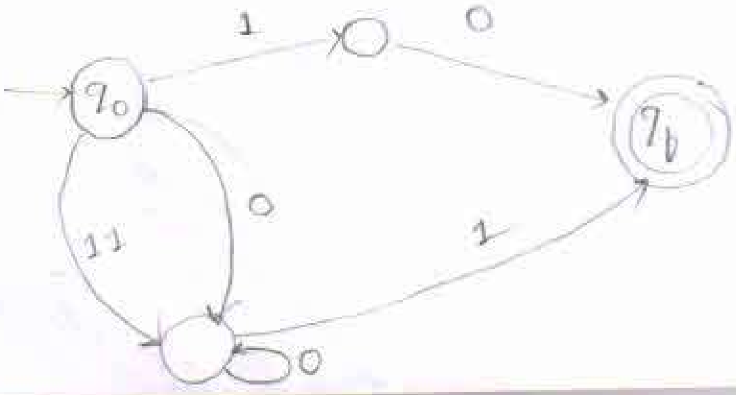
$$10+(0+\emptyset 11)0^*1$$

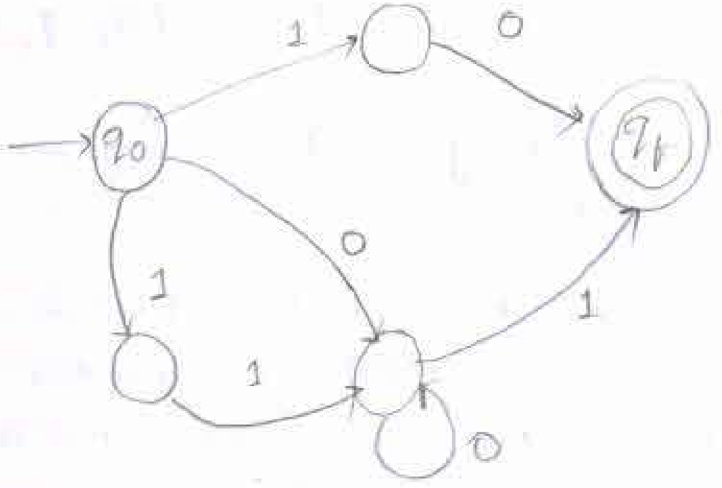
elimination of union



elimination of concatenation

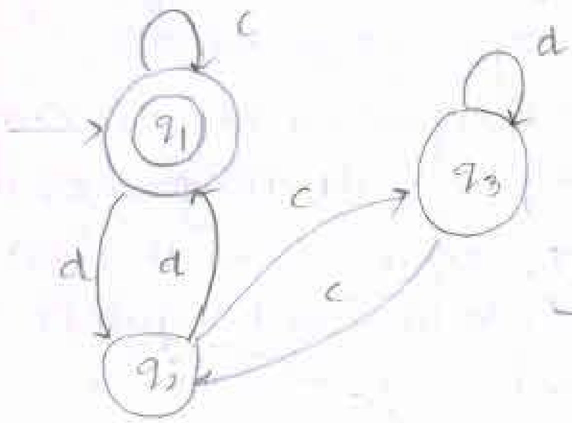




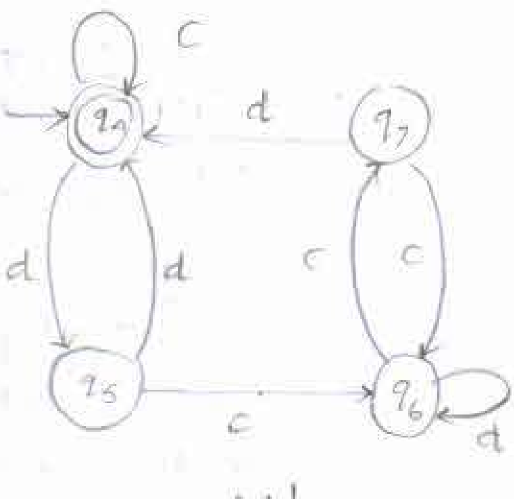


COMPARISION METHOD OF DFA's:

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



M



M'

	(q_c, q_c')	(q_d, q_d')
(q_1, q_4)	(q_1, q_4)	(q_2, q_5)
(q_2, q_5)	(q_3, q_6)	(q_1, q_4)
(q_3, q_6)	(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 '2' 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 the 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' cases

Case 1:

If reach a pair (q, q') such that q is the final state of M & q' is the non-final state of M' or viceversa 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'

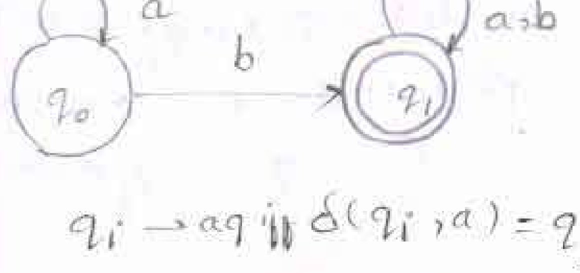
→ we construct a another DFA $M' = (Q, \Sigma, \delta, q_0, F')$

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:



$q_i \to a q$ iff $\delta(q_i, a) = q_j$ with $q_j \notin F$

$q_i \to a q_j$, $q_i \to a$ iff $\delta(q_i, a) \in F$

$q_0 \rightarrow a q_0$

$q_0 \rightarrow b q_1$ $\quad$ $q_0 \rightarrow b$

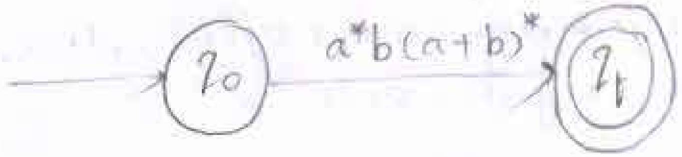
$q_1 \rightarrow a q_1$ $\quad$ $q_1 \rightarrow a$

$q_1 \rightarrow b q_1$ $\quad$ $q_1 - b$

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

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

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



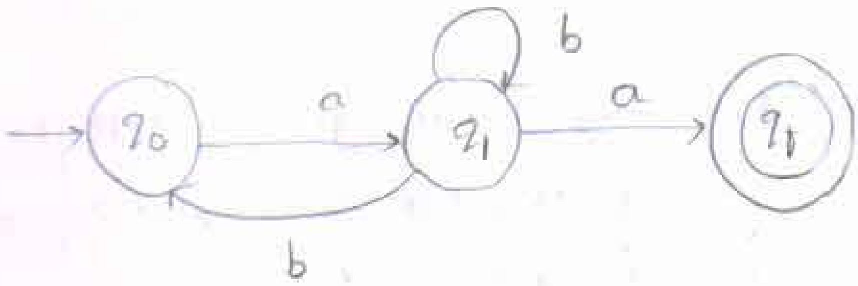
Construction of transition system for a given regular grammar

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

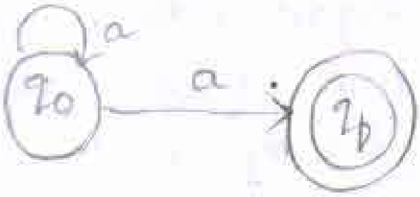
each production $A_k \rightarrow a$ induces a transition from q_k to q_f with label 'a'.

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

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



$S \to aS/a$



$S \to aS/\ bA/b$

$A \to aA/bS/a$

Pumping lemma:

(1)

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

(2)

8 chose 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 $xy^iz \notin L$ this 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) $w = a^{n^2}$

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

let $w = xyz$

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

(3) $w = xyz$

$xy^i z = xy^2 z$

$|xy^2z| = |xyz| + |y|$

$= n^2 + |y|$

$> n^2$ —— (1)

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

$|y| \leq n$

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

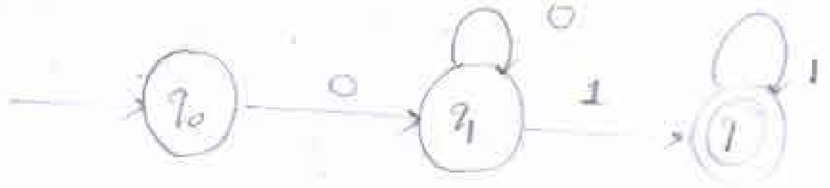
$< n^2 + n + n + 1$

$< (n+1)^2$ —— (2)

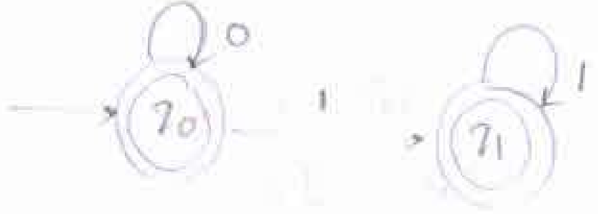
from (1) & (2),

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

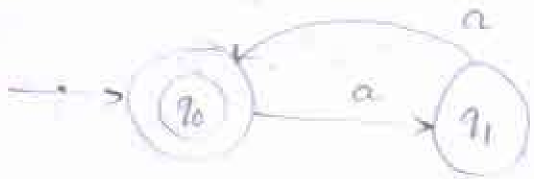
$0^m 1^n$ where $m,n \geq 1$



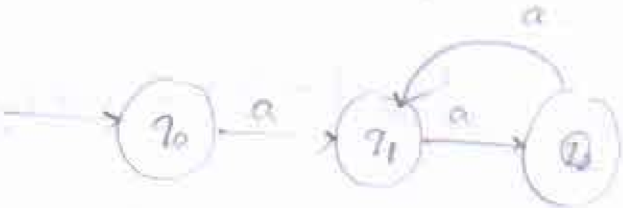
$0^m 1^n$ where $m,n \geq 0$

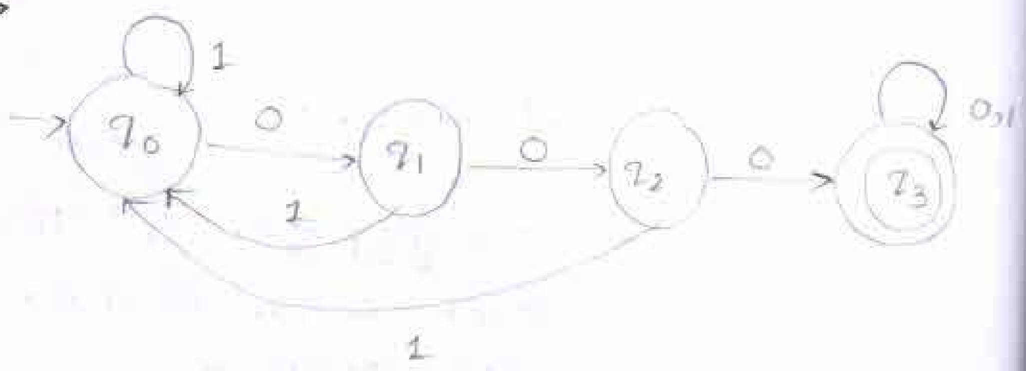


a^{2n}, $n \geq 0$



a^{2n}, $n \geq 1$





Convert into regular grammar.

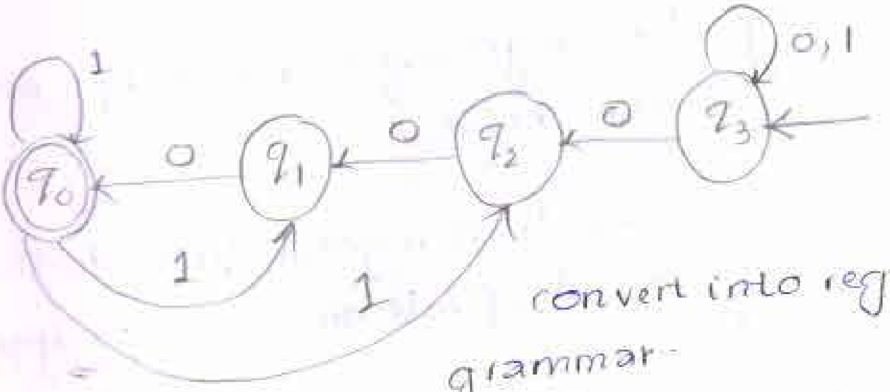
$q_0 \longrightarrow 0q_1 / 1q_0$

$q_1 \longrightarrow 0q_2 / 1q_0$

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

$q_3 \longrightarrow 0q_3 / 1q_3 / 0 / 1$

Right linear grammar



convert into regular grammar

$q_0 \longrightarrow 1q_0 / 1q_1 / 1q_2 / 1$

$q_1 \longrightarrow 0q_0 / 0$

$q_2 \longrightarrow 0q_1$

$q_3 \longrightarrow 0q_2 / 0q_2 / 1q_3$

left linear grammar

$\downarrow$

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

$q_1 \rightarrow q_0 0 / 0$

$q_2 \rightarrow q_1 0$

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

Find the regular expression for the following

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

$\downarrow$

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

$a^m b^{2n} c^{3p}$ $m, n, p, \geq 1$

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

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

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

verify 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) $w = a^{pm}$

let, $m > n$

$|w| > n$

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

(3) $w = xy^iz$

let $i = m+1$

$|xy^iz| = |xyz| + |y^{i-1}|$

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

$|xy^iz| = m + m|y|$

$= m(1+|y|)$ not prime

def:

If q_i, q_j are said to be k-equivalance $\forall k \geq 0$ then q_i, q_j are said to be equivalance.

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

3. FORMAL LANGUAGES:

Chomsky → classification of languages

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

left context right context

Grammar is basically 4-tuple

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

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

Σ – it is finite set collection of terminals

here,

$V_N \cap \Sigma = \phi$

S – starting symbol where $S \in V_N$.

S ⟶ <Noun><verb>

S ⟶ <Noun><verb><adverb>

Noun → Priya, Raj

verb → run, ate

adverb → quickly, slowly.

P – collection of productions in the form of $\alpha \to \beta$ where $\alpha, \beta \in V_N \cup \Sigma$

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

$S \to 0S1 / \Lambda$

$0S1$

$00S11$

$0^2 S 1^2$

$00^2 S 11^2$

$0^3 S 1^3$

replacing s by Λ we get, $0^3 1^3$.

eg,

→ In a production of the form $\phi A \psi \to \phi \alpha \psi$ 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

→ $abAbcd \longrightarrow ab\,AB\,bcd$

$ab \to$ left context

$bcd \to$ right context

here $\alpha = AB$

→ $AC\Lambda \to A\Lambda\Lambda$

left context is A

right context is Λ

and $\alpha = \Lambda$

→ $C \to \Lambda$ (here 'c' is erased)

leftcontext & 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 $\phi A \psi \to \phi \alpha \psi$ is called a type 1 production if $\alpha \neq \Lambda$ (i.e, $A \neq \Lambda$)

In type-1 production erasing of A is not permitted.

eg,

(1) $a\underline{A}bcD \to a\underline{bcD}bcD$.

type 1 grammar

(2) $\underline{A}B \longrightarrow \underline{A}bBc$

left left

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

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

The production $S \to \Lambda$ 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 type1 or context sensitive language.

In a context sensitive grammar G we allow $S \to \Lambda$ apart from $S \to \Lambda$ all the other productions donot decrease the length of the working string

TYPE - 2

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

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

eg,

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

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 $A \to a$ (or) $A \to aB$ where $A, B \in V_N$ and $a \in \Sigma$ 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 $S \to \Lambda$ 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 number which can be applied to following grammars

(a) $S \to Aa \to$ type 2

$A \to c \mid Ba$

c → type-3, Ba → type-2

$B \to abc \to$ type 2

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

ASB → type 2, d → type 3

$A \to aA$ → type 3

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

aS → type 3, ab → type 2

REGULAR EXPRESSION & REGULAR GRAMMAR:

Any terminal or element of Σ is regular expression

for eg, a in Σ, Φ is also regular expression

$\{a\}$ ↑ empty set

Λ is regular expression

$\{\Lambda\}$

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 $\{a, b\}$

CONCANTENATION:

Concantenation 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

$\Lambda, a, aa, aaa \cdots\cdots$ 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 precedance next highest is for concantenation and last is for union

IDENTIFIER NOTATION:

notation: $l(l/d/s)^*$

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 $\{\Lambda, a, aa, aaa \ldots\}$

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

$\{\Lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots\ldots\ldots\}$

Regular set	Regular expression
$\{101\}$	101.
$\{\Lambda, a\}$	$\Lambda + a$

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

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

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

$1(11)^*$ or $(11)^*1$

Strings of 0's and 1's with atleast '2' connective zeros

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

All strings of 0's and 1's begining with 1 or 0 and not having '2' 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

$(I)_1$ $\phi + R = R$

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

I_3 $\Lambda R = R\Lambda = R$

I_4 $\Lambda^* = \Lambda$ & $\phi^* = \Lambda$

I_5 $R + R = R$

I_6 $R^*R^* = R^*$

I_7 $RR^* = R^*R$

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

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

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

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

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

$\rightarrow$ '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:

Case(i) $R = Q + RP$

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

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

$R = QP^*$ (from I_9)

Case (ii):

$R = Q + RP$

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

$= Q + QP + RP^2$

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

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

⋮

$= Q + QP + QP^2 + \cdots + QP^i + QP^{i+1}$

$= Q(\Lambda + P + P^2 + \cdots P^i) + RP^{i+1}$

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

$= Q(P^*)$

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

Proove that the regular expression $R = \Lambda + 1^*(011)^*(1^*(011)^*)^*$

$= (1+011)^*$

$L.H.S = \Lambda + 1^*(011)^*(1^*(011)^*)^*$

let $R = 1^*(011)^*$

$L.H.S = \Lambda + R(R)^*$

$= R^*$ by I_9

$L.H.S = (1^*(011)^*)^*$

$= (1+011)^*$ by I_{11}

$= R.H.S$

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

$r(st) = (rs)t$

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

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

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

→ $\phi^* = \Lambda$

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 to 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:

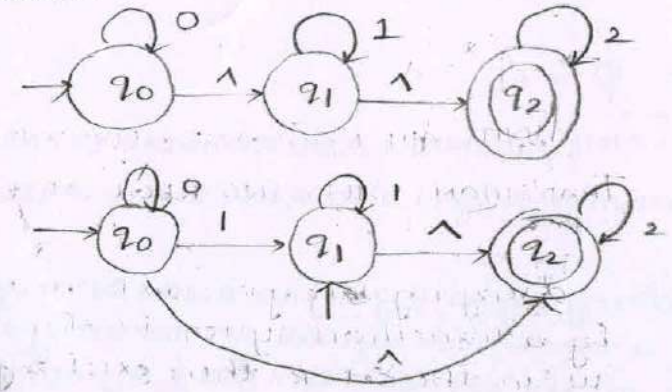
If v_1 is the initial state, make v_2 also as initial state

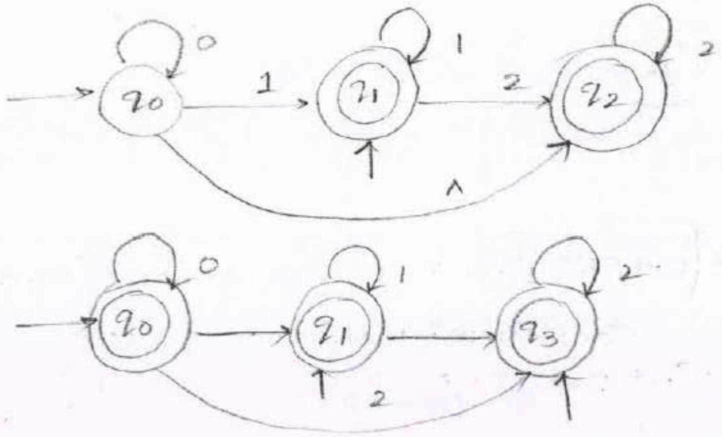
Step 4:

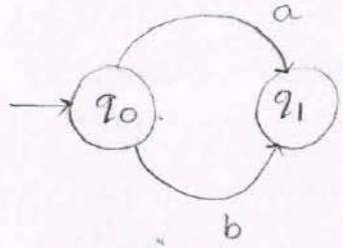
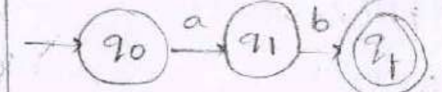
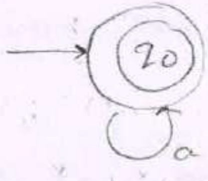
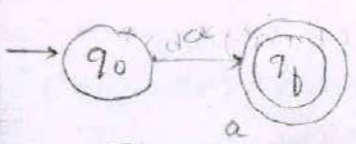
If v_2 is a final state make v_1 as the final state

eg,

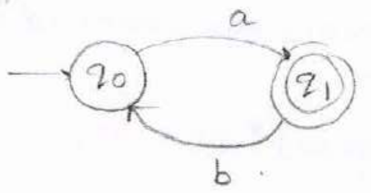
Consider a finite automata will null moves & obtain a equivalent automata without null moves





Finite automata	Regular set	Regular expression
	{a, b}	a+b
	{ab}	ab
	{Λ, a, aa, aaa ....}	a^*
	{a, aa, aaa ...}	a^+
	{Λ, aa, aaa ....}	$(aa)^*$

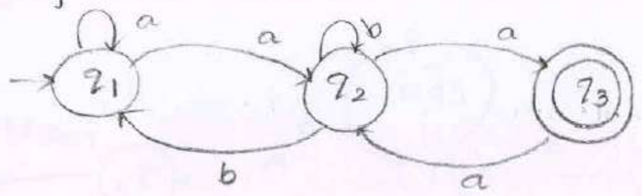
7



$\{a, aba, ababa \ldots\}$ $a(ba)^*$

ALGEBRIC METHOD USING ARDEN'S THEOREM

Consider the trancition system and find out its equivalent regular expression using arden's theorem



$$q_1 = q_1a + q_2b + \Lambda$$

$$q_2 = q_1a + q_2b + q_3a$$

$$q_3 = q_2a$$

$$q_2 = q_1a + q_2b + q_2aa$$

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

$$R = Q + RP$$

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

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

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

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

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

$$q_3 = q_2a$$

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

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

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 $V_1, V_2, \ldots V_n$

(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 $\alpha_{ij} = \phi$ consequently, we get a following set of equations in V_1 to V_n

$$V_1 = V_1\alpha_{11} + V_2\alpha_{21} + \ldots + V_n\alpha_{n1} + \Lambda$$

$$V_2 = V_1\alpha_{12} + V_2\alpha_{22} + \ldots + V_n\alpha_{n2}$$

$$\vdots$$

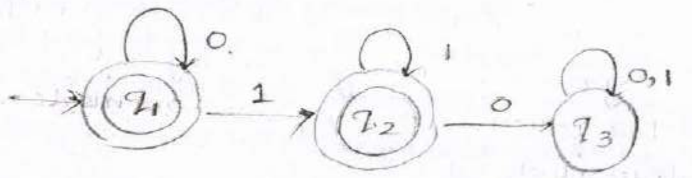
$$V_n = V_1\alpha_{1n} + V_2\alpha_{2n} + \ldots + V_n\alpha_{nn}$$

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

$R = Q r^*$, $R = Q P^*$

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



$q_1 = q_1 0 + \Lambda$
$R = RP$
$q_2 = q_2 1 + q_1 1$ $\quad R = Q + RP$
$= Q(P)^*$
$q_3 = q_2 0 + q_3 0 + q_3 1$
$= q_2 0 + q_3 (0+1)$

$q_1 = \Lambda + q_1 0$
$q_1 = \Lambda 0^*$
$q_1 = 0^*$

$q_2 = q_2 1 + \underline{q_1 1}$
$R = RP + Q \quad = Q(P)^*$
$q_2 = q_1 1 1^*$ by ardens theorem.

$q_2 = 0^* 1 1^*$

$q_2 = 0^* 1^*$

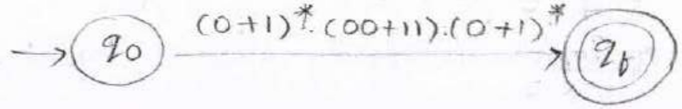
$q_1 + q_2 = 0^* + 0^* 1^* $

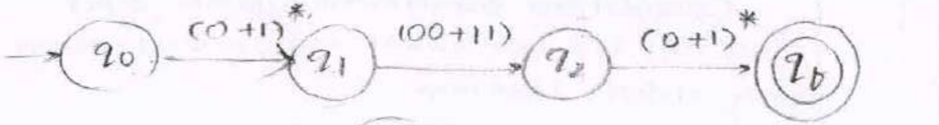
$= 0^* (\Lambda + 1^*)$

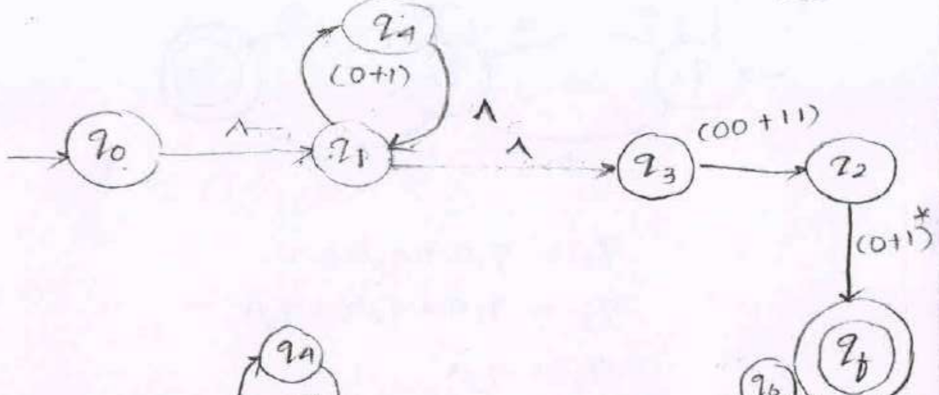
$= 0^* 1^*$

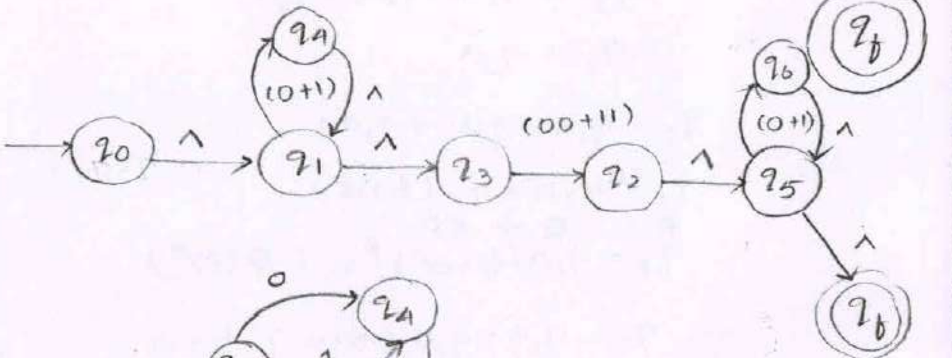
Constrnct the finite automata equivalent to the regular expression

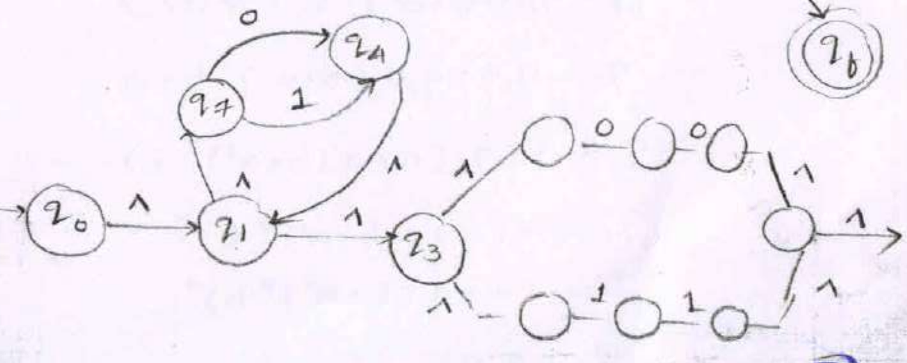
$\underline{(0+1)^* (00+11)(0+1)^*}$



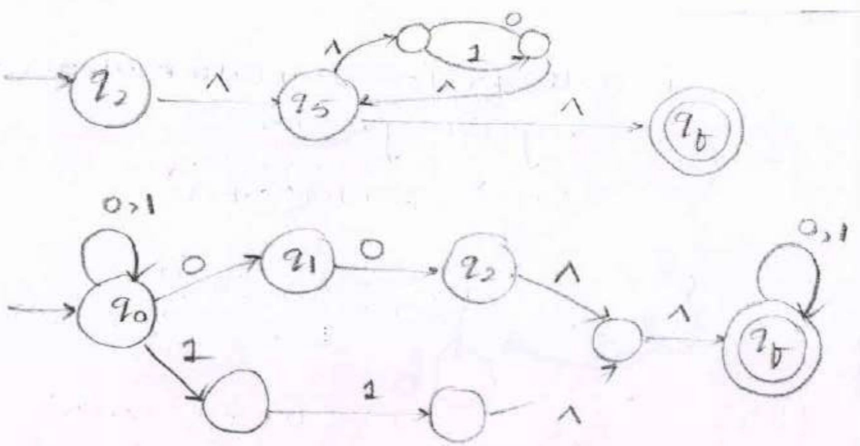


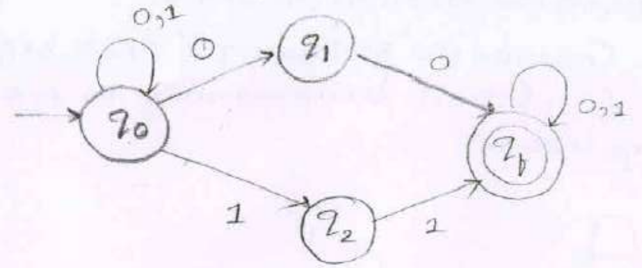






9





NFA

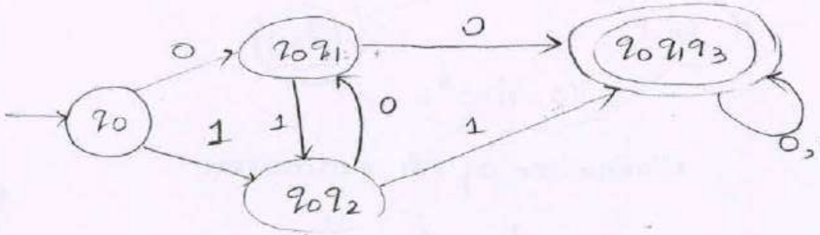
NFA table:

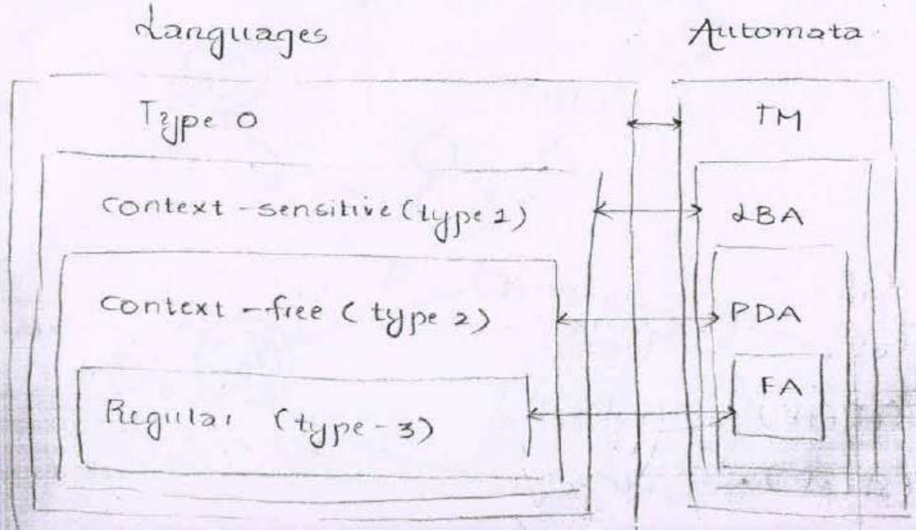
	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0 q_2\}$
q_1	q_3	
q_2		q_3
q_3	q_3	q_3

NFA-to DFA table

	0	1
$\rightarrow q_0$	$[q_0 q_1]$	$[q_0 q_2]$
$[q_0 q_1]$	$[q_0 q_1 q_3]$	$[q_0 q_2]$
$[q_0 q_2]$	$[q_0 q_1]$	$[q_0 q_2 q_3]$ (∴ $[q_0 q_1 q_3]$)
$[q_0 q_1 q_3]$	$[q_0 q_1 q_3]$	$[q_0 q_2 q_3]$ ($[q_0 q_1 q_3]$)
$[q_0 q_2 q_3]$	$[q_0 q_1 q_3]$	$[q_0 q_2 q_3]$ ×

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





where

TM - Turing machine

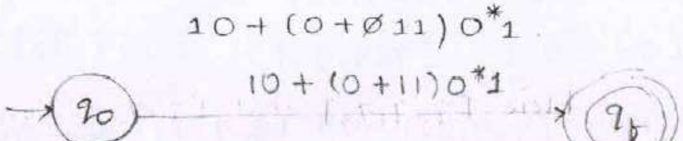
LBA - linear bounded automaton

PDA - push down automaton

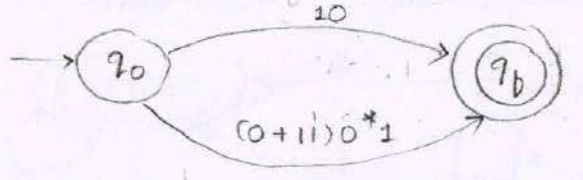
FA - Finite automata

Construct DFA with reduced states equivalent of to the regular expression

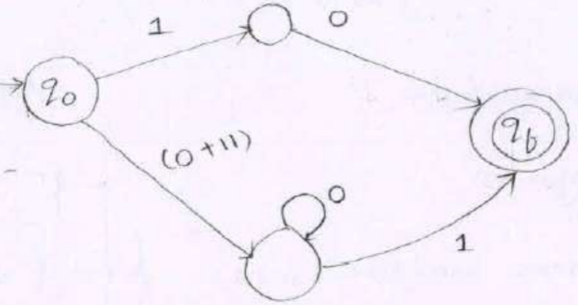
$$10 + (0 + \emptyset 11) 0^* 1$$

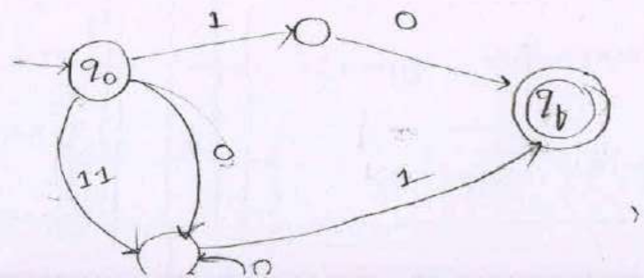


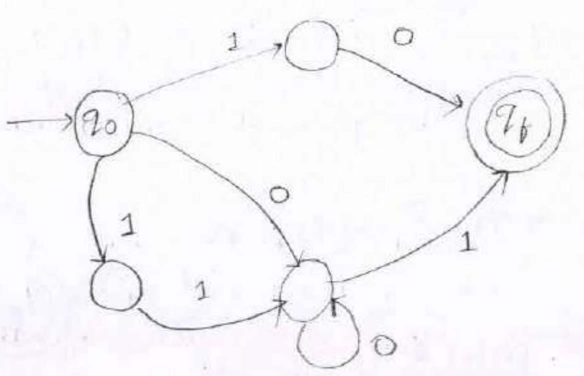
elimination of union



elimination of concantenation

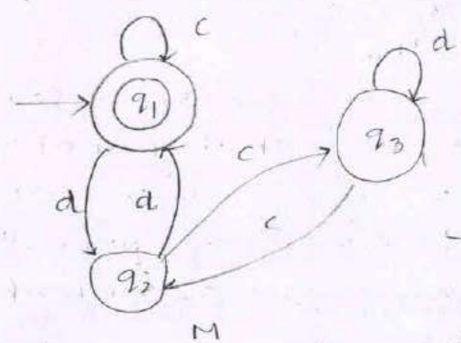




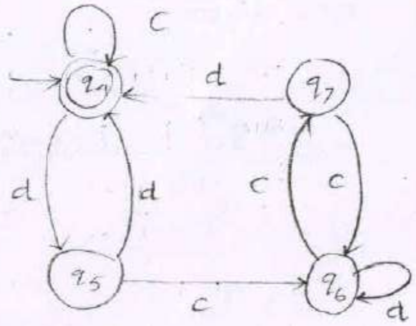


COMPARISION METHOD OF DFA'S:

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



M



M'

	(q_c, q_c')	(q_d, q_d')
(q_1, q_4)	(q_1, q_4)	(q_2, q_5)
(q_2, q_5)	(q_3, q_6)	(q_1, q_4)
(q_3, q_6)	(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 '2' 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 the 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 rowwise construction is repeated. There are '2' cases

Case 1:

If reach a pair (q, q') such that q is the final state of M & q' is the non-final state of M' or viceversa 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.

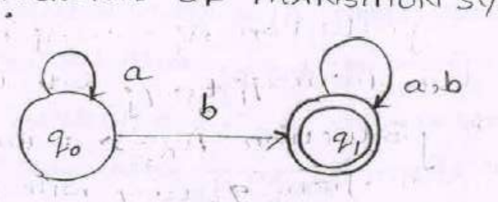
CLOSURE PROPERTIES OF REGULAR SETS:

- 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 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:



$q_i \to a q_j$ iff $\delta(q_i, a) = q_j$ with $q_j \notin F$

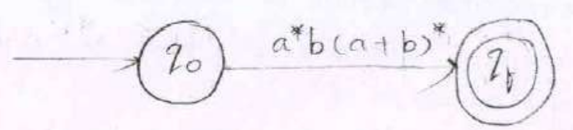
$q_i \to a q_j$, $q_i \to a$ iff $\delta(q_i, a) \in F$

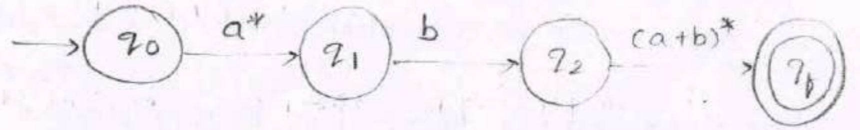
$q_0 \rightarrow aq_0$
$q_0 \rightarrow bq_1$ $\quad$ $q_0 \rightarrow b$
$q_1 \rightarrow aq_1$ $\quad$ $q_1 \rightarrow a$
$q_1 \rightarrow bq_1$ $\quad$ $q_1 - b$

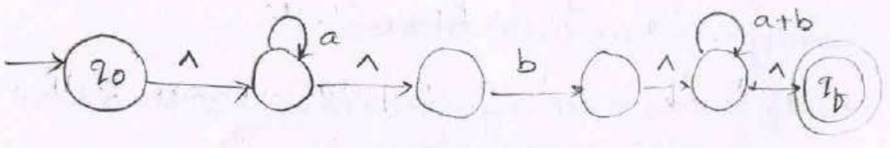
construct a regular grammar generalset represented by $a^*b(a+b)^*$

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

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







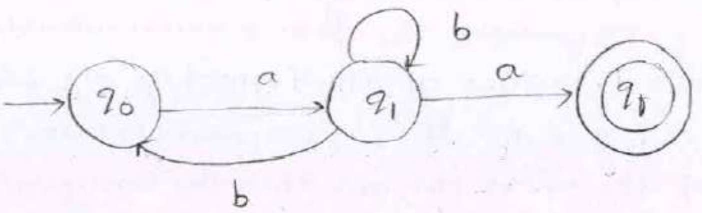
CONSTRUCTION OF TRANSITION SYSTEM FOR A GIVEN REGULAR GRAMMAR:

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

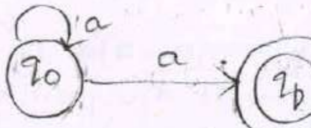
each production $A_k \rightarrow a$ induces a transition from q_k to q_f with label 'a'.

let $G = (\{A_0, A_1\}, \{a, b\}, P, A_0)$ 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

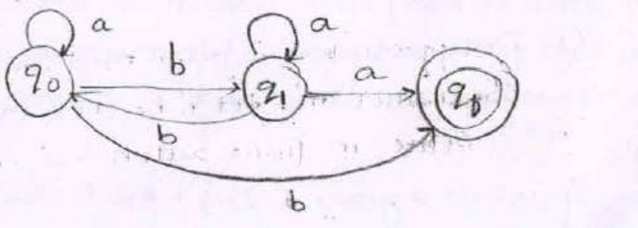
$M = \{ \{q_0, q_1, q_f\}, \{a, b\}, \delta, q_0, \{q_f\} \}$



$S \rightarrow aS/a$



$S \rightarrow aS/bA/b$
$A \rightarrow aA/bS/a$



17

PUMPING LEMMA:

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

(2) I chose 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 $xy^iz \notin L$ this 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) $w = a^{n^2}$

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

let $w = xyz$

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

(3) $w = xyz$

$xy^iz = xy^2z$

$|xy^2z| = |xyz| + |y|$

$= n^2 + |y|$

$> n^2$ —(1)

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

$|y| \leq n$

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

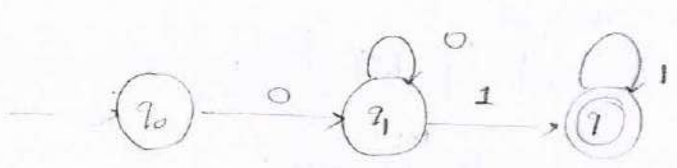
$< n^2 + n + n + 1$

$< (n+1)^2$ —(2)

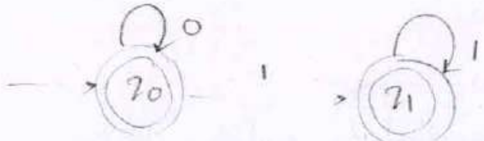
from (1) & (2),

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

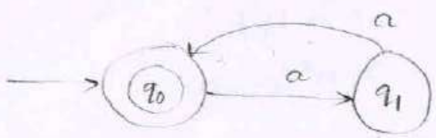
$0^m 1^n$ where $m, n \geq 1$



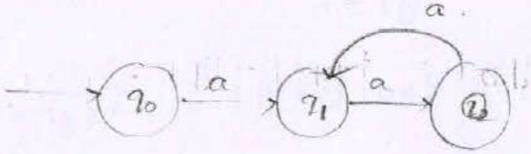
$0^m 1^n$ where $m, n \geq 0$



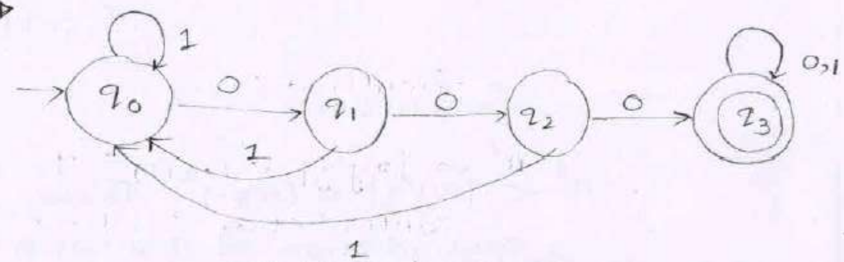
a^{2n}, $n \geq 0$



a^{2n}, $n \geq 1$



→



Convert into regular grammar.

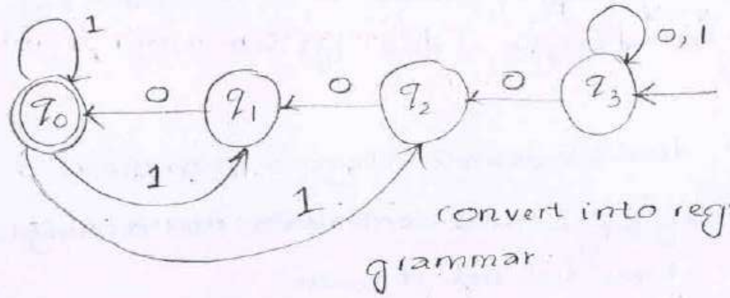
$q_0 \longrightarrow 0q_1 / 1q_0$

$q_1 \longrightarrow 0q_2 / 1q_0$

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

$q_3 \longrightarrow 0q_3 / 1q_3 / 0 / 1$

Right linear grammar.



convert into regular grammar

$q_0 \longrightarrow 1q_0 / 1q_1 / 1q_2 / 1$

$q_1 \longrightarrow 0q_0 / 0$

$q_2 \longrightarrow 0q_1$

$q_3 \longrightarrow 0q_2 / 0q_2 / 1q_3$

left linear grammar

↓

$q_0 \to q_0 1 / q_1 1 / q_2 1 / 1$

$q_1 \to q_0 0 / 0$

$q_2 \to q_1 0$

$q_3 \to q_2 0 / q_3 0 / q_3 1$

Find the regular expression for the following

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

↓

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

$a^m b^{2n} c^{3p}$ $m, n, p \geq 1$

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

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

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

verify 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) $w = a^m$

let, $m > n$

$|w| > n$

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

(3) $w = xy^i z$

let $i = m+1$

$|xy^i z| = |xyz| + |y^{i-1}|$

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

$|xy^i z| = m + m|y|$

$= m(1+|y|)$ not prime

CONTEXT FREE GRAMMAR:

obtain the context free grammar for signed integers.

S → <sign><Integer>

sign → +/-

Integer → <digit><Integer>/<digit>

digit → 0/1/ .... /9

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

$V_N = \{S, sign, Integer, digit\}$.

eg, +1250

S → <sign><Integer>

sign → +

Integer → <Digit><Integer>

→ 1 <Integer>

→ 1 <digit><Integer>

→ 12 <Integer>

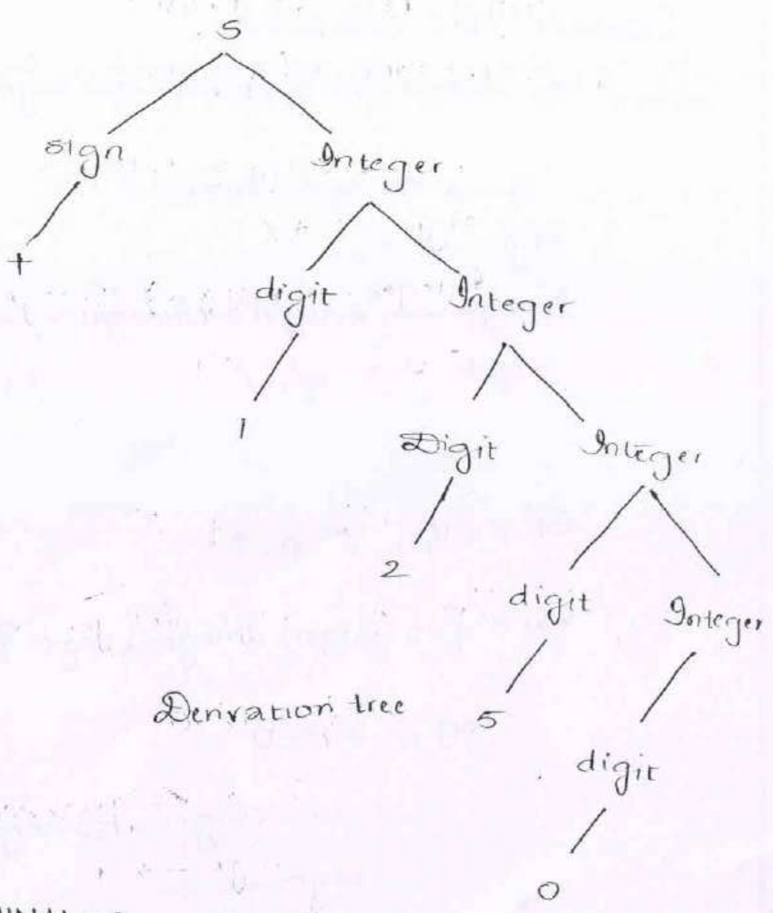
→ 12 <Digit><Integer>

→ 125 <Integer>

→ 125 <digit>

→ 1250

S → <Sign><Integer>



Derivation tree

TERMINAL : a
nonterminal : S
productions : $S \to aS$
$S \to \Lambda$

$S \to aS/\Lambda$

Λ, as, $\underline{aaS}$, $\underline{aaaS}$
Λ, aΛ, $\overline{aa\Lambda}$, $\overline{aaa\Lambda}$
Λ, a, aa, aaa
Here,
$L(G) = a^*$

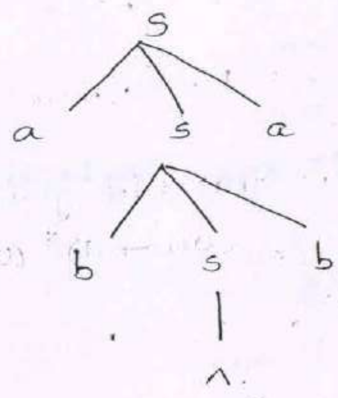
Define context-free grammar for polindrome over a,b.

Λ, a, b, aa, bb, aaa ....

polindrome = Initial and final must be same in the middle it should contain polindrome.

productions:

$S \to aSa$
$S \to bSb$
$S \to \Lambda/a/b$.



~~deter~~
derivation tree

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

productions:

$S \to SaSbS / SbSaS / S / \Lambda$

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

abb, baa, bbba, abbab, ababa, aabbaaa, bbaab, aba, ba~~b~~, a, b.

let

$X \rightarrow SaSbS / SbSaS / \Lambda$

$S \rightarrow U / V \rightarrow |a| < |b|$

$\downarrow$

$|a| > |b|$

$U \rightarrow XaU / XaX$

$V \rightarrow XbV / XbX$

$X \rightarrow XaXbX / XbXaX / \Lambda$

Give the context free grammar for

$(011 + 11)^* (01)^*$

$(011 + 11)^* (01)^*$

$S \rightarrow AB$

$A \rightarrow CA / \Lambda$

$C \rightarrow 011 / 11$

$B \rightarrow DB / \Lambda$

$D \rightarrow 01$

Derive the context free grammar for $a^n b^{2n}, n \geq 1$

$S \rightarrow aSbb / abb.$

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

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

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

$A \rightarrow 0A / 1A / \cancel{0} / \cancel{1} / \Lambda$

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

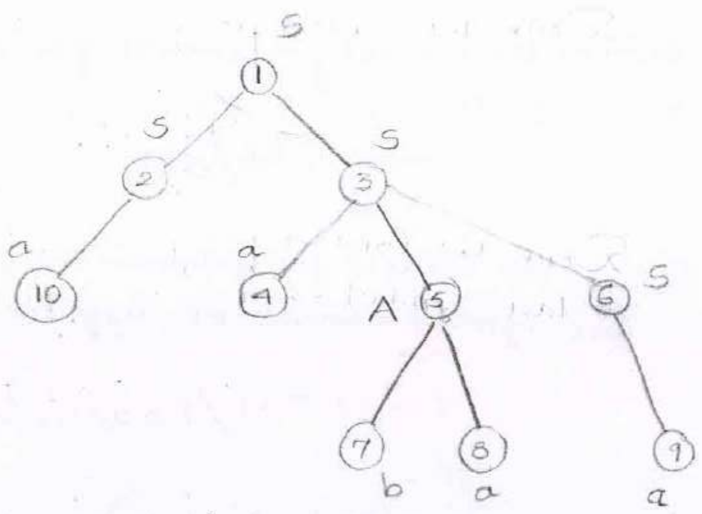
In a tree, every vertex has a label which is a variable or terminal or null

The root has the label 'S'.

Internal ~~terminal~~ vertex is the non-terminal or variable

~~The~~ If the vertices $n_1, n_2, \ldots n_k$ are written with the labels $X_1, X_2 \ldots X_k$ are the sons of the vertex n with label 'A' then $A \rightarrow X_1, X_2 \ldots X_k$ is a production in 'P'

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



Productions: $S \rightarrow SS$
$S \rightarrow a$
$S \rightarrow aAS$
$A \rightarrow ba$

ODERING 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.

$S \rightarrow SS \rightarrow aS \rightarrow aaAS \rightarrow aabaS \rightarrow aabaa$

A subtree of a derivation tree 'T' is a tree

(1) whose root is vertex 'v' of 'T' i.e, v(T)
(2) whose vertices are the descendents of v together with their labels
(3) whose edges are those connecting the descendence of v

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

eg,

$S \rightarrow aab$ ×

cannot be formed using the before 4 productions

SENTENTIAL FORM:

Yield is also known as sentential form

Consider G whose productions are

$S \to aAS/a$
$A \to SbA/SS/ba$

Show that $S \to aabbaa$ and Consider a derivation tree whose yield is aabbaa.

$S \to aAS/a$
$A \to SbA/SS/ba$

(1) $S \Rightarrow aabbaa$

LEFT MOST DERIVATION:

$S \to aAS$ — A is replaced by SbA
$S \to a\underline{S}bAS$ — S replaced by a
$S \to aab\underline{A}S$
$S \to aabba\underline{S}$ — A replaced by ba, S by a

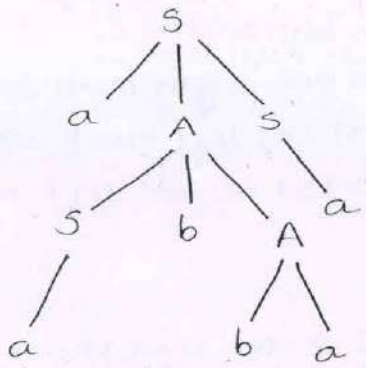
(2) $S \to aabaa$

RIGHTMOST DERIVATION:

$S \to aAS$
$\downarrow (S \to a)$
$a\underline{A}S$
$\downarrow$
$aSbAa \ (A \to sba)$
$\downarrow$
$a\underline{S}bbaa \ (S \to a)$
$\downarrow$
$aabbaa$

(3) $S \to a\underline{A}S \to aSbA\underline{S} \to a\underline{S}bAa \to aab\underline{A}a$
$A \to SbA \quad S \to a \quad S \to a \quad \downarrow A \to ba$
aabbaa

DERIVATION TREE



String $\to$ aabbaa

In derivation (1) whenever we replace a variable x using a production, there is no variable to the left x.
In derivation (2) there are no variables to the right of x.
But in derivation (3) no such conditions are satisfied

LEFTMOST DERIVATIONS:

The derivation $A \Rightarrow w$ is left most 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 G then there is a left most derivation of w.

Every 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 every level consider the productions 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 be a grammar to a $S \to 0B/1A$, $A \to 0B/0S/1AA$, $B \to 0/1S/0BB$ for the string 00110101

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

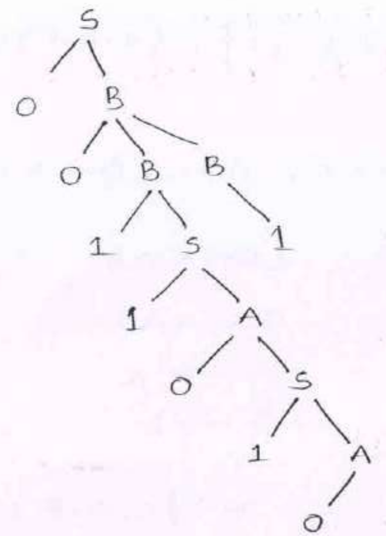
LEFT MOST DERIVATION:

$S \to 0B/1A,\ A \to 0/0S/1AA,\ B \to 1/1S/0BB$

$S \to 0\underline{B}$	$S \to 0B$
$\downarrow$	$S \to 00B\underline{B}$
$00\underline{B}B$	$S \to 00B0BB \leftarrow$
$\downarrow$	$S \to 00B0B1 \leftarrow$
$001\underline{S}B$	$S \to 00B0 \leftarrow$
$\downarrow$	$S \to 00B1S \leftarrow$
$0011\underline{A}B$	$S \to 00B10B \leftarrow$
$\downarrow$	$S \to 00B101 \leftarrow$
$00110\underline{S}B$	$S \to 001S101$
$\downarrow$	$S \to 0011B101 \leftarrow$
$001101\underline{A}B$	$S \to 00110101 \leftarrow$
$\downarrow$	
$0011010\underline{B}$	
$\downarrow$	
00110101	

RIGHT MOST DERIVATION:

DERIVATION TREE:



AMBIGUITY CONTEXT FRAGMENTATION:

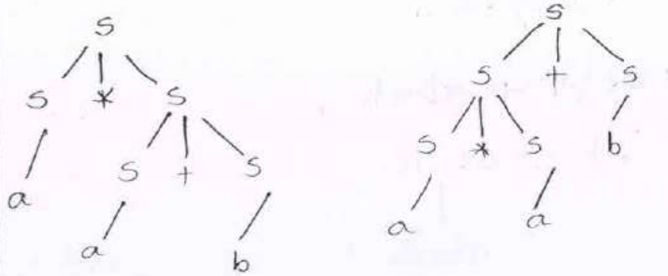
In books selected information is given the word "selected" may refer to books or information so the sentence may be parsed into 2 different way The same situation may arise in Context free languages The same terminal string may be the yield of 2 derivation trees. So there may be 2 different left most derivation of w by the theorems This leads to the definition of ambiguous sentences in grammar

The terminal strings are here $w \in L(G)$ is ambiguous if there exists 2 or more derivation tree for w

Eg, $G = (\{S\}, \{a, b, +, *\}, P, S)$

where P consists of

$S \to S+S / S*S / a / b / a*a+b$



LEFT MOST DERIVATION:

$S \to S+S \to S*S+S \to a*S+S \to a*a+S \to a*a+b$

$S \to S*S \to a*S \to a*S+S \to a*a+S \to a*a+b$

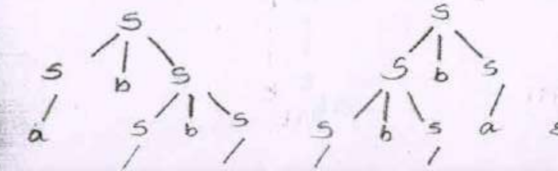
Here we are getting 2 leftmost derivations or 2 different derivation trees so, $a*a+b$ is ambigous

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

S.T that $S \to SbS / a$ is ambigous

$S \to \underline{S}bS \to ab\underline{S} \to aba$

$S \to \underline{S}bS \to \underline{S}bSbS \to ab\underline{S}bS \to abab\underline{S} \to ababa$



'2' derivation tree so, L(G) is ambigous.

Show that the grammar $S \to a/abSb/aAb$, $A \to bS/aAAb$ is ambiguous

$S \to a/abSb/aAb$

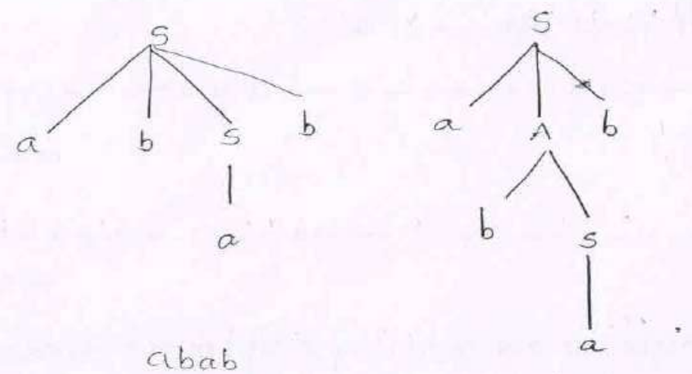
$A \to bS/aAAb$

$S \to abSb \to abab$.

$S \to aAb \to absb$

$\downarrow$

$abab$

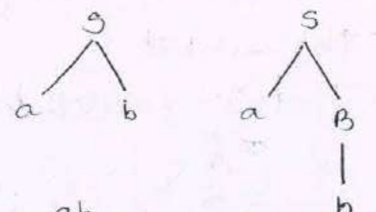


abab   abab.

So, the grammar is ambigous as the string abab is the string having 2 derivation trees

Show that the grammar $S \to aB/ab$ $A \to aAB/a$

$B \to ABb/b$



ab   ab.

The string ab has the 2-derivation trees so, the grammar has 2-derivating trees so, the grammar is ambigous

SIMPLIFICATION OF CONTEXT FREE GRAMMAR:

Consider the example.

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

where

$P = \{S \to AB, A \to a, B \to b, B \to C, E \to c/\Lambda\}$

$L(G_1) = ab$.

$S \to AB$.

$A \to a$

$B \to b$

$V_N' = \{S, A, B\}, \Sigma' = \{a, b\}$

It is easy to see that $L(G_1) = ab$ so, $G_1' = (\{S, A, B\}, \{a, b\}, P', S)$

where P' consists of,

$S \to AB$

$A \to a$

$B \to b$

$L(G_1) = L(G_1')$

we have eleminated the symbols C, E and c and the productions $B \to C$, $E \to c/\Lambda$

we note the following points regarding the symbols and productions which are elemination

(1) 'C' does not derive any terminal string

(2) 'E' and 'C' donot appear in any sentencial form

(3) $E \to \Lambda$ is a null production

(4) $B \to C$ 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 $A \to B$

CONSTRUCTION OF REDUCED GRAMMAR

THEOREM 1:

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

Step 1. Construction of V_N'

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

$$W_1 = \{ A \in V_N \mid \text{there exist a production } A \to w \text{ where } w \in \Sigma^* \}$$

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

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

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

$\therefore W_k = W_{k+j}$ for $j \geq 1$. so, here we define $V_N' = W_k$

step 2:

Construction of P'

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

& $G' = (V_N', \Sigma, P', S)$

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

Construction of V_N'

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

Since $A \to a$, $B \to b$, $E \to c$ are productions with a terminal string on R·H·S

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

$$W_2 = W_1 \cup S$$

$W_2 = \{A, B, E, S\}$ from the production $S \to AB$

$W_3 = W_2 \cup \{ A_1 / A_1 \in V_N, A_1 \to \alpha$
$\alpha \in (\Sigma, A, B, E, S)^* \}$

$W_3 = W_2 \cup \phi$

$W_3 = W_2$.

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

Construction of P'.

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

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

here,

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

THEOREM 2:

for every CFG $G = (V_N, \Sigma, P, S)$ we can construct an equivalent grammar $G' = (V_N', \Sigma', P', S)$ 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 $S \overset{*}{\underset{G'}{\Rightarrow}} \alpha$ and x is a symbol in the string α)

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

(a) Construction of W_i for $i \geq 1$.

(i) initially $W_1 = \{S\}$

(ii) $W_{i+1} = W_i \cup \{ x \in V_N \cup \Sigma \mid \exists$ a production $A \to \alpha$ with $A \in W_i$ and α containing the symbol $x \}$.

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

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

(b) Construction of V_N', Σ' and P'

we define $V_N' = V_N \cap W_k$

$\Sigma' = \Sigma \cap W_k$

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

Eg,

Consider $G = (\{S, A, B, E\}, \{a, b, c\}, P, S)$ where P Consists of $S \to AB$, $A \to a$, $B \to b$,
$E \to c$

$W_1 = \{S\}$

$W_2 = W_1 \cup \{ x \mid x \in V_N \cup \Sigma \ \exists$ a production $A_1 \to \alpha$ with $A \in W_1$ and α containing $x \}$
(5)

So, here $S \in W_1$ and $S \to AB$.

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

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

$W_3 = W_2 \cup \{ x \in V_N \cup \Sigma \mid \exists$ a production $A_1 \rightarrow \alpha$ with $A_1 \in W_2$ and $\{S, A, B\}$ α containing $x \}$

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

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

$W_4 = W_3 \cup \{ x \in V_N \cup \Sigma \mid \exists$ a production $A_1 \rightarrow \alpha$ with $A_1 \in W_3$ and α 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', P' and Σ'

$V_N' = V_N \cap W_K$

$= V_N \cap W_3$

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

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

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

$\Sigma' = \Sigma \cap W_K$

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

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

$= \{a, b\}$

$P' = \{ A_1 \rightarrow \alpha / A \in W_K \}$

$= \{ A_1 \rightarrow \alpha / A \in W_3 \}$

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

the required grammar $G' = (V_N', \Sigma', P', S)$

THEOREM 3:

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

Proof:

we construct the reduced grammar in 2 steps

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

Step 2: we construct a grammar $G_1' = (V_N', \Sigma', P')$ 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. G_1' is the required reduced grammar

By step2 every symbol x in G_1' appears in some sentential form, say $\alpha x \beta$

By step1 every symbol in $\alpha x \beta$ derives some terminal string

$\therefore S \rightarrow \alpha x \beta \Rightarrow w$ in Σ^* i.e, G_1' is reduced

NOTE:

To ~~sole~~ get a reduced grammar we must first apply theorem1 and then theorem2

If you apply theorem2 first and then theorem1 we may not a 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:

$W_1 = A / A \rightarrow w,\ w \in \Sigma^*$

$W_1 = \{A, C\}$ from the productions $A \rightarrow a,\ C \rightarrow b$.

$W_2 = W_1 \cup \{A \rightarrow \alpha / \alpha \in (\Sigma, A, C)^*\}$

$W_2 = \{A, C\} \cup \{S\} = \{S, A, C\}$ from the production $S \rightarrow CA$

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

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

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

$= W_2 = W_k = W_{k+1}$

$V_N' = W_k = W_2 = \{S, A, C\}$.

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

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

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

step 2:

$W_1 = \{S\}$

$W_2 = W_1 \cup \{x \mid x \in (V_N' \cup \Sigma), A_1 \rightarrow \alpha.\ A_1 \in W_1$ and α containing symbol $x\}$

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

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

$W_3 = W_2 \cup \{x / x \in (V_N' \cup \Sigma), A_1 \rightarrow \alpha\ \ A_1 \in W_2$ and α containing symbol $x\}$

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

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

$W_K = W_3 = \{S, C, A, a, b\}$.

$V_N' = V_N \cap W_K$

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

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

$\Sigma' = \Sigma \cap W_K$

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

$= \{a, b\}$

$P' = \{A_1 \to \alpha \,/\, A \in W_K\}$

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

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

Construct a reduced grammar equivalent to the grammars,

$S \to aAa,$
$A \to Sb / bCC / DaA$
$C \to abb / DD$
$E \to aC$
$D \to aDA$

Step 1:

$W_1 = \{C\}$

$W_2 = W_1 \cup \{A_1 / A_1 \in V_N,\ A_1 \to \alpha,\ \alpha \in (C, a, b)^*$

from the productions

$A \to bCC,\ E \to aC$

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

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

$W_3 = W_2 \cup \{A_1 / A_1 \in V_N,\ A_1 \to \alpha,\ \alpha \in (A, C, E, a, b)^*\}$

from the productions $S \to aAa$

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

$W_4 = W_3 \cup \{A_1 / A_1 \in V_N,\ A_1 \to \alpha,\ \alpha \in (S, A, C, E, a, b)^*\}$

from the production

$A \to Sb$

$W_4 = \{S, A, C, E\} \cup \{A\}$

$W_4 = \{S, A, C, E\} = W_3 = W_K$

$V_N' = W_3 = W_K = \{S, A, C, E\}$

$P' = \{S \to aAa,\ A \to bCC,$
$A \to Sb,\ C \to abb,$
$E \to aC\}$

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

STEP 2:

$W_1 = \{S\}$

$W_2 = W_1 \cup \{ x \mid x \in V_N \cup \Sigma,\ A_1 \to \alpha$
$A_1 \in W_1$ & α containing symbol $x \}$

$W_2 = \{S\} \cup \{a, A\}$

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

$W_3 = W_2 \cup \{ X \mid x \in V_N \cup \Sigma,\ A_1 \to \alpha,$
$A_1 \in W_2$ & α containing symbol $x \}$.

$W_3 = \{S, A, a\} \cup \{b, C\}$

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

$W_4 = W_3 \cup \{ X \mid x \in V_N \cup \Sigma,\ A_1 \to \alpha,$
$A_1 \in W_3$ & α containing symbol $x \}$

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

$= \{S, A, C, a, b\} = W_3 = W_K$.

$V_N' = V_N \cap W_3$

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

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

$\Sigma' = \Sigma \cap W_2$

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

$= \{a, b\}$

$P' = \{ S \to aAa,\ A \to Sb / bCC,\ C \to abb \}$

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

ELIMINATION OF NULL PRODUCTIONS:

A variable 'A' in a context free grammar is nullable if $A \to \Lambda$

THEOREM:

If $G = (V_N, \Sigma, P, S)$ is a CFG then we can find CFG G_1 having no null productions such that $L(G_1) = L(G) - \{\Lambda\}$

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

Step 1: Construction of the set of nullable variables

we find the nullable variables recursively

(i) $W_1 = \{ A \in V_N \mid A \to \Lambda \text{ is in } P \}$
(ii) $W_{i+1} = W_i \cup \{ A \in V_N \mid A \to \alpha,\ \alpha \in W_i^* \}$.

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 $K \le |V_N|$ so, $W_{K+j} = W_K$ $\forall j \ge 0$.

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 $A \to x_1 x_2 \ldots x_k$ is in P, the productions of form $A \to \alpha_1 \alpha_2 \ldots \alpha_k$ are included in P' where $\alpha_i = x_i$ if $x_i \notin W$

$\alpha_i = x_i / \Lambda$ if $x_i \in W$ and $\alpha_1 \alpha_2 \ldots \alpha_k \neq \Lambda$

Actually (ii) gives several productions in P' so, the productions are obtained either by not erasing any nullable variable on R.H.S of $A \to x_1 x_2 \ldots x_k$ or by erasing some or all nullable variables provided some symbol appears on the R.H.S after erasing now $G_1 = (V_N, \Sigma, P', S)$ has no null productions.

$S \to aS / AB,\ A \to \Lambda,\ B \to \Lambda,\ D \to b$ find G_1 ($L(G_1) = L(G) - \{\Lambda\}$)

Step 1:

$W_1 = \{A, B\} \mid A_1 \in V_N,\ A_1 \to \Lambda \quad K \subseteq V_N$

$W_2 = \{A, B, S\} \mid A_1 \in V_N,\ A_1 \to \alpha,\ \alpha \in W^*$

$W_3 = W_2 = W_k = W$

P	2	$S \to AB$
	00	$S \to AB$
	01	$S \to A$
	10	$S \to B$
	11	$S \to \Lambda$

Step 2:

Construction of P'

(i) $D \to b$

(ii) $S \to aS$

$S \to a$

$S \to AB,\ S \to A,\ S \to B$

$S \to A,\ A \to B,\ B \to C,\ C \to \Lambda,\ D \to a,\ D \to aA,$
$D \to AE$

Step 1

$W_1 = \{C\}$

$W_2 = W_1 \cup \{X \mid X \to \alpha,\ \alpha \in W_1^*\}$

$W_2 = \{C, B\}$

$W_3 = W_2 \cup \{X \mid X \to \alpha,\ \alpha \in W_2^* (\{C, B\}^*)\}$

$W_3 = \{C, B\} \cup \{A, S\}$

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

$W_4 = W_3 \cup \{X \mid X \to \alpha,\ \alpha \in W_3^*\}$

$W_4 = \{S, A, B, C\} \cup \Phi$

$W_4 = \{S, A, B, C\} = W_3 = W_k = W$

Step 2: Construction of P'

(i) $D \to a$

(ii) $D \to \underset{\alpha_1}{\underset{\downarrow}{A}}\ \underset{\alpha_2}{\underset{\downarrow}{E}}$

$D \to AE$

$D \to E$

$S \to A, \quad A \to B \quad B \to C$

$S \to \Lambda \quad A \to \Lambda \quad B \to \Lambda.$

So, $L(G_1) = L(G_1) - \{\Lambda\}$

COROLLARY 1 :

There exists an algorithm to decide whether $\Lambda \in L(G)$ for a given CFG '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)

(i) Construct W

(ii) Test whether $S \in W$

COROLLARY 2 :

If $G = (V_N, \Sigma, P, S)$ is a CFG we can find an equivalent CFG $G_1 = (V_N', \Sigma, P_1, S_1)$ without 'Λ' productions except $S_1 \to \Lambda$ when Λ is in $L(G)$ If $S_1 \to \Lambda$ is in P_1, S_1 does not appear on the R.H.S of any production in P_1

proof:

By corollary 1, we can decide whether 'Λ' is in $L(G)$

Case (i)

If Λ is not in $L(G)$

G_1 obtained by using theorem (4) is a required equivalent grammar

Case (ii)

If Λ is in $L(G)$, Construct $G' = (V_N, \Sigma, P', S)$ using theorem 4 $L(G') = L(G) - \{\Lambda\}$

Define $G_1 = (V_N \cup \{S_1\}, \Sigma, P, S_1)$, where

$$P_1 = P' \cup \{S_1 \to S ; S_1 \to \Lambda\}$$

S_1 does not appear on the R.H.S of any production in P_1, and so G_1 is the required grammar with $L(G_1) = L(G)$.

ELIMINATION OF UNIT PRODUCTIONS:

def:

A unit production (or a chain rule) In CFG 'G' is a production of the form $A \to B$, where A & B are variables in G'.

THEOREM

If G is a CFG we can find a CFG G_1 which has no null productions or unit productions such that $L(G_1) = L(G)$

proof: we can apply Corollary 2 of theorem 4 to grammar 'G' to get a grammar $G' = (V_N, \Sigma, P, S)$ without null productions such that $L(G') = L(G)$

Let A, be any variable in V_N

<u>step 1</u>

Construction of the set of variables derivable from A:

define $W_i(A)$ recursively as follows:

$\omega_0(A) = \{A\}$

$\omega_{i+1}(A) = \omega_i(A) \cup \{B / B \in V_N, C \to B$ is in P 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, $\omega_{k+1}(A) = \omega_k(A)$ for some $k \le |V_N|$

so, $\omega_{k+j}(A) = \omega_k(A)\ \forall, j \ge 0$. let $\omega(A) = \omega_k(A)$ then $\omega(A)$ is the set of all variables derivable from A

Step 2:

Construction of A-productions in G_1,
The A-productions in G_1 are either
(1) The non-unit production in G_1
(2) $A \to \alpha$ whenever $B \to \alpha$ is in G_1, with $B \in \omega(A)$ and $\alpha \notin V_N$ (actually 2 covered 1 as $A \in \omega(A)$). now we define $G_1 = (V_N, \Sigma, P_1, S)$ where P_1 is constructed using step2 $\forall, A \in V_N$.

Ex:

Let G be $S \to AB$, $A \to a$, $B \to C/b$, $C \to D$, $D \to E$, $E \to a$ eliminate unit productions and get an equivalent grammar

$\omega(S)$ Step 1:

$\omega_0(S) = \{S\}$

$\omega_1(S) = \omega_0(S) + \{x \mid x \in V_N, Y \to x$ is in P, $Y \in \omega_0(S)\}$

$= \{S\} \cup \phi$

$\omega_1(S) = \{S\}$

$= \omega_0(S)$

$\therefore \omega(S) = \{S\}$

$\omega(A)$

$\omega_0(A) = \{A\}$

$\omega_1(A) = \omega_0(A) + \{x / x \in V_N, Y \to x$ is in P, $Y \in \omega_0(A)\}$

$\omega_1(A) = \{A\} + \phi$

$\omega_1(A) = \{A\}$

$= \omega_0(A)$

$\omega(A) = \{A\}$

$\omega(B)$

$\omega_0(B) = \{B\}$

$\omega_1(B) = \omega_0(B) + \{x / x \in V_N, Y \to x$ is in P, $Y \in \omega_0(B)\}$

$= \{B\} + \{C\}$

$= \{B, C\}$

$\omega_2(B) = \omega_1(B) + \{D\}$

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

$w_3(B) = \{B, C, D\} \cup \{E\}$

$= \{B, C, D, E\}$

$w_4(B) = \{B, C, D, E\} \cup \phi$

$= \{B, C, D, E\}$

$= w_3(B)$

$W(B) = \{B, C, D, E\}$

W(C)

$w_0(C) = \{C\}$

$w_1(C) = \{C\} \cup \{D\} = \{C, D\}$

$w_2(C) = \{C, D\} \cup \{E\}$

$= \{C, D, E\}$

$w_3(C) = w_2(C) \cup \phi$

$= \{C, D, E\}$

$= W_2(C)$

$W(C) = \{C, D, E\}$

W(D)

$w_0(D) = \{D\}$

$w_1(D) = \{D\} \cup \{E\}$

$= \{D, E\}$

$w_2(D) = \{D, E\} \cup \phi$

$= w_1(D)$

$W(D) = \{D, E\}$

W(E)

$w_0(E) = \{E\}$

$w_1(E) = w_0(E) \cup \phi$

$= \{E\}$

$= w_0(E)$

$w(E) = \{E\}$.

Step 2:

Construction of S in G_1

$S \to AB$

A in G_1

$A \to a$

B in G_1

$B \to b$,

$B \to a$

C in G_1

$C \to a$

D in G_1

$D \to a$

E in G_1

$E \to a$

Productions in G_1 are $S \to AB$

$A \to a$

$B \to b/a$

$C \to a$

$D \to a$

$E \to a$.

COROLLARY:

If G is CFG we can construct an equivalent grammar G' 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' equivalent to G_2 (theorem 3)

G' is the required grammar equivalent to G.

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 R.H.S and the nature of symbols in the R.H.S of production.

def:

A CFG G is in CNF if every production is of the form $A \to a$ (or) $A \to BC$ and $S \to \Lambda$ is in G if $\Lambda \in L(G)$. when null is in L(G), we assume that S does not appear on the R.H.S of any production

for eg, Consider G whose productions are

$S \to AB / \Lambda$, $A \to a$, $B \to b$ then G is in CNF.

Remark:

for a grammar in CNF the derivation tree has the following property:

every node has atmost 2 descendants either 2 internal vertices (variables) or a single leaf (terminals)

When a grammar is in CNF, some of the proofs and Constructions are simpler.

Reduction to chomsky normal form:

Now we develop a method of Constructing a grammar in CNF equivalent to a given CFG

Let us first Consider an example.

Let G be $S \to ABC/aC$, $A \to a$, $B \to b$, $C \to c$ except $S \to aC/ABC$ all the productions are in the form required for CNF. The terminal a $S \to aC$ can be replaced by a new variable D by adding a new production $D \to a$, the effect of applying $S \to aC$ can be achieved by $S \to DC$ and $D \to a$. $S \to ABC$ is not in the required form, and hence this production can be replaced by $S \to AE$ and $E \to BC$.

Thus an equivalent grammar is $S \to AE/DC$, $E \to BC$, $A \to a$, $B \to b$, $C \to c$, $D \to a$

The techniques applied in this eg are used in the following theorem.

THEOREM: (reduction to CNF)

for every CFG, there is an equivalent grammar G_2 in CNF.

proof:

Construction of grammar in CNF

Step 1:

Elimination of null & unit productions

we apply theorem 4 to eliminate null production, we then apply theorem 5 to the resulting grammar to eliminate chain (unit) productions

Let the grammar thus obtained be $G = (V_N, \Sigma, P, S)$

Step 2: Elimination of terminals on R.H.S

we define $G_1 = (V_N', \Sigma, P_1, S)$ where P_1 and V_N' are Constructed as follows:

(i) All the productions in P of the form $A \to a$ or $A \to BC$ are included in P_1, All the variables in V_N are included in V_N'

(ii) Consider $A \to x_1 x_2 \cdots x_n$ with some terminal on R.H.S if x_i is a terminal say a_i add a new variable C_{a_i} to V_N' and $C_{a_i} \to a_i$ to P_1

In production $A \to x_1 x_2 \cdots x_n$ 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 $G_1 = (V_N', \Sigma, P_1, S)$

Step 3 Restricting the no. of variables on R.H.S

for any production in P_1 the R.H.S Consists of either a single terminal or (Λ in $S \to \Lambda$) two or more variables. we define $G_2 = (V_N'', \Sigma, P_2, S)$ 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_N' are added to V_N''

(ii) Consider $A \to A_1 A_2 \cdots A_m$ where $m \geq 3$. we introduce new productions $A \to A_1 C_1$, $C_1 \to A_2 C_2 \cdots C_{m-2} \to A_{m-1} A_m$ and 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_2 to CNF

G is $S \to aAD$, $A \to aB \,/\, bAB$, $B \to b$, $D \to d$

step 1:

In a given grammar there are no null productions and unit productions

Step 2:

(a): $B \to b$, $D \to d$

(b) $S \to aAD$ is replaced by $S \to C_aAD$, $C_a \to a$

$A \to aB$ is replaced by $A \to C_aB$

$A \to bAB$ is replaced by $A \to C_bAB$, $C_b \to b$

$V_N' = (S, A, B, D, C_a, C_b)$

Step 3:

$B \to b$, $D \to d$

$A \to C_aB$, $C_a \to a$, $C_b \to b$

$S \to C_aC_1$, $C_1 \to AD$

$A \to C_bC_2$, $C_2 \to AB$

$V_N'' = (S, A, B, D, C_a, C_b, C_1, C_2)$.

$A \to a$

$A \to AB$

$A \to a\underline{B}$

eg,

$A \to AC_aBC_bDE$

$A \to AC_1$ $\quad C_1 = C_aBC_bDE$

$C_1 \to C_aC_2$ $\quad C_2 = BC_bDE$

$C_2 \to BC_3$ $\quad C_3 = C_bDE$

$C_3 \to C_bC_4$ $\quad C_4 = DE$

$C_4 \to DE$

$S \to aAb\cancel{b}B$, $A \to aA/a$, $B \to bB/a$

Step 2

(a): $A \to a$, $B \to a$

(b) $S \to aAbB$ is replaced by $S \to C_aAC_bB$

$C_a \to a$, $C_b \to b$

$A \to aA$ is replaced by $A \to C_aA$

$B \to bB$ is replaced by $B \to C_bB$

$V_N' = (\cancel{S}\ A, B, C_a, C_b)$.

Step 3

$A \to C_aA$, $B \to C_bB$.

$S \to \cancel{C_aC_b}\ \cancel{AB}\ C_aAC_bB$ $\to$ AB

$V_N'' = (S, A, B, C_a, C_b)$

GREIBACH NORMAL FORM:

def:

A CFG is in GNF if every production is of the form $A \to a\alpha$ where $\alpha \in V_N^*$ and $a \in \Sigma$ (α may be Λ) and $S \to \Lambda$ is in G if $\Lambda \in L(G)$ when $\Lambda \in L(G)$ we assume that 'S' does not appear on the R.H.S of any production for eg, G given by $S \to aAB / \Lambda$, $A \to bC$, $B \to b$, $C \to c$ is in GNF

Lemma 1

Let $G = (V_N, \Sigma, P, S)$ be a CFG let $A \to B\gamma$ be an A-production in P Let the B-productions be $B \to \beta_1 / \beta_2 \ldots / \beta_s$ define $P_1 = (P - \{A \to B\gamma\}) \cup \{A \to \beta_i \gamma / 1 \le i \le s\}$

then $G_1 = (V_N, \Sigma, P_1, S)$ is a CFG equivalent to G

NOTE.

Lemma 1 is useful for deleting a variable 'B' appearing as the first symbol of the R.H.S of A-productions, provided no B-production has B as the first symbol on R.H.S.

The Construction given in lemma-1 is simple. To eliminate B in $A \to B\gamma$ is simply replaced B by the right hand side of every B-production

for eg, we can replace $A \to Bab$ by $A \to aAab$, $A \to bBab$, $A \to aaab$, $A \to ABab$ when the B-productions are aA, bB, aa, AB.

If

$A \to A\alpha_1 / A\alpha_2 / \ldots / \beta_1 / \beta_2 \ldots$

then

$A \to \beta_1 / \beta_2 / \ldots$

$A \to \beta_1 z_1 / \beta_2 z_2$

$A \to \alpha_1 / \alpha_2 \ldots$

$A \to \alpha_1 z_1 / \alpha_2 z_1 /$

Lemma 2:

Let $G = (V_N, \Sigma, P, S)$ be CFG let the set of A-productions be $A \to A\alpha_1 / \ldots / A\alpha_r / \beta_1 / \ldots / \beta_s$ (β_i donot start with A)

Let z be a new variable let $G_1 = (V_N \cup \{z\}, \Sigma, P_1, S)$ where P_1 is defined as follows,

(1) The set of A-productions in P_1 are $A \to \beta_1 / \beta_2 \ldots / \beta_s$

$$A \to \beta_1 z / \beta_2 z \ldots / \beta_s z$$

The set of z-productions in P_1 are

(ii) $z \to \alpha_1 / \alpha_2 \ldots / \alpha_r$

$$z \to \alpha_1 z / \alpha_2 z \ldots / \alpha_r z$$

(iii) The productions for the other variables are as in P then G_1 is a CFG and equivalent to G.

Apply lemma 2 to the following A-productions in a CFG in G.

$$A \to \underset{\beta_1}{aBD} / \underset{\beta_2}{bDB} / \underset{\beta_3}{c} / \underset{\alpha_1}{AB} / \underset{\alpha_2}{AD}$$

(i) $A \to aBD / bDB / c$

$A \to aBDz / bDBz / cz$

(ii) $z \to B / D$

$z \to Bz / Dz$

CONVERSION OF CNF to GNF :

Step 1:

Eliminate null & unit productions then convert it into CNF (which are not in required form).

Rename the variables as $A_1, A_2 \ldots A_n$ with $S = A_1$. Now G has $(\{A_1 A_2 \ldots A_n\}, \Sigma, P, A_1)$.

Step 2:

Convert the productions of the form $A_i \to A_j \gamma$ where $i > j$ into $A_i \to A_i \gamma'$ by applying lemma 1 iteratively.

Step 3:

Apply the lemma 2 for the productions of the form $A_i \to A_i \gamma'$

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) of R.H.S of a variable Z_i

Construct a grammar in GNF equivalent to the grammar $S \to AA / a$ $A \to SS / b$ →

Step 1

The given grammar has no null & unit productions so, it is in CNF.

Rename the variables as S as A_1 and A as A_2.

so,

$A_1 \to A_2 A_2 / a$

$A_2 \to A_1 A_1 / b$

CNF

step 2:

$A_1 \to \boxed{A_2} A_2 / a$ & $A_2 \to b$ are required form

Consider $A_2 \to A_1 A_1$

by applying lemma 1 we get,

$A_2 \to A_1 A_1$

$A_2 \to A_2 A_2 A_1 / a A_1$

Step 3:

Applying lemma 2. $A_2 \to A_2 A_2 A_1$

(i) $A_2 \to aA_1$, $A_2 \to aA_1Z_2$ —— (1) $A_2 \to b$.

(ii) $Z_2 \to A_2A_1$; $Z_2 \to A_2A_1Z_2$

Step 4:

The production $A_1 \to a$ is in the required form and $A_1 \to A_2A_2$ is not in the required form so, apply lemma 1

$$A_1 \to aA_1A_2 / aA_1Z_2A_2 \longrightarrow (2)$$

$$A_1 \to a$$

Step 5:

Consider the production $Z_2 \to A_2A_1 / A_2A_1Z_2$ & apply lemma 1,

$$\left.\begin{array}{l} Z_2 \to aA_1A_1 / aA_1Z_2A_1 \\ Z_2 \to aA_1A_1Z_2 / aA_1Z_2A_1Z_2 \end{array}\right\} \to (3).$$

Convert the grammar to GNF $B \to a$ $C \to a$

$S \to aB$ $S \to AB$, $A \to a$, $B \to C / b$ $C \to D$, $D \to E$, $E \to a$ $D \to a$

Convert the grammar to GNF

$S \to AB$, $A \to BSB$, $A \to BB$, $B \to aAb$ / $C_b \to b$

$B \to a$, $A \to b$.

Convert the grammar to GNF

$S \to AB$, $A \to BS / b$ $B \to SA / a$.

Step 1

Given grammar does not have unit & null productions and it is in CNF

Rename the variables, S, A, B as A_1, A_2, A_3 respectively.

∴ The productions are

$$A_1 \to A_2A_3$$
$$A_2 \to A_3A_1 / b$$
$$A_3 \to A_1A_2 / a$$

Step 2

The production $A_1 \to A_2A_3$ are in required form the production $A_2 \to A_3A_1 | b$ & $A_3 \to a$ is in required from but the production

$A_3 \to A_1A_2$ ($j \not< i$).

by applying lemma 1,

$A_3 \to A_2A_3A_2$ ($j < i$)

by applying lemma 1,

$$A_3 \to A_3A_1A_3A_2.$$

$$A_3 \to b\,A_3A_2$$

Step 3:

Apply lemma 2 for the production $A_3 \to A_3A_1A_3A_2$

$A_3 \to bA_3A_2$ $\quad\quad$ $A_3 \to bA_3A_2Z_3$ —— (1)

$\cancel{A_3} \to A_1A_3A_2$ $\quad\quad$ $Z_3 \to A_1A_3A_2Z_3$

Step 4:

$A_2 \to b$ is in the required form but $A_2 \to A_3A_1$ is not the GNF

$A_2 \to A_3A_1$

applying lemma 1,

$A_2 \to bA_3A_2\underline{A_1} \mid bA_3A_2Z_3A_1$ and

$A_2 \to b$ —— (2).

$A_1 \to A_2A_3$ is not in GNF

applying lemma 1,

$A_1 \to bA_3A_2A_1A_3$

$A_1 \to bA_3A_2Z_3A_1A_3$ ——→ (3)

~~Applying~~ bA_3.

Step 5

$$\left.\begin{array}{l} Z_3 \to bA_3A_2A_1A_3A_3A_2 \\ Z_3 \to bA_3A_2Z_3A_1A_3A_3A_2 \\ Z_3 \to bA_3A_3A_2 \end{array}\right\} 4$$

$$\left.\begin{array}{l} Z_3 \to bA_3A_2A_1A_3A_3A_2Z_3 \\ Z_3 \to bA_3A_2Z_3A_1A_3A_3A_2Z_3 \\ Z_3 \to bA_3A_3A_2Z_3 \end{array}\right\} 4$$

PROPERTIES OF CF languages:

1) CFL's are closed under union.

eg,

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 respecti-vely.

$\therefore S \to S_1/S_2$ is also a CFL

2) CFL's are closed under Concantenation

$S \to S_1S_2$

3) CFL's are closed under closure.

$S \to S_1^*$

$S \to S_1S/\Lambda$

4) CFL's are closed under reversal

5) CFL's are closed under cyclic shift

eg,

$S \to abc$ by cyclicshift $S \to cab$ is also CFL.

$S \to \alpha\beta\gamma$ by cyclic shift $S \to \gamma\alpha\beta$ is also CFL

6) CFL's are not closed under intersection.

7) CFL's are not closed under difference.

8) CFL's are not closed under Compliment

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 recognisation also in processing the spoken word.

5) The expressive power of CFG is too limited adequately capture all natural language phenomenon. Therefore extensions of CFG are of interest for Competitional linguistics.

DIFFERENCE b/w AMBIGIOUS & ANAMBIGIOUS GRAMMARS:

AMBIGIOUS GRAMMAR	ANAMBIGIOUS GRAMMAR
There exists more than one LMD or RMD for a string	Unique LMD/RMD
LMD & RMD represents different parse trees	LMD & RMD represents same parse tree
More than 1 parse tree for a string	Unique parse tree

$E \rightarrow id / E+E / E*E / E-E$

$\rightarrow id+id+id$

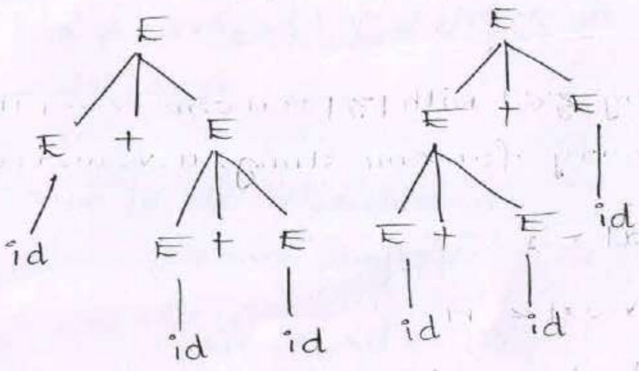
$E \rightarrow E+E$

$E \rightarrow id+E$

$E \rightarrow id+E+E$

$E \rightarrow id+id+E$

$E \rightarrow id+id+id$



the string id+id+id has '2' derivation trees. So, the string id+id+id is ambigous. So, the given grammar is ambigous.

PUMPING LEMMA FOR CFG:

Let 'L' be a CFG then we can find a natural number 'n' such that

→ $a^n b^n$ is a CFG i.e, $S \longrightarrow aSb$.

→ $a^n b^n c^n \longrightarrow$ not a CFG

$S \rightarrow aSt$

ww^T

→ $w \in (a,b)^*$

$S \rightarrow aSa / bSb / a / b / aa / bb$

It is a CFG.

(1) Every $z \in L$ with $|z| \geq n$ can be written as $uvwxy$ for some strings u, v, w, x, y

(2) $|vx| \geq 1$

(3) $|vwx| \leq n$

(4) $uv^k wx^k y \in L \ \forall \ k \geq 0$.

We use the pumping lemma to show that a language L is not a CFL. We assume that L is CF. 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 $|z| \geq n$ write $z = uvwxy$ using the pumping lemma

Step 3

find a suitable k so that $uv^k wx^k y \notin L$

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 CF let 'n' be the natural number obtained by using the pumping lemma

Step 2

let p be a prime number > n

then $z = a^p \in L$

we write $z = uvwxy$ then

Step 3

let $k = 0$ then the string

$uv^k wx^k y = uv^0 wx^0 y = uwy$

$|uwy| = q$ ($\because q$ is prime)

assume $|vx| = n$

then ~~apply~~ assume k as q

then $uv^q wx^q y$

$|uv^q wx^q y| = |uwy| + |v^q x^q|$

$= q + q|vx|$

$= q + qn$

$= q(1+n)$ This is not a prime number so, $uv^k wx^k y \notin L$ This is a Contradiction so, L is not a CFG

DECISION ALGORITHMS:

Algorithm1:-for determining a given CFL is empty

After application of theorem 1, if $S \in \omega$ then the given CFL is not empty otherwise the CFL is empty

Algorithm2:

Algorithm for determining a CFL is infinite.

Construct a non redundant CFG 'G' in CNF.

$L - \{\wedge\}$

we draw a directed graph whose vertices are variables in G, if $A \to BC$ is a production there are directed edges from $A \to B$ & $A \to C$

L 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 F.A M accepting L

we construct the set of all states reachable from the 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 tabular 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 F.A M (transition diagram) accepting L. L is infinite iff M 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 $Q \times (\Sigma \times \{\Lambda\}) \times \Gamma$ 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

Example: for $a^n b^n$.

transition function rules:

$\delta(q_0, a, z_0) = (q_0, az_0)$

$\delta(q_0, a, a) = (q_0, aa)$.

$\delta(q_0, b, a) = (q_1, \wedge)$.

$\delta(q_1, b, a) = (q_1, \wedge)$.

$\delta(q_1, \wedge, z_0) = (q_f, \wedge)$.

eg, aaabbb.

$$\begin{aligned}\delta(q_0, aaabbb, z_0) &= \delta(q_0, aabbb, az_0)\\ &= \delta(q_0, abbb, aaz_0)\\ &= \delta(q_0, bbb, aaaz_0)\\ &= \delta(q_1, bb, aaz_0)\\ &= \delta(q_1, b, az_0)\\ &= (q_1, \wedge, z_0)\\ &= (q_f, \wedge).\end{aligned}$$

Equal no. of 0'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

$\delta(q_0, a, z_0) = (q_0, az_0)$.

$\delta(q_0, b, z_0) = (q_0, bz_0)$

$\delta(q_0, a, a) = (q_0, aa)$.

$\delta(q_0, b, b) = (q_0, bb)$

$\delta(q_0, a, b) = (q_0, \Lambda)$

$\delta(q_0, b, a) = (q_0, \Lambda)$

$\delta(q_0, \Lambda, z_0) = (q_f, \Lambda)$

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

$\delta(q_0, 0, z_0) = (q_0, 0z_0)$

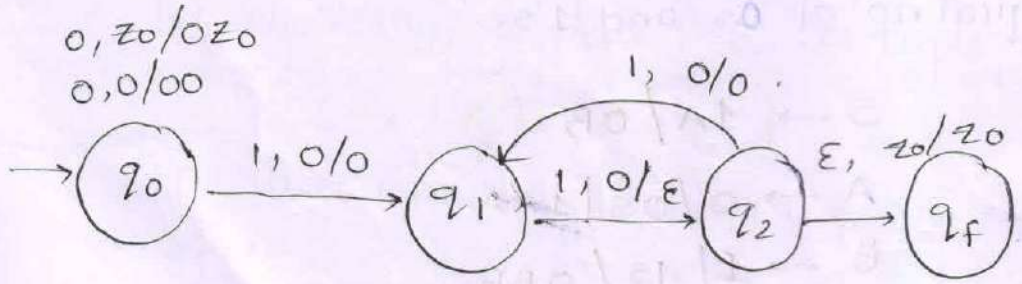
$\delta(q_0, 0, 0) = (q_0, 00)$

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

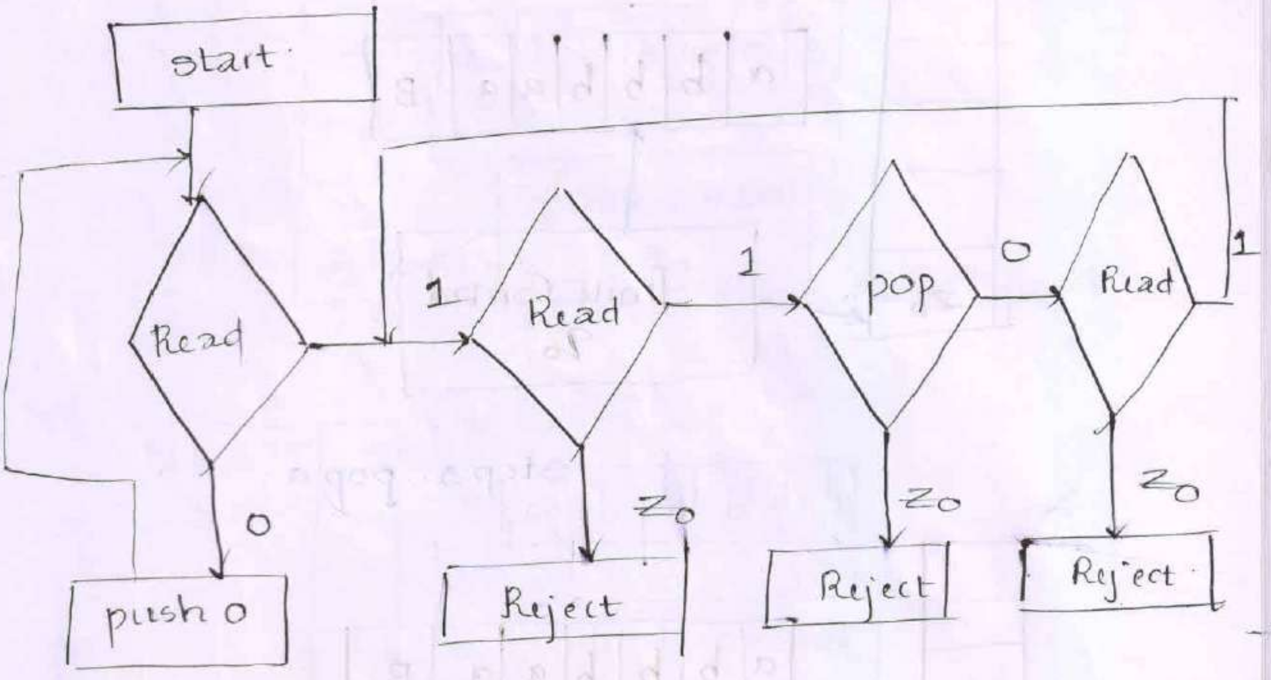
$\delta(q_1, 1, 0) = (q_2, \varepsilon)$

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

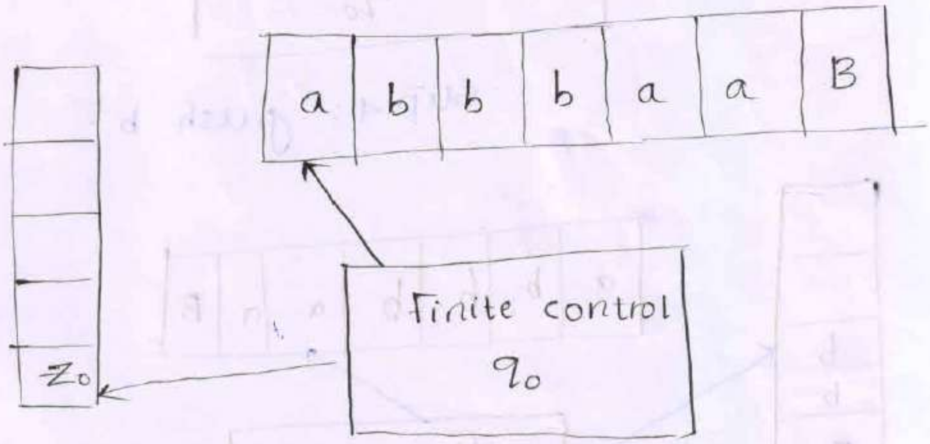
$\delta(q_2, \varepsilon, z_0) = (q_f, z_0)$



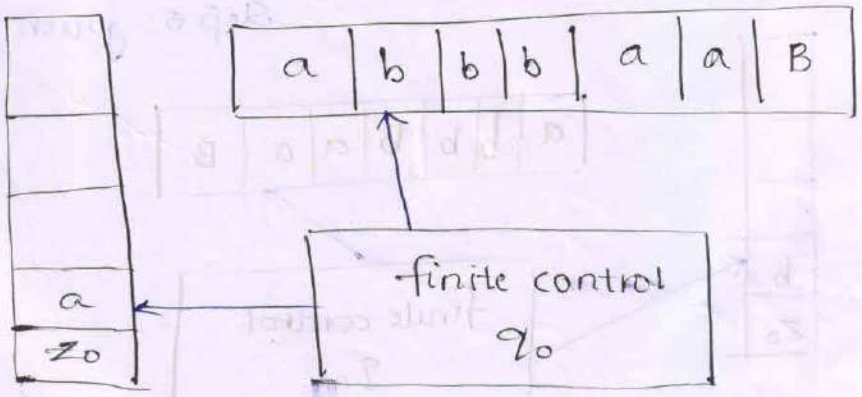
5 (3)



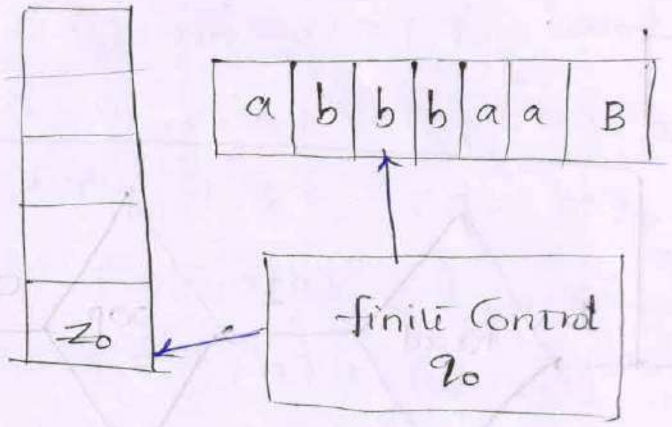
equal no. of a's and equal no. of b's.



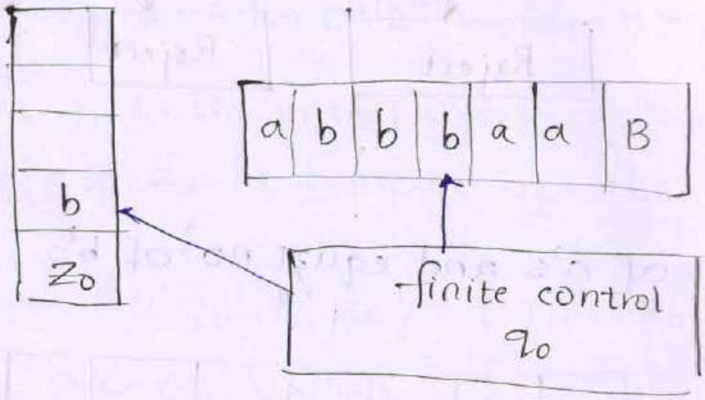
Step1: initial configuration



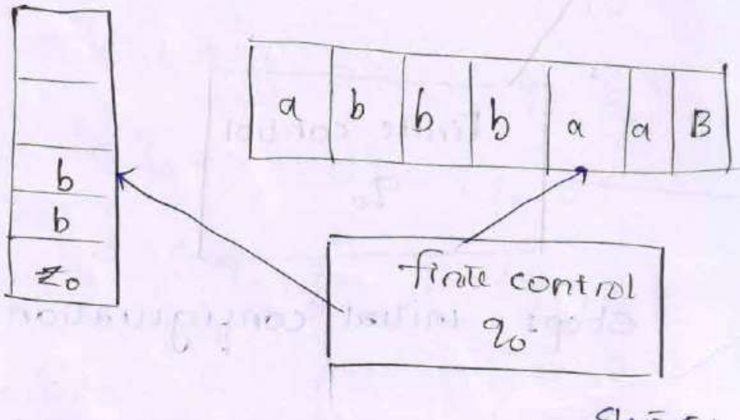
Step2: a is pushed



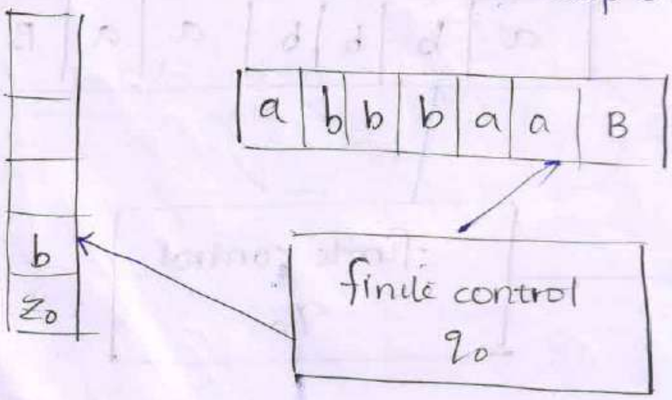
Step 3: pop a.



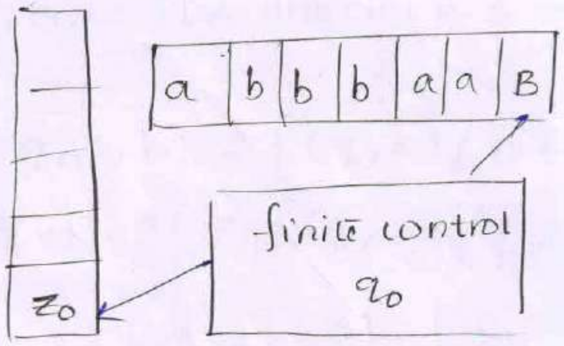
Step 4: push b.



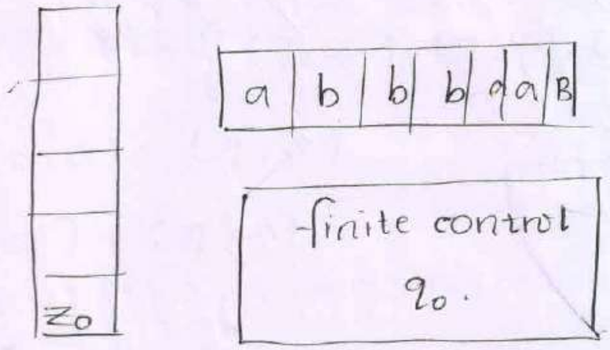
Step 5: push b



Step 6: pop b



step 7: pop b



step 8 : final state

Convert the CFG to PDA the CFG is $S \to 0BB$, $B \to 0S/1S/0$.

R_1: $\delta(q, \Lambda, A) = \{(q, \alpha) / A \to \alpha$ is in $P\}$

R_2: $\delta(q, a, a) = \{(q, \Lambda) /$ for every a in $\Sigma\}$

$\delta(q, \Lambda, S) = (q, 0BB)$

$\delta(q, \Lambda, B) = \{(q, 0S), (q, 1S), (q, 0)\}$

$\delta(q, 0, 0) = (q, \Lambda)$

$\delta(q, 1, 1) = (q, \Lambda)$.

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 $S \overset{*}{\Rightarrow} UA\alpha$ by a left most derivation then

$(q, uv, S) \vdash (q, v, A\alpha)$

eg, $S \to aaBA$

(i) $(q, aaA, S) \vdash (q, A, BA)$

(ii) $(q, aAa, S) \vdash (q, Aa, aBA)$.

CONVERSION FROM PDA to CFG: If $A(Q, \Sigma, \Gamma, \delta, q_0 F)$ is a PDA Then there exist a CFG then $L(G) = N(A)$

Step 1:

Construction of G we define $G(V_N, \Sigma, P, S)$ where $V_N = \{S\} \cup \{[q, z, q'] / q, q' \in Q, z \in \Gamma\}$

7

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

$\Gamma = \{z_0, z_1\}$

$\delta(q_1 a_1 z) \cdot (q' \wedge) \to \delta(q_1 z, q') \to a$

The non-terminals of V_N is $S, [q_0, z_0, q_0]$,

//ly $[q_0, z_0, q_1]$, $[q_0, z_0, q_1]$, $[q_1, z_0, q_0]$,

$[q_0 z_1, q_0]$, $[q_0, z_1, q_1]$, $[q_1, z_1, q_1]$.

$[q_1, z_1, q_0]$

The productions are

R_1: S - productions are given by $S \to [q_0, z_0, q]$ for every q in Q.

ex: $S \to [q_0, z_0, q_0]$

$S \to [q_0, z_0, q_1]$

R_2: each move erasing a push down symbol given by $(q', \wedge) \in \delta(q, a, z)$ induces the production.

$[q, z, q'] \to a$

ex: $\delta(q_0, a, z_0) = (q_1, \wedge)$

$[q_0, z_0, q_1] \to a$.

R_3: Each move not erasing a push down symbol given by $(q_1, z_1 z_2 \ldots\ldots z_n) \in \delta(q, a, z')$ induces many productions of the form $[q, z, q'] \to a[q_1, z_1, q_2][q_2, z_2, q_3] \ldots\ldots [q_m, z_m, q']$.

where each of the states $q', q_2, \ldots q_m$

can be any state in Q.

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

$\Gamma = \{z_0, z\}$

$\boxed{\delta(q_0, b, z_0) = (q_0, zz_0)}$

$[q_0, z_0, q_0] \longrightarrow b[q_0, z, q_0][q_0, z, q_0]$

$[q_0, z_0, q_0] \longrightarrow b[q_0, z, q_1][q_1, z, q_0]$

Construct a PDA accepting $a^n b^m a^n$ $m, n \geq 1$.
Construct the corresponding CFG accepting the same sets.

$\delta(q_0, a, z_0) = (q_0, az_0)$

$\delta(q_0, a, a) = (q_0 aa)$.

$\delta(q_0, b, a) = (q_1, a)$.

$\delta(q_1, b, a) = (q_1, a)$.

$\delta(q_1, a, a) = (q_2, \wedge)$

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

$\delta(q_2, \wedge, z_0) = (q_f, \wedge)$

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

$\Gamma = \{z_0\}$

R_1: S-productions are given by $S \rightarrow [q_0, z_0, q]$ for every q in Q

$S \rightarrow [q_0, z_0, q_0] / [q_0, z_0, q_1] / [q_0, z_0, q_2] /$
$[q_0, z_0, q_f]$

$\delta(q_0, a, Z_0) = (q_0, aZ_0)$ gives

$[q_0, Z_0, q_0] \longrightarrow a[q_0, a, q_1][q_1, Z_0, q_0]$

$[q_0, Z_0, q_1] \longrightarrow a[q_0, a, q_2][q_2, Z_0, q_1]$.

$\delta(q_0, a, a) = (q_0, aa)$ gives

$[q_0, a, q_0] \longrightarrow a[q_0, a, q_1][q_1, a, q_0]$.

$\delta(q_2, b, a) = (q_2, \Lambda)$.

$[q_2, a, q_2] \rightarrow a$

$\delta(q_1, a, a) = (q_2, \Lambda)$

$[q_1, a, q_2] \longrightarrow a$

$\delta(q_2, \Lambda$

P_1: $S \rightarrow [q_0, z_0, q_0] / [q_0, z_0, q_1] / [q_0, z_0, q_2] /$
$[q_0, z_0, q_f]$

$\delta(q_0, a, z_0) = (q_0, az_0)$ gives

P_2: $[q_0, z_0, q_0] \rightarrow a[q_0, a, q_0][q_0, z_0, q_0]$
P_3: $[q_0, z_0, q_0] \rightarrow a[q_0, a, q_1][q_1, z_0, q_0]$
P_4: $[q_0, z_0, q_0] \rightarrow a[q_0, a, q_2][q_2, z_0, q_0]$
P_5: $[q_0, z_0, q_0] \rightarrow a[q_0, a, q_f][q_f, z_0, q_0]$
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: $[q_0, z_0, q_1] \rightarrow a[q_0, a, q_f][q_f, z_0, q_1]$
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}: $[q_0, z_0, q_2] \rightarrow a[q_0, a, q_f][q_f, z_0, q_2]$
P_{14} $[q_0, z_0, q_f] \rightarrow a[q_0, a, q_0][q_0, z_0, q_f]$
P_{15} $[q_0, z_0, q_f] \rightarrow a[q_0, a, q_1][q_1, z_0, q_f]$
P_{16}: $[q_0, z_0, q_f] \rightarrow a[q_0, a, q_2][q_2, z_0, q_f]$
P_{17}: $[q_0, z_0, q_f] \rightarrow a[q_0, a, q_f][q_f, z_0, q_f]$

$\delta(q_0, b, a) = (q_1, a)$ gives.

$P_1: [q_0, a, q_0] = b[q_1, a, q_0]$

$P_2: [q_0, a, q_1] = b[q_1, a, q_1]$

$P_3: [q_0, a, q_2] = b[q_1, a, q_2]$

$P_4: [q_0, a, q_f] = b[q_1, a, q_f]$.

Turing Machine

Turing machine is represented by 3-types

(1) ID (Instanteneous description) using move relations

(2) Transition table

(3) Transition diagram (Transition grammar).

ID·

Moves in turing machine.

Suppose $\delta(q, x_i) = (p, y, L)$.

$$x_1 x_2 \cdots\cdots x_{i-1}\, q\, x_i \cdots\cdots x_n$$

$$\delta(q, x_i) = (p, y, L)$$

$$x_1 x_2 \cdots\; p\, x_{i-1}\, y \cdots\cdots x_n$$

$$\delta(q, x_i) = (P, y, R)$$

$$x_1 x_2 \cdots\cdots x_{i-1}\, q\, x_i \cdots\cdots x_n$$

$$x_1 x_2 \cdots\cdots x_{i-1}\, y\, p \cdots\cdots x_n$$

TRASITION TABLE:

Present state	b	0	1
$\rightarrow q_1$	$1Lq_2$	$0Rq_1$	
q_2	bRq_3	$0Lq_2$	$1Lq_2$
q_3		bRq_4	bRq_5
q_4	$0Rq_5$	$0Rq_4$	$1Rq_4$
q_5	$0Lq_2$		

Turing Machine

NOTE:

If $\delta(q, a) = (\gamma, \alpha, \beta)$ we write α, β, γ under the a column and in the q row. So, if we get α, β, γ in the table it means α is written the Current cell (a is replaced by α), β gives the movement of the head (L or R) and γ 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 $0^n 1^n$.

	0	1	x	y	b
$\rightarrow q_1$	xRq_2			yq_4R	
q_2	$0Rq_2$	yLq_3		yRq_2.	
q_3	$0Lq_3$		xRq_1	yLq_3	
q_4				yRq_4	bRq_5
q_5					

Design a Turing machine $1^n 2^n 3^n$.

	1	2	3	b
$\rightarrow q_1$	bRq_2			bRq_1
q_2	$1Rq_2$.	bRq_3		bRq_2
q_3		$2qR_3$	bRq_4	bRq_3
q_4			$3Lq_5$	bRq_b
q_5		$2Lq_6$		bLq_5
q_6	$1Lq_7$	$2Lq_6$		bLq_6
q_7	$1Lq_7$			bRq_1

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

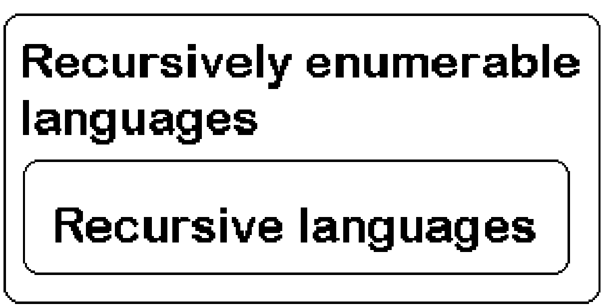
Remember that there are *three* possible outcomes of executing a Turing machine over a given input. The Turing machine may

- 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.)



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 If L2 are two recursive languages, their union L1∪L2 will also be recursive because if TM halts for L1 and halts for L2, it will also halt for L1∪L2.
- **Concatenation:** If L1 and If L2 are two recursive languages, their concatenation L1.L2 will also be recursive. For Example:
- L1= $\{a^n b^n c^n | n>=0\}$
- L2= $\{d^m e^m f^m | m>=0\}$
- L3= L1.L2
- = $\{a^n b^n c^n d^m e^m f^m | m>=0 \text{ and } n>=0\}$ 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 L1is recursive, its kleene closure L1* will also be recursive. For Example:

L1= $\{a^n b^n c^n | n>=0\}$

L1*= $\{ a^n b^n c^n || n>=0\}^*$ is also recursive.

- **Intersection and complement**: If L1 and If L2 are two recursive languages, their intersection L1 ∩ L2 will also be recursive. For Example:
- L1= $\{a^n b^n c^n dm | n>=0 \text{ and } m>=0\}$
- L2= $\{a^n b^n c^n d^n | n>=0 \text{ and } m>=0\}$
- L3=L1 ∩ L2
- = $\{ a^n b^n c^n d^n | n>=0\}$ 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, complementof recursive language L1 which is ∑*-L1, will also be recursive.

RE - Recursive Enumerable REC- Recursive Language

Note: As opposed to REC languages, RE languages are not closed under complementon 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 $\boldsymbol{w}$ in the alphabet of the standard TM, and will halt if and only if the standard TM halts on $\boldsymbol{w}$.

UNIT – V

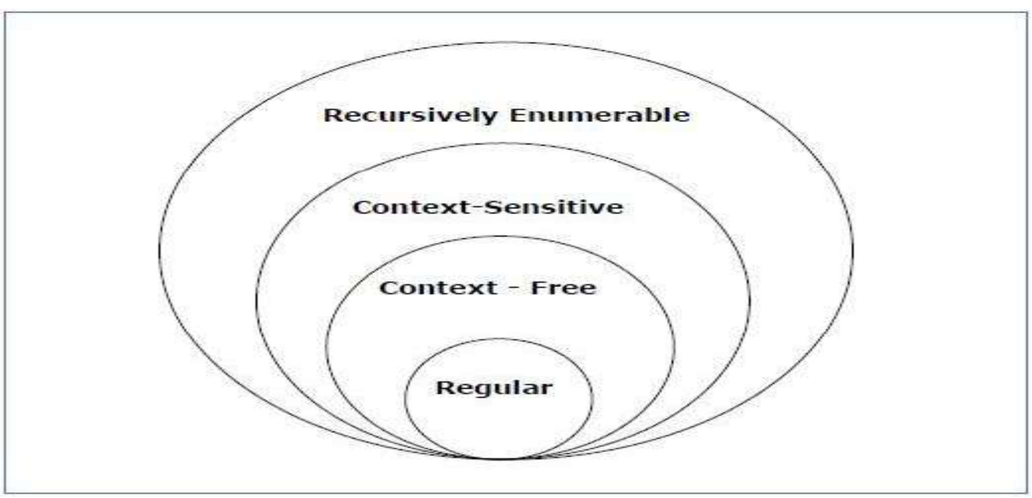
Syllabus:
Computability Theory: Chomsky hierarchy of languages, Linear Bounded Automata and Context Sensitive Language, LR(0) grammar, Decidability of problems, Universal Turing Machine, Undecidability of Posts Correspondence Problem, Turing Reducibility, Definition of P and NP problems.

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 −



Grammar Type	Production Rules	Language Accepted	Automata	Closed Under
Type-3 (Regular Grammar)	A→a or A→aB where A,B ∈N(non terminal) and a∈T(Terminal)	Regular	Finite Automata	Union, Intersection, Complementation, Concatenation, Kleene Closure
Type-2 (Context Free Grammar)	A->ρ where A∈N and $\rho \in (T \cup N)^*$	Context Free	Push Down Automata	Union, Concatenation, Kleene Closure
Type-1 (Context Sensitive Grammar)	α→β where α,β∈ (T∪N)* and len(α) <= len(β) and α should contain atleast 1 non terminal.	Context Sensitive	Linear Bound Automata	Union, Intersection, Complementation, Concatenation, Kleene Closure
Type-0 (Recursive Enumerable)	α → β where $\alpha,\beta \in (T \cup N)^*$ andα 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 → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S** → **ε** is allowed if **S** does not appear on the right side of any rule.

Example

$X \rightarrow \varepsilon$

$X \rightarrow a \mid aY$

$Y \rightarrow b$

Type - 2 Grammar:

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

S → Xa
X → a
X → aX
X → abc
X → ε

Type - 1 Grammar:

Type-1 grammars **generate context-sensitive languages.**
The productions must be in the form
α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule **S → ε** 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 → AbBc
A → bcA
B → b

Type - 0 Grammar:

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase 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 **α → β** 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

S → ACaB
Bc → acB
CB → DB
aD → Db

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:

< q0a1a2a3a4a5.......an >

Formal Definition:

Formally Linear Bounded Automata is a non-deterministic Turing Machine, M=(Q, P, Γ, δ, F ,q0, t, r)

- Q is set of all states
- P is set of all terminals
- Γ is set of all tape alphabets P ⊂ Γ
- δ is set of transitions
- F is blank symbol
- q0 is the initial state
- < is left marker and > is right marker
- t is accept state
- r is reject state

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 $A \rightarrow \in$ 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
  $L(G) = \{w \mid w \in \sum* \text{ and } S \Rightarrow+ G\ w\}$
- CSG for $L = \{ a^n b^n c^n \mid n \geq 1 \}$
  - N : {S, B} and P = {a, b, c}
  - P : $S \rightarrow aSBc \mid abc\ cB \rightarrow Bc\ bB \rightarrow bb$
- 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 $S \rightarrow SABC$
$\underline{S}ABC$	rule $S \rightarrow aBC$
$aB\underline{CA}BC$	rule $CA \rightarrow AC$
$aBA\underline{CB}C$	rule $CB \rightarrow BC$
$a\underline{BA}BCC$	rule $BA \rightarrow AB$
$\underline{aA}BBCC$	rule $aA \rightarrow aa$
$a\underline{aB}BCC$	rule $aB \rightarrow ab$

aabBCC rule bB → bb
aabbCC rule bC → bc
aabbcC 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
- Intesection

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 → ε
    - B → · is an item

Example: Items

- Given our example grammar:
  - S' → Sc, S → SA|A, A → aSb|ab
- The items for the grammar are:

S'→·Sc, S'→S·c, S'→Sc·
S→·SA, S→S·A, S→SA·, S→·A, S→A·

$A \rightarrow \cdot aSb$, $A \rightarrow a \cdot Sb$, $A \rightarrow aS \cdot b$, $A \rightarrow aSb \cdot$, $A \rightarrow \cdot ab$, $A \rightarrow a \cdot b$, $A \rightarrow ab \cdot$

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

    - A → a·b for prefix a
    - A → ·ab for prefix ε
  - A→ab· is a complete item, ∴ Ac 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
  - ∀ prefixes γ of G where A→α· 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 → E$
2. E → E+(E)
3. E → id

The LR(0) items (simply place a dot at point in every production)

1. S → • E$
2. S → E• $
3. S → E$•
4. E → • E+(E)
5. E → E• +(E)
6. E → E+• (E)
7. E → E+(• E)
8. E → E+(E•)
9. E → E+(E)•
10. E → • id
11. E → id•

Creating states from Items

States are composed of **closures** constructed from items. Initially the only closure is **{S → • E$}**. Next, we construct the closure like so:
Closure(I) = Closure(I) ∪ {A → • α | B → β• Aγ ∈ I}

Basically, for a non-terminal \(A\) in \(I\) with a • before it, add all items of the form "A → • …".

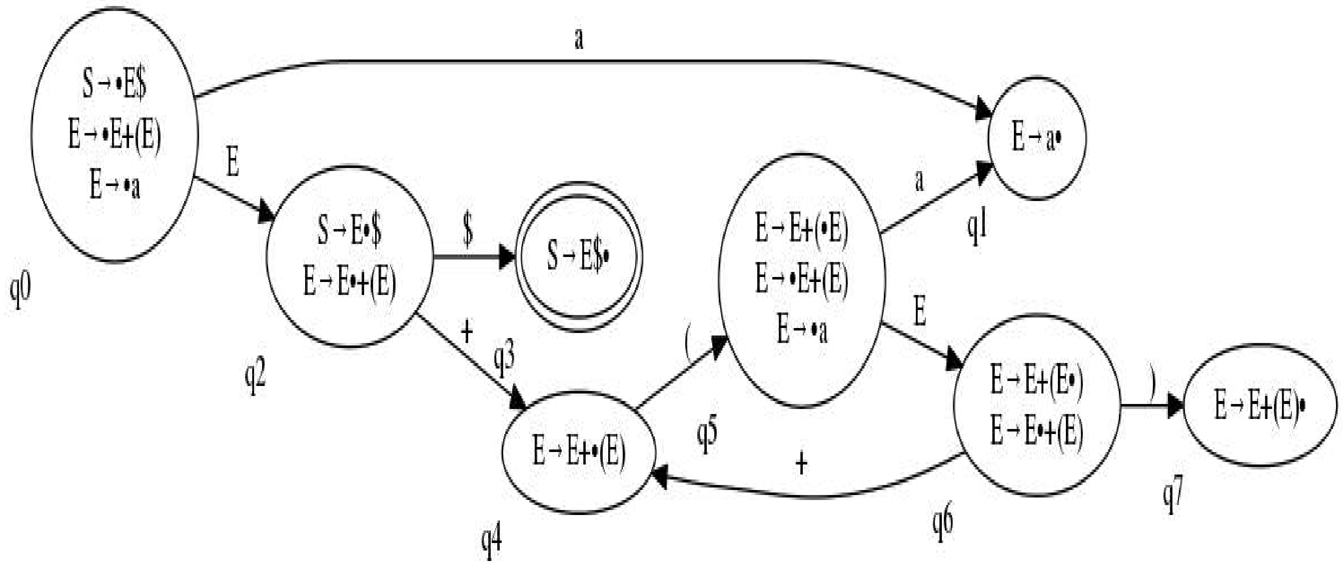
Given our example (**Initial = {S → • E$}**) we create the following closure:
Closure({S → • E$ } = {S → • E$, E → • E+(E), E → • id}

to create more closures we define a "goto" function that creates new closures. Given a closure \(I\) and a symbol \(a\) (terminal or non-terminal):
goto(I, a) = {B → α a• β | B→α • aβ ∈ I}

Basically, For every item in \(I\) that has a • before \(a\) we create a new closure by pushing the • one symbol forward. For instance, given our example closure and the symbol \(E\) we get:
goto({S → • E$, E → • E+(E), E → • id}, E) = {S → E• $, E → E• +(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 transitions between the closures. Transition from state q_1 to state q_2 given symbol a \(\iff\) 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)

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 =Null and M-L =Null.
  Hence (L-M) U (M-L) = Null, then L,M are equivalent.
- Membership of a CFL?
  We can always find whether a string exist in a given CFL by using on 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>=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

always unsure whether a contradiction exists or not and hence we term this problem as an **Undecidable Problem**.

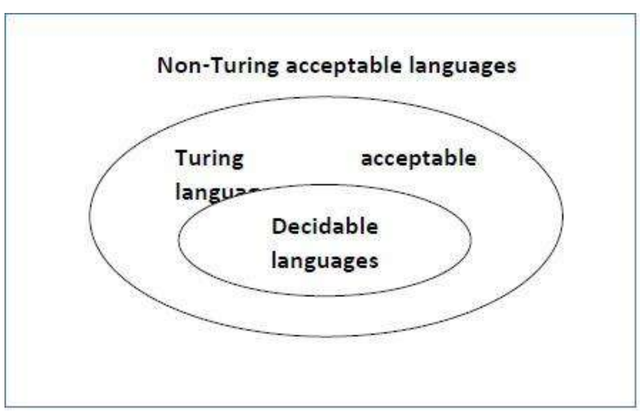
Examples – These are few important **Undecidable Problems**:

- 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:

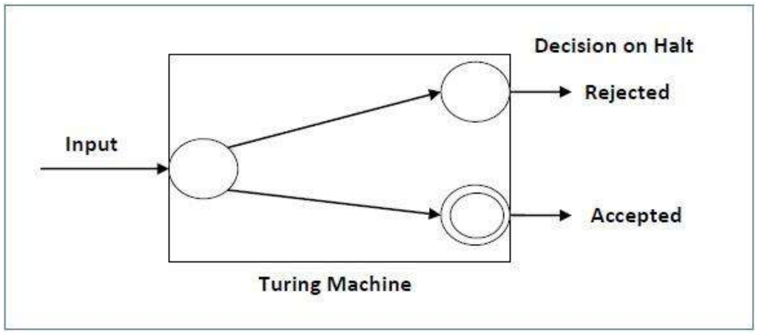
- **Membership** problem of a Turing Machine?
- **Finiteness** of a Turing Machine?
- **Emptiness** of a Turing Machine?
- Whether the language accepted by Turing Machine is regular or CFL?

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.

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 '√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'.

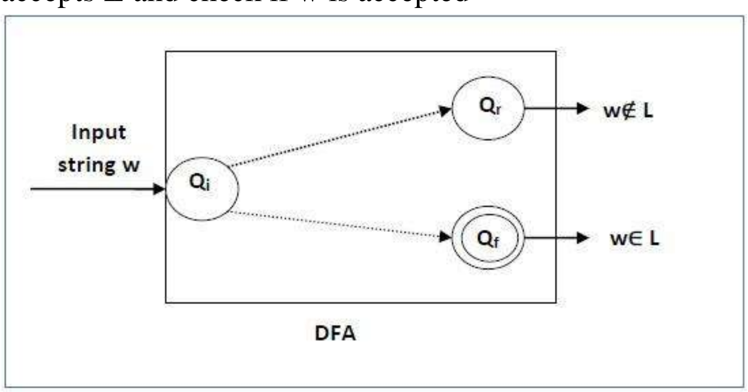
Hence, it is a decidable problem.

Example 2

Given a regular language **L** and string **w**, how can we check if **w ∈ 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.

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 ∑ is stated as follows −
Given the following two lists, **M** and **N** of non-empty strings over ∑ −
$M = (x_1, x_2, x_3, \ldots\ldots\ldots, x_n)$
$N = (y_1, y_2, y_3, \ldots\ldots\ldots, y_n)$
We can say that there is a Post Correspondence Solution, if for some $i_1, i_2, \ldots\ldots\ldots\ldots i_k$, where $1 \leq i_j \leq n$, the condition $x_{i1} \ldots\ldots x_{ik} = y_{i1} \ldots\ldots y_{ik}$ satisfies.

Example 1

Find whether the lists M = (abb, aa, aaa) and N = (bba, aaa, aa) have a Post Correspondence Solution?

Solution

	X1	X2	X3
M	abb	aa	aaa
N	bba	aaa	aa

Here,
$\mathbf{x_2x_1x_3}$ **= 'aaabbaaa'**

and $\mathbf{y_2y_1y_3}$ **= 'aaabbaaa'**

We can see that

$\mathbf{x_2x_1x_3 = y_2y_1y_3}$

Hence, the solution is **i = 2, j = 1, and k = 3.**

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.

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 = [k Time(n 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

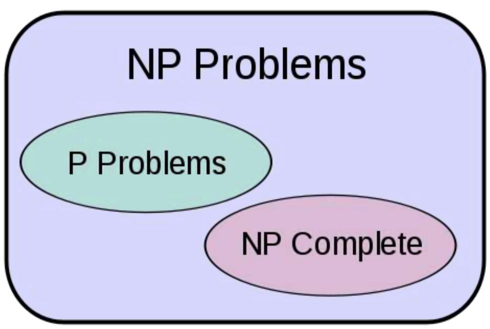
A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

Proof.

(⇒) 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 < w, c >.
- 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.

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 ≠ 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.