1.1 INTRODUCTION TO AUTOMATA THEORY

Automata theory is the study of abstract machines and the computational problems can be solved using these machines. Abstract machines are called automata. The name comes from the Greek word (Αυτόματα). It means doing something by itself. An automaton can be a finite representation of a formal language that may be an infinite set. Automata are used as theoretical models for computing machines, and are used for proofs about computability.

The automata theory is essential for,

- The study of the limits of computation
- Designing and checking the behaviour of digital circuits.
- Pattern searching in Websites
- Verifying systems of all types that have a finite number of distinct states, such as communications protocols or protocols for secure exchange information

Ex1: To design a machine that accepts all binary strings ends in 0 and reject all other that does not ends in 0.

11011010 – Accept

Ex2: To design a machine to accepts all valid 'C' codes

Machine will check the binary equivalent of this code and from this binary equivalent it tells weather it is valid piece of C code or invalid.

Question : Is it possible to design a machine?

Yes – The best example is Compiler.

1.1.1 Introduction To Formal Languages

Formal languages are the system used to train the machines in recognizing certain commands or instructions. These languages are the abstraction of natural languages, since they are expended by the machines. Formal languages are of five types. They are: r

- Regular Languages (RL)
- Context free Languages (CFL)
- Context Sensitive Languages (CSL)
- Recursive Languages r Recursively Enumerable Languages (RE)

✓ These languages are recognized by specific automata/machines and grammars.

- Regular grammars (type 3) and finite automata recognize regular languages.
- Context free grammars (Type 2) and push down automata recognize context free languages.
- Context sensitive grammars (Type 1) and Linear Bounded Automata (LBA) recognize context sensitive languages.
- Unrestricted grammars (phrase structure grammar) (Type 0).
- Turing machines recognize recursively enumerable languages.

✓ Total Turing Machines (TTM) that halt for every input are used to recognize recursive languages.

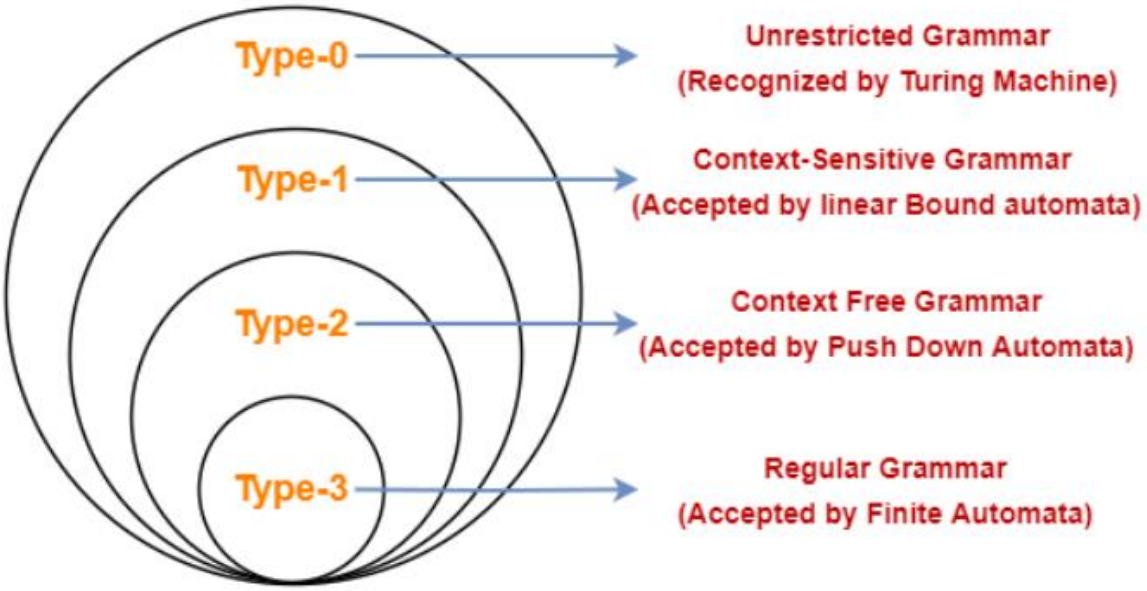
1. Formal Language Theory

Formal language theory describes languages as a set of operations over an alphabet. It is closely linked with automata theory, as automata are used to generate and recognize formal languages. Automata are used as models for computation; formal languages are the preferred mode of specification for any problem that must be computed.

2. Computability theory Computability

theory deals primarily with the question of the extent to which a problem is solvable on a computer. It is closely related to the branch of mathematical logic called recursion theory.

3. Models of Computation



LAYERS AND LEVELS IN THEORY OF COMPUTATION:

- FSM – Finite State Machine – Simplest model of Computation and it has very limited memory.

Perform low level computation and calculations

- CFL – Context Free Language

Performs some higher level of computation.

- Turing Machine – Much powerful model perform very high level computation designed by Alan Turing in 1940.
- Undecidable – Problem cannot be solved mechanically is falls under undecidable layer.

Basic Units of Regular Language:

Alphabets (∑) : { a, b} or {0,1}

String(w) : Collection of input alphabets

Language (L) : Collection of Strings Empty Set : Ø

NULL String : ε or λ

1. Find L with 0'S and 1's with odd no. of 1's ∑ ={ 0,1}
   w ={ 1,01,10,100,010, 111,1011……..}
   w ={ 1,01,10,100,010, 110 ,111,1011……..}
   L = {w/w consists of odd no. of 1's}

2. Find L with 0's and 1's with even no. of 1's
   ∑ ={ 0,1} w ={Λ ,11,011,101,110,0110, 1010, ……..}
   w = {Λ ,11,011,101,100, 110,0110, 1010, …..}
   L = {w/w consists of even no. of 1's}

Regular Expression:

• A Mathematical notation used to describe the regular language. • This is formed by using 3 Symbols:
(i). [dot operator] – for concatenation
(ii) + [Union operator] –at least 1 occurrence eg) 1+ = {1,11,111,-------}
(iii) {*} [Closure Operator] – Zero or more occurrences eg) 1* = {Λ ,1,11,111,…..}

1.6 Basic Regular Expressions:

- Ø is a RE and denotes the empty set.
- ε is a RE and denotes the set { ε }
- For each a in ∑, a is a RE and denotes the set {a}
- If r and s are RE that denoting the languages R and S respectively, then,
  - (r+s) is a RE that denotes the set (RUS)
  - (r.s) is a RE that denotes the set R.S
  - (r)* is a RE that denotes the set R*

1.7 Problems on RE:

1. **Write the RE for the language of even no. of 1's.**

   ∑={0,1}

   W={ʌ,11,011,101,110,1111,1100,....}

   RE=(11)*

2. **Write the RE for the language of odd no. of 1's.**

   ∑={0,1}

   W={1,10,01,100,111,1110,.....}

   RE=(11)*.1

9. **Write the RE for identifiers in 'C' Programming.**

   Letter=(a-z)

   Digit=(0-9)

   RE=(letter+_).(letter+_+digit)*

1.8 What is Finite Automata?

- Simplest model of a computing device.
- Finite automata are used to recognize patterns.
- A machine that accepts Regular Language.
- It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata can either move to the next state or stay in the same state.
- Finite automata have two states, Accept state or Reject state. When the input string is processed successfully, and the automata reached its final state, then it will accept.

Applications:

- Compilers
- Text processing
- Hardware design.

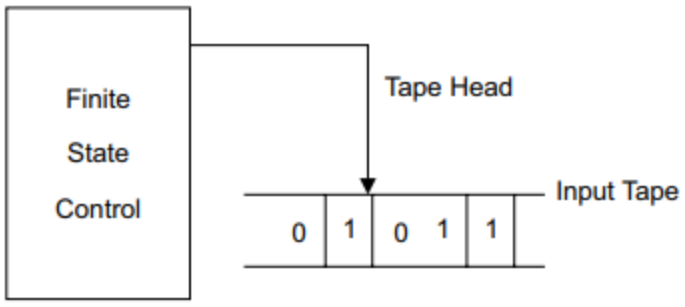


Fig. 1.1 The Working Model of a Finite Automata

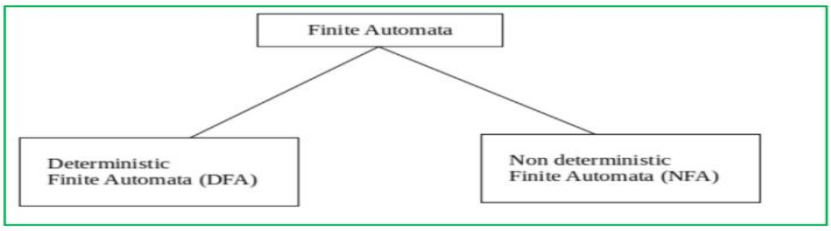


Figure 1.2 Types of Automata

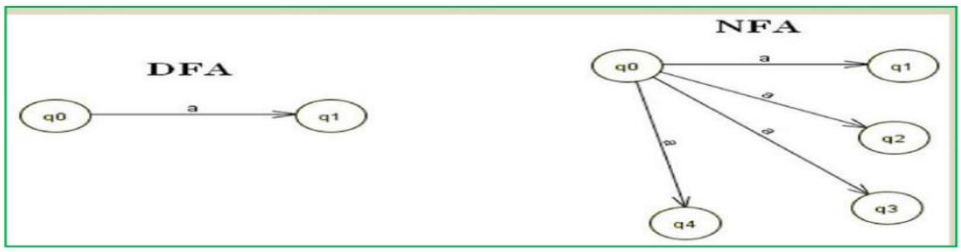


Figure 1.3 Difference between DFA and NFA

2. DFA (Deterministic Finite Automata):

- Only one path for specific input from the current state to the next state.
- DFA does not accept the null move.
- It is used in Lexical Analysis phase in Compilers.

  Example: RE=$(a+b)^+$

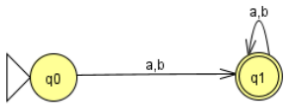


Figure 1.4 Sample DFA

Definition of DFA:

A finite automaton is a collection of 5-tuple (Q, ∑, δ, q0, F), where:

Q: finite set of states

∑: finite set of the input symbol

q0: initial state

F: final state

δ: Transition function

2.2 Construction of DFA:

1. Construct DFA to accept strings of a's and b's having a substring aa.

W={aa,aaa,baa,aab,aabb,abaa…..}

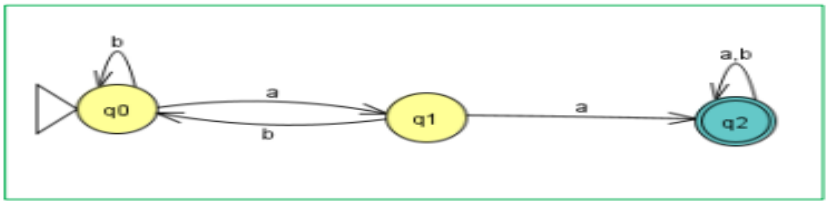


Figure 1.5 State Diagram

Figure 1.5 State Diagram

DFA Definition

M=(Q, ∑,q_0, δ,A)

Q={q0,q1,q2}

∑={a,b}

q_0 =q0

A = q2

δ –Transition Function

δ(q0,a)=q1

δ(q0,b)=q0

δ(q1,a)=q2

δ(q1,b)=q0

δ(q2,a)=q2

δ(q2,b)=q2

3.NON-DETERMINISTIC FINITE AUTOMATA:

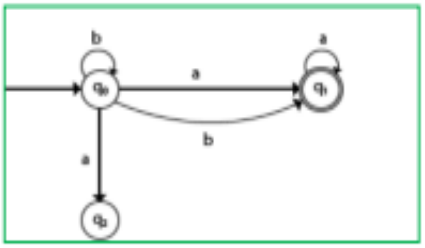


Figure 1.13 NFA

- When there exist many paths for specific input from the current state to the next state.
- Every NFA is not DFA, but each NFA can be translated into DFA.
- Types
  - (i) NFA without Λ
  - (ii) NFA with Λ

Advantages of NFA over DFA:

- DFAs are faster but more complex.
- Build a FA representing the language that is a union, intersection, concatenation etc. of two (or more) languages easily by using NFA's.

Definition:

NFA has 5 tuples M=(Q,∑,q0,A,δ), where

Q: finite set of states

Obtain an NFA to accept the language L={ w| w ε abab^n or aba^n}

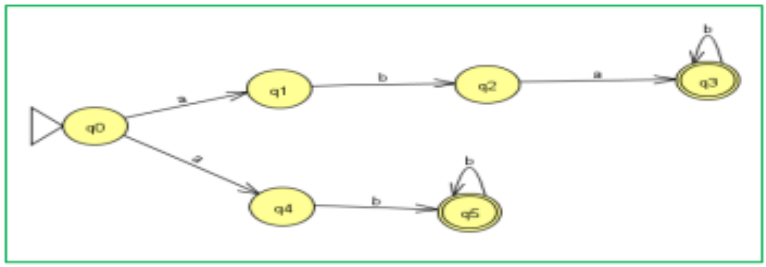


Figure 1.4 NFA State Diagram

Q={q0,q1,q2,q3,q4,q5}

∑={a,b}

q0={q0}

A={q3,q3}

δ:

δ(q0,a)={q1,q4}

δ(q0,b)= Ø

δ(q1,a)= Ø

δ(q1,b)={q2}

δ(q2,a)={q3}

δ(q2,b)= Ø

δ(q3,a)= Ø

δ(q3,b)={q3}

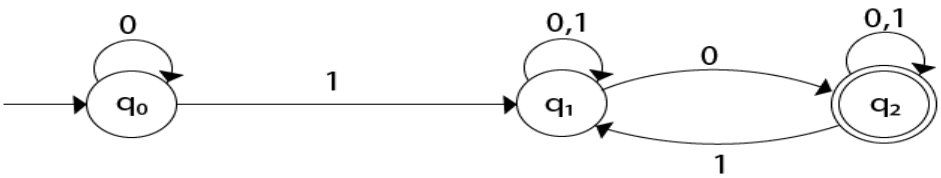
δ(q4,a)= Ø

δ(q4,b)={q5}

δ(q5,a)={q5}

δ(q5,b)= Ø

Convert the given NFA to DFA.

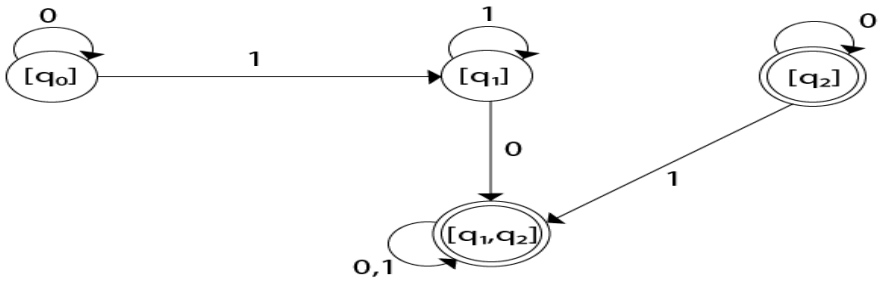


Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	q0	q1
q1	{q1, q2}	q1
*q2	q2	{q1, q2}

State	0	1
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]
*[q1, q2]	[q1, q2]	[q1, q2]

The Transition diagram will be:



The state q2 can be eliminated because q2 is an unreachable state.

Now we will obtain δ' transition for state q0.

δ'([q0], 0) = [q0]
δ'([q0], 1) = [q1]

The δ' transition for state q1 is obtained as:

δ'([q1], 0) = [q1, q2] (**new** state generated)
δ'([q1], 1) = [q1]

The δ' transition for state q2 is obtained as:

δ'([q2], 0) = [q2]
δ'([q2], 1) = [q1, q2]

Now we will obtain δ' transition on [q1, q2].

δ'([q1, q2], 0) = δ(q1, 0) ∪ δ(q2, 0)
= {q1, q2} ∪ {q2}
= [q1, q2]
δ'([q1, q2], 1) = δ(q1, 1) ∪ δ(q2, 1)
= {q1} ∪ {q1, q2}
= {q1, q2}
= [q1, q2]

The state [q1, q2] is the final state as well because it contains a final state q2. The transition table for the constructed DFA will be:

Myhill Nerode Theorem

Myhill Nerode theorem tells that the given language might be regular

Theorem :- A given language L is a regular language if the set of equivalence classes of L is finite

Proof: Let finite classes be $C_0, C_1, C_2 \dots C_n$ for some fixed value of n. We assume that this classes as the states of FA such that

a) The class containing ϵ (null) is initial state
b) For all $w_i \in L$ the class having w_i is the final state
c) A transition from the state C_i to C_j with label a is there if $x \in C_i$ & $xa \in C_j$

Q Prove that $L = a^n b^n$ is regular or not using Myhill Nerode Theorem.

Let L be a language over Σ

Two string are x and y in Σ^* are distinguishable with respect to L if there is a string $z \in \Sigma^*$ (which may depend on x and y) so that exactly one of the string xz and yz is in L

the string z is said to be distinguishable x and y with respect to L.

=) Either Both in L or both not in L

Eg ~~=)~~ $L = \{ x \in \{a,b\}^* \mid x \text{ ends with } ab \}$

Eg =) Prove that $L = \{a^n b^n \mid n > 0\}$ is non regular using My Nill Nerode Theorem.

Let us assume $S = \{a^n \mid n > 0\}$. This set S is over alphabet $\{a, b\}$ and it is infinite. we have to show its string are pairwise distinguishable with respect to Language L.

Assume that a^i and a^j are arbitary two different member of the set S, where i and j are positive integer and $i \neq j$

Let b^i as a string to be appended to a^i and a^j. It then becomes $a^i b^i$ and $a^j b^i$. From these string $a^i b^i$ is in L and $a^j b^i$ is not in L. Hence we can say that a^i and a^j are arbitary string of S and are distinguishable w.r.t. L. Thus satisfy the condition of My Nill Nerode theorem.

~~Context Free Grammar~~

Minimization of finite Automata

i) By Equivalence Theorem

ii) By Myhill Nerode Theorem (Table Filling Method)

i) Using State equivalence Method

→ Initially the state are divided into two groups i.e. final & non-final state.

→ For each group repeat the following steps until no more groups can be splitted.

→ transition on the input symbol is checked for every state

→ If the transition state falls into two different groups, then group is splitted.

Algorithm

S1. we will divides states Q into two different sets. One set consain all final states & other set contain all non-final state. this partition is called Π_0

S_2. Initially $K=1$

S_3 Find Π_K by partitioning the different set of Π_{K-1}

S_4 : Stop when $\boxed{\Pi_K = \Pi_{K-1}}$

S_5 All the states of 1 set are merged into 1 Number of state in minimized DFA will be

equal to number of set in Π_k.

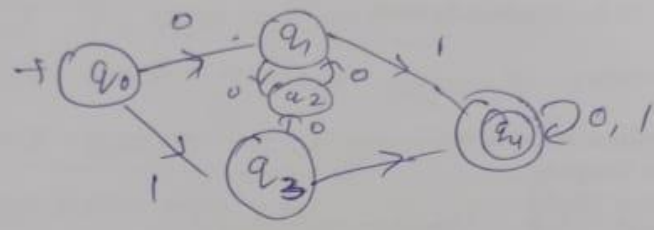
Q. Find the Π_0 state equivalence

Π_0 = [ES] [NFS]

= {q_4} {q_0, q_1, q_2, q_3}

	q_0	q_1	q_2	q_3
0	q_1			
1	q_2			

(1) Minimize the following DFA



By ~~Nykill/Myhill Theorem~~ State / Equivalence

Sol: Π_0 = [FS] [NFS]

{q_4} . {q_0 q_1 q_2 q_3}

1 2

.	0	1
q_0	q_1	q_3
q_1	q_2	q_4
q_2	q_1	q_4
q_3	q_2	q_4

2	0	1
q_0	2	2
q_1	2	1
q_2	2	1
q_3	2	1

$S_2 = \Pi_1 =$ {q_4} {q_0} {q_1, q_2, q_3}

1 3 4

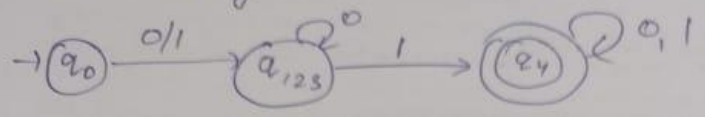
	q_1	q_2	q_3
0	q_2	q_1	q_2
1	q_4	q_4	q_4

	q_1	q_2	q_3
0	4	4	4
1	1	1	1

$\pi_0 = \pi_1$

So $q_1 \equiv q_2 \equiv q_3$

Transition diagram:-



Transition Table

	0	1
→ q_0	q_{123}	q_{123}
→ q_{123}	q_{123}	q_4
* q_4	q_4	q_4

Using Table Filling / Myhill Nerode Theorem

S_1. Construct a table for all pair of states (P, Q) Initially all are unmarked.

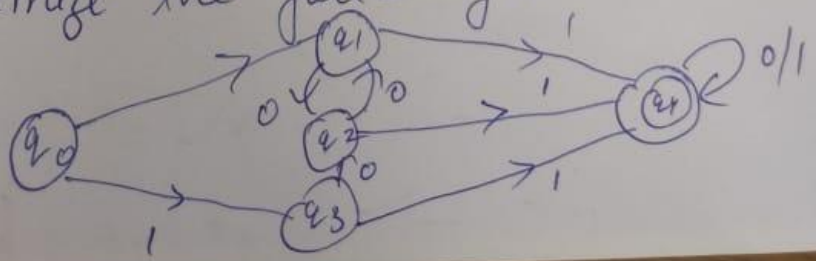
S_2. Consider every state pair (P, Q) in the DFA. where P is final & Q is final and vice versa

S_3: Then mark the pair (P, Q)

S_4: Repeat this step until no more mark can be made.

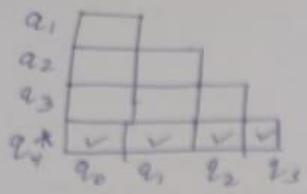
S_5: Combine all the unmarked pair and make them as a single state in minimized DFA.

Q. Minimize the following DFA



Using Myhill Nerode Theorem

S_1



S_2 check (q_0, q_1)

	0	1
→q_0	q_1	q_3 → marked
q_1	q_2	q_4

(q_0, q_1) → distinguishable

check (q_0, q_2)

	0	1
q_0	q_1	q_3 → marked
q_1	q_1	q_4

(q_0, q_2) → distinguishable

check (q_1, q_2)

	0	1
q_1	q_2	q_4
q_2	q_1	q_4

equivalent state

check (q_1, q_3)

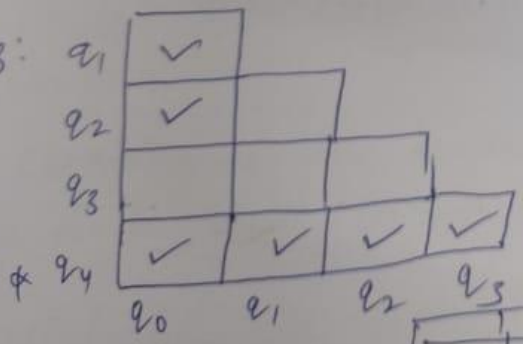
	0	1
q_1	q_2	q_4
q_3	q_2	q_4

q_1 q_3 are equivalent

check (q_2, q_3)

Similarly (q_2, q_3) are equivalent

S_3:

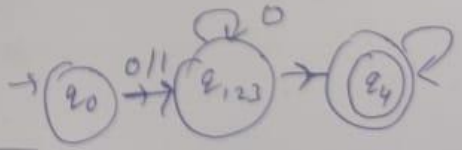


	0	1
q_0	q_{123}	q_{123}
q_{123}	q_{123}	q_4
q_4	q_4	q_4

S_4: unmarked pairs

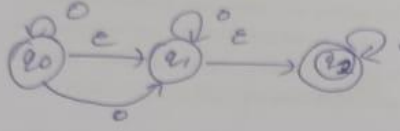
$q_1 \equiv q_2$, $q_1 \equiv q_3$

$q_2 \equiv q_3$



Finite automata with ε Moves

If a finite automata is malified to permit transition without input signals along with zero one or more transition transition on input symbols then we get a NFA with ε-transition.

Eg 

Conversion of ε-NFA to NFA

1. Take the start state of ε-NFA as start state of NFA. If ε-NFA accept ε, then mark the start state as final state.
2. Take the final state of ε-NFA as final state of NFA.
   $F' = F \cup \{q\}$ if ε-closure of q contain a member of F
3. Perform $S_N(S, a) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q, a))$
   where S_N is the transition f'n of resulting NFA, δ is the transition f'n of ε-NFA

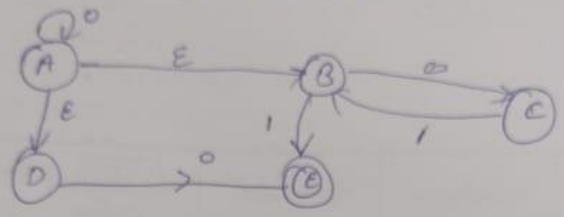
To find closure

To find closure, ε-closure (q), say P we do following step

add q to P

Find all the sets $\delta(q, \varepsilon)$ for each element $q \in P$ and add to P all the elements of these sets that are not included. In P, stop when this step does not changed to P

Q. Convert the following ε-NFA into equivalent NFA without ε-transition



The equivalent NFA is given by

Start state A,

Now $S_N(A,0) = \varepsilon\text{-Closure}\{\delta(\varepsilon\text{-Closure}(A),0)\}$

$= \varepsilon\text{-Closure}(\delta(A,B,D),0)$

$= \varepsilon\text{-Closure}(A,C,E)$

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

$S_N(A,1) = \varepsilon\text{-Closure}(\delta(\varepsilon\text{-Closure}\{A\},1)$

$\varepsilon\text{-Closure}\{\delta(A,B,C,D),1)$

$\varepsilon\text{-Closure}(\{E\})$

$S_N(B,0) = \varepsilon\text{-Closure}\{\delta(\varepsilon\text{-Closure}(B),0)$

$\varepsilon\text{-Closure}(\delta(\{B\},0)$

$\varepsilon\text{-Closure}(\{C\})$

$\{C\}$

$\delta_N(B,0)$ = ε-closure (δ (ε-closure (B); 0)
ε-closure (δ ({B}, 0)
ε-closure ({C})
= {C}

$\delta_N(B,1)$ = ε-closure (δ (ε-closure (B), 1)
ε-closure (δ ({B}, 1))
= {E}

$\delta_N(C,1)$ = ε-closure (δ (ε-closure (C), 1)
ε-closure (δ ({C}, 1))
ε-closure ({B})
{B}

$\delta_N(C,0)$ ε-closure (δ (ε-closure (C), 0))
= ε-closure (δ ({C}, 0)
= ε-closure ({∅}) = ∅

$\delta_N(D,0)$ = ε-closure (δ (ε-closure (D), 0))
ε-closure ({E})

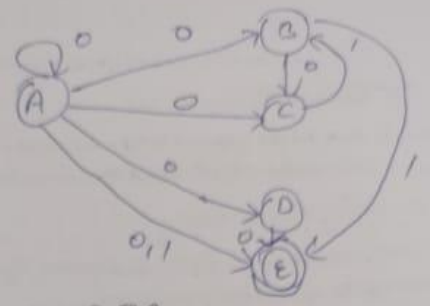
$\delta_N(D,1)$ = ε-closure (δ (ε-closure (D), 1))
= ε-closure (δ ({D}, 1))
= ε-closure ({∅})

$\delta_N(E,0)$ = ε-closure (δ (ε-closure (E), 0))
= ε-closure ({δ{E}, ∅)
= ε-closure ({∅})
= ∅

$\delta_N(E,1)$ = ε-closure (δ (ε-closure (E), 1))
ε-closure ({δ{E}, 1)
= ε-closure ({∅})
= ∅

Thus the resulting NFA is

Σ / δ	0	1
A	{A, B, C, D, E}	{E}
B	{C}	{E}
C	{D}	{E}
D	{E}	ϕ
E	ϕ	ϕ



Conversion of ε-NFA to DFA

Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be an ε-NFA then the equivalent DFA $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ is defined as follows.

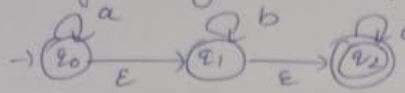
1. Q_D = set of subset of Q_E
2. q_D = ε-closure (q_0)
3. F_D = set of those state that contains atleast one accepting state of E.
   i.e. $F_D = \{S / S$ is in Q_D and $S \cap F \neq \phi\}$
4. For all $a \in \Sigma$ and set $S \in Q_D$, $\delta_D(S, a)$ is computed as,

(a) let $S = \{P_1, P_2 \dots P_k\}$

(b) compute $\bigcup_{i=1}^{k} \delta_E(P_i, a)$ Let this set be $\{r_1, r_2 \dots r_m\}$

(c) Then $\delta_D(S, a) = \bigcup_{s=1}^{m} \varepsilon\text{-closure}(r_s)$

Convert following ε-NFA into DFA



Sol Start state of DFA = ε-Closure (start state of ε-NFA)

$= \varepsilon\text{-Closure}(q_0)$

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

$\delta_D(\{q_0, q_1, q_2\}, a) = \varepsilon\text{-Closure}(\delta_\varepsilon(\{q_0, q_1, q_2\}, a))$

$= \varepsilon\text{-Closure}(\{q_0\})$

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

$\delta_D(\{q_0, q_1, q_2\}, b) = \varepsilon\text{-Closure}(\delta_\varepsilon(\{q_0, q_1, q_2\}, b))$

$= \varepsilon\text{-Closure}(q_1)$

$= \{q_1, q_2\}$

$\delta_D(\{q_0, q_1, q_2\}, c) = \varepsilon\text{-Closure}(\delta_\varepsilon(q_0, q_1, q_2\}, c))$

$\varepsilon\text{-Closure}(q_2)$

$\{q_2\}$

Similarly $\delta_D(\{q_1, q_2\}, a) = \varepsilon\text{-Closure}(\delta_\varepsilon(\{q_1, q_2\}, a))$

$= \varepsilon\text{-Closure}(\emptyset)$

$= \{\emptyset\}$

$\delta_D(\{q_1, q_2\}, b) = \varepsilon\text{-Closure}(\delta_\varepsilon(\{q_1, q_2\}, b))$

$\varepsilon\text{-Closure}(\{q_1\})$

$= \{q_1, q_2\}$

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

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

$= \{q_2\}$

Similarly $\delta_D(q_2, a) = \varepsilon\text{-closure}(\delta(\{q_2\}, a))$

$= \varepsilon\text{-closure}(\{\phi\})$

$= \phi$

$\delta_D(q_2, b) = \varepsilon\text{-closure}(\delta(\{q_2\}, b))$

$= \varepsilon\text{-closure}(\{\phi\})$

$= \phi$

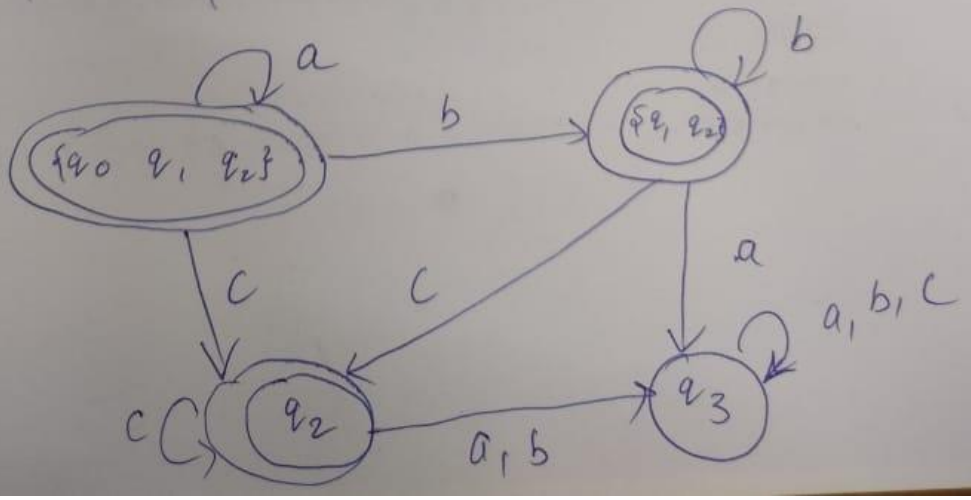
$\delta_D(q_2, c) = \varepsilon\text{-closure}((\delta\{q_2\}, c))$

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

$\{q_2\}$

Transition Table

δ_D / ε	a	b	c
$\rightarrow\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
$*\{q_1, q_2\}$	$\{q_3\}$	$\{q_1, q_2\}$	$\{q_2\}$
$*\{q_2\}$	$\{q_3\}$	$\{q_3\}$	$\{q_2\}$
$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$



Regular expression, Regular language & loops

* Regular expression are algebraic expression used for representing regular language i.e. language accepted by Finite Automata

1. A terminal symbol (i.e. $x \in \Sigma$) is regular expression
2. The union of two different R.E is also regular
3. Concatenation " "
4. The closure of R.E "
5.

Designing Regular Expression

1. zero or more a's → a^*
2. One or more a's → a^+
3. zero or one a → $(a+\varepsilon)$
4. All string of $\Sigma = (a, b, c)$ ⇒ $(a+b+c)^*$

Q Find the regular expression representing the sets of all string of the form

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

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

(c) $a^n b\, a^{2m} b^2$ where $m \geq 0$ $n \geq 1$

(a) $a\, a^*\, b\, b^*\, c\, c^*$

(b) $a\, a^*\, b\, b\, (bb)^*\, ccc\, (ccc)^*$

(c) $a\, a^*\, b\, (aa)^*\, bb$

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

Q a) All strings that contain exactly one a

b) All string beg with ab

c) All string that contains either the substring 'aaa' or 'bbb'

Sol a) $b^* a\, b^*$

b) $ab(a+b)^*$

c) $(a+b)^*(aaa + bbb)(a+b)^*$

Simplification of Regular expression

1. $\phi + R = R$
2. $\phi R = R\phi = \phi$
3. $\varepsilon R = R\varepsilon = R$
4. $\varepsilon^* = \varepsilon$ and $\phi^* = \varepsilon$
5. $R + R = R$
6. $(R^*)^* = R^*$
7. $(P\phi^*)P = P(\phi P)^*$
8. $(P+Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$
9. $(P+\phi)R = PR + \phi R$
10. $R + RR^* = R^* = \varepsilon + R^*R$

Properties of Regulare Expression

1. Union
2. Concatenation
3. Kleene Closure
4. Intersection
5. Complement
6. Transpose

Converting R.E to automata

Case I for union

* Converting Regular Expression to automata :- (Thompson's Construction)

Theorem :- Every Language defined by regular expression is also defined by finite automata. ∴ for any language R regular expression 'r' there is an ε-NFA that accept the same language represented by r.

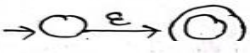
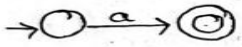
Proof : Let L = L(r) be the language to regular expression 'r'. Now we have to show that there is an ε-NFA E such that L(E) = L.

This proof can be done through structural induction on r.

Basic step :-

⇒ Φ, ε, a are regular expression for representing language {Φ}, {ε}, {a} respectively.

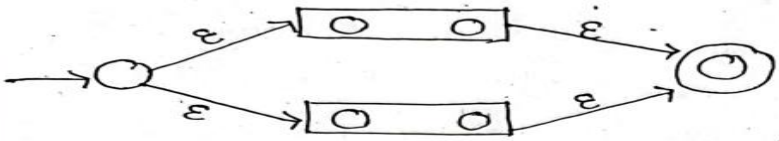
⇒ The ε-NFA accepting these language can be constructed as,

1.  for r = Φ
2.  for r = ε
3.  for r = a

Case 1 :- for union '+'

from basic step we can construct ε-NFA for r_1 and r_2. let ε-NFA be M_1 and M_2 respectively.

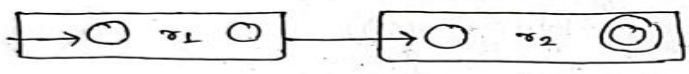
hen $r = r_1 + r_2$ can be constructed as,



The language of this automaton is $L(r_1) \cup L(r_2)$ which is also the anguage represented by regular expression $r_1 + r_2$.

ase 2 : **for concatenation :-**

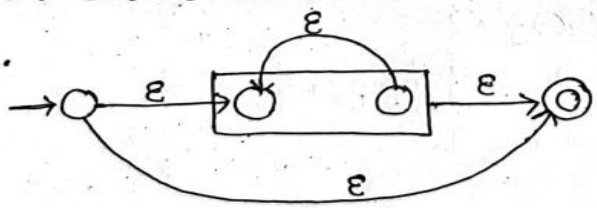
Now, $r = r_1 \cdot r_2$ can be constructed as



Here the path from starting to accepting state go first through the automation for r_1 where it must follow a path labelled by string in $L(r_1)$ and then

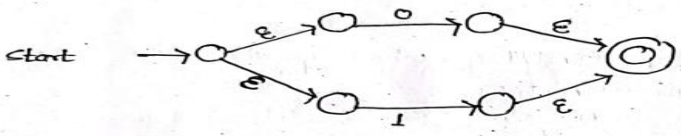
Case 3: for Kleene closure :- (*)

Now $r = r^*$ can be constructed as

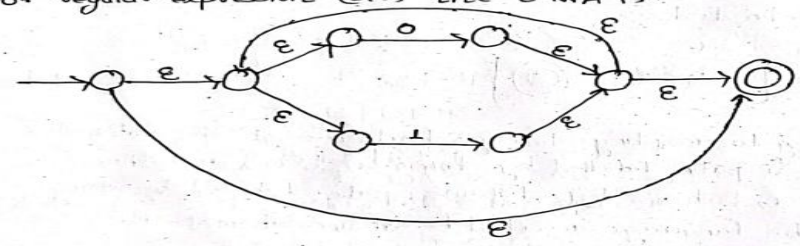


Examples

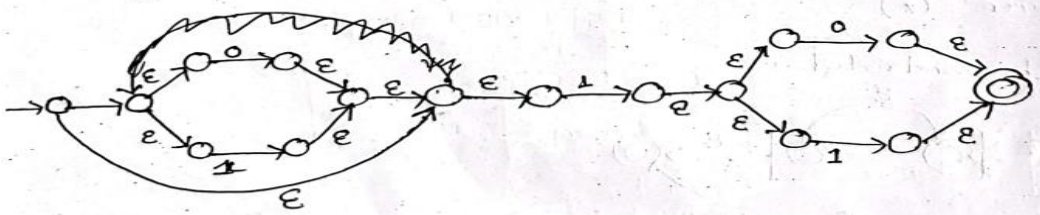
1. for regular expression (1+0) the ε-NFA is



2. for regular expression $(1+0)^*$ the ε-NFA is



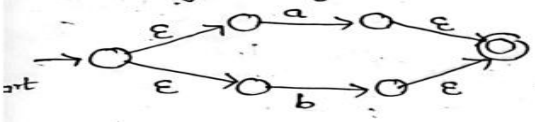
3. for regular expression $(1+0)^*1(1+0)$



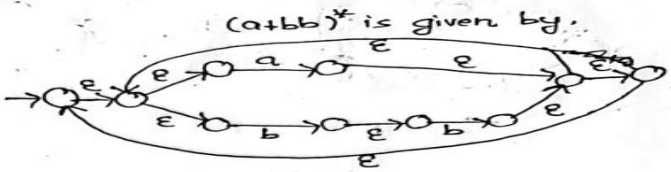
onstruct a NFA for language $L = (a+b)^* b(a+bb)^*$

:olution Here $L = (a+b)^* b (a+bb)^*$

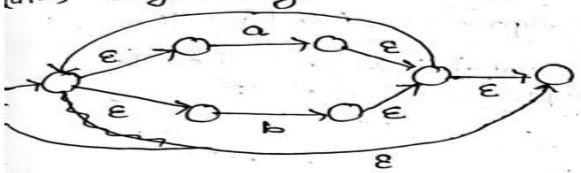
$(a+b)$ is given by.



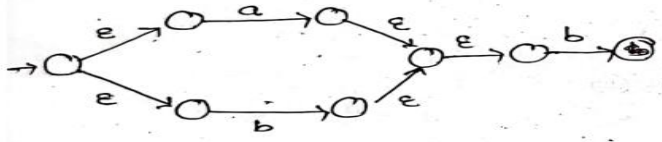
$(a+bb)^*$ is given by.



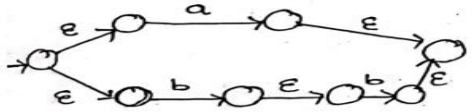
$(a+b)^*$ is given by.



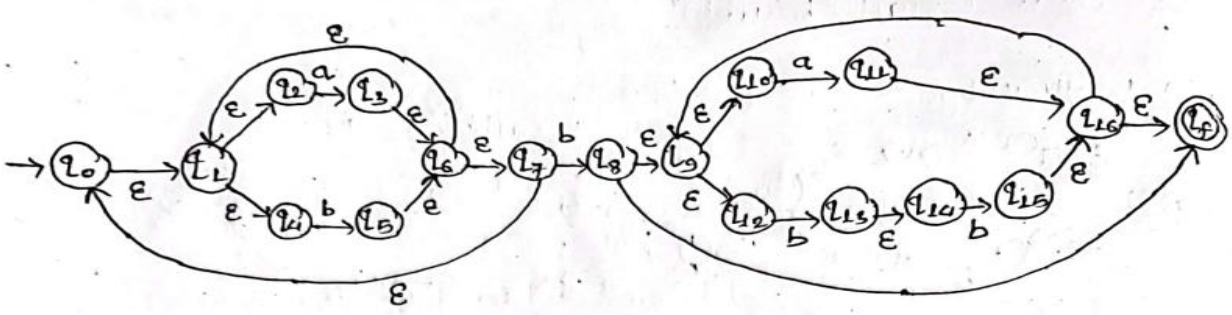
$(a+b)^*b$ is given by.



$(a+bb)$ is given by



Now final NFA of $(a+b)^*b(a+bb)^*$



*Conversion from finite automata to regular expression:- (Ardon's Theorem)

Let P and Q be the regular expression over alphabet Σ, if P doesnot contain empty string then $r = q+rp$ has a unique solution $r = qP^*$

Proof:

Here, $r = q+rp$ — (1)

Let us put value of $r = q+rp$ in eqn (1)

$$r = q+(q+rp)p$$

$$r = q+qp+rp^2 \quad -(2)$$

Again, put the value of r in eqn (2)

$$r = q+qp+(q+rp)p^2$$

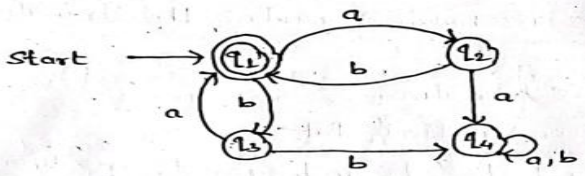
$$r = q+qp+qp^2+rp^3 \quad -(3)$$

Similarly, $r = q+qp+qp^2+qp^3+\ldots$

$$r = q(\varepsilon+p+p^2+p^3+\ldots)$$

$r = qP^*$ Hence proved.

Example find the regular expression for following DFA.



Now, lets from the equation for each state

$$q_1 = q_2 b + q_3 a + \varepsilon \quad - (1)$$

$$q_2 = q_1 a \quad - (2)$$

$$q_3 = q_1 b \quad - (3)$$

$$q_4 = q_2 a + q_3 b + q_4 a + q_4 b \quad - (4)$$

Now, putting the value of q_2 and q_3 in eqn (1)

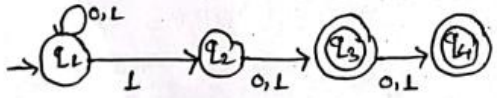
$$q_1 = q_1 ab + q_3 a + \varepsilon$$

$$q_1 = q_1 ab + q_1 ba + \varepsilon$$

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

$$= \varepsilon (ab + ba)^*$$

Example 2 Given the following NFA, configure the equivalent Regular Expression.



- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where $\delta: Q \times \sum \rightarrow Q$
- **X** is the output transition function where $X: Q \times \sum \rightarrow O$
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).

Finite automata may have outputs corresponding to each transition. There are two types of finite state machines that generate output −

- Mealy Machine
- Moore machine

Mealy Machine

A Mealy Machine is an FSM whose output depends on the present state as well as the present input.

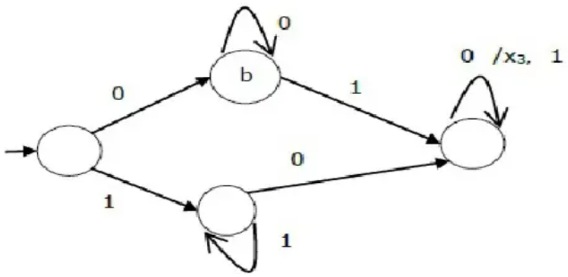
It can be described by a 6 tuple $(Q, \sum, O, \delta, X, q_0)$ where −

- **Q** is a finite set of states.
- $\sum$ is a finite set of symbols called the input alphabet.

The state table of a Mealy Machine is shown below −

	Next state			
Present state	**input = 0**		**input = 1**	
	State	**Output**	**State**	**Output**
→ a	b	x_1	c	x_1
b	b	x_2	d	x_3
c	d	x_3	c	x_1
d	d	x_3	d	x_2

The state diagram of the above Mealy Machine is −



Moore Machine

Moore machine is an FSM whose outputs depend on only the present state.

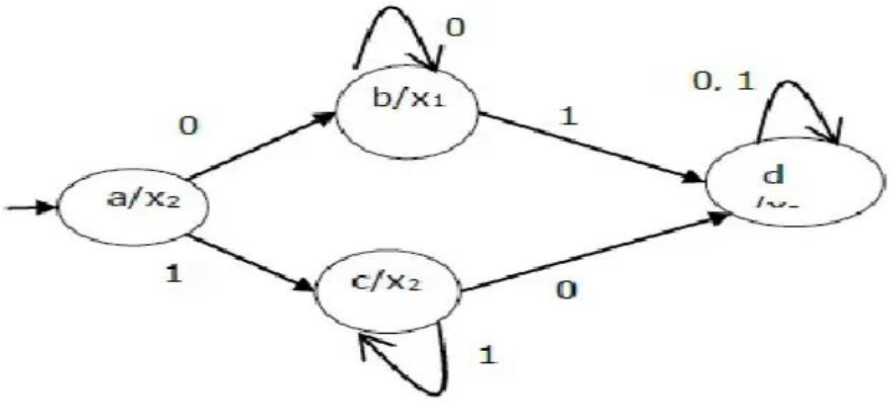
A Moore machine can be described by a 6 tuple $(Q, \sum, O, \delta, X, q_0)$ where −

- **Q** is a finite set of states.
- $\sum$ is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where $\delta: Q \times \sum \rightarrow Q$
- **X** is the output transition function where $X: Q \rightarrow O$

The state table of a Moore Machine is shown below −

Present state	Next State		Output
	Input = 0	Input = 1	
→ a	b	c	x_2
b	b	d	x_1
c	c	d	x_2
d	d	d	x_3

The state diagram of the above Moore Machine is −



Mealy Machine vs. Moore Machine

The following table highlights the points that differentiate a Mealy Machine from a Moore Machine.

Mealy Machine	Moore Machine
Output depends both upon present state and present input.	Output depends only upon the present state.
Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
Output changes at the clock edges.	Input change can cause change in output change as soon as logic is done.
Mealy machines react faster to inputs	In Moore machines, more logic is needed to decode the outputs since it has more circuit delays.

Moore Machine to Mealy Machine

Input − Moore Machine

Output − Mealy Machine

Step 1 − Take a blank Mealy Machine transition table format.

Step 2 − Copy all the Moore Machine transition states into this table format.

Step 3 − Check the present states and their corresponding outputs in the Moore Machine state table; if for a state Q_i output is m, copy it into the output columns of the Mealy Machine state table wherever Q_i appears in the next state.

Let us consider the following Moore machine −

Present State	Next State a = 0	Next State a = 1	Output
→ a	d	b	1
b	a	d	0
c	c	c	0
d	b	a	1

Now we apply Algorithm 4 to convert it to Mealy Machine.

Step 1 & 2 −

Present State	Next State a = 0		Next State a = 1	
	State	Output	State	Output
→ a	d		b	
b	a		d	
c	c		c	
d	b		a	

Step 3 −

Present State	Next State a = 0		Next State a = 1	
	State	Output	State	Output
=> a	d	1	b	0
b	a	1	d	1
c	c	0	c	0
d	b	0	a	1

Mealy Machine to Moore Machine

Algorithm 5

Input − Mealy Machine

Output − Moore Machine

Step 1 − Calculate the number of different outputs for each state (Q_i) that are available in the state table of the

Mealy machine.

Step 2 – If all the outputs of Qi are same, copy state Q_i. If it has n distinct outputs, break Q_i into n states as Q_{in} where **n** = 0, 1, 2.....

Step 3 – If the output of the initial state is 1, insert a new initial state at the beginning which gives 0 output.

Example

Let us consider the following Mealy Machine –

Present State	Next State			
	a = 0		a = 1	
	Next State	Output	Next State	Output
→ a	d	0	b	1
b	a	1	d	0
c	c	1	c	0
d	b	0	a	1

Here, states 'a' and 'd' give only 1 and 0 outputs respectively, so we retain states 'a' and 'd'. But states 'b' and 'c' produce different outputs $1 and 0$. So, we divide **b** into $\mathbf{b_0}$, $\mathbf{b_1}$ and **c** into $\mathbf{c_0}$, $\mathbf{c_1}$.

Present State	Next State		Output
	a = 0	a = 1	
→ a	d	b_1	1
b_0	a	d	0
b_1	a	d	1
c_0	c_1	C_0	0
c_1	c_1	C_0	1
d	b_0	a	0

Closure Properties of Regular Languages

Let L and M be regular languages. Then the following languages are all regular:

- *Union:* $L \cup M$
- *Intersection:* $L \cap M$
- *Complement:* $\overline{N}$
- *Difference:* $L \setminus M$
- *Reversal:* $L^R = \{w^R : w \in L\}$
- *Closure:* L^*.
- *Concatenation:* $L.M$
- *Homomorphism:*
$h(L) = \{h(w) : w \in L, h$ is a homom. $\}$
- *Inverse homomorphism:*
$h^{-1}(L) = \{w \in \Sigma : h(w) \in L, h : \Sigma \to \Delta^*$ is a homom. $\}$

$h(a_1\, a_2 \ldots a_n) = h(a_1)h(a_2)\ldots h(a_n)$

Pumping lemma

It is used to check whether the given string is accepted by regular set or not.

Theorem: Let L be a regular language set then there is a constant 'n' such that if z is any word or any string in L and $|z| \geq n$ then we can write $z = uvw$ such that

i) $|uv| \leq n$

ii) $|v| \geq 1$

iii) for all $i \geq 0$, $uv^i w \in L$

Q Show that the given language is regular or not.

$L = \{ a^n b^n / n \geq 1 \}$

S1: Assume L is a regular language

S2: Identify the language

$L = \{ ab, a^2b^2, a^3b^3 \ldots \}$

S3: Take any one above string as z

$z = a^2 b^2$

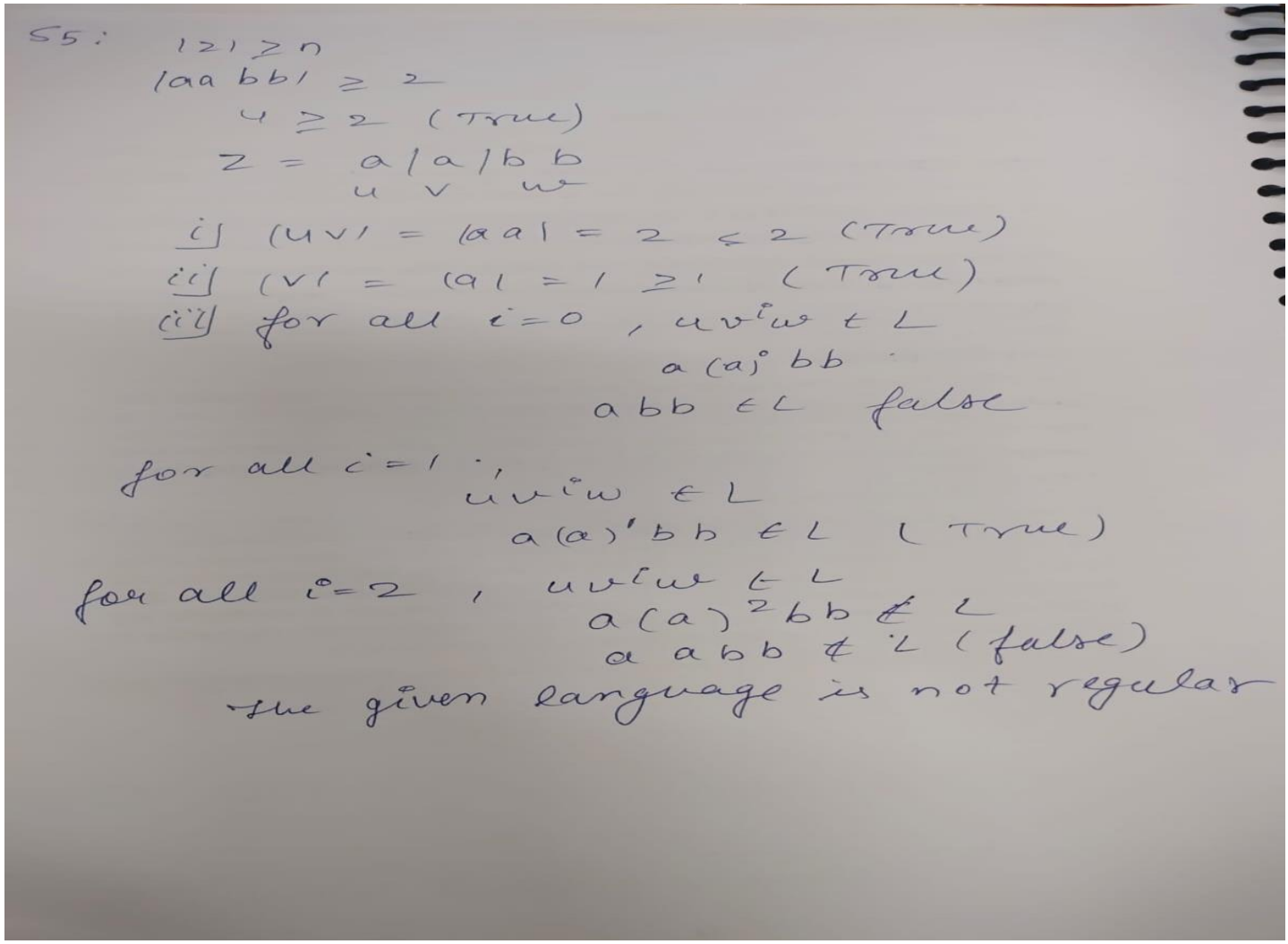
S4: Write Pumping lemma.

if $|z| \geq n$, then $z = uvw$,

To prove i) $|uv| \leq n$

ii) $|v| \geq 1$

iii) for all $i \geq 0$ then $z = uv^i w \in L$



CONTEXT FREE LANGUAGES AND NORMAL FORMS

1. Context Free Grammar

Definition – A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple

G= (N, T, P, S)

Where,

A ... Z

- **N** is a set of non-terminal symbols (N is also represented as V- the set of variables).
- **T** is a set of terminals where **N ∩ T = NULL.** a, b, c
- **P** is a set of rules, **P: N → (N ∪ T)***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
- **S** is the start symbol.

Terminals:
- Defines the basic symbols from which a string in a language are composed.
- Represented in lower case letters.

Non Terminals :
- They are special symbols that are described recursively in terms of each other and terminals. Represented in upper case letters.

Production rules:
- It Defines the way in which NTs may be built from one another and from terminal.

Start Symbol:
- It is a special NT from which all the other strings are derived. It is always present in the first rule on the LHS.

1.1 Problems:

1. **Given a CFG, find the language generated by G**

(i) G=(N,T,P,S) where N={S}, T={a,b}
P={S→aSb, S→ab}

S=>ab | S=>aSb =>aaSbb =>aaabbb =>aaaabbbb | S=>aSb =>aabb | S=>aSb =>aaSbb =>aaaSbbb

W= { ab, aabb,aaabbb,aaaabbbb,……..}

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

(ii) G=(P,{Ɛ,0,1},P,P}
P: P→0 | 1 | Ɛ |0P0 |1P1
P=>0
P=>1
P=>Ɛ
P=>0P0 | P=>0P0 | P=>0P0 | P =>0P0

=>010 =>00 =>000 =>00P00 =>00100

P =>1P1 =>10P01 =>10101 | P =>0P0 =>00P00 =>001P100 =>0010100 |

W={Λ, 0,1,00,010,000, 00100, 0010100, 10101,….}

L(G)={language of palindromes}

1.2 CFG Corresponding To A Language

1.2.3 For the given L(G), design a CFG.

i.Language consisting of even number of 1's

T={1,Ɛ}

W={Ɛ,11,1111,111111,....}

P:
S -> Ɛ
S -> 1S1

G=({S},{1,Ɛ},P,S)

ii. Design a CFG for a language consisting of arithmetic expression.

T={id,+,-,*,/,(,)}

W={id,id+id,id-id,id*id,id/id, id+id*id,(id-id)/id,……}

P: S -> id
S -> S+S
S -> S-S
S ->S*S
S ->S/S
(or)

S->id |S+S |S-S|S*S |S/S |(S)

G=({S},{+,-,*,/,id},P,S)

iv. L=$\{a^n b^m \mid m > n \text{ and } n \geq 0\}$

T={a,b}
W={b,bb,bbb,…,abb,abbb,…,aabbb,aaabbbb,…}

n=0, m=1=> b
n=0, m=2=>b, bb, bbb,bbbb,….
n=1, m=2 =>abb,abbb,abbbb,…
n=2, m=3=>aabbb,aaabbbb,….
P:
B->b |bB
S->aSb |B
G=({S,B}, {a,b} , P,S)

vii. Construct the CFG for the language having any number of a's followed by any number of b's over the set ∑= {a}

W={ Ɛ ,aaaa,bbbb,aabb,abbb,…..}
a*.b*
x=aabb
P:
S->A.B
A->aA|Ɛ
B->bB| Ɛ

G=({S,A,B},{a,b, Ɛ },P,S)

Regular Grammar
A Grammar G=(N,T,P,S) is regular if every production takes one of the following forms:

B→aC
B→a

Where B & C are NT and 'a' is a T

1.3 Regular Expression to CFG

i. Find the CFG equivalent to a Regular Expression

RE=ab. (a+bb)*

Generate the production for the language L1={ab}

A->ab

Generate the production for the language L2=a+bb

B->a | bb

Generate the production for the language L2*=(a+bb)*

C->Ɛ |BC

RE=ab. (a+bb)*

P: S->A.C

C->Ɛ |BC

A->ab

B->a | bb

G=({S,A,B,C}, {a,b, Ɛ },P, }

Leftmost derivation & Rightmost Derivation

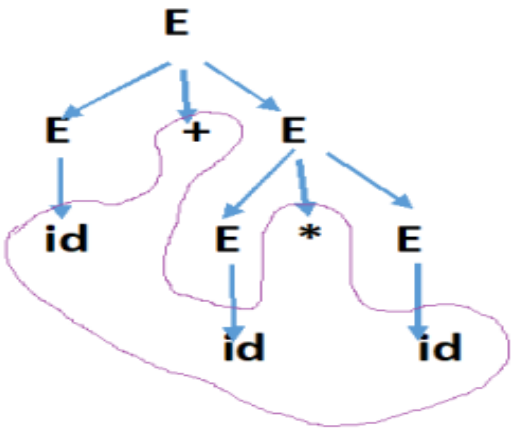
2.3 Leftmost derivation

In the derivation process, if the leftmost variable is replaced at every step then the derivation is leftmost derivation.

E->E+E | E*E | (E) |id

String:id+id*id

$E \overset{lmd}{\Longrightarrow}$ E+E	E->E+E
$\overset{lmd}{\Longrightarrow}$ id+E	E->id
$\overset{lmd}{\Longrightarrow}$ id+E*E	E->E*E
$\overset{lmd}{\Longrightarrow}$ id+id*E	E->id
$\overset{lmd}{\Longrightarrow}$ id+id*id	E->id



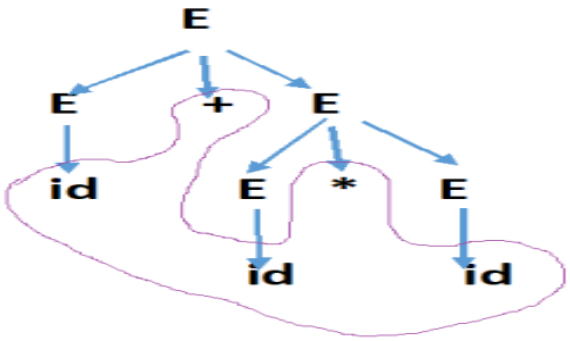
2.4 Rightmost Derivation

In the derivation process, if the rightmost variable is replaced at every step then the derivation is rightmost derivation.

E->E+E | E*E | (E) |id

String:id+id*id

$E \overset{rmd}{\Longrightarrow} E+E$	E->E+E
$\overset{rmd}{\Longrightarrow} E+E*E$	E->E*E
$\overset{rmd}{\Longrightarrow} E+E*id$	E->id
$\overset{rmd}{\Longrightarrow} E+id*id$	E->id
$\overset{rmd}{\Longrightarrow} id+id*id$	E=>id



2.5 Problems:

1. **For the Grammar G defined by**
   S->AB
   B->a|Sb
   A->Aa|bB
   Give the derivation trees for the following sentential forms

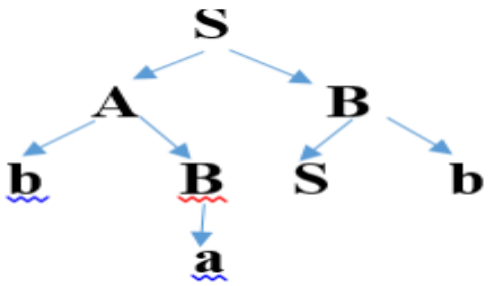
(i) baSb

S=>AB | S->AB

=>bBB | A->bB

=>baB | B->a

=>baSb | B->Sb



3. Ambiguity

Definition: An Ambiguous CFG
A CFG G is ambiguous if there is atleat one string in L(G) having two or more distinct derivation trees(or equivalently two or more distinct LMD).

i.Is the following grammar ambiguous:

E->E+E | E*E | (E) |id

Consider the String: id+id*id

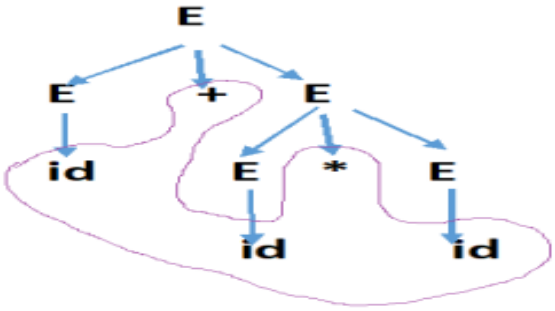
$E \overset{lmd}{\Longrightarrow} E+E$ | E->E+E

$\overset{lmd}{\Longrightarrow} id+E$ |E->id

$\overset{lmd}{\Longrightarrow} id+E*E$ |E->E*E

$\overset{lmd}{\Longrightarrow} id+id*E$ |E->id

$\overset{lmd}{\Longrightarrow} id+id*id$ |E->id



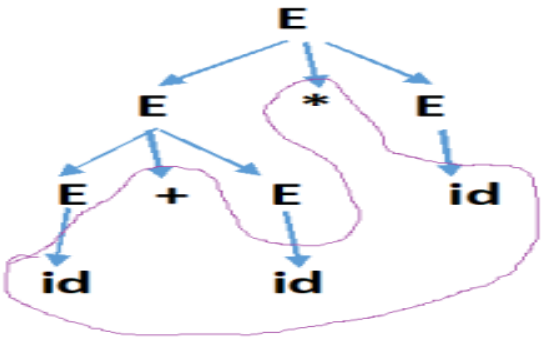
$E \overset{lmd}{\Longrightarrow} E*E$ | E-> E*E

$\overset{lmd}{\Longrightarrow} E+E*E$ | E->E+E

$\overset{lmd}{\Longrightarrow} id+E*E$ | E->id

$\overset{lmd}{\Longrightarrow} id+id*E$ | E->id

$\overset{lmd}{\Longrightarrow} id+id*id$ | E->id



There are 2 parse trees or 2 leftmost derivations for the string 'id+id*id'. So the given grammar is ambiguous.

Example Design a CFG for the language $L = \{a^n b^m : n \neq m\}$

solution If $n \neq m$ then there are only two cases are possible

Case 1. $n > m$

let us say language L on condition $n>m$ is

L_1 and L_2 = $\{a^n b^m : n > m\}$

Let G_L be the CFG for the language L_1,

$G_L = (V_N^L, \Sigma^L, P^L, S^L)$

$V_N^L = \{S_1, A, S\}$

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

Then production P_L are defined as.

$$S^1 \rightarrow AS_1$$

$$S_1 \rightarrow aS_1b \mid \varepsilon$$

$$A \rightarrow aA \mid a$$

Case 2 $\quad n<m$

let us say L on condition $n<m$ is L_2

$$L_2 = \{0^n 1^m : n<m\}$$

and let CFG for G_2 be $G_{12} = (V_N^2, \Sigma^2, P^2, S)$

$$V_N^2 = \{S^2, S_2, B\}$$

$$\Sigma^2 = \{a, b\}$$

P^2 are defined as,

$$S^2 \rightarrow S_2B$$

$$S_2 \rightarrow aS_2b \mid \varepsilon$$

$$B \rightarrow bB \mid b$$

By combining G_L and G_2 we can write the CFG for L as

$$S \rightarrow S^1 \mid S^2$$

* Derivation

⇒ CFG generates string according to the following process called derivation.

(a) We start by writing down the start symbol of the CFGs.

(b) At each step of the derivation, we may replace any non-terminal symbol of the string generated so far by the right hand side of any production that has the symbol on the left.

(c) The process ends when it is impossible to apply a step of type described in (b).

⇒ If the string at the end of this process consists entirely of terminals, it is a string in the language generated by the grammar called yield.

⇒ There are two approaches of derivation

1. Body to head approach (Bottom up)
2. Head to Body (Top down)

1. Body to head :-

⇒ Here, we take strings known to be in the language of each of the variables of the body, concatenate them in the proper order with any terminals appearing in the body, the resulting string is the language of the variable in the head.

Consider grammar,

$S \to S+S$

$S \to S|S$

$S \to (S)$

$S \to S-S$

$S \to S*S$

$S \to a$

Here given a+(a*a)|a-a

now, applying body to head approach,

S.N	String inferred	Variable	Production	string used.
1.	a	S	S→a	
2.	a*a	S	S→S*S	string ①
3.	(a*a)	S	S→(S)	string ②
4.	(a*a)|a	S	S→S|S	string ② and ①
5.	(a*a)|a-a	S	S→S-S	string ④ and ①
6.	a+(a*a)|a-a	S	S→S+S	string ⑤ and ①

Thus in this approach we start with any terminal appearing in the body and use the available rules from body to head.

2. Head to Body :-

Here, we used production from head to body. we expand the start symbol using a production, whose head is the start symbol. Here we expand the resulting string until all strings of terminal are obtained. Here we have two approaches

(A) Left most Derivation :-

Here, leftmost symbol (variable) is replaced first

(B) Right most Derivation :-

Here, rightmost symbol is replaced first.

Example Consider the grammar G1 with production

$$S \to aSS \mid b$$

Now, we have

$$\begin{aligned} S &\Rightarrow aSS \\ &\Rightarrow aaSSS \\ &\Rightarrow aabSS \\ &\Rightarrow aabaSSS \\ &\Rightarrow aababSS \\ &\Rightarrow aababbS \\ &\Rightarrow aababbb \end{aligned}$$

The sequence followed is left-most derivation

Now right-most derivation:

$$\begin{aligned} S &\Rightarrow aSS \\ &\Rightarrow aSb \\ &\Rightarrow aaSSb \\ &\Rightarrow aaSaSSb \\ &\Rightarrow aaSaSbb \\ &\Rightarrow aaSabbb \\ &\Rightarrow aababbb \end{aligned}$$

Parse Tree or Derivation Tree

⇒ The strings generated by a context free grammar (V_N, Σ, P, S) can be represented by a hierarchical structure called tree. Such trees representing derivations are called derivation trees or parse tree.

⇒ characteristics of a Parse Tree

A parse tree for a context-free grammar G has the following characteristics

1. Every vertex of a parse tree has a label which is a variable or terminal or ε.
2. The root of parse tree has label S (start symbol).
3. The label of an internal vertex is a variable.
4. If a vertex A has k children with labels $A_1, A_2 \ldots A_k$ then $A \to A_1, A_2, \ldots A_k$ will be a production in context free Grammar G.
5. A vertex 'n' is a leaf if its label is $a \in \Sigma$ or ε.
6. 'n' is the only son of its father if its label is ε.

Example Derive the string "aabbaa" and construct a derivation tree for "aabbaa" for a grammar G with productions $S \to aAS|a$, $A \to SbA|SS|ba$

Solution

$S \to aAS|a$

$A \to SbA|SS|ba$

String "aabbaa" is derived from S as,

$S \Rightarrow aAS$

$\Rightarrow aSbAS$

$\Rightarrow aabAS$

$\Rightarrow aabbaS$

$\Rightarrow aabbaa$

The derivation tree is shown as.

Simplification of Context Free Grammar

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

Simplification consists of the following steps:-

1) Reduction of CFG 2) Removal of Unit Productions 3) Removal of Null Productions.

Reduction of CFG

CFG are reduced in two phases

phase1:- Derivation of an equivalence grammar 'G', from the CFG, G, such that each variable ~~devices~~ derives some terminal string

Derivation Procedure :-

Step 1:. Include all symbols W_1, that derives some terminal and initialize $i = 1$

Step 2: Include symbols W_{i+1}, that derives W_i

Step 3: Increment i and repeat step 2, until $W_{i+1} = W_i$

Step 4:- Include all production rules that have W_i in it.

Phase 2:- Derivation of an equivalent grammar "G", from the CFG, 'G', such that each symbol appears in a sentential form.

Derivation Procedures:-

Step 1: Include the start symbol in Y_1 and initialize $i=1$

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

Step 3: Increment i and repeat step 2, until $Y_{i+1} = Y_i$

Example :-

Example:- Find a reduced grammar equivalent to the Grammar G, having production rules

P: S → AC | B , A → a , C → c | BC, E → aA | e

Phase 1:

T = {a, c, e}

W_1 = {A, C, E} — set derive terminals

W_2 = {A, C, E, S} — set of element of W_1

W_3 = {A, C, E, S}

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

P: S → AC , A → a , C → c , E → aA | e

phase 2:-

Y_1 = { S }

Y_2 = { S, A, C }

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

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

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

P:- S → AC , A → a , C → c

Removal of Unit Productions

Any production Rule of the form A → B where A, B ∈ Non Terminals is called Unit Production.

Procedure for Removal:-

Step 1:- To remove A → B, add production A → x to the grammar rule whenever B → x occurs in the grammar. [x ∈ Terminal, x can be Null]

Step 2:- Delete A → B from the grammar.

Step 3:- Repeat from step1 until all Unit Productions are removed

Example:- Remove Unit Productions from the Grammar whose production rule is given by

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

$Y \to Z$, $Z \to M$, $M \to N$

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

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

2) Since $M \to a$, we add $Z \to a$

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

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

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

→ from Start Symbol S we can not reach to Z, M, N.

Remove the unreachable symbols

P: $S \to XY$, $X \to a$, $Y \to a|b$

Removal of Null Productions

In a CFG, a Non-Terminal Symbol 'A' is a nullable variable if there is a production $A \to \epsilon$ or there is a derivation that starts at 'A' and leads to ϵ. (Like $A \to \dots \to \epsilon$)

Production for Removal:-

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

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

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

Example:- Remove Null Productions from the following Grammar.

$S \to ABAC$, $A \to aA|\epsilon$, $B \to bB|\epsilon$, $C \to c$

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

1) To eliminate $A \rightarrow \epsilon$

$S \rightarrow ABAC$

$S \rightarrow ABEC$

$\rightarrow ABC \mid BAC \mid BC$

$\rightarrow$

$A \rightarrow aA$

$A \rightarrow a$

New production: $S \rightarrow ABAC \mid ABC \mid BAC \mid BC$

$A \rightarrow aA \mid a$, $B \rightarrow bB \mid \epsilon$, $C \rightarrow c$

2) To eliminate $B \rightarrow \epsilon$

$S \rightarrow AAC \mid AC \mid C$, $B \rightarrow b$

New production: $S \rightarrow ABAC \mid ABC \mid BAC \mid BC \mid AAC \mid AC \mid C$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

$\epsilon \rightarrow c$

4. Normal Forms:

Chomsky Normal Form(CNF)

Greibach Normal Form(GNF)

4.1Chomsky Normal Form(CNF):

A CFG is in CNF if every production is one of two types

A->BC

A->a

Where A,B and C are Non terminals and 'a' is a terminal

4.1.1.Converting a CFG to CNF

i.Let G be the CFG with productions

S->AACD
A->aAb | Λ
C->aC |a
D ->aDa | bDb |Λ

Step 1:

Eliminating Λ productions

Any production A for which P contains the production A->Λ is nullable

Nullable variable: A->Λ D->Λ

S->AACD | ACD |CD | AAC |AC |C

A->aAb | ab

C->aC |a

D ->aDa | bDb |aa |bb

Step 2:

Eliminating unit productions S->C

S->AACD | ACD |CD | AAC |AC | aC |a

A->aAb | ab

C->aC |a

D ->aDa | bDb |aa |bb

Step 3:Restricting the right sides of the productions to single terminals or strings of two or more variables(NON TERMINALS).

S->AACD | ACD |CD | AAC |AC | X_a C |a

A-> X_a A X_b | X_a X_b

C-> X_a C |a

D -> X_aD X_a | X_b D X_b | X_a X_a | X_b X_b

X_a->a

X_b->b

Step 4:

S->AT_1 |AT_2 |CD |AT_3 |AC | X_a C |a

T_1->AT_2

T_2->CD

T_3->AC

A-> $X_a$$T_4$ | X_a X_b

T_4-> A X_b

C-> X_a C |a

D->X_aT5 | $X_b$$T_6$| X_a X_a | X_b X_b

T_5->D X_a

T_6-> D X_b

X_a->a

X_b->b

4.3 Greibach Normal Form (GNF)

Let G=(N, T, P, S) be a CFG. The CFG 'G' is said to be in GNF, if all the production are of the form:

$$A \rightarrow a\alpha$$

Where a Ɛ T and α Ɛ N*

A non-terminal generating a terminal which is followed by any number of non-terminals.

For example, A → a.

S → aASB.

Step 1: Convert the grammar into CNF.

Step 2: Rename the non-terminals to A1, A2, A3,…

Step 3: In the grammar, convert the given production rule into GNF form.

Problem 1:

S-> AB1 |0S | ε

A->00A | B

B-> 1A1

Step 1: Eliminate null productions.

S-> ε

S->AB1 | 0S | 0

A->00A |B

B->1A1

Step 2: Eliminate unit productions

A->B is the unit production.

S->AB1 |0S |0

A->00A |1A1

B->1A1

Step 3: Restricting right hand side production with single terminal symbol or two or more non terminals.

X->0

Y->1

S->ABY | XS |0

A->XXA |YAY

B->YAY

Step 4: Final CNF

X->0 Y->1

S->AT1 T1->BY

S->XS | 0

A->XT2 T2->XA A->YT3 T3->AY

B->YT3

Step 5: Rename Non terminal as A1, A2, A3,……..

S=A1, A= A2, B=A3, X=A4, Y=A5, T1=A6, T2=A7, T3=A8
X->0 Y->1
S->AT1 T1->BY
S->XS | 0
A->XT2 T2->XA A->YT3 T3->AY
B->YT3

A4->0 A5->1
A1->A2A6 | A4A1 | 0
A2->A4A7 | A5A8
A3->A5A8
A6->A3A5
A7->A4A2
A8->A2A5

Step 6: Obtain productions to the form A->aα

Final CFG is

A4->0 A5->1

A2->0A7 |1A8

A3->1A8

A7->0A2

A8->0A7A5 | 1A8A5

A6->1A8A5

A1->0A7A6 | 1A8A6 | 0A1 |0

Pumping Lemma for Context-free Language :-

⇒ Pumping Lemma is used to prove that a language is not regular but we can't prove that the language is regular.

⇒ Similarly, in the case of context-free Language we cannot prove that a context-free language and the pumping lemma given are only for finite languages.

⇒ If L is any context-free language, then we can find a natural number 'n' such that:

(i) every $w \in L$ where $|w| \geq n$ can be written as $uvwxy$ for some strings u, v, w, x and y.

(ii) $|vx| \geq 1$

(iii) $|vwx| \leq n$

(iv) $uv^i wx^i y \in L \quad \forall\ i \geq 0$

Example :- Show that $L = \{a^n b^n c^n \mid n \geq 1\}$ is not a CFL.

Soln. Let the given Language 'L' is a context-free language.

Let $n = 3$

$L = \{abc, aabbcc, aaabbbccc, \ldots\}$

$w = aaabbbccc$

checking (1) case :- $w = aaabbbccc$

$|w| \geq n$

$|aaabbbccc| \geq 3$

$9 \geq 3$, which is true.

checking 2 case :

$w = \underbrace{aa}_{u}\underbrace{a}_{v}\underbrace{bbb}_{w}\underbrace{b}_{x}\underbrace{ccc}_{y}$

$u = aa$, $v = a$, $w = b$, $x = b$, $y = ccc$

$|vwx| \leq n \Rightarrow |a.b.b| \leq 3$

$3 \leq 3$

checking 3 case :-

i.e $|vx| \geq 1$

$|ab| \geq 1$

$2 \geq 1$, true

for $i = 0$, $uv^i wx^i y \in L$ is in L.

$aa.a^0.b.b^0.bccc$

$\Rightarrow aabbccc \notin L$, which is false

Thus, the given Languge is not CFL.

Closure Properties of CFL :-

⇒ Whenever we apply operations like union, intersection, concatenation, complement, star closure etc on given CFL and the result of these operations are again a CFL, then we say that the family of CFL is closed under the operations done.

⇒ The CFL's are closed under some operation means 'after performing that particular operation on these CFLs the resultant language is CFL. These propaties are :

1. The context free languages are closed under union.
2. The context free languages are closed under concatenation.
3. The context free Languages are closed under kleen closure.
4. The context free Languages are not closed under intersection.
5. The context free Languages are not closed under complement.

Theorem: If L_1 and L_2 are context-free Languages then $L = L_1 \cup L_2$ is also context-free. That is, the CFLs are closed under union.

Proof :- We will consider two Languages L_1 and L_2 which are context-free Languages. We can give these Languages using context-free Grammars G_1 and G_2 such that $G_1 \in L_1$ and $G_2 \in L_2$. The G_1 can be given as

$G_1 = \{V_1, \Sigma, P_1, S_1\}$ where P_1 can be given as,

$$P_1 = \{ S_1 \to A_1 S_1 A_1 \mid B_1 S B_1 \mid \varepsilon$$

$$A_1 \to a$$

$$B_1 \to b$$

Here $V_1 = \{S_1, A_1, B_1\}$ and S_1 is a start symbol.

Similarly, we can write $G_2 = \{V_2, \Sigma, P_2, S_2\}$

$V_2 = \{S_2, A_2, B_2\}$ and S_2 is a start symbol.

P_2 can be given as:

$$P_2 = \{ S_2 \to a A_2 A_2 \mid b B_2 B_2$$

$$A_2 \to b$$

$$B_2 \to a$$

$$\}$$

Now, $L_1 \cup L_2$ gives $G \in L$ This G can be written as.

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

$$V = \{ S_1, A_1, B_1, S_2, A_2, B_2 \}$$

$$P = \{ P_1 \cup P_2 \}$$

S is a start symbol.

$P = \{$

$S \to S_1 | S_2$

$S_1 \to A_1 S_1 A_1 | B_1 S B_1 | \varepsilon$

$A_1 \to a$

$B_1 \to b$

$S_2 \to a A_2 A_2 | b B_2 B_2$

$A_2 \to b$

$B_2 \to a$

$\}$

Thus grammar G is context-free Grammar which produces Languages L which is context free Language.

* Decision algorithm for CFLs

⇒ There are many questions about CFL that are undecidable that mean there doesnot exist any algorithm to answer these questions. Such questions are called the questions about undecidable context free Languages

1. Whether or not two different context-free Languages define the same Language?
2. Whether given CFL is ambiguous or not?
3. Whether complement of given CFL is context free language?
4. Whether the intersection of two context free Language is a context free?

⇒ There is no algorithm to answer these questions. But there is a certain set of questions for which the answer can be obtained. Such questions are about context free language that are decidable

⇒ following is a list of those questions

1. Does the CFG G_1 generates any string? (emptiness)
2. Is the language generated by G_1 is finite or not? (finiteness).
3. Suppose G_1 is a grammar and W is a string, does G_1 generates W? (membership).

1. Push Down Automata

1.1. Drawback of Finite Automata

- can remember only a finite amount of information
- No memory is used in FA

2. Introduction

- PDA can remember an infinite amount of information.
- Memory used – Stack
- A PDA is more powerful than FA.
- Any language which can be acceptable by FA can also be acceptable by PDA.
- PDA also accepts a class of languages which cannot be accepted by FA.
- PDA recognizes CFL.

FA + stack = PDA

3. Definition

The PDA can be defined as a collection of 7 tuples:

$$M=(Q, \sum, \Gamma, q_0, Z_0, F, \delta)$$

Q: the finite set of states

$\sum$: the input set

Γ: a stack symbol which can be pushed and popped from the stack q_0:

the initial state

Z_0: a start symbol which is in Γ.

F: a set of final states.

δ: Transition function which is used for moving from current state to next state.

δ: Q x $\{\Sigma \cup \varepsilon\}$ x Γ -> Q x Γ^*

(i.e) $\delta(q,a,x)=(p,\alpha)$

from state 'q' for an input symbol 'a', and stack symbol 'x', goto state 'p' and x is replaced by string 'α'.

3.1. Instantaneous Description (ID)

- An instantaneous description is a triple ID

$$(q, w, \alpha)$$

Where:

- **Q** describes the current state.
- **w** describes the remaining input.
- **α** describes the stack contents, top at the left.

Example Derivation: $(p, b, T) \vdash (q, w, \alpha)$

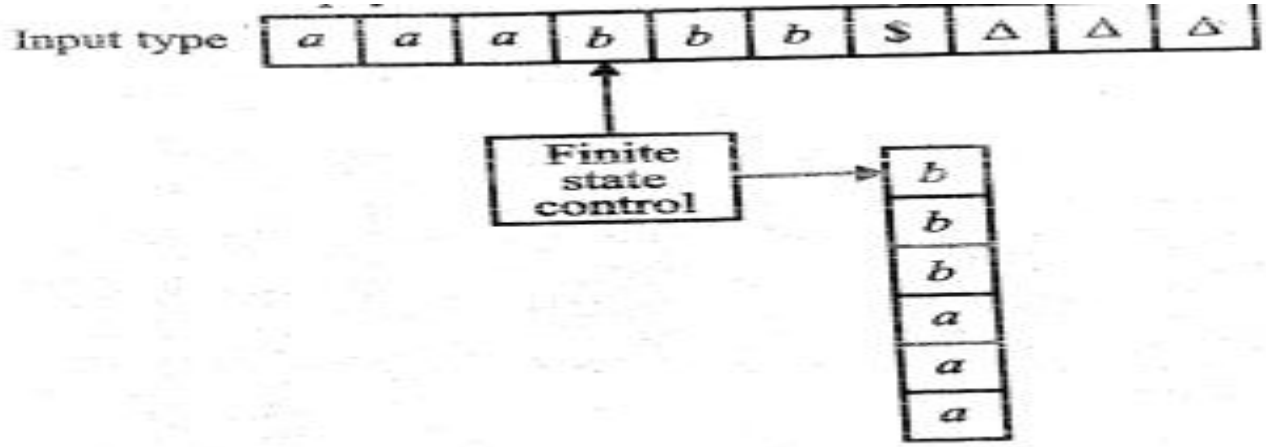


Fig. pushdown automata

3.2. Definition: Acceptance by a PDA

1. Acceptance by Final State:
   If M =(Q, ∑, Γ, δ, q0, Z, F) is a PDA and the language L(M) accepted by the final state is given by: x∈∑* and x is accepted by M if,
   $L(M) = \{x \mid (q0, x, Z_0) \vdash^* (q, \Lambda, \alpha)\}$
   Where q∈Aα∈Γ*

2. Acceptance by Empty Stack:
   For each PDA, M=(Q, ∑, Γ, δ, q0, Z, F) the language accepted by empty stack is given by
   $L(M) = \{x \mid (q0, x, Z_0) \vdash^* (q, \Lambda, \Lambda)\}$
   For any state q∈A and x∈∑*

Language Acceptance:
A language L ⊆∑* is said to be accepted by M, if L is precisely the set of string accepted by M.
L=L(M)

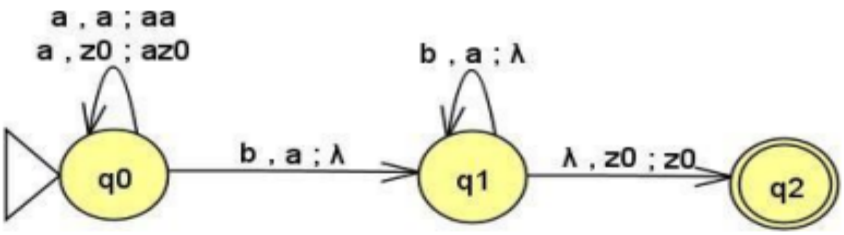
4. Construction of PDA

1. Design a PDA for accepting a language $\{a^n b^n \mid n>=1\}$.

Solution:

1. Decide the nature of the language b's followed by 'a'.
2. Execution procedure using stack:
   - Push all a's on to stack.
   - For every 'b' pop out an 'a'.
3. Define the states
   - q0 – push all a's on to the stack.
   - q1 – when a 'b' encounters, pop 'a' from stack.
   - q2 – accepting state.

PDA Diagram



Transition Table:

Move No.	**State**	**Input Symbol**	**Top of stack**	**Moves**
1	q_0	a	Z0	(q_0,az_0)
2	q_0	a	a	(q_0,aa)
3	q_0	b	a	(q_1,Λ)
4	q_1	b	a	(q_1,Λ)
5	q_1	Λ	Z0	(q_2,z_0)
All other combinations	None			

Trace the moves: a^3b^3 => aaabbb

Move No.	Resulting state	Input	Stack
-	q_0	aaabbb	Z_0
1	q_0	aabbb	az_0
2	q_0	abbb	aaz_0
2	q_0	bbb	$aaaz_0$
3	q_1	bb	aaz_0
4	q_1	b	az_0
4	q_1	Λ	Z_0
5	q_2	Λ	Z_0
Accepted			

Trace the moves: a^2b=> aab

Move No.	Resulting state	Input	Stack
-	q_0	aab	Z_0
1	q_0	ab	az_0
2	q_0	b	aaz_0
3	q_1	Λ	az_0
Rejected			

Instantaneous Description (ID)

$(q_0, aabb,z_0)$ |- (q_0, abb, az_0)

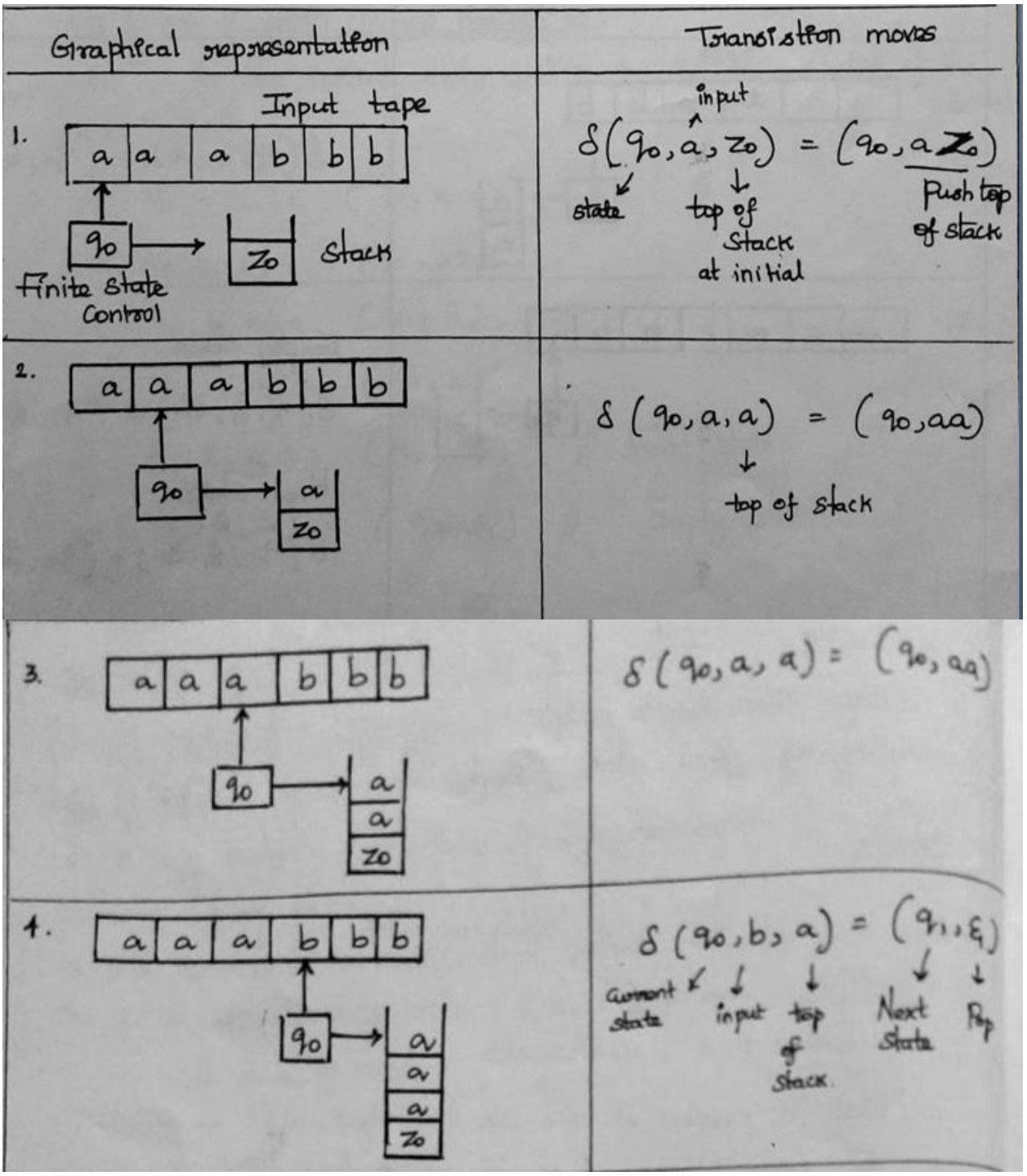
|- (q_0, bb, aaz_0)

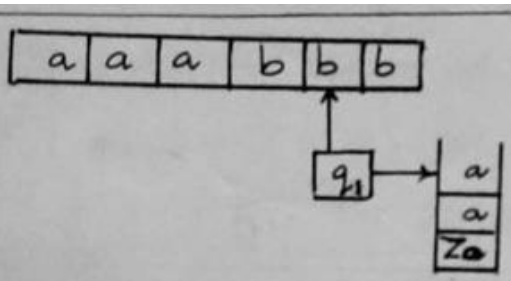
$\vdash(q_1, b, az_0)$

$\vdash(q_1, \Lambda, z_0)$

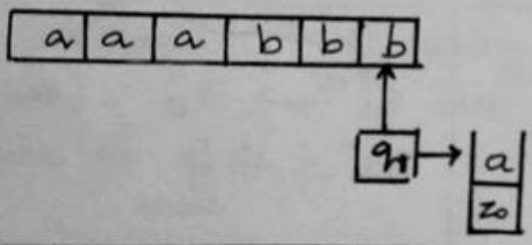
$\vdash(q_2, \Lambda, z_0)$

String Accepted

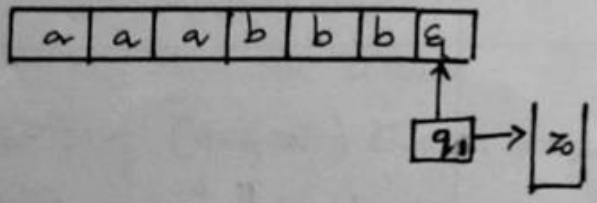


5. 

$\delta(q_1, b, a) = (q_1, \epsilon)$

6. 

$\delta(q_1, b, a) = (q_1, \epsilon)$

7. 

Empty stack:

$\delta(q_1, \epsilon, Z_0) = (q_2, \epsilon)$

Final state:

$\delta(q_1, \epsilon, Z_0) = (q_2, Z_0)$

$L = \{ 0^n 1^{2n} / n \geq 0 \}$

4) $L = \{ a^n b^{2n} / n \geq 0 \}$ (or) Design a PDA with set of strings with twice as many b's than a's with a as the starting string (or) 2 occurences of b's for each a's. accept by final state

Step1 : $L = \{ \epsilon, abb, aabbbb, \ldots \}$

Idea: To Design this PDA, is that when we read single 'a' we insert / push 2 a's on stack.

Then when we read 'b' we pop each 'a' on the top of stack and when reading Z_0 on stack, we reach final state.

Step 3: Transition function :

$\delta(q_0, a, z_0) = (q_0, aaz_0)$

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

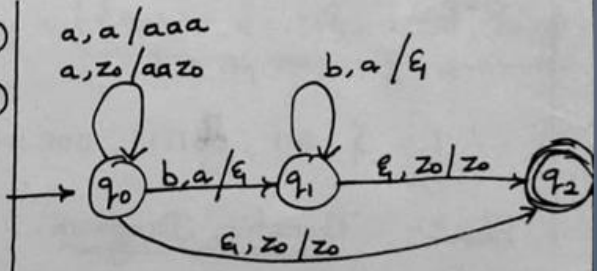
$\delta(q_0, b, a) = (q_1, \varepsilon)$

$\delta(q_1, b, a) = (q_1, \varepsilon)$

$\delta(q_1, \varepsilon, z_0) = (q_2, z_0)$

$\delta(q_0, \varepsilon, z_0) = (q_2, z_0)$

Step 2: Transition diagram.



PDA for $L = \{a^n b^{2n} / n \geq 0\}$

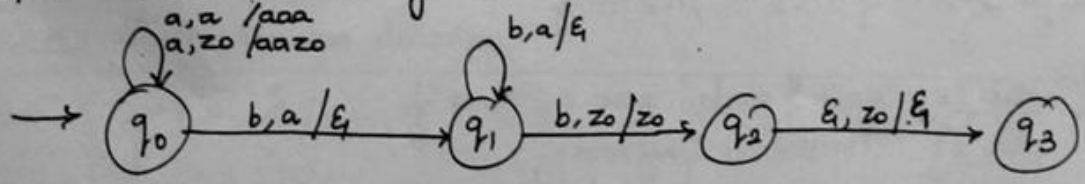
PDA $P_F = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \{q_2\})$

---

9) $L = \{a^n b^{2n+1} / n \geq 1\}$ by empty stack.

Step 1: $L = \{abbb, aabbbbb, \ldots\}$

Step 2: Transition Diagram:



PDA $P_N = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \phi)$

Step 3: Transition Function

$\delta(q_0, a, z_0) = (q_0, aaz_0)$

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

$\delta(q_0, b, a) = (q_1, \epsilon)$

$\delta(q_1, b, z_0) = (q_0, z_0)$

$\delta(q_2, \epsilon, z_0) = (q_3, \epsilon)$

$\delta(q_1, b, a) = (q_1, \epsilon)$

Step 1: Instantaneous description:

$w_1 = abbb$

$(q_0, abbb, z_0) \vdash_P (q_0, bbb, aaz_0)$

$\vdash_P (q_1, bb, az_0)$

$\vdash_P (q_1, b, z_0)$

$\vdash_P (q_0, \epsilon, z_0)$

$\vdash_P (q_3, \epsilon, \epsilon)$

∴ String is accepted.

$w_2 = abb$

$(q_0, abb, z_0) \vdash_P (q_0, bb, aaz_0)$

$\vdash_P (q_1, b, az_0)$

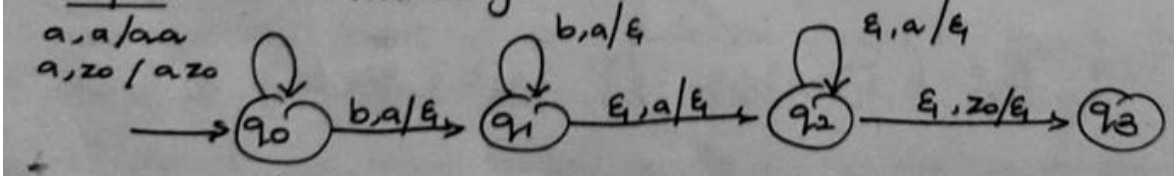
$\vdash_P (q_1, \epsilon, z_0)$

There is no transition ∴ String is not accepted.

11) $L = \{a^n b^m / n > m\}$

Step 1: $L = \{aab, aaab, aaabb, \ldots\}$

Step 2: Transition Diagram:



PDA $P_N = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \phi)$

Step 3: Transition function.

$\delta(q_0, a, z_0) = (q_0, az_0)$

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

$\delta(q_0, b, a) = (q_1, \epsilon)$

$\delta(q_1, b, a) = (q_1, \epsilon)$

$\delta(q_1, \epsilon, a) = (q_2, \epsilon)$

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

$\delta(q_2, \epsilon, z_0) = (q_3, \epsilon)$

Step 4: Instantaneous description

$w_1 = aab$

$\delta(q_0, aab, z_0) \vdash_P (q_0, ab, az_0)$

$\vdash_P (q_0, b, aaz_0)$

$\vdash_P (q_1, \epsilon, az_0)$

$\vdash_P (q_2, \epsilon, z_0)$

$\vdash_P (q_3, \epsilon, \epsilon)$

∴ String is accepted.

$w_2 = ab$

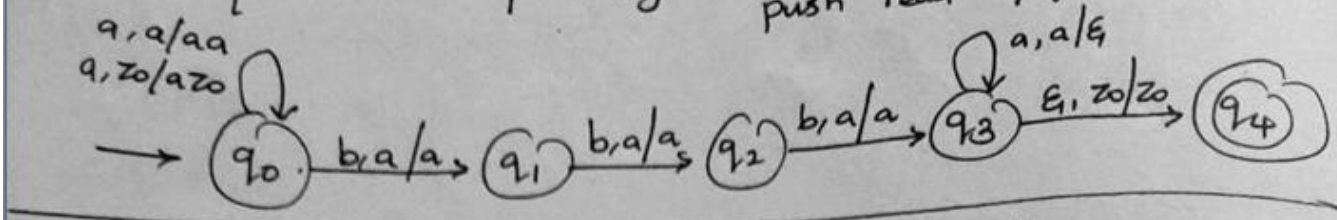
$(q_0, ab, z_0) \vdash_P (q_0, b, az_0)$

$\vdash_P (q_1, \epsilon, z_0)$

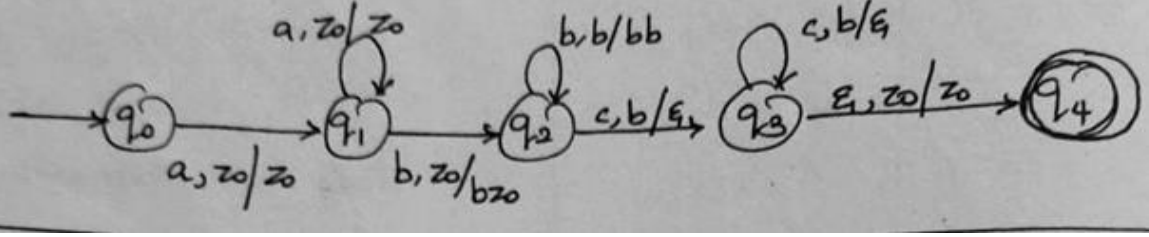
There is no transition for $\delta(q_1, \epsilon, z_0)$

⇒ The string is not accepted.

(4) $L = \{a^n b^3 a^n \mid n \geq 1\}$ a^n b^3 a^n
push read pop



15) $L = \{a^n b^m c^m \mid n, m \geq 1\}$ a^n b^m c^m
read push pop

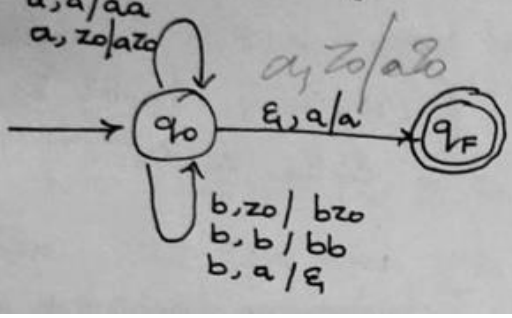


Design a PDA for the language

1. $L = \{w \mid w \in (a+b)^*$ and $n_a(w) > n_b(w)\}$

$n_a(w)$ means total no. of a's in input string and $n_b(w)$ means total no. of b's in input string, problem states that total no. of a's are more than total no. of b's in input string.

2. PDA Transition diagram:



PDA $P_F = (\{q_0, q_F\}, \{a, b\}, \{a, b, z_0\}, \delta, q_0, z_0, \{q_F\})$

3. Transition moves

$\delta(q_0, a, z_0) = (q_0, az_0)$

$\delta(q_0, b, z_0) = (q_0, bz_0)$

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

$\delta(q_0, b, b) = (q_0, bb)$

$\delta(q_0, a, b) = (q_0, \varepsilon)$

$\delta(q_0, b, a) = (q_0, \varepsilon)$

$\delta(q_0, \varepsilon, a) = (q_f, a)$

1. Instantaneous description

$\delta(q_0, aababab, z_0)$

$\vdash_P (q_0, ababab, az_0)$

$\vdash_P (q_0, babab, aaz_0)$

$\vdash_P (q_0, abab, az_0)$

$\vdash_P (q_0, bab, aaz_0)$

$\vdash_P (q_0, ab, az_0)$

$\vdash_P (q_0, b, aaz_0)$

$\vdash_P (q_0, \varepsilon, az_0)$

$\vdash_P (q_F, a)$

$\therefore$ It is accepted.

DETERMINISTIC PUSH DOWN AUTOMATA (DPDA)

Definition:

A PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ is deterministic if and only if it satisfies the following conditions

1. $\delta(q, a, x)$ has only one member for any given q in Q, a in Σ or $a = \epsilon$, and x in Γ
2. If $\delta(q, a, x)$ is non empty for some a in Σ, then $\delta(q, \epsilon, x)$ must be empty.

Problem:

1. $L = \{wc w^R \mid w \text{ is in } (0+1)^*\}$ / odd Palindrome

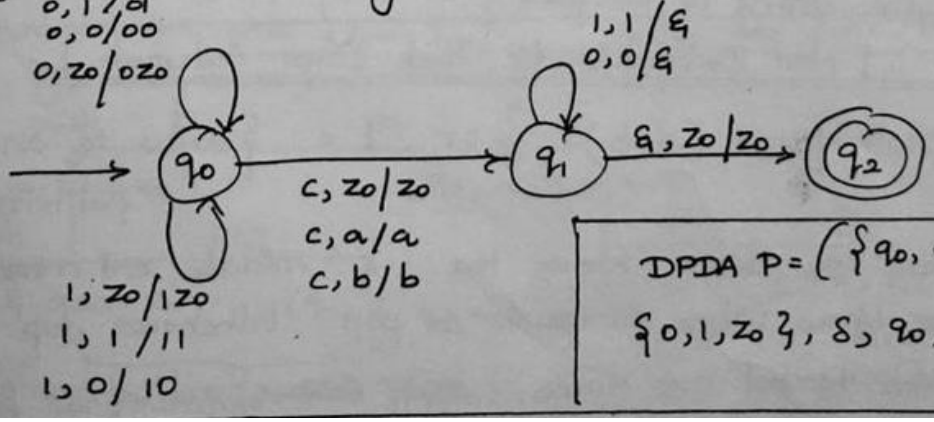
Idea: The PDA for this is designed in such a way that DPDA is to store 0's and 1's on stack until it sees the middle end marker. After this, it goes to another state in which it matches i/p symbols against stack symbols and pops the stack if they match or else rejected. Thus the PDA is strictly DPDA.

It does not have a choice of move in the start state using the same input and stack symbol.

Step 1: $L = \{c, 0c0, \overline{1c1}, \overline{0\overline{1c1}0}, 11011, 00c00, \ldots\}$

Step 2: Transition diagram.

push — wc — read — w^R — pop



DPDA $P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, z_0\}, \delta, q_0, z_0, \{q_2\})$

Step 3: Transition function

$\delta(q_0, 0, z_0) = (q_0, 0z_0)$

$\delta(q_0, 0, 0) = (q_0, 00)$

$\delta(q_0, 0, 1) = (q_0, 01)$

$\delta(q_0, 1, z_0) = (q_0, 1z_0)$

$\delta(q_0, 1, 1) = (q_0, 11)$

$\delta(q_0, 1, 0) = (q_0, 10)$

$\delta(q_0, c, z_0) = (q_1, z_0)$

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

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

$\delta(q_1, 1, 1) = (q_1, \epsilon)$

$\delta(q_1, 0, 0) = (q_1, \epsilon)$

$\delta(q_1, \epsilon, z_0) = (q_2, z_0)$

Step 4: Instantaneous description:

$w_1 = 011c110$

$\delta(q_0, 011c110, z_0) \vdash_P (q_0, 11c110, 0z_0)$

$\vdash_P (q_0, 1c110, 10z_0)$

$\vdash_P (q_0, c110, 110z_0)$

$\vdash_P (q_1, 110, 110z_0)$

$\vdash_P (q_1, 10, 10z_0)$

$\vdash_P (q_1, 0, 0z_0)$

$\vdash_P (q_1, \epsilon, z_0)$

$\vdash_P (q_2, \epsilon, z_0)$

$\therefore$ The string is accepted.

$w_2 = 01c1$

$\delta(q_0, 01c1, z_0)$

$\vdash_P (q_0, 1c1, 0z_0)$

$\vdash_P (q_0, c1, 10z_0)$

$\vdash_P (q_0, 1, 10z_0)$

$\vdash_P (q_1, \epsilon, 0z_0)$

No transition moves

$\therefore$ String is not accepted.

NPDA [Non Deterministic Push Down Automata]

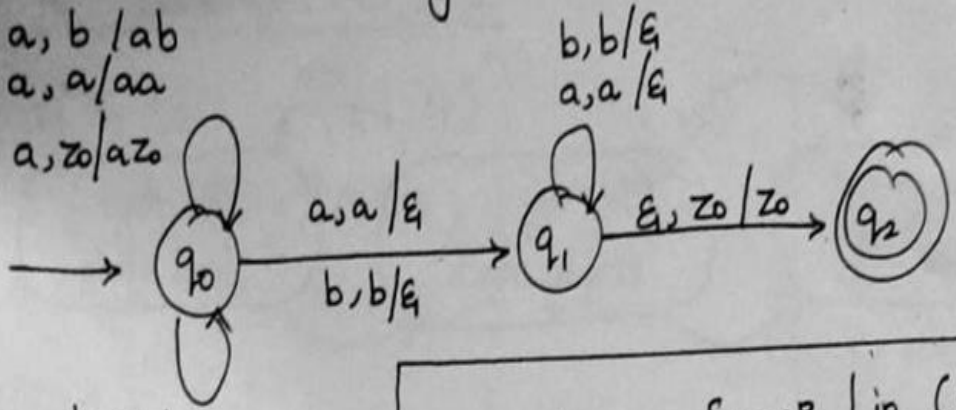
$L = \{ wcw^R \mid w \in (a,b)^* \}$ or $L = \{ w \mid w$ is an even palindrome$\}$.

Idea: Here we don't know the 'c' middle end marker. So not able to know when to push or pop. Whenever top of stack and input symbol are same, then corresponding one change to centre (ie) when top of stack = input symbol, we have to assume that might be centre has come or not.

PDA → Accept : Centre has come.
PDA → Reject : Centre has not come.

Step 1: $L = \{ \varepsilon, aa, bb, abba, aaaa, baab, \ldots \}$

Step 2: Transition diagram.



NPDA for $L = \{ wcw^R \mid$ in $(a,b)^* \}$

NPDA $P = (\{q_0, q_1, q_2\}, \{a,b\}, \{a,b,z_0\}, \delta, q_0, z_0, \{q_2\})$.

Step 3: Transition function:

$\delta(q_0, a, z_0) = (q_0, az_0)$

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

$\delta(q_0, a, b) = (q_0, ab)$

$\delta(q_0, b, z_0) = (q_0, bz_0)$

$\delta(q_0, b, b) = (q_0, bb)$

$\delta(q_0, b, a) = (q_0, ba)$

$\delta(q_0, a, a) = (q_1, \varepsilon)$

$\delta(q_0, b, b) = (q_1, \varepsilon)$

$\delta(q_1, a, a) = (q_1, \varepsilon)$

$\delta(q_1, b, b) = (q_1, \varepsilon)$

$\delta(q_1, \varepsilon, z_0) = (q_2, z_0)$

Step 4: Instantaneous description $\boxed{w = aaaa}$

$\delta(q_0, aaaa, z_0) \vdash_P (q_0, aaa, az_0)$

centre hasn't come (Push) | Centre has come (Pop)

(q_0, aa, aaz_0) — (q_1, aa, z_0) ×

(Dead configuration)

No transition

push | pop

$(q_0, a, aaaz_0)$ — (q_1, a, az_0)

push | pop — pop

$(q_0, \varepsilon, aaaaz_0)$ — $(q_1, \varepsilon, aaz_0)$ — (q_2, ε, z_0)

× × — Final state

Dead configuration [No Transition] ∴ String is accepted

PROBLEM :

1) Construct the PDA for the following grammar :

$E \rightarrow E+E \mid E*E \mid a$

Solution:

Step 1: $G : E \rightarrow E+E \mid E*E \mid a$

$V = \{E\}$ $\quad P = \{E \rightarrow E+E,\ E \rightarrow E*E,\ E \rightarrow a\}$

$T = \{+, *, a\}$ $\quad S = E$

Step 2: PDA $P = (\{q\}, \{+, *, a\}, \{E, *, +, a\}, \delta, q, E)$

Step 3: Transition function of PDA

For Nonterminal Variable 'E'

$\delta(q, \varepsilon, E) = \{(q, E+E), (q, E*E), (q, a)\}$

For terminal $+, *, a$

$\delta(q, +, +) = \{(q, \varepsilon)\}$

$\delta(q, *, *) = \{(q, \varepsilon)\}$

$\delta(q, a, a) = \{(q, \varepsilon)\}$

Step 4: Instantaneous description

$w = a*a+a$

Selecting string :

LMD :

$E \xrightarrow[lmd]{} E*E$ $\quad (E \rightarrow E*E)$

$\xrightarrow[lm]{} a*E$ $\quad (E \rightarrow a)$

$\xrightarrow[lm]{} a*E+E$ $\quad (E \rightarrow E+E)$

$\xrightarrow[lm]{} a*a+E$ $\quad (E \rightarrow a)$

$\xrightarrow[lm]{} a*a+a$ $\quad (E \rightarrow a)$

$(q, a*a+a, E) \vdash_P (q, \varepsilon\, a*a+a, E)$

$\vdash_P (q, a*a+a, E*E)$

$\vdash_P (q, a*a+a, a*E)$

$\vdash_P (q, *a+a, *E)$

$\vdash_P (q, a+a, E)$

$\vdash_P (q, a+a, E+E)$

$\vdash_P (q, a+a, a+E)$

$\vdash_P (q, +a, +E)$

$\vdash_P (q, a, E)$

$\vdash_P (q, a, a)$

$\vdash_P (q, \varepsilon, \varepsilon)$

Thus the CFG accepts the string a*a+a and it is accepted by PDA by empty stack.

CONVERSION FROM PDA TO CFG

RULE 1: If q_0 is start state then Q is set of states of PDA, then start production is given by $\boxed{S \rightarrow [q_0, Z_0, Q]}$

$S \rightarrow$ [initial state, initial stack symbol, each state in Q]

RULE 2: Production Rule for instantaneous description of the form

PUSH: $\delta(q_i, a, z_0) = (q_{i+1}, Z_1 Z_2)$ then

$[q_i, z_0, q_{i+k}] \longrightarrow a\,[q_{i+1}, Z_1, q_m][q_m, Z_2, q_{i+k}]$

POP:

$\delta(q_i, a, z_0) = (q_{i+1}, \varepsilon)$

$[q_i, z_0, q_{i+1}] \longrightarrow a$

READ:

$\delta(q_i, a, z_0) = (q_{i+1}, Z_1)$

$[q_i, z_0, q_{i+m}] \longrightarrow a\,[q_{i+1}, Z_1, q_{i+m}]$

No. of Variables : $V = Q^2 M + 1$ [Q → No. of states, M → No. of stack symbols]

1) **Convert PDA to CFG**

$P = (\{p, q\}, \{0, 1\}, \{x, z\}, \delta, q, z)$ where δ P defined by

$\delta(q, 1, z) = (q, xz)$, $\delta(q, 1, x) = (q, xx)$, $\delta(q, \varepsilon, x) = (q, \varepsilon)$

$\delta(q, 0, x) = (p, x)$, $\delta(p, 1, x) = (p, \varepsilon)$, $\delta(p, 0, z) = (q, z)$.

Solution:

$V = \{ S, \overset{A}{[pxp]}, \overset{B}{[pxq]}, \overset{C}{[qxp]}, \overset{D}{[qxq]}, \overset{E}{[pzp]}, \overset{F}{[pzq]}, \overset{G}{[qzp]}, \overset{H}{[qzq]} \}$

[$V = Q^2M + 1 = 2^2 \times 2 + 1 = 9$]

$T = \Sigma = \{0, 1\}$; $S = S$

Production p:

* For start symbol S, $S \rightarrow [q,z,p] \mid [q,z,q]$

$S \rightarrow G_1 \mid H_1$

1/4 p, p 2/3 p, q q, p q, q

* For transition functions, (i) Push

$\delta(q,1,z) = (q, xz)$

$[q,z,p] \rightarrow 1\,[q,x,p]\,[p,z,p]$ ∴ $G_1 \rightarrow 1CE$

$[q,z,p] \rightarrow 1\,[q,x,q]\,[q,z,p]$ ∴ $G_1 \rightarrow 1DG_1$

$[q,z,q] \rightarrow 1\,[q,x,p]\,[p,z,q]$ $H \rightarrow 1CF$

$[q,z,q] \rightarrow 1\,[q,x,q]\,[q,z,q]$ $H \rightarrow 1DH$

ii) Push

$2 \times 2 = 4$ Productions (xx, (p,q))

$\delta(q,1,x) = (q,xx)$

$[q,x,p] \rightarrow 1\,[q,x,p]\,[p,x,p]$ ∴ $C \rightarrow 1CA$

$[q,x,p] \rightarrow 1\,[q,x,q]\,[q,x,p]$ $C \rightarrow 1DC$

$[q,x,q] \rightarrow 1\,[q,x,p]\,[p,x,q]$ $D \rightarrow 1CB$

$[q,x,q] \rightarrow 1\,[q,x,q]\,[q,x,q]$ $D \rightarrow 1DD$

iii) Pop

$2^0 = 1$ production

$\delta(q,\varepsilon,x) = (q,\varepsilon)$

$[q,x,q] \rightarrow \varepsilon$ ∴ $D \rightarrow \varepsilon$

iv) Read

$2^1 = 2$ Productions

$\delta(q,0,x) = [p,x)$

$[q,x,p] \rightarrow 0\,[p,x,p]$ $C \rightarrow 0A$

$[q,x,q] \rightarrow 0\,[p,x,q]$ $D \rightarrow 0B$

v) Pop:

$\delta(p,1,x) = (p,\varepsilon)$

$[p,x,p] \rightarrow 1$ $A \rightarrow 1$

vi) Read :

$\delta(p,0,z) = (q,z)$

$[p,z,p] \rightarrow 0\,[q,z,p]$

$[p,z,q] \rightarrow 0\,[q,z,q]$

$E \rightarrow 0G$

$F \rightarrow 0H$

$S \rightarrow G/H$

$G \rightarrow 1CE/1DG$

$H \rightarrow 1CF/1DH$

$C \rightarrow 1CA/1DC/0A$

$D \rightarrow 1CB/1DD/0B/\epsilon$

$A \rightarrow 1$; $E \rightarrow 0G$; $F \rightarrow 0H$

Since **B** has no transition (Production), Remove **B**,

$S \rightarrow G/H$

$G \rightarrow 1CE/1DG$

$H \rightarrow 1CF/1DH$

$C \rightarrow 1CA/1DC/0A$

$D \rightarrow 1DD/\epsilon$

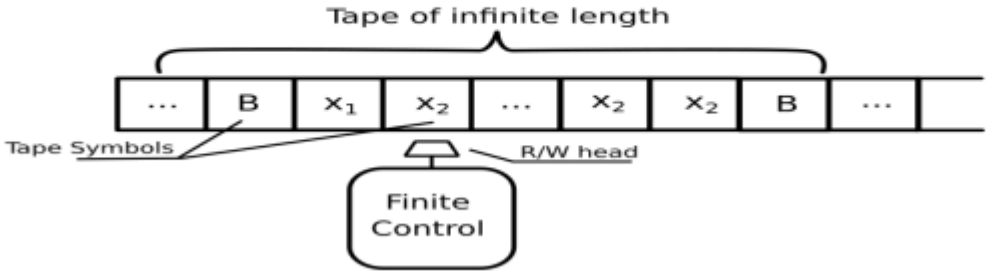
$A \rightarrow 1$; $E \rightarrow 0G$; $F \rightarrow 0H$

INTRODUCTION – TURING MACHINE (TM):

- During the year 1936, Alan Turing introduced a new mathematical model called Turing Machine
- Turing Machine is an abstract machine (or) mathematical model to represent a real computer.
- Turing Machine is a tool, for studying the computability of mathematical function.
- Turing Hypothesis believed that a function is computable if and only if it can be computed by turing machine.
- Turing machine can solve any problem that a modern computer can solve.
- Turing machine is used to define the language and to compile the integer functions.
- Turing machine accepts recursive language or recursive enumerable language.

- Turing machine differs from PDA and FA.
- FA has finite memory and PDA has infinite memory and access in LIFO order
- But TM has both infinite memory and no restriction in accessing the input.

1.3. Turing machine proposal



- Tape head is centered on one of the squares of the tape.
- Tape head reads the symbol in the current square(Fig 4.1)
- Moving the tape head one square to the

  Left | Right | Stationary => I | R | S

2. Definition of TM

A TM can be formally described as a 7-tuple abstract machine

$(Q, \Gamma, \sum, \delta, q_0, B, F)$

where –

Q- Finite set of states

Γ – Finite set of allowable tape symbol

∑ - Set of input symbol

B – Symbol of Γ - blank symbol(Δ)

q0 – Start state

F – Final state

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$

EX: $\delta (q1, x) = (q2, y, D)$

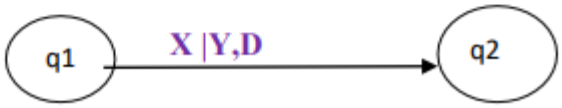


Figure 4.2 Sample Transition

- From state q1 with x, replace X |Y , go to state q1, and move the tape head either D = { L,R, S}(Fig 4.2)

Turing machine can

(i) **Crash:** If in this situation D=L but the tape head is scanning square 0, the leftmost square, the tape head is not allowed to move.

(ii) **Halt:** r=h, the move causes the turing machine to halt.

Turing Machine can be represented by

1. Transition Table
2. Instantaneous Description
3. Transition Diagram

2.1. Instantaneous Description (ID)

Instantaneous Description of a turing machine is given by $\boldsymbol{\alpha_1 q \alpha_2}$

where, q- is the current state of M, $q \epsilon Q$

$\alpha_1, \alpha_2 \epsilon \Gamma^*$ - the contents of the tape upto the rightmost non blank symbol.

Initial ID: $q_0 \alpha_1 \alpha_2$

Final ID: $\alpha_1 \alpha_2 q_B$

Turing Machine can do one of the following things:

(i) Halt and accept by entering into the final state.

(ii) Halt and reject (δ is not defined)

(iii) Turing machine will never halt and enters into an infinite loop.

2.2. construction of Turing Machines

1. Obtain TM to accept the language

$L = \{ 0^n 1^n | n >= 1 \}$

Solution:

$W = \{01,0011,000111,\ldots\ldots\}$

Δ	0	0	1	1	Δ	Δ	Δ	Δ	Δ	Δ

Execution Procedure:

Δ0 0 1 1 Δ

Δ 0 0 1 1 Δ

Δ X 0 1 1 Δ

B X 0 1 1 B

B X 0 Y 1 B

B X 0 Y 1 B

B X 0 Y 1 B

B X X Y 1 B

B X X Y 1 B

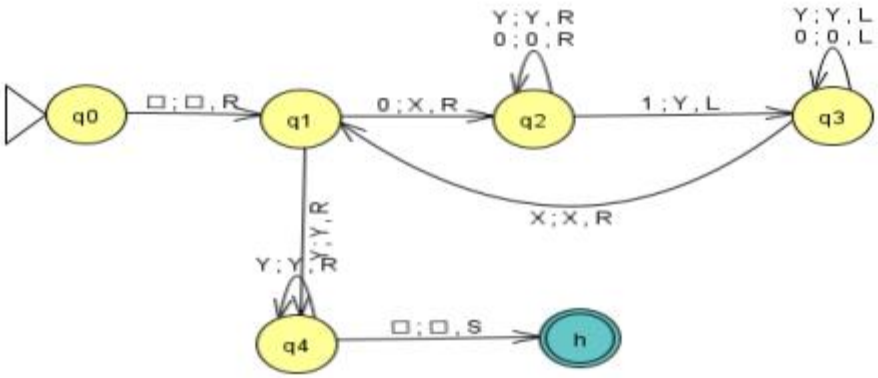
B X X Y Y B

B X X Y Y B

B X X Y Y B

B X X Y Y B

B X X Y Y B



Define Tuples

q_s-start state

Q={qs,qo,q1,q2,q3,h}

F={h}

∑={0,1}

Γ={0,1,X,Y,B}

δ : Transition table :

state	0	1	X	Y	B
Qs	-	-	-		(q0,B,R)
q0	(q1,X,R)			(q3,y,R)	
q1	(q1,0,R)	(q2,Y,L)		(q1,Y,R)	
q2	(q2,0,L)		(q0,X,R)	(q2,Y,L)	
q3				(q3,y,R)	(h,B,S)

Sequence of Moves: 0011 (ID)

(q0, B 0 0 1 1 B) |- (q1, B 0 0 1 1B)

|- (q2, B X 0 1 1 B)

|- (q2, B X 0 1 1 B)

|- (q3, B X 0 Y 1 B)

|- (q3, B X 0 Y 1 B)

|- (q1, B X 0 Y 1 B)

|- (q2, B X X Y 1 B)

|- (q2, B X X Y 1 B)

|- (q3, B X X Y Y B)

|- (q3, B X X Y Y B)

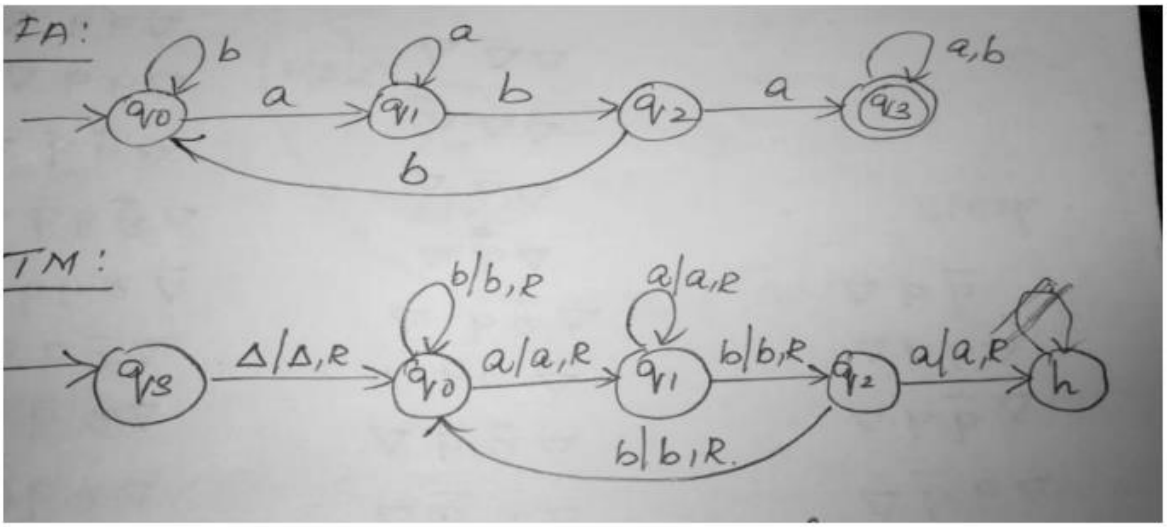
|- (q1,B X X Y Y B)

|- (q4, B X X Y Y B)

|- (q4, B X x Y Y B)

|- (q5, B X X Y Y B)

3. L= { x ∈ {a,b} * | x contains the sub string aba }



2. Construct a Turing Machine to accept palindrome over {a,b}

- Even palindrome - abba
- Odd palindrome- aba
- Not a palindrome -abb

Even Palindrome: abba

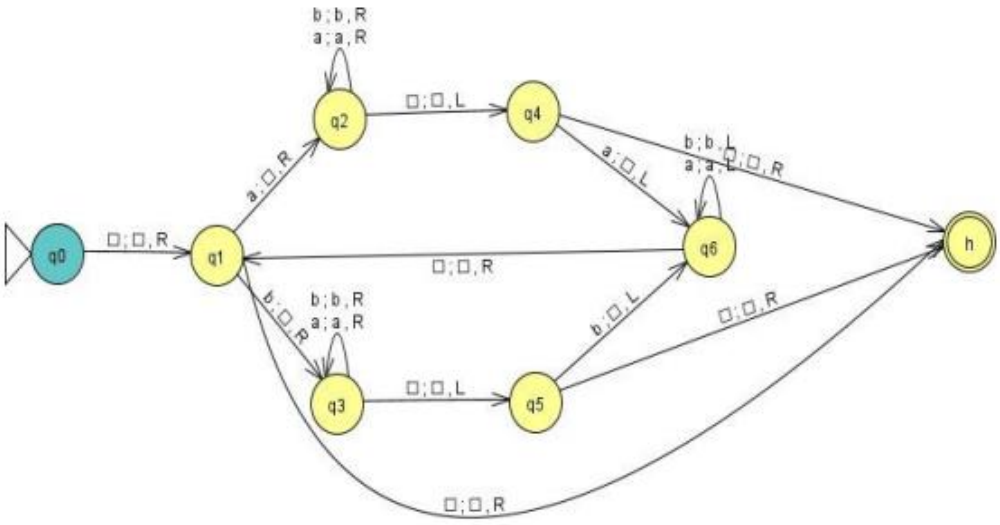
Δ	a	b	b	a	Δ	Δ	Δ	……..

Odd Palindrome: aba

Δ	a	b	a	Δ	Δ	Δ	Δ	Δ	………..

Not a Plalindrome: abb

Δ	a	b	b	Δ	Δ	Δ		……



Define Tuples

q_s-start state

Q={qs,q0,q1,q2,q3,q4,q5,h }

F={h }

∑={a,b}

Γ={ a,b,Δ}

δ : Transition table :

state	a	b	Δ
qs	**-**	**-**	**(q0, Δ,R)**
q0	**(q1, Δ,R)**	**(q4, Δ,R)**	**(h, Δ,R)**
q1	**(q1,a,R)**	**(q1,b,R)**	**(q2,B,L)**
q2	**(q3, Δ,L)**		**(h, Δ,R)**
q3	**(q3,a,L)**	**(q3,b,L)**	**-**
q4	**(q4,a,R)**	**(q4,b,R)**	**(q6, Δ,L)**
q5	**-**	**(q3, Δ,L)**	**(h, Δ,R)**

Instantaneous Description:

String: aba

Sequence of Moves for **aba**

(qs, Δ aba Δ) ⊢ (q0, Δ a b a Δ)

⊢ (q1, Δ Δ b a Δ)

⊢(q1, Δ Δ b a Δ)

⊢(q1, Δ Δ b a Δ)

⊢(q2, Δ Δ b aΔ)

⊢(q3, Δ Δ bΔΔ)

⊢(q3, Δ Δ bΔΔ)

⊢(q0, Δ Δ bΔΔ)

⊢(q4, Δ Δ ΔΔΔ)

⊢(q5, Δ Δ ΔΔΔ)

⊢(h, Δ Δ ΔΔΔ)

Accepted

---

7. Design a TM to perform 1's complement of a no. over $\Sigma = \{0,1\}$.

SOLUTION:

On Reading the I/P;

→ If the symbol = 0 replaces it by '1' & move right

→ If the symbol = 1 replace it by '0' & move right

→ Perform step 1 & 2 until the i/p symbols are processed from left to right

→ Halt the m/c when it encounters the 1st Blank symbol.

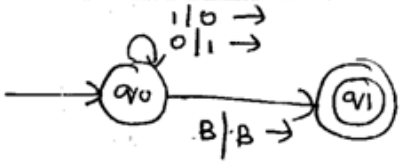
Example: 1011 ⟶ 0100

I/P O/P

STEP 2: TRANSITION TABLE:

	0	1	B
→ q_0	$(q_0, 1, R)$	$(q_0, 0, R)$	(q_1, B, R)
* q_1	(-	-	-

STEP 3: TRANSITION DIAGRAM.



STEP 4: TM Definition $M=(\{q_0, q_1\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_1\})$

STEP 5: Lp. $w=101$

$\delta(q_0, 101B) \vdash_m (q_0101B) \vdash_m (0q_001B) \vdash_m (01q_01B) \vdash_m (010q_0B)$

$\vdash_m (010Bq_1)$

String accepted and is complement is implemented.

6. Design the TM to compute the fn $f(w) = wcw^R$. where w is any string of a's & b's.

SOLUTION:

STEP 1: IDEA OF CREATION.

· The idea to create this TM is that to read the string w and to create wcw^R.

→ Here we initially read all the symbols in the string w upto 'B' and then moves on the left one position and symbol.

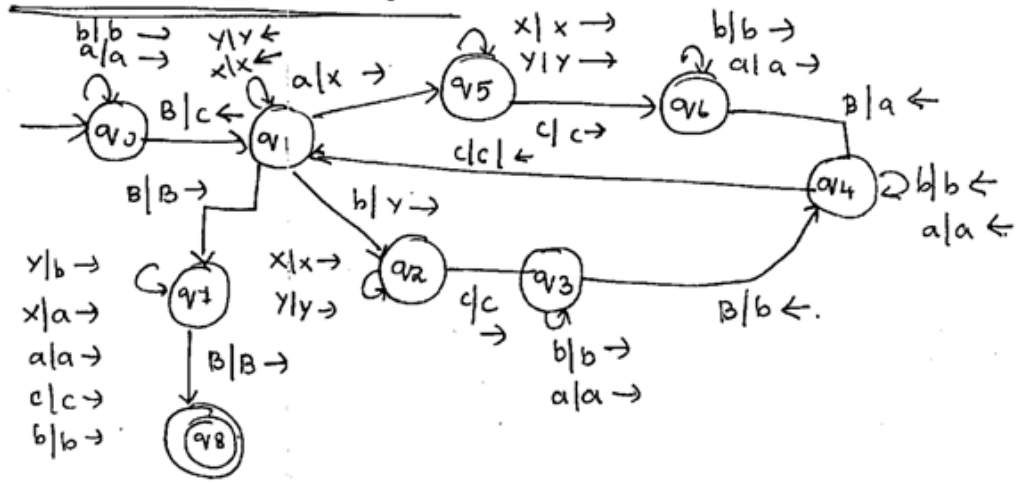
→ If the symbol is 'a', then we replace it by x and if the symbol is 'b', it is replaced by Y.

→ After replacing the symbol, we move to the right and replace B by 'a' or 'b' based on the symbol read before the B.

→ After processing all the strings w and we replace 'x' by 'a' and 'Y' by b.

→ After replacing the entire string symbol in 'w', we move to the right side until blank symbol.

STEP 2: Transition Diagram.



Rejecting state

$\delta(q_1, a) = (q_{reject}, a, L)$

$\delta(q_1, b) = (q_{reject}, b, L)$.

STEP 3 : TRANSITION TABLE

STEP 5: ID - any string.

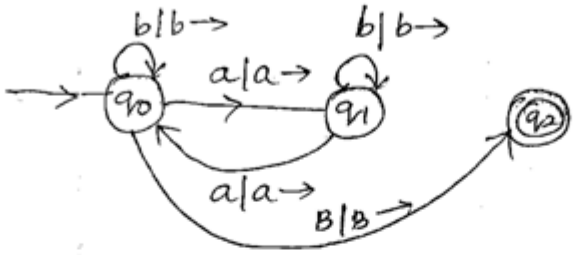
STEP 4: TM Definition

q_7, q_8

$M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{a, b, c\}, \{a, b, B\}, \delta, q_0, B, \{q_8\})$

Qn: Design the TM to accept the set of all strings over alphabet {a, b} with even number of a's.

Solution:

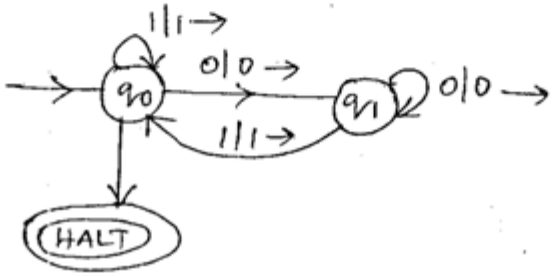


TM $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, B\}, \delta, q_0, B, \{q_2\})$

Qn: Design a TM that accepts the language of odd integers written in binary.

Soln:

Logic: The binary string that ends with 1 is always an odd integer. Hence the TM will be



9. Construct a turing machine for subtraction

f(m,n) = m-n , m>n

0 , m≤n

Case 1: m>n

Input tape:

Δ	0	0	0	1	0	0	Δ	…..

Output Tape:

Δ	Δ	Δ	Δ	0	Δ	Δ	Δ	…..

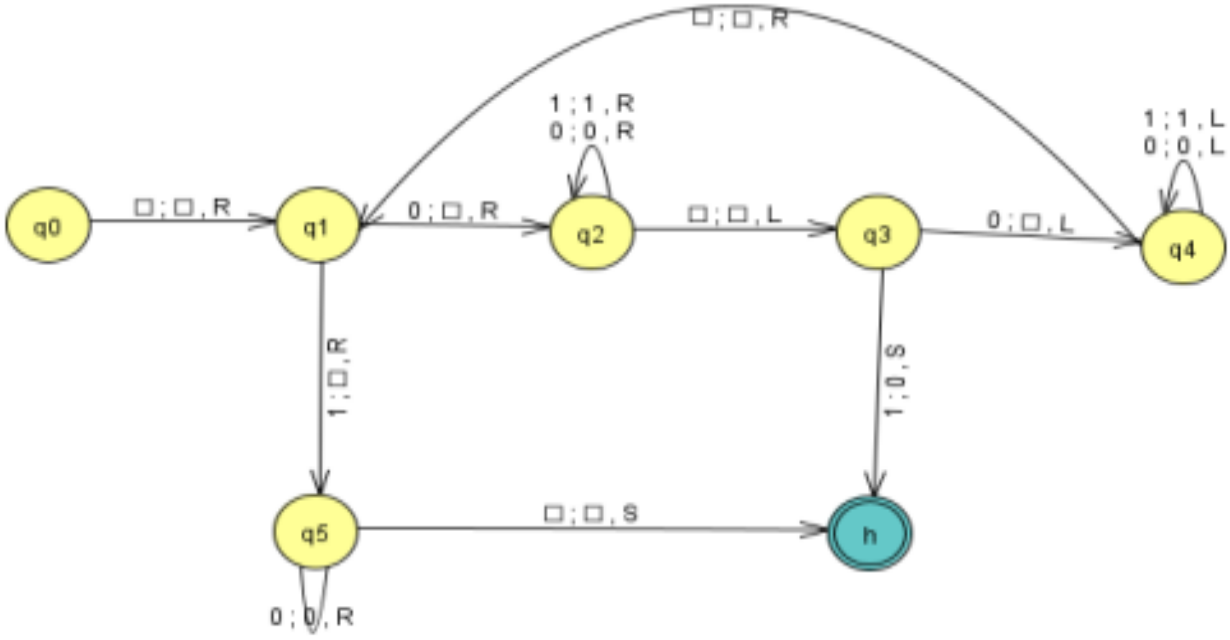
Case 2: m<=n

Input tape:

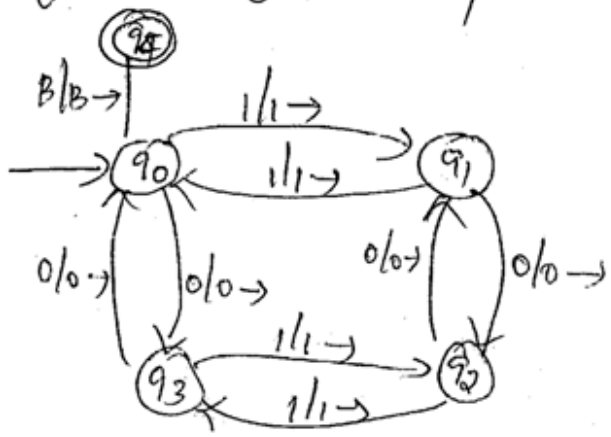
Δ	0	0	1	0	0	0	Δ	Δ	…..

Output Tape:

Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	…..

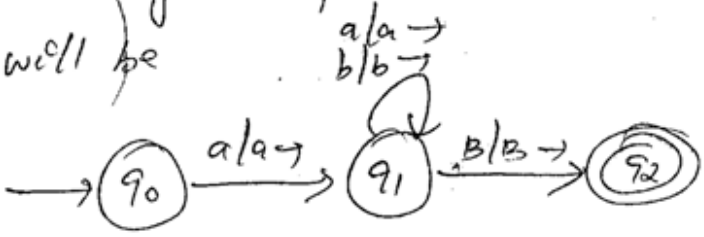


Design a TM to accept the string with even number of 0's & 1's over the alphabet {0 & 1.



Design a TM with not more than three states that accepts the language $a(a+b)^*$. Assume $\Sigma = \{a, b\}$.

Soln Let Regular Expression = $a(a+b)^*$ The corresponding TM will be



4. Design a TM to implement the concatenation function $f(x, y) = xy$ (or) to implement addition function $f(x, y) = x+y$

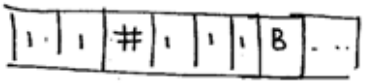
SOLUTION:

STEP1:

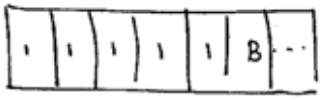
Let us assume that x is represented by the 1^x and y is represented by 1^y in the input tape. The 1^x and 1^y is separated by the separator symbol '#' and is shown below.

$x=2$ $y=3$

Input :



output :



$x+y = 2+3 = 5$

The sum of 2 values are performed by replacing the last '1' by Blank symbol and the step 3 are as follows :

a. At initial state q_0, when it reads '1', it skips the 1's and remain in the same state.

b. At state q_0, when it reads '#' it reaches the state q_1 and changes '#' to '1' and moves right

c. At state q_1, it skips all 1's and searches for 'B' by moving righ

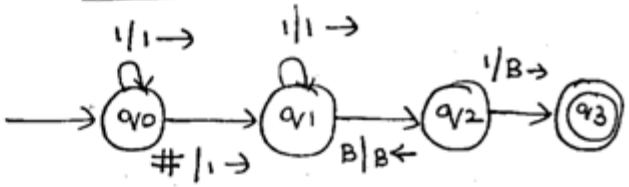
d. At state q_1, when it sees blank symbol, it moves left and changes state to q_2.

e. At state q_2, when it finds '1' it replaces '1' to B and enters the Final state q_3.

STEP 2: TRANSITION TABLE

State	1	#	B
→ q_0	$(q_0, 1, R)$	$(q_1, 1, R)$	-
q_1	$(q_1, 1, R)$	-	(q_2, B, L)
q_2	(q_3, B, R)	-	-
* q_3	-	-	-

STEP 3: TRANSITION DIAGRAM.



TM for $f(x,y) = x+y$.

STEP 4: TM definition $M = (\{q_0, q_1, q_2, q_3\}, \{1\}, \{1, \#, B\}, \delta, q_0, B, \{q_3\})$,

STEP 5: ID EXAMPLE $x=2$ $y=3$

$\delta(q_0, 11\#111B) \vdash_m (q_0 11\#111B) \vdash_m (1q_0 1\#111B) \vdash_m (11q_0\#111B)$

$\vdash_m (111q_1 111B) \vdash_m (1111 q_1 11B) \vdash_m (11111q_1 1B) \vdash_m (111111 q_1 B)$

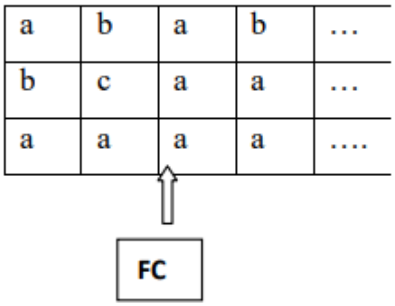
$\vdash_m (11111 q_2 1) \vdash_m (11111 B q_3 B)$

String accepted. The function $f(x,y) = x+y$ is implemented

3.Variations of Turing Machine

1. Multiple track Turing Machine:

- A k-tack Turing machine(for some k>0) has k-tracks and one R/W head that reads and writes all of them one by one.
- A k-track Turing Machine can be simulated by a single track Turing machine



Example: Design a TM using multiple tracks to check whether the given input number is prime or not.

Soln:

* Store the i/p symbol in the 1st track of i/p tape
* Store the number 2 in binary in the 2nd tracks of i/p tape
* Copy the i/p in the 3rd Track also.
* All the symbols in the three tracks of the TM are in binary form.
* Now subtract the 2nd Track from third Track until we get '0' or any remainder.

* If the remainder is zero, then the number is not prime, since the prime number is one which is divided by 1 and itself.

* If the remainder is non-zero value, then the 2nd track value is incremented by 1 and again subtraction procedure is continued.

* If the value of the 2nd & 1st track is equal, then the number is prime number. Let us take an i/p value 5 and it is stored as,

Track 1	1	0	1	B	...
Track 2	B	1	0	B	...
Track 3	1	0	1	B	...

Divide the value 2 in 2nd track from value in 3rd track

1	0	1	B	...	
B	1	0	B	...	→
1	0	1	B	...	

1	0	1	B	..	
B	1	0	B	..	→
0	1	1	B	..	

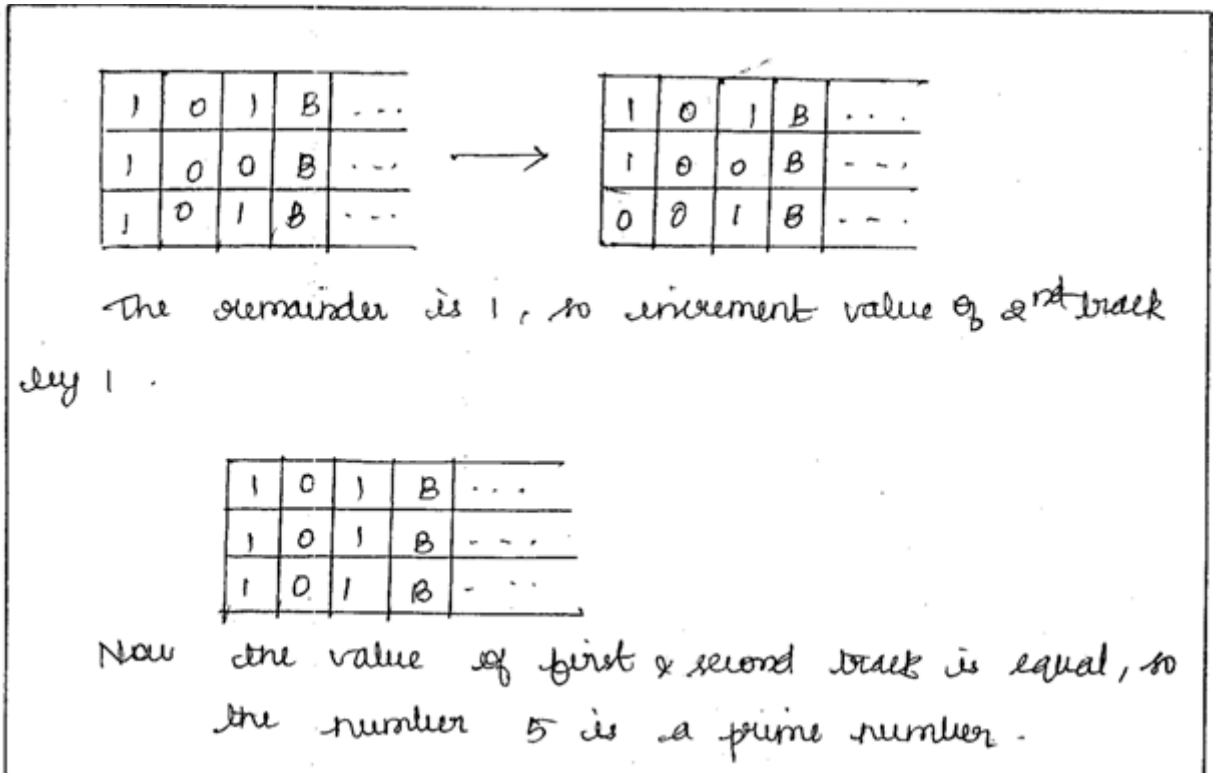
1	0	1	B	..	
B	1	0	B	..	
B	0	1	B	..	

The remainder is 1, so increment the value of 2nd track by 1.

1	0	1	B	...	
B	1	1	B	...	→
1	0	1	B	..	

1	0	1	B	...	
B	1	1	B	...	
0	1	0	B	...	

The remainder is 2, so increment the value of 2nd track

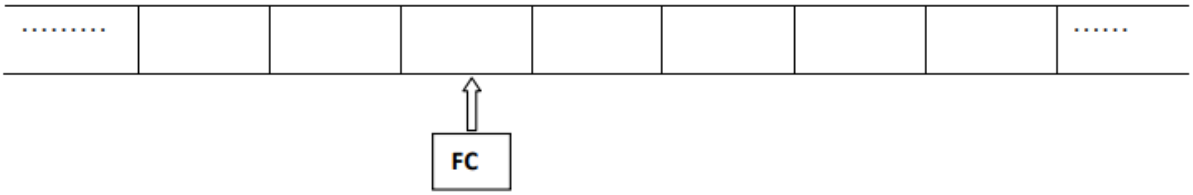


---

2. Two-way infinite Tape Turing Machine:

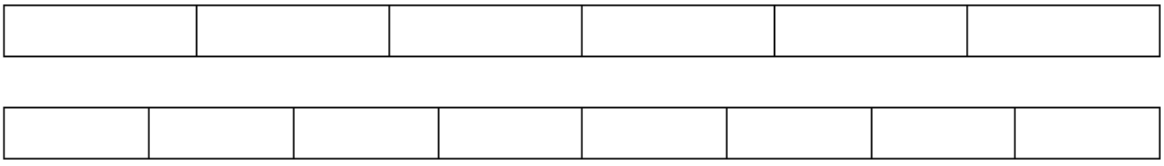
A two way infinite tape turing machine is a turing machine with it's a input tape infinite in both directions, the other components being the same as that of the basic model.

- Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.
- Two-way infinite tape Turing machine can be simulated by one-way infinite Turing machine (standard Turing machine).

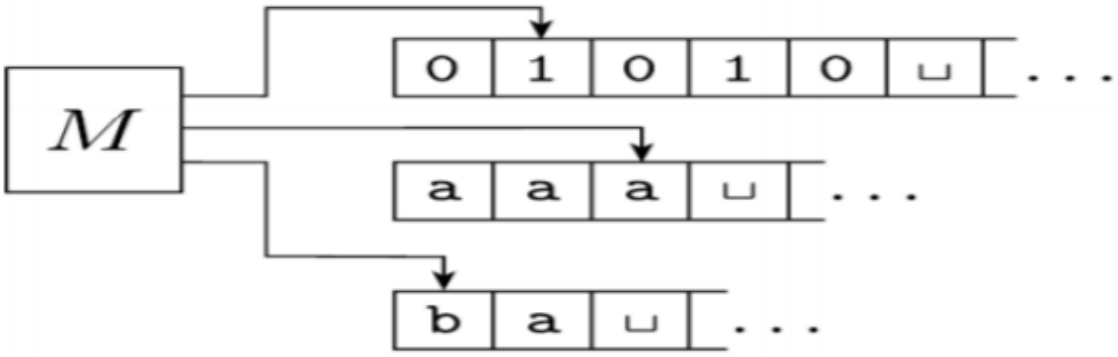


3. Multi-tape Single-head Turing Machine:

- It has multiple tapes and controlled by a single head.
- The tape head scans the same position on all tapes.



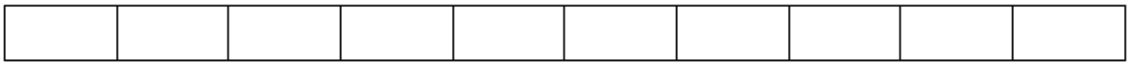
4. Multi-tape Multi-head Turing Machine:



- The multi-tape Turing machine has multiple tapes and multiple heads
- Each tape controlled by separate head
- Multi-Tape Multi-head Turing machine can be simulated by standard Turing

5. Multi-head Turing Machine:

- A multi-head Turing machine contain two or more heads to read the symbols on the same tape.
- In one step all the heads sense the scanned symbols and move or write independently.
- Multi-head Turing machine can be simulated by single head Turing machine.



6. Non-deterministic Turing Machine:

- A non-deterministic Turing machine has a single, one way infinite tape.
- For a given state and input symbol has atleast one choice to move (finite number of choices for the next move), each choice several choices of path that it might follow for a given input string.
- A non-deterministic Turing machine is equivalent to deterministic Turing machine.

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

4. Universal Turing Machine

A UTM is a specified Turing machine that can simulate the behavior of any TM.

A UTM is capable of running any algorithm.

It is a Turing Machine whose input consists of 2 parts:

- A string specifying some special purpose Turing Machine, T1.
- A string Z that is an input to T1.

The Turing Machine, Tu then simulates the processing of Z by T1.

4.1. Construction of Tu: $\sum=\{0,1\}$

Step 1: Formulate a notational system

- For each tape symbol (including Δ) as string of 0's
- For each state (including h)
- 3 directions
- For beginning of string and ending of string – 11
- For comma, encoding is 1

e - encoding function

Tu – represents the Universal Turing Machine

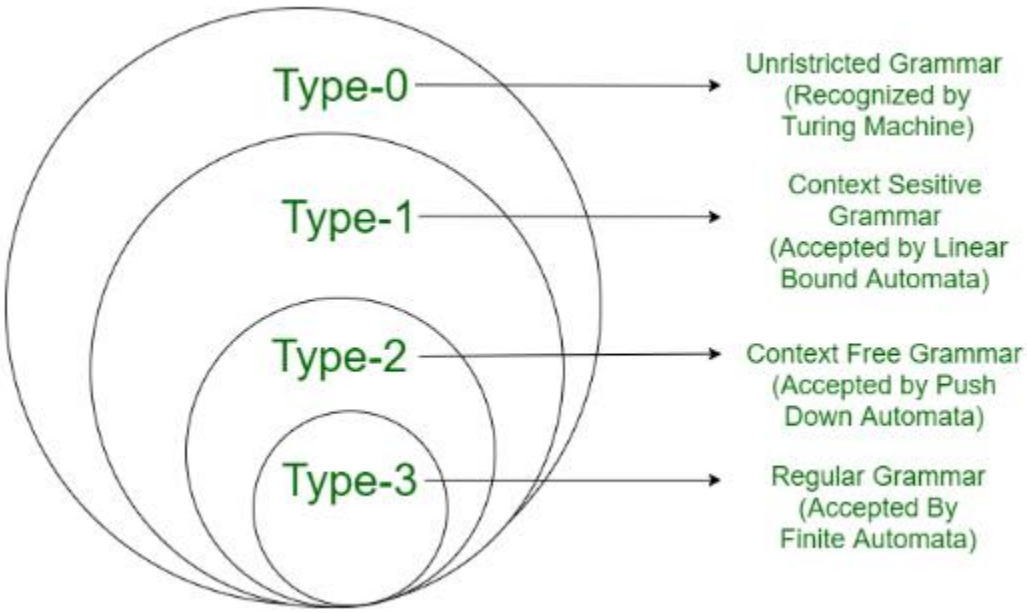
T1 – represents the name of the special Turing machine

Tu=e(T1).e(Z)

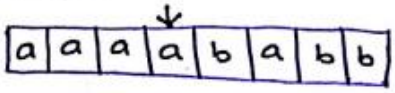
Chomsky Hierarchy :-

The chomsky Hierarchy as originally defined by Noam chomsky, comprises four types of languages and their associated grammars and machines.

Language	Grammar	Machine	Example
Regular Language	Regular Grammar	Finite Automata	a^*
Context-Free Language	Context free Grammar	Push down Automata	$a^n b^n$
Context Sensitive Language	Context Sensitive Grammar	Linearly bounded Automata	$a^n b^n c^n$
Recursively Enumerable Language	Unrestricted Grammar	Turing Machine	Any Computable function

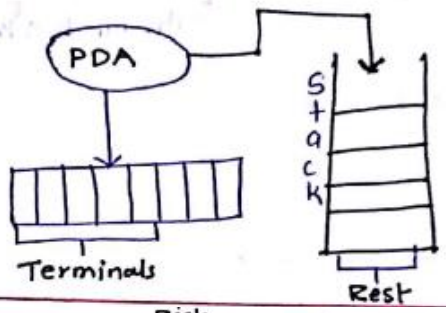


FSM : The Input String



PDA : → The Input String

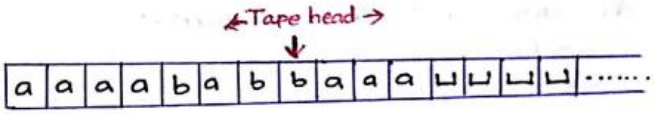
→ A Stack



Rest

TURING MACHINE:

→ A Tape

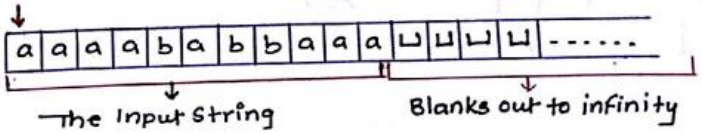


Tape Alphabets : $\Sigma = \{0, 1, a, b, x Z_0\}$

The Blank ⊔ is a special Symbol $\quad ⊔ \notin \Sigma$

The blank is a special symbol used the fill the infinite tape.

Initial configuration:



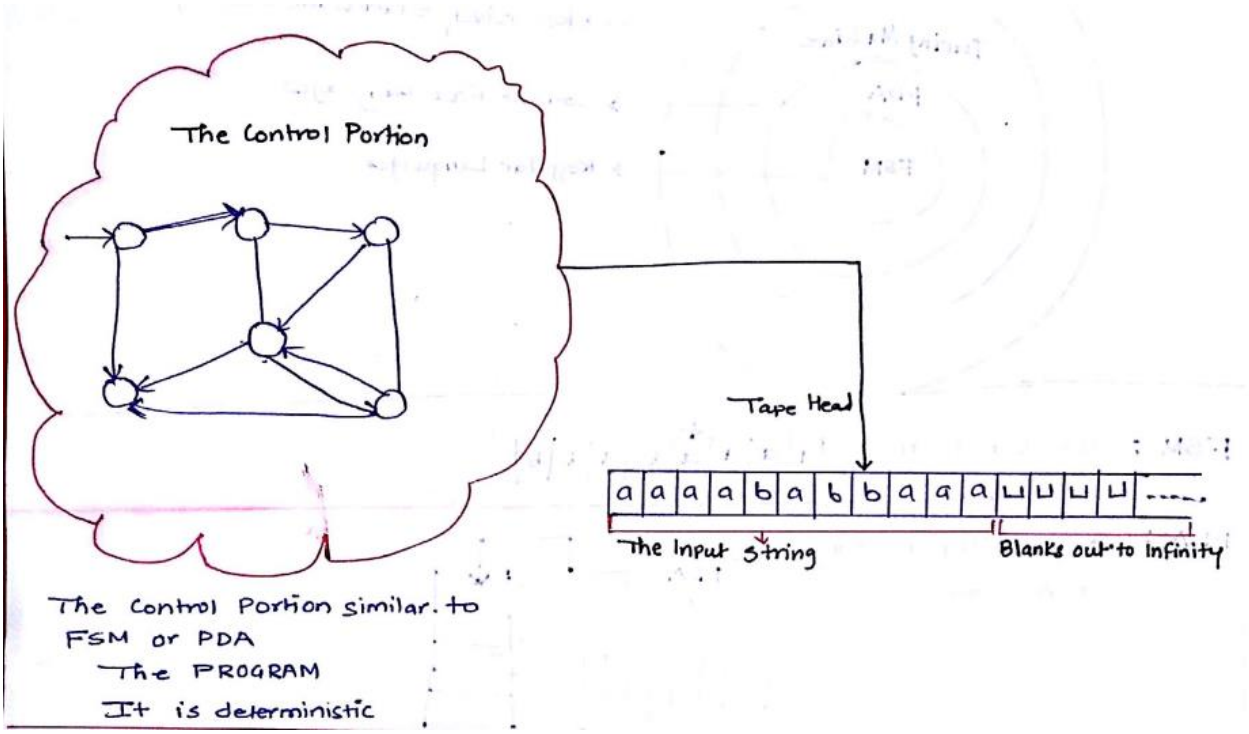
Operations on the Tape :

→ Read / Scan symbol below the Tape Head.

→ Update / Write a Symbol below the TapeHead

→ Move the Tape Head one step LEFT

→ Move the Tape Head one step RIGHT.



Type 0 Grammar:

- Type-0 grammars include all formal grammar. Type 0 grammar languages are recognized by turing machine. These languages are also known as the Recursively Enumerable languages.
- Grammar Production in the form of $\alpha \rightarrow \beta$

  where \alpha is $(V + T)^* V (V + T)^*$

  V : Variables T : Terminals.

  β is $(V + T)^*$.
- In type 0 there must be at least one variable on the Left side of production.

Type 1: Context-Sensitive Grammar:

Type 1 grammar is known as Context Sensitive Grammar. The context sensitive grammar is used to represent context sensitive language. The context sensitive grammar follows the following rules:

- The context sensitive grammar may have more than one symbol on the left hand side of their production rules.
- The number of symbols on the left-hand side must not exceed the number of symbols on the right-hand side.
- The rule of the form A → ε is not allowed unless A is a start symbol. It does not occur on the right-hand side of any rule.

- The Type 1 grammar should be Type 0. In type 1, Production is in the form of V → T

  Where the count of symbol in V is less than or equal to T.

  $\alpha \rightarrow \beta$

  $|\alpha| <= |\beta|$

  Also $\beta \in (V + T)^+$

  i.e. β can not be ε.

- For Example:

  S --> AB

  AB --> abc

  B --> b

Type 2: Context-Free Grammar:

- Type-2 grammars generate context-free languages. The language generated by the grammar is recognized by a Pushdown automata. In Type 2:
- First of all, it should be Type 1.
- The left-hand side of production can have only one variable and there is no restriction on β.
- The production rule is of the form

  $A \rightarrow \alpha$

  Where A is any single non-terminal and is any combination of terminals and non-terminals.

- **For example:**

  $A \rightarrow aBb$

  $A \rightarrow b$

  $B \rightarrow a$

Type 3 Grammar:

- Type 3 Grammar is known as Regular Grammar. Regular languages are those languages which can be described using regular expressions. These languages can be modeled by NFA or DFA.
- Type 3 is most restricted form of grammar. The Type 3 grammar should be Type 2 and Type 1.
- Type 3 should be in the given form only :
- **V --> VT / T** (left-regular grammar)

  (or)
- V --> TV /T (right-regular grammar)

- For example:

  S --> a

  The above form is called strictly regular grammar.

PROPERTIES OF RECURSIVE AND RE LANGUAGES

1. The union of two recursive language is recursive
2. The language L and its complement L¯are recursively enumerable, then L is recursive.
3. The complement of a recursive language is recursive.
4. . The Union of two recursively enumerable languages is recursively enumerable.
5. The intersection of two recursive language is recursive.
6. The intersection of two recursively enumerable language is recursively enumerable

Linear Bounded Automata

A linear bounded automaton is a multi-track non-deterministic Turing machine with a tape of some bounded finite length.

Length = function (Length of the initial input string, constant c)

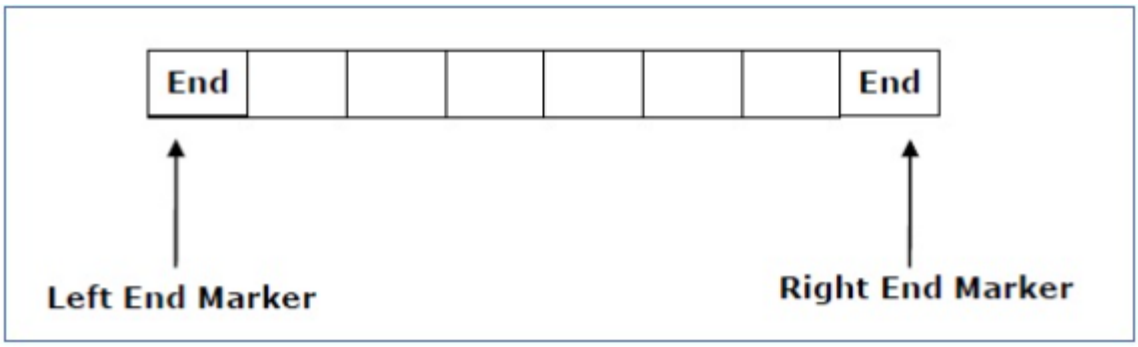
Here,

Memory information ≤ c × Input information

The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape.

A linear bounded automaton can be defined as an 8-tuple (Q, X, Σ, q_0, ML, MR, δ, F) where –

- **Q** is a finite set of states
- **X** is the tape alphabet
- **Σ** is the input alphabet
- $\mathbf{q_0}$ is the initial state
- $\mathbf{M_L}$ is the left end marker
- $\mathbf{M_R}$ is the right end marker where $M_R \neq M_L$
- **δ** is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant 'c') where c can be 0 or +1 or -1
- **F** is the set of final states



A deterministic linear bounded automaton is always **context-sensitive** and the linear bounded automaton with empty language is **undecidable.**.

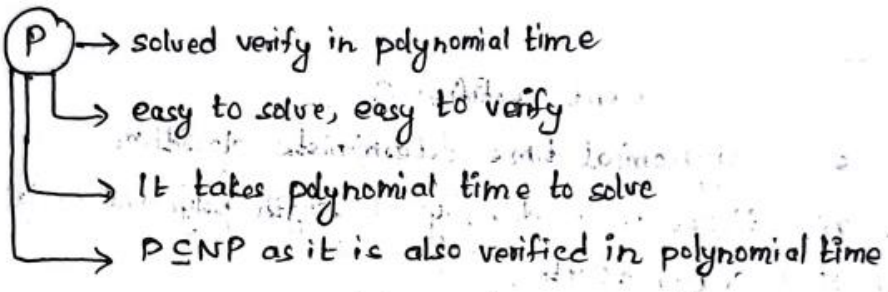
Unit 5

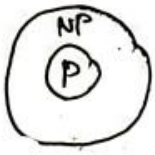
* Class P Problem :-

→ P is a set of problem that can be solved in polynomial time using deterministic algorithm

→ If a problem can be solved in polynomial time on a regular computer, that problem is known as class P problem.

→ As P class problem verified in time also, hence it is also a subset of NP class problem.





→ In complexity theorem P is also known as PTime or DTime is one of the most fundamental classes.

→ It contains all the decision problems that are solved by a deterministic turing machine using polynomial amount of computation time. i.e for a problem with size N, there must some way to solve the problem in f(n) steps.

Defination :-

The Language L is said to be class P, if there exist a Polynomial bounded Turing Machine (Deterministic) such that TM is of time complexity P(n) for some polynomial P and TM accepts L. This Language is also called polynomial decidable.

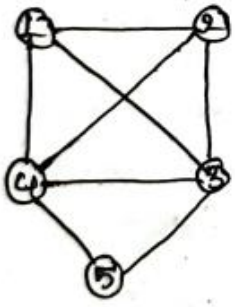
Examples:

- Eulerian and Hamilton graph problem
- Optimization problem
- Integer partition problem

Explanation: Hamilton Cycle

⇒ Hamilton cycle is a path in a given graph that visit each vertex exactly once and there is a edge/path between first and last vertex.

⇒ There may be any number of Hamiltonian cycle exist in a graph



1-2-3-5-4-1
1-3-5-4-2-1

⇒ Here we have to determine whether a given graph contains hamiltonian cycle or not. If it contains then print the path.

⇒ The cyclic condition ensures that the circuit is closed and the requirement that all the nodes are included (with no repeats) ensures that circuit doesnot cross over itself and passes through every node.

* Class NP Problems:-

→ NP is a set of problems that can be solved in polynomial time using non-deterministic algorithms.

→ NP problems are problems which can be solved in polynomial time on non-deterministic turing machine.

→ It can be verified in polynomial time.

→ NP problems can be solved by a polynomial time using "non-deterministic algorithm" a magical algorithm that always makes a right guess among the given set of choices.

→ Example:

Graph Coloring:-

Graph coloring in such a way that each adjacent nodes has different color.

→ This NP class problem is solved in exponential time, but solution verification is easy because once a solution is provided it is much easier to check for it's correctness.

→ Defination:- A language L is in class NP if there is polynomially bounded non-deterministic turing machine such that TM is of time complexity P(n) for some polynomial P and TM accept L.

Example Explanation: Travelling Salesman Problem:

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that a salesman has to visit every city at exactly once and return to the starting point.

For a given graph in which nodes (cities) are connected by the directed edges (routes) where the weight of an edge is the distance between two cities.

⇒ Assume a salesman has travels to n cities.

⇒ He starts from particular city, visiting each city once and then returns to his starting point.

⇒ The objectives is to select the sequence in which the cities are visited (with shortest possible route) in such a way that his total travelling cost is minimized.

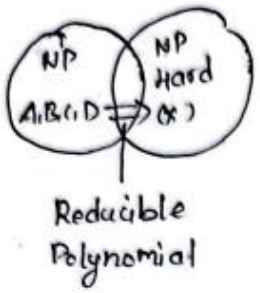
⇒ The only know solution that grows exponentially with the problem size. (i.e number of cities).

⇒ There is an example of 'NP complete problem for which no known efficient (i.e polynomial time) algorithm exists

Np Hard and NP Complete :-

* NP-Hard :-

⇒ A problem is NP-Hard if every problem in NP can be polynomial reduced to it.



X is NP Hard

⇒ A problem is NP-hard of an algorithm for solving it can be translated into one for solving any other NP problem. NP hard therefore means "at least as hard as any NP-problem.

⇒ Although it might in fact be harder. The NP-hard class is the superset containing NP complete class this is potentially harder to solve than NP complete problem.

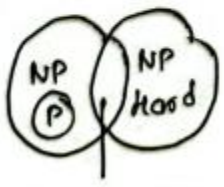
* NP-Complete :-

⇒ A decision Yes/No problem is NP-complete if:

(a) L is in NP

(b) L is in NP-hard

(Every problem in NP is reduced to L in polynomial time)



⇒ The most important property of NP complete problem is polynomial time reducibility. Any NP complete polynomial can be transformed into any other NP complete problem in polynomial time.

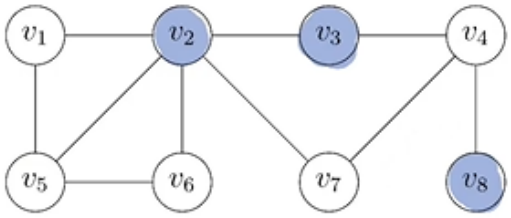
⇒ Thus if it could be proved that, any NP complete problem is formally intractable or such algorithm exist.

⇒ Once we have some NP-complete problem we can prove a new problem to be NP-complete by reducing some known NP-complete problem to it by using polynomial time reduction.

⇒ Examples: Hamilton Cycle problem
Traveling Salesman problem

Vertex Cover

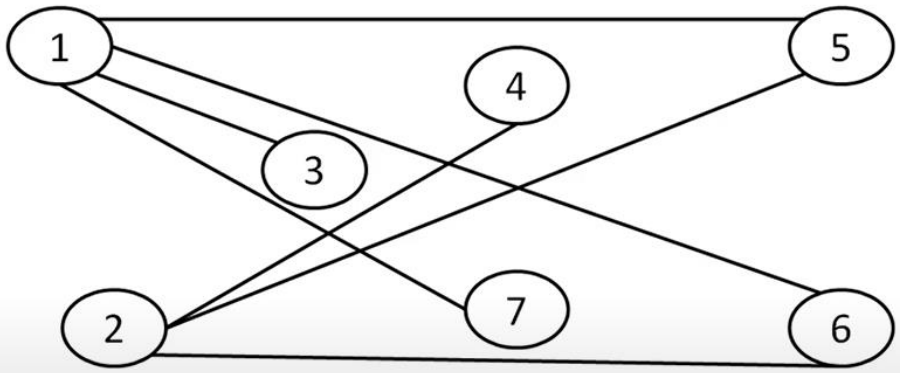
Definition 12. An **vertex cover** is a subset V' of the vertices of graph $G = (V, E)$ such that for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$.



- $\{v_1, v_3, v_5, v_6, v_7, v_8\}$ is a vertex cover.
- $\{v_2, v_4, v_5\}$ is a vertex cover.
- $\{v_2, v_3, v_8\}$ is **not** a vertex cover.

Prove that VERTEX COVER Problem is NP-Complete

- **G=(V, E)**



Prove that VERTEX COVER Problem is NP-Complete

- **Step 1:**Write a polynomial time verification algorithm to prove that the given problem is NP
- **Inputs:** <G, k,V'>
- **Verifier Algorithm**:
  1. count = 0
  2. for each vertex v in V' remove all edges adjacent to v from set E
     1. increment count by 1
  3. if count = k and E is empty then the given solution is correct
  4. else the given solution is wrong
- This algorithm will execute in polynomial time. Therefore **VERTEX COVER problem is a NP problem.**

Prove that VERTEX COVER Problem is NP-Complete

- **Step 2:**Write a polynomial time reduction algorithm from CLIQUE problem to VERTEX COVER problem
- **Algorithm**

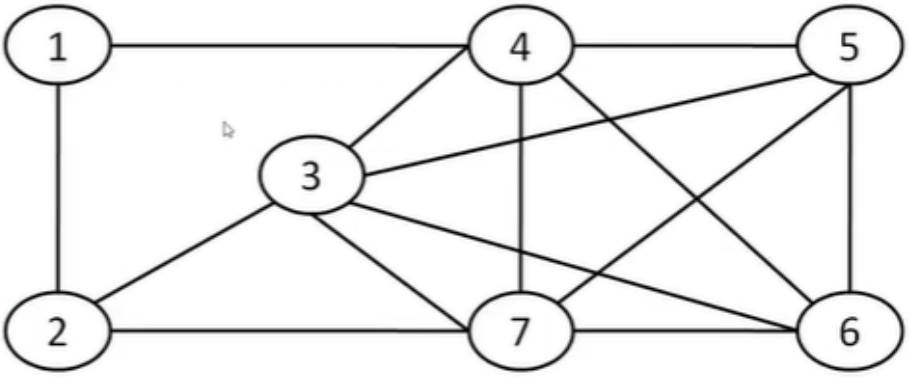
  Inputs: <G=(V,E), k>

  1. Construct a graph G', which is the complement of Graph G
  2. If G' has a vertex cover of size |V| - k, then G has a clique of size k.

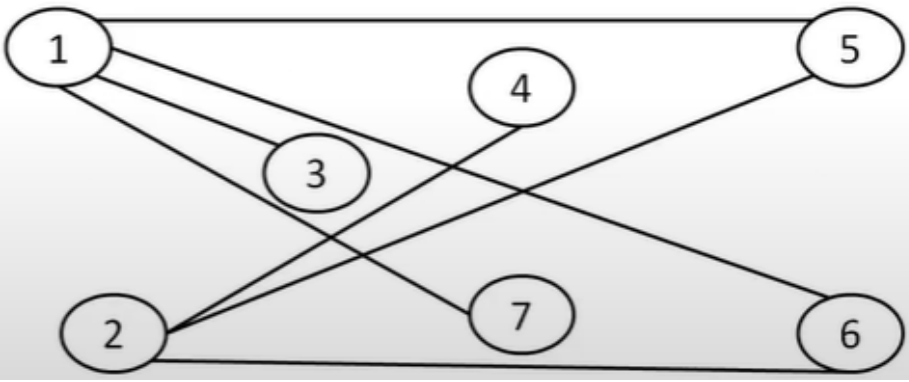
- This reduction algorithm(CLIQUE to VERTEX COVER) is a polynomial time algorithm
- So **VERTEX COVER problem is NP Hard**.

Prove that VERTEX COVER Problem is NP-Complete

- **G=(V, E)**



- **G'=(V, E')**



- **Conclusion**
  - VERTEX COVER problem is NP and NP-Hard.
  - So it is NP-Complete