Unit-1

Introduction to Finite Automata: Structural Representations, Automata and Complexity, the Central Concepts of Automata Theory–Alphabets, Strings, Languages, Problems. Nondeterministic Finite Automata: Formal Definition, an application, Text Search, Finite Automata with Epsilon-Transitions. Deterministic Finite Automata: Definition of DFA, How a DFA Process Strings, The language of DFA, Conversion of NFA with €- transitions to NFA without €-transitions. Conversion of NFA to DFA, Moore and Melay machines.4

AUTOMATA THEORY:

- Automaton = an abstract computing device. A mathematical device which acts as a computer for computation.
- Note: A “device” need not even be a physical hardware.
- The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting".
- Automaton is singular and Automata is plural.

Why study automata theory? or Applications of automata Theory

- The **lexical analyzer** and **Syntax analyzers** of a typical Compiler
- Software for designing and checking the behavior of digital circuits
- Software for scanning large bodies of text such as collections of Web pages to find occurrences of words, phrases or other patterns.
- The software for Natural Language Processing take the help of an automata theory **(Chat boat Application).**

1.1 INTRODUCTION TO FINITE AUTOMATA:

1.1.1. STRUCTURAL REPRESENTATIONOF FINITE AUTOMATA:

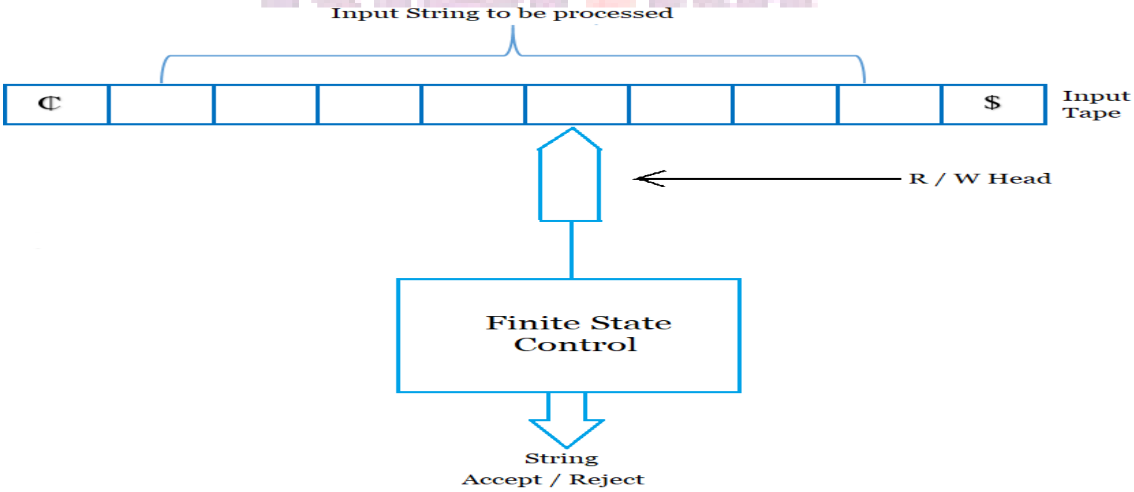


Fig: Block Diagram of Finite Automata

An automaton with a finite no of states is called finite automaton or Finite state machine.
It consists of three components 1) Input Tape 2) Read/Write Head 3) Finite Control

- Input Tape: i) the input tape is divided in to squares, each square contains a single symbol from the input alphabet ∑. ii) The end squares of each tape contain end markers different from symbols of ∑. iii) Absence of end markers indicate the tape is of infinite length. iv) The symbols between end markers is the input string to be processed.
- Read/Write Head: The R/W head examines only one square at a time and can move one square either to the left or the right.
- Finite control: Finite control can be considered as the control unit of an FA. An automaton always resides in a state. The reading head scans the input from the input tape and sends it to finite control. In this finite control, it is decided that 'the machine is in this state and it is getting this input, so it will go to this state'. The state transition relations are written in this finite control.

1.1.2. The Central Concepts of Automata Theory

- **Symbol:**A Symbol is an abstract entity. It cannot be formerly defined as points in geometry.
  - Ex: letters or digits or special symbols like !,@,#,$..
- **Alphabet**: A finite set of symbols denoted by ∑.
  - Ex: ∑= {a,b,..z} is called english alphabet
  - ∑={0,1} is called Binary Alphabet
- **String/Word**: Finite sequence of letters from the alphabet. It Is denoted by S or W Ex: S= computer is a string defined over ∑ ={a,b,c,..z}
  Ex: W=010100 is a binary word defined over ∑={0,1}
- **Length of a string**: It is the number of symbols present in a given string. It is denoted by |S|. Ex: S= computer then |S|=8
- **Empty/Null string (ε)** :If |S|= 0 then it is called an empty string. It is denoted by λ or ε.
- **Powers of an alphabet ($\sum^k$):**if ∑ is an alphabet then $\sum^k$ is the set of strings of length k.
  Ex: $\sum^0$ ={ε }, $\sum^1$ ={0,1 }, $\sum^2$ ={00,11,01,10 }
- **Kleene /Star Closure (∑*)**: The infinite set of all possible strings of all possible lengths over ∑ including
  ε.i.e., $\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup$ ....where $\sum^k$ is the set of all possible strings of length k.
  Ex: If ∑ = {a, b} then ∑* = {ε, a, b, aa, ab, ba, bb, ...}
- **Positive Closure ($\sum^+$):**The infinite set of all possible strings of all possible lengths over ∑ excluding
  ε.i.e., $\sum^+ = \sum^1 \cup \sum^2 \cup$ ....where $\sum^k$ is the set of all possible strings of length k.
  Ex: If ∑ = {a, b} then $\sum^+$ = { a, b, aa, ab, ba, bb,… ....}
- **Strings Concatenation:**Let S1 and S2 be two strings. The Concatenation of S1 and S2 is adding the string S2 at the of string S1.
  Ex: S1= Computer, S2=Science then S1S2=ComputerScience and S2S1=Science Computer
- **Language:**A non Empty Subset of ∑* is called a language. It is denoted by L.
  Ex: Let ∑={0,1}
  ∑* ={ε ,0,1,00,11,01,10,111,000,101,011,..}
  L={0,00,10,110,…} is called even binary numbers language.

OPERATIONS ON LANGUAGES:

If L1, L2 are two languages then

Union operation: It is denoted as L1UL2 or L1 + L2 , L1/L2 and is defined as L1UL2={s|s is in L1 or s is in L2}.

Intersection operation: It is denoted as L1∩L2 ,and is defined as L1∩L2={s|s is in L1 and s is in L2}.

Concatenation operation: It is denoted as L1L2 and is defined as L1L2 = { xy | L1 ϵ x and L2 ϵ y}

Difference operation: It is denoted as L1-L2 ,and is defined as L1-L2={s|s is in L1 and s is not in L2}.

Keen Closure operation (L*):It is the language consisting of all words that are Concatenations of 0 or more words in the original language (including null string).

- **Problems in Automata Theory:** It is the question of deciding whether a given string is a member of some particular language. Precisely, if ∑ is an alphabet and L is a language over ∑ then the problem L is a given a string W in ∑* decide whether or not w is in L.

1.1.3. DEF: FINITE AUTOMATA

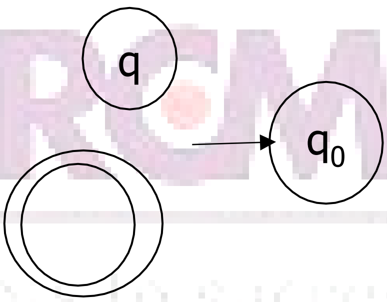
A finite automaton is a collection of 5-tuple M=(Q, ∑, δ, q0, F), where:

- Q: finite set of states
- ∑: finite set of the input symbol
- q0: initial state
- F: Set of final states
- δ: QX ∑ ⟶ Q is a Transition function

1.1.3.1. REPRESENTATION OF FA: Finite automata can be represented in two ways: (i) Graphical representation and (ii) Tabular representation.

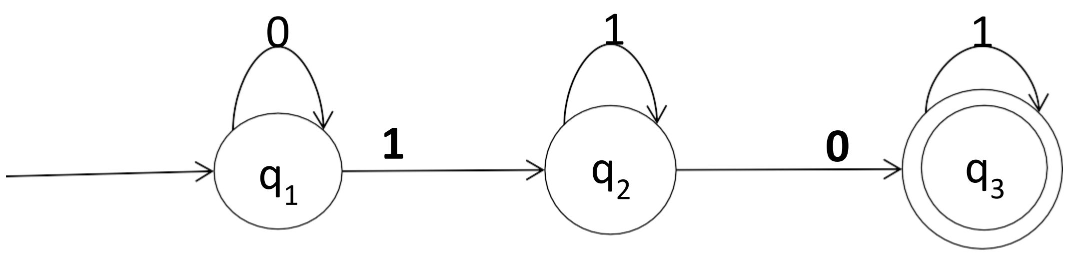
Graphical Representation of FA:

- It is called as transition graph or diagram
- It is a collection of states and transitions
- A state is represented by a circle
- A beginning/initial state is represented as
  q2
- A final state is represented as



- A directed edge indicates the transition from one state to another state and edges are labeled with input symbols.

EX: GRAPHICAL REPRESENTATION OF FA



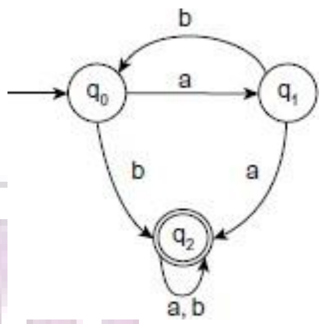
Tabular Representation: Transition table

- It is a table of order mXn.
- First row indicates inputs and first column indicates states and the corresponding entities are outputs of a transition function.
- Start state is marked with arrow and final state is marked with * or circle.

δ	0	1
→ q1	q1	q2
q2	q3	q2
q3	--	q3

Ex: Consider an automata M=(Q, ∑, δ, q0, F) where Q={q0,q1,q2}, ∑={a,b}, F={q2} , δ(q0,a) =q1, δ(q0,b)=q2, δ(q1,a)=q2, δ(q1,b)=q0, δ(q2,a)=q2, δ(q2,b)=q2. Draw transition diagram and transition table.

Δ/∑	a	b
→ q_0	q_1	q_2
q_1	q_2	q_0
*q_2	q_2	q_2



1.1.4. PROPERTIES OF TRANSITION FUNCTION (δ):

- δ(q,ε)=q i.e., If the input symbol is null for a given state q, it remains in the same state.
- For all strings w and input symbol a, δ(q, aw) = δ(δ(q,a),w)

1.1.5. ACCEPTANCE OF A STRING BY FA: A string w is accepted by a finite automata M=(Q,∑,δ,q0,F) if δ(q0,w) = q for some q ϵ F.

Ex: Now let us consider the finite state machine whose transition function δ is given in the form of transition table. Where Q= {q0,q1,q2,q3}, ∑={0,1} & F={q0}.Test whether the string 110101 is accepted or not

δ	0	1
→* q0	q2	q1
q1	q3	q0
q2	q0	q3

- Sol:

δ(q0,110101)= δ(q1,10101)= δ(q0,0101)= δ(q2,101) = δ(q3,01)= δ(q1,1)= δ(q0, ε) = q0

Here q0 is not a final state. Hence the string is rejected.

1.1.6. TYPES OF FINITE AUTOMATA:

There are two types of finite automata DFA-Deterministic Finite Automata
NFA -Nondeterministic Finite Automata

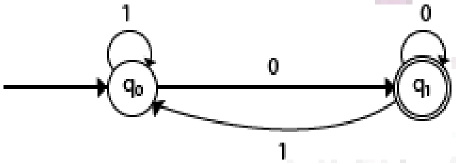
DFA: It refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. In the DFA, the machine goes to one state only for a particular input character. DFA does not accept the null move.

NFA: It is used to transmit any number of states for a particular input. It can accept the null move.

- Some important points about DFA and NFA:
- Every DFA is NFA, but the converse need not be true i.e., every NFA need not be DFA.
- There can be multiple final states in both NFA and DFA.
- DFA is used in Lexical Analysis in Compiler.
- Construction of NFA is easier than the construction of DFA
- To test string is Accepted or not easier in DFA than in NFA

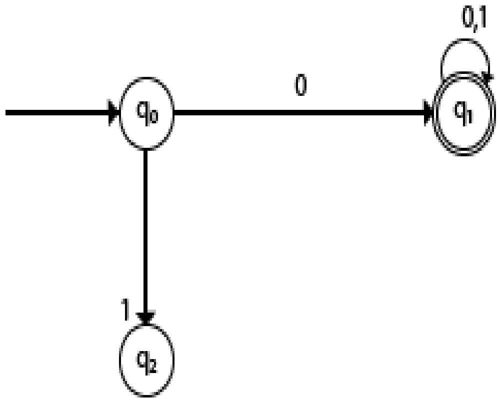
1.2. DETERMINISTIC FINITE AUTOMATA (DFA): A DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where

- Q is a finite set of states.
- ∑ is a finite set of symbols called the alphabet.
- δ is the transition function where δ: Q × ∑ → Q
- q0 is the initial state from where any input is processed (q0 ∈Q).
- F is a set of Final states
  - **Design a DFA which accepts strings ending with 0 defined over ∑ = {0, 1} Transition Diagram: Transition Table:**



δ	0	1
q0	q1	q0
q1	q1	q0

- **Design a DFA to accept all strings starting with 0 defined over ∑ = {0, 1}**
  Transition Diagram: Transition Table:

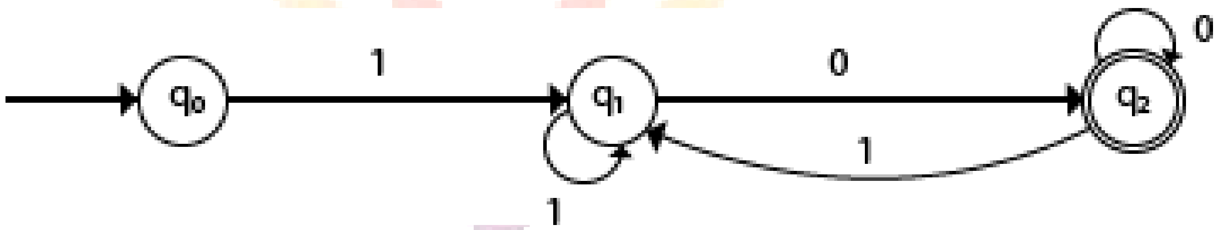


	0	1
q0	q1	q2
q1	q1	q1
q2	--	--

Test whether the string 0101010 is accepted or not

- **Design a FA which accepts strings starts with 1 and ends with 0 defined over ∑ = {0, 1}**

Transition Diagram:



Transition Table:

δ	0	1
→q0		
q1		
*q2		

Test whether the string 11010101 is accepted or not

- **Design a FA which accepts the only input 101 defined over ∑ = {0, 1}**

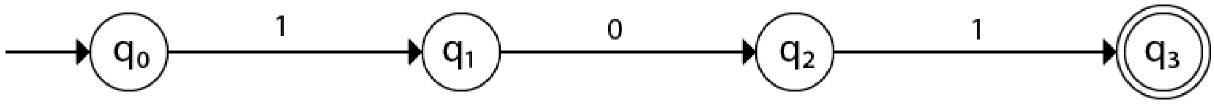


Fig: FA

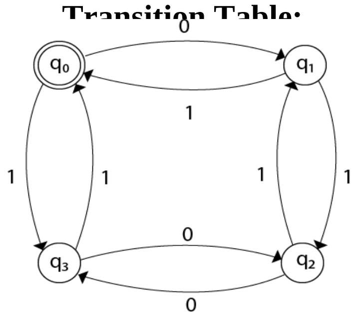
Transition Diagram:

Transition Table:

- **Design FA which accepts even number of 0's and even number of 1's over ∑ = {0, 1}**

Transition Diagram:

Transition Table:



	0	1
q0	q1	q3
q1	q0	q2
q2	q3	q1
q3	q2	q0

- **Design FA which accepts odd number of 0's and odd number of 1's defined over ∑ = {0, 1}**

Transition Diagram: **Transition Table**

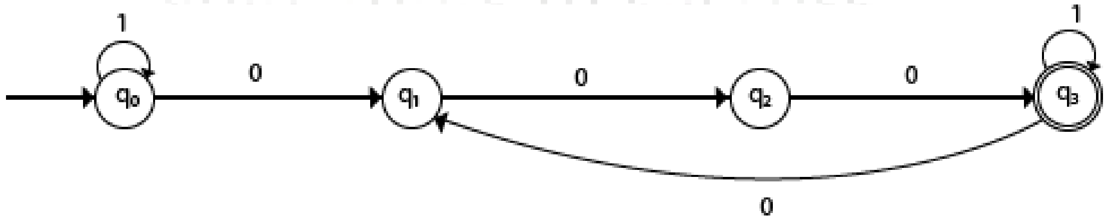
- **Design FA accepts even number of 0's and odd number of 1's defined over ∑ = {0, 1}**

Transition Diagram: **Transition Table**

- **Design FA which accepts odd number of 0's and even number of 1's defined over ∑ = {0, 1}**

Transition Diagram: **Transition Table**

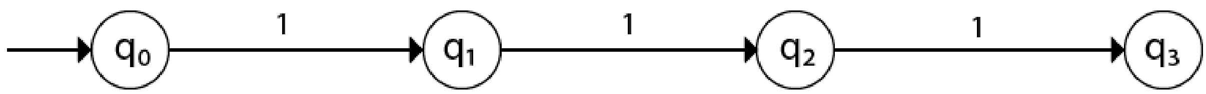
- **Design FA which accepts the set of all strings with three consecutive 0's.**



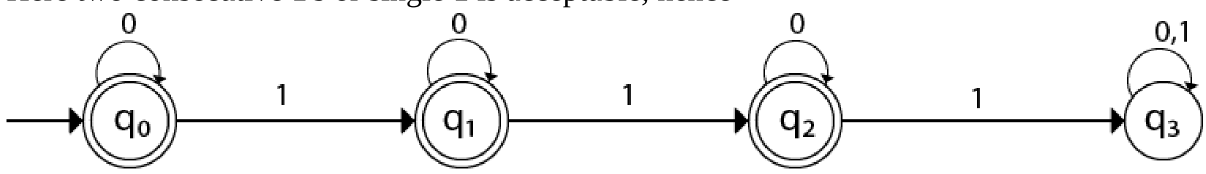
Transition Table:

- **Design a DFA for L(M) = {w | w ε {0, 1}*} and W is a string that does not contain three consecutive 1's}.**

- When three consecutive 1's occur the DFA will be:



Here two consecutive 1's or single 1 is acceptable, hence

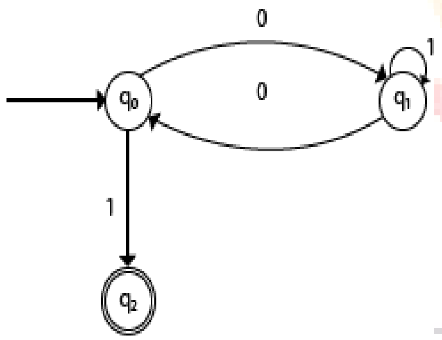


The stages q0, q1, q2 are the final states. The DFA will generate the strings that do not contain consecutive 1's like 10, 110, 101,.......................... etc.

Transition Table:

- **Design a FA which accepts the strings with an even number of 0's followed by single 1**

Transition Diagram:



Transition Table:

Practice Problems

 Design a FA with ∑ = {0, 1} accepts the strings with an even number of 0's followed by single 1

 Design a finite automata that recognizes i) even no of a’s ii) odd no of b’s defined over ∑ = {a, b}

 Design a DFA that contains 001as a substring defined over ∑ = {0, 1}

 Design a FA to accept strings of a’s and b’s ending with abb defined over ∑ = {a, b}

 Design a DFA which accepts the strings starting with 1 and ending with 0.

 Obtain the DFA that recognizes the language L(M)={W/W is in $\{a, b\ c\}^*$ and W contains the pattern abac}

 Design a DFA for the language $L=\{0^m1^n: m>=0,n>=1\}$

- Design a DFA for the language L={0^m1^n: m>=1,n>=1}

Note: Decimal to Binary
{ 0-0, 1-1, 2-10 , 3-11, 4-100, 5-101,
6-110, 7-111, 8-1000, 9-1001, 10-1010, 11-
1011, 12-1100, 13-1101, 14-1110,…….}

- Design a FA which checks whether a given binary number is even

1.3. Non-Deterministic Finite Automata (NFA):

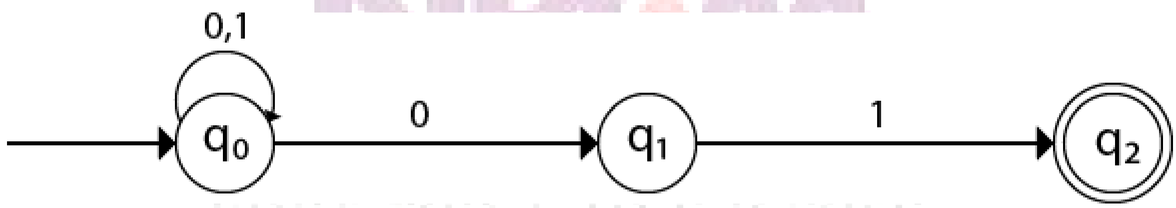
- NFA stands for Non-Deterministic Finite Automata. It is easy to construct an NFA than DFA for a given regular language.
- The finite automata are called NFA when there exist many paths for specific input from the current state to the next state.
- Every NFA is not DFA, but each NFA can be translated into DFA.
- NFA is defined in the same way as DFA but with the following two exceptions, it contains multiple next states, and it contains ε transition.

Formal definition of NFA:

A NFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where

- **Q** is a finite set of states.
- ∑ is a finite set of symbols called the alphabet.
- δ: Q x ∑ →2^Q is a transition function
- q0: initial state
- F: Set of final states

Ex: Design an NFA with ∑ = {0, 1} accepts all string ending with 01



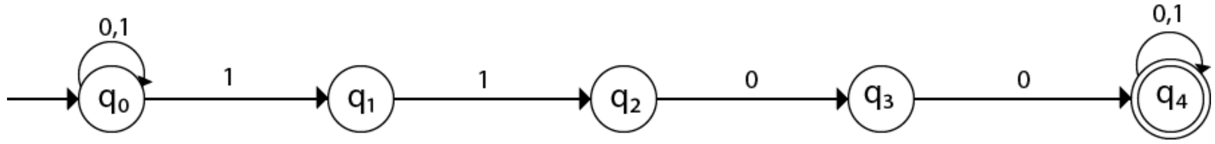
Transition Table:

	0	1
→q0	{q0,q1}	{q0}
q1	--	{q2}

		}
* q 2	--	- -

Ex: Design an NFA with $\sum = \{0, 1\}$ in which double '1' is followed by double '0'.

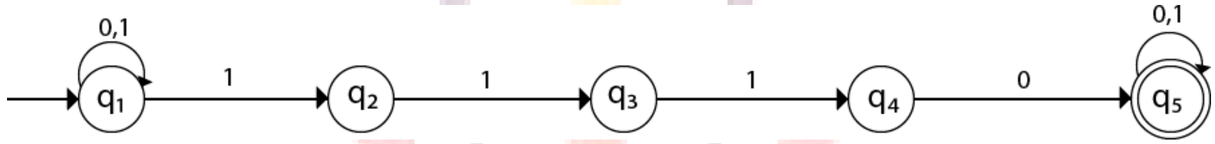
Transition Diagram:



Transition Table:

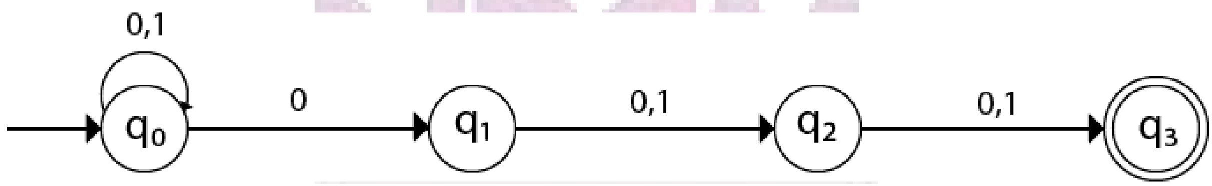
Ex: Design an NFA in which all the string contain a substring 1110

Transition Diagram:



Transition Table:

Ex: Design an NFA with $\sum = \{0, 1\}$ accepts all string in which the third symbol from the right end is always 0.



S.No	DFA	NFA
1	The transition from a state is to a single particular next state for each input symbol. Hence it is called deterministic	The transition from a state can be to multiple next states for each input symbol. Hence it is called non-deterministic.
2	Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.

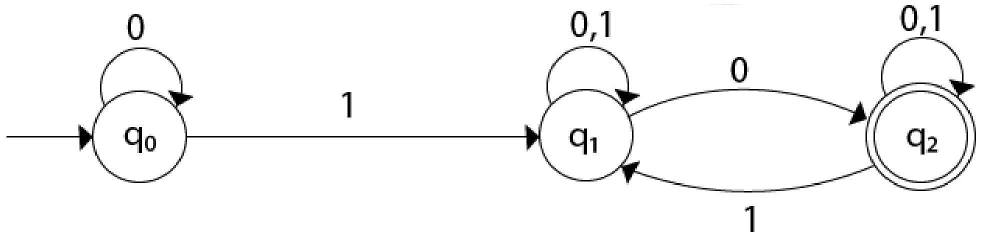
3	Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
4	Requires more space.	Requires less space.
5	A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

1.4. CONVERSION OF NFA to DFA:

- Let, M = (Q, ∑, δ, q0, F) is an NFA which accepts the language L(M). There should be equivalent DFA denoted by M' = (Q', ∑', q0', δ', F') such that L(M) = L(M').

  Steps for converting NFA to DFA:
- **Step 1:** Start from the initial state of NFA. Take the state with the '[ ]'.
- **Step 2:** place the next states for the initial state for the given inputs in the next columns put them also in [ ].
- **Step 3:** If any new combination of state appears in next state column then take the combination in the present state column.
- **Step 4:** If no new combination of state appears then stop the process.
- **Step 5:** The initial state for the constructed DFA will be the initial state of NFA.
- **Step 6:** The Final state(s) for the constructed DFA will be the combinations of states containing at least one final state of NFA.

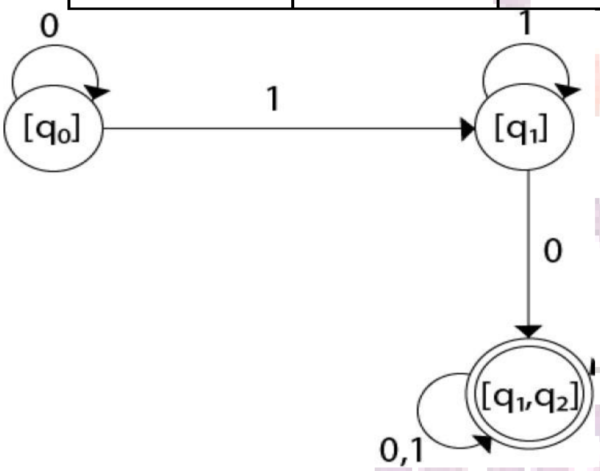
  EX: CONVERT THE GIVEN NFA TO DFA



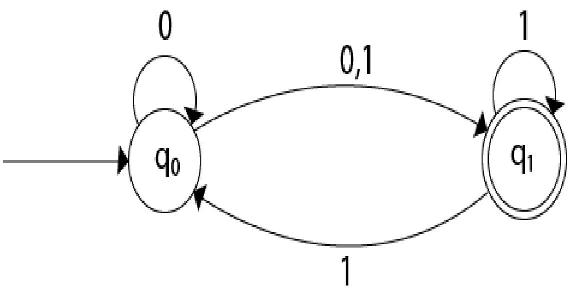
	0	1
→q0	{q0}	{q1}
q1	{q1, q2}	{q1}
*q2	{q2}	{q1,q2}

Now we will obtain δ' transition for state q0.
δ'([q0], 0) = [q0] δ'([q0], 1) = [q1] **(new state generated)**
δ'([q1], 0) = [q1, q2] (**new state generated**)
δ'([q1], 1) = [q1]
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 because it contains a final state q2.

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



EX:NFA TO DFA CONVERSION



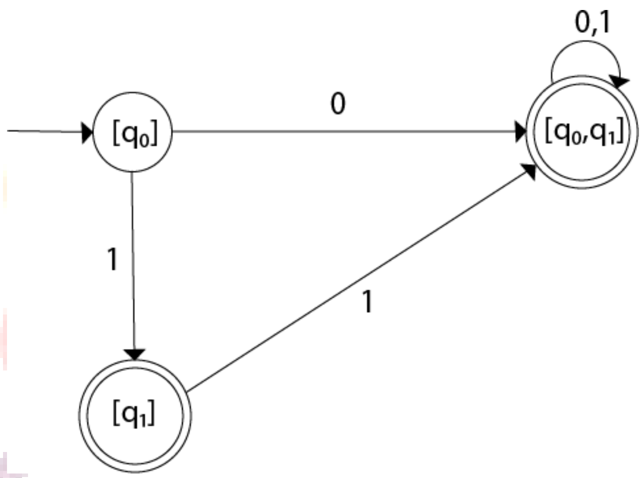
	0	**1**
→q0	{q0,q1}	{q1}
*q1	--	{q0,q1}

Now we will obtain δ' transition for state q0.

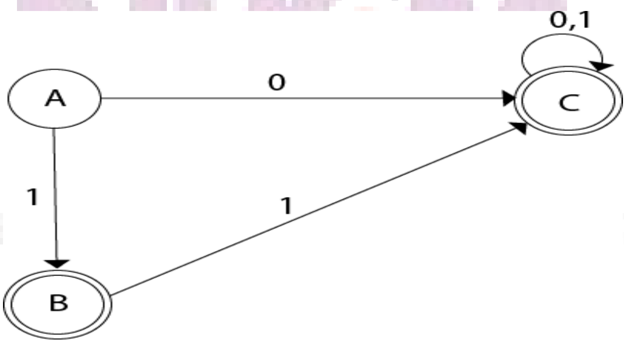
δ'([q0], 0) = {q0, q1} = [q0, q1] **(new state generated)**
δ'([q0], 1) = {q1} = [q1] **(new state generated)**
The δ' transition for state q1 is obtained as:
δ'([q1], 0) = ϕ , δ'([q1], 1) = [q0, q1]
Now we will obtain δ' transition on [q0, q1].
δ'([q0, q1], 0) = δ(q0, 0) ∪ δ(q1, 0) = {q0, q1} ∪ ϕ = {q0, q1} = [q0, q1]
Similarly,
δ'([q0, q1], 1) = δ(q0, 1) ∪ δ(q1, 1) = {q1} ∪ {q0, q1} = {q0, q1} = [q0, q1]

As in the given NFA, q1 is a final state, then in DFA wherever, q1 exists that state becomes a final state. Hence in the DFA, final states are [q1] and [q0, q1]. Therefore set of final states F = {[q1], [q0, q1]}.

	0	1
→[q0]	[q0, q1]	[q1]
*[q1]	ϕ	[q0, q1]
*[q0, q1]	[q0, q1]	[q0, q1]

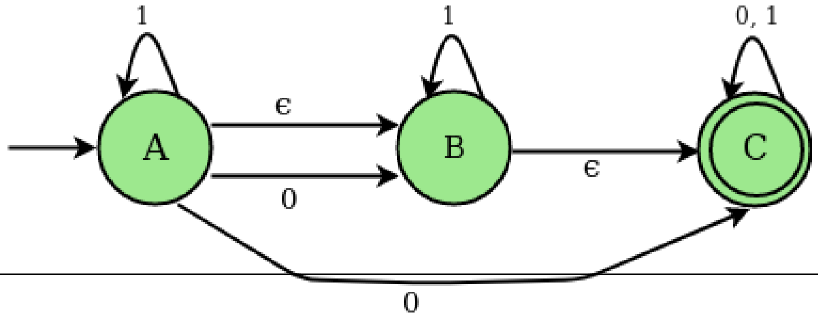


Even we can change the name of the states of DFA.
Suppose A = [q0] B = [q1] C = [q0, q1]
With these new names the DFA will be as follows:

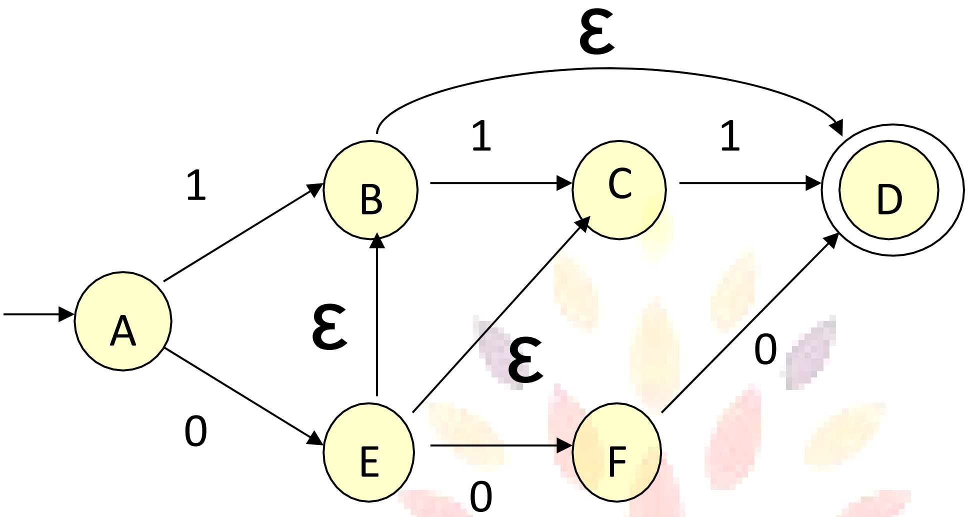


NFA WITH EPSILON TRANSITIONS

Def: If any finite automata contain ε (null) move or transition, then that finite automaton is called NFA with ∈ moves



STATES	0	1	EPSILON
A	{B, C}	{A}	{B}
B	–	{B}	{C}
C	{C}	{C}	–



1.5. EPSILON (∈) – CLOSURE:

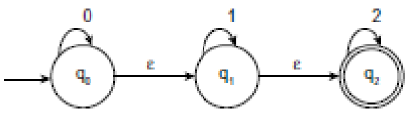
- Epsilon closure for a given state X is a set of states which can be reached from the states X with only (null) or ε moves including the state X itself. In other words, ε-closure for a state can be obtained by union operation of the ε-closure of the states which can be reached from X with a single ε move in recursive manner.
- For the above example ∈ closure are as follows :
- ∈ closure(A) : {A, B, C} , ∈closure(B) : {B, C}, ∈ closure(C) : {C}

Construction of ∈ -NFA:

Ex: Construct ∈-NFA with e-transitions and it accepts strings of the form$\{O^n l^m 2^o/n,m,o>=0\}$,thatis,anynumberof0'sfollowedbyanynumberofl'sfollowed by any number of 2's.

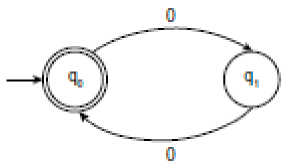
Transition Diagram:

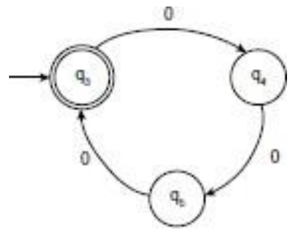
Transition Table:



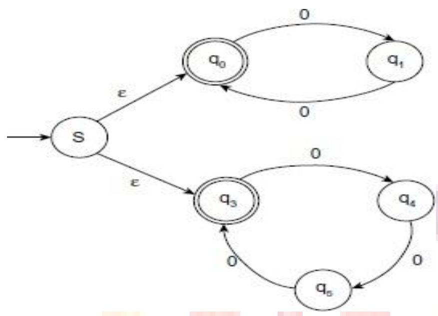
	0	1	2	∈
q0	{q0}	--	--	{q1}
q1	--	{q1}	--	{q2}
*q2	--	--	{q2}	--

Ex: Design NFA for language L ={0^KIk is multiple of 2 or 3}.





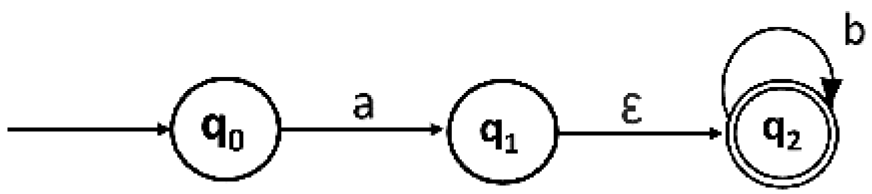
NFA for multiple of 3
NFA for multiple of 2



Conversion of∈ -NFA TO NFA or elimination of ∈ transitions

- Find ε-closure {qi} for all qi ∈ Q.
- Find δ^ (q,a)= ε-closure (δ (δ^ (q, ε),a)) =ε-closure (δ (ε-closure(q),a))
- Repeat Step-2 for each input symbol and each state of given NFA.
- Using the resultant states, the transition table for equivalent NFA without ε can be built.
- If the ε-closure of a state contains a final state then make the state as final.

Ex: Convert the following ∈ -NFA TO NFA



Solutions: We will first obtain

ε-closures of q0, q1 and q2 as follows:

ε-closure(q0) = {q0} , ε-closure(q1) = {q1, q2}

ε-closure(q2) = {q2}

Now the δ^transition on each input symbol is obtained as:

δ^ (q0, a) = ε-closure(δ(δ^(q0, ε),a)) = ε-closure(δ(ε-closure(q0),a)) = ε-closure(δ(q0, a))= ε- closure(q1)
= {q1, q2}

δ^ (q0, b) = ε-closure(δ(δ^(q0, ε),b)) = ε-closure(δ(ε-closure(q0),b)) = ε-closure(δ(q0, b)) = Φ

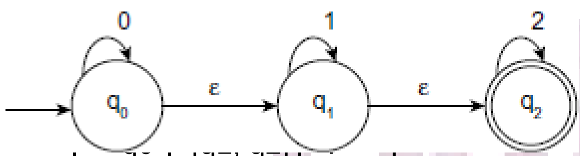
δ^(q2, a) = ε-closure(δ(δ^(q2, ε),a)) = ε-closure(δ(ε-closure(q2),a)) = ε-closure(δ(q2, a))

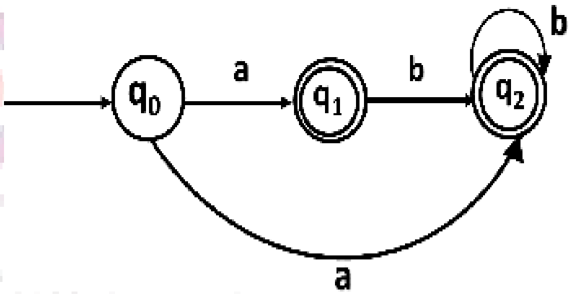
= ε-closure(Φ) = Φ

δ^(q2, b) = ε-closure(δ(δ^(q2, ε),b)) = ε-closure(δ(ε-closure(q2),b))

= ε-closure(δ(q2, b)) = ε-closure(q2) = {q2}
Now we will summarize all the computed δ' transitions:

δ^(q0, a) = {q0, q1} δ^(q0, b) = Φ δ^(q1, a) = Φ **, δ^(q1, b) = {q2}** δ^(q2, a) = Φ , **δ^(q2, b) = {q2} .**
State q1 and q2 become the final state as ε-closure of q1 and q2 contain the final state q2.
Ex: Convert the following ∈ -NFA TO NFA





*q1	Φ	{q2}
*q2	Φ	{q2}

The transition table is

	a = 0	a = 1	a = 2	a = ε
→q_0	q_0	Ø	Ø	q_1
q_1	Ø	q_1	Ø	q_2
*q_2	Ø	Ø	q_2	Ø

CONVERSION FROM ε-NFA TO DFA

Step 1: If ε-closure(q_0)= {P1,P2,..Pn} then [P1P2..Pn] becomes the starting state of DFA.

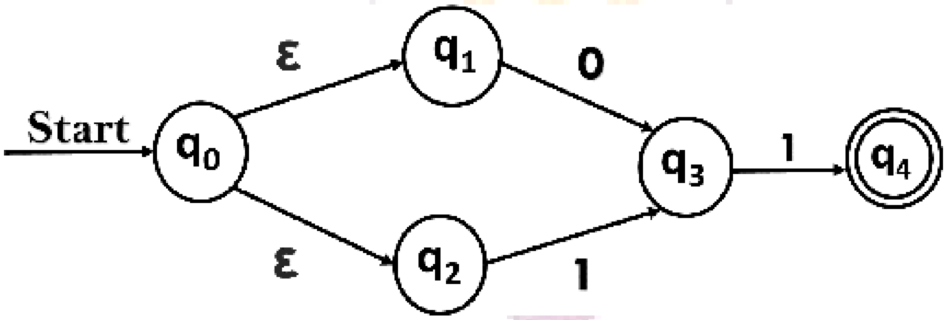
Step 2: Find δ_D ([P1P2..Pn] ,a)= ε-closure(δ(P1,P2,..Pn),a))

Step 3: If we found a new state, take it as current state and repeat step 2.

Step 4: Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.

Step 5: Mark the states of DFA as a final state which contains the final state of NFA.

EX: CONVERT THE NFA WITH ε INTO ITS EQUIVALENT DFA.



Let us obtain ε-closure of each state.

ε-closure {q0} = {q0, q1, q2}

ε-closure {q1} = {q1} ε-closure {q2} = {q2} ε-closure {q3} = {q3}
ε-closure {q4} = {q4}

Now, let ε-closure {q0} = {q0, q1, q2} be state A.

Hence

δ'(A, 0) = ε-closure {δ((q0, q1, q2), 0) } = ε-closure {δ(q0, 0) ∪δ(q1, 0) ∪δ(q2, 0) }

= ε-closure {q3} = {q3} **call it as state B**.

δ'(A, 1) = ε-closure {δ((q0, q1, q2), 1) } = ε-closure {δ((q0, 1) ∪δ(q1, 1) ∪δ(q2, 1)}

= ε-closure {q3} = {q3} = B.

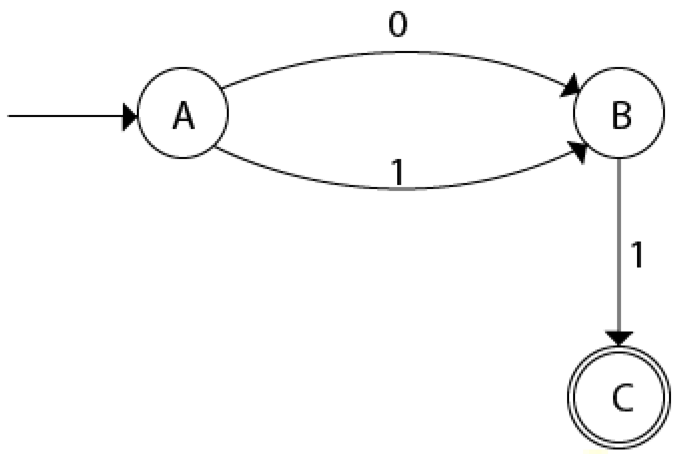
Now,

δ'(B, 0) = ε-closure {δ(q3, 0) } = ϕ

δ'(B, 1) = ε-closure {δ(q3, 1) }= ε-closure {q4} = {q4} **i.e. state C**

For state C: δ'(C, 0) = ε-closure {δ(q4, 0) } = ϕ δ'(C, 1) = ε-closure {δ(q4, 1) } = ϕ

The DFA will be



Ex: Convert the given NFA with epsilon into its equivalent DFA

L= any no of a's followed by any no of b's followed by any no of c's
Solution: Let us obtain the ε-closure of each state.

ε-closure(q0) = {q0, q1, q2}

ε-closure(q1) = {q1, q2}

ε-closure(q2) = {q2}

Now we will obtain δ' transition.

Let ε-closure(q0) = {q0, q1, q2} call it as **state A**.

δ'(A, 0) = ε-closure{δ((q0, q1, q2), 0)} = ε-closure{δ(q0, 0) ∪δ(q1, 0) ∪δ(q2, 0)}

= ε-closure{q0}= {q0, q1, q2}

δ'(A, 1) = ε-closure{δ((q0, q1, q2), 1)} = ε-closure{δ(q0, 1) ∪δ(q1, 1) ∪δ(q2, 1)}

= ε-closure{q1} = {q1, q2} **call it as state B**

δ'(A, 2) = ε-closure{δ((q0, q1, q2), 2)} = ε-closure{δ(q0, 2) ∪δ(q1, 2) ∪δ(q2, 2)}
= ε-closure{q2} = {q2} **call it state C**
Thus we have obtained

δ'(A, 0) = A δ'(A, 1) = B δ'(A, 2) = C

Now we will find the transitions on states B and C for each input.

Hence

δ'(B, 0) = ε-closure{δ((q1, q2), 0)} =ε-closure{δ(q1, 0) ∪δ(q2, 0)}= ε-closure{ϕ} = ϕ

δ'(B, 1) = ε-closure{δ((q1, q2), 1)}= ε-closure{δ(q1, 1) ∪δ(q2, 1)} = ε-closure{q1}= {q1, q2} **i.e. state B itself**

δ'(B, 2) = ε-closure{δ((q1, q2), 2)} = ε-closure{δ(q1, 2) ∪δ(q2, 2)} = ε-closure{q2} = {q2} **i.e. state C itself**

Thus we have obtained

δ'(B, 0) = ϕ δ'(B, 1) = B δ'(B, 2) = C

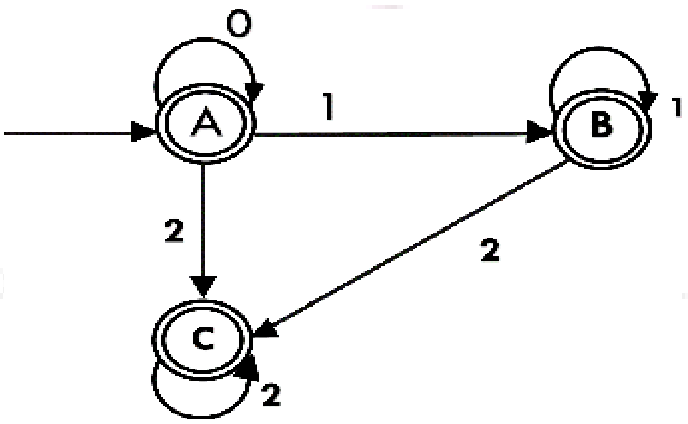
Now we will obtain transitions for C:

δ'(C, 0) = ε-closure{δ(q2, 0)} = ε-closure{ϕ} = ϕ

δ'(C, 1) = ε-closure{δ(q2, 1)}= ε-closure{ϕ}= ϕ

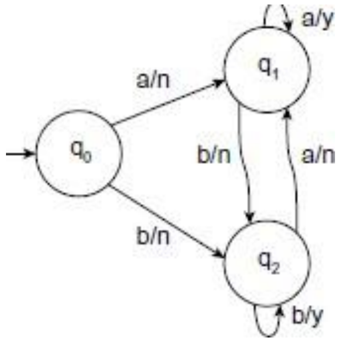
δ'(C, 2) = ε-closure{δ(q2, 2)}= {q2}

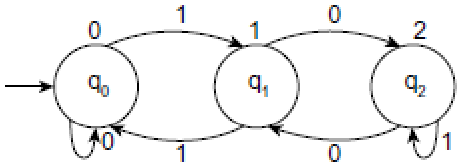
As A = {q0, q1, q2} in which final state q2 lies hence A is final state. B = {q1, q2} in which the state q2 lies hence B is also final state. C = {q2}, the state q2 lies hence C is also a final state.



1.6. FINITE AUTOMATA WITH OUTPUTS: MOORE& MEALY M/C

- Finite automata may have outputs corresponding to state or transition. There are two types of finite state machines that generate output: (i) Moore Machine (ii) Mealy Machine
- If the output associated with state then such a machine is called Moore machine, and if the output is associated with transition then it is called mealy machine.





Moore Machine

Mealy Machine

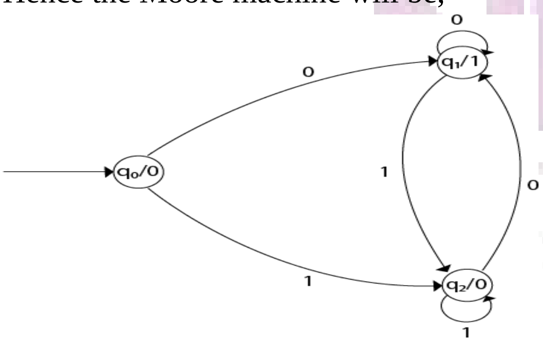
1.6.1. MOORE MACHINE:

- Moore machine is a finite state machine in which the next state is decided by the current state and current input symbol. The output symbol at a given time depends only on the present state of the machine.
- Def: Moore machine can be described by 6-tuple M=(Q, ∑ , Δ, δ,q0, λ) where
- Q: finite set of states
- ∑: finite set of input symbols
- Δ: output alphabet
- q0: initial state of machine
- δ: Q × ∑ → Q is a transition function
- λ: Q → Δ output function

Ex: Design a Moore machine to generate 1's complement of a given binary number.

Solution: To generate 1's complement of a given binary number the simple logic is that if the input is 0 then the output will be 1 and if the input is 1 then the output will be 0. That means there are three states. One state is start state. The second state is for taking 0's as input and produces output as 1. The third state is for taking 1's as input and producing output as 0.

Hence the Moore machine will be,



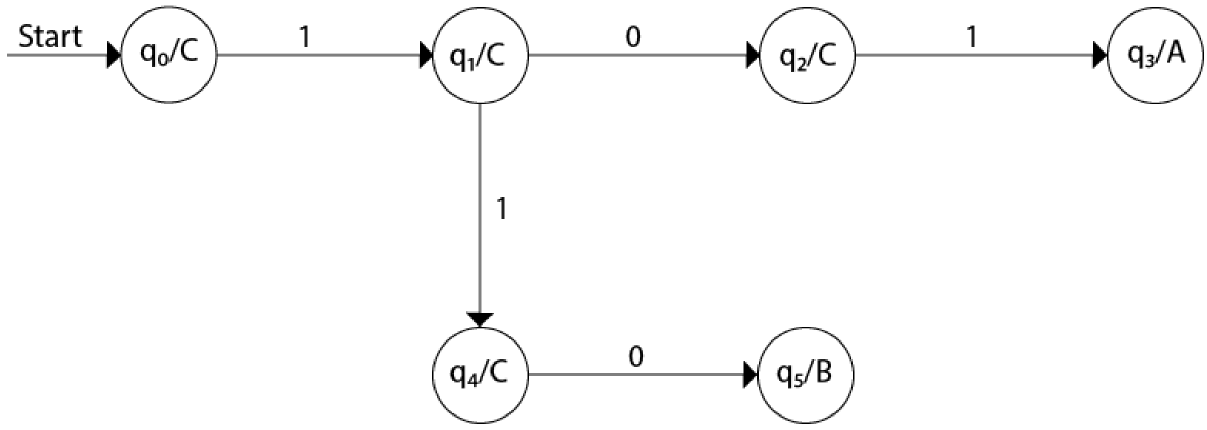
Current State	Next State (δ) 0	Next State (δ) 1	Output (λ)
→ q_0	q_1	q_2	0
q_1	q_1	q_2	1
q_2	q_1	q_2	0

For instance, take one binary number 1011 then

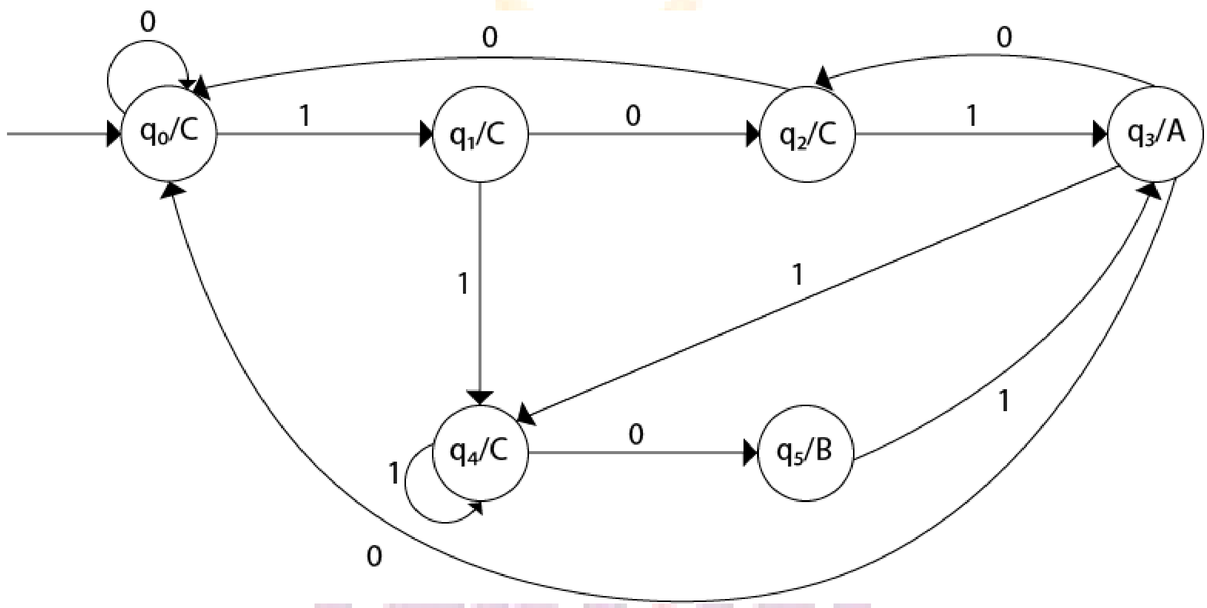
Input		1	0	1	1
State	q0	q2	q1	q2	q2
Output	0	0	1	0	0

Thus we get 00100 as 1's complement of 1011, we can neglect the initial 0 and the output which we get is 0100 which is 1's complement of 1011.
Note: The output length for a Moore machine is greater than input by 1.

Ex: Design a Moore machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.
Solution: For designing such a machine, we will check two conditions, and those are 101 and 110. If we get 101, the output will be A, and if we recognize 110, the output will be B. For other strings, the output will be C.
The partial diagram will be:

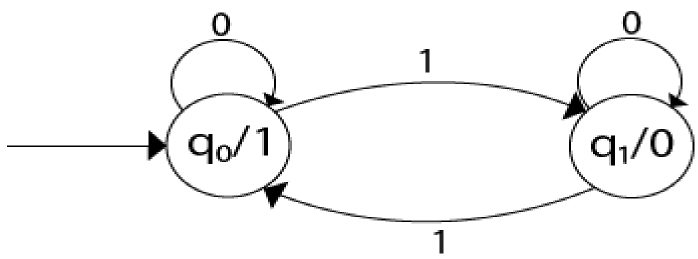


Now we will insert the possibilities of 0's and 1's for each state. Thus the Moore machine becomes:



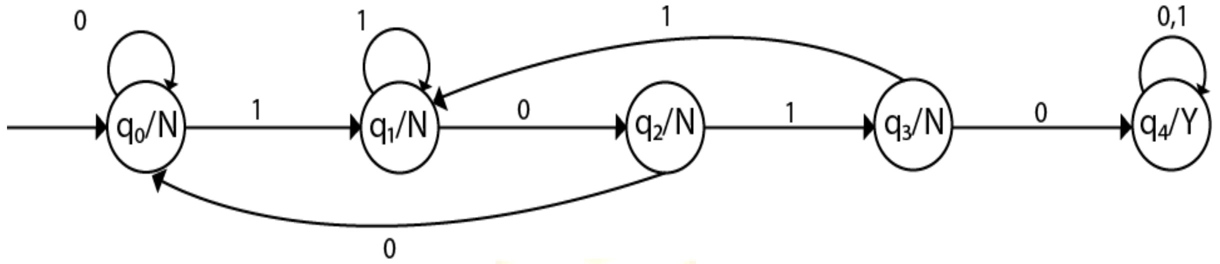
Ex: Construct a Moore machine that determines whether an input string contains an even or odd number of 1's. The machine should give 1 as output if an even number of 1's are in the string and 0 otherwise.

Sol: The Moore machine will be:



This is the required Moore machine. In this machine, state q1 accepts an odd number of 1's and state q0 accepts even number of 1's. There is no restriction on a number of zeros. Hence for 0 input, self-loop can be applied on both the states.

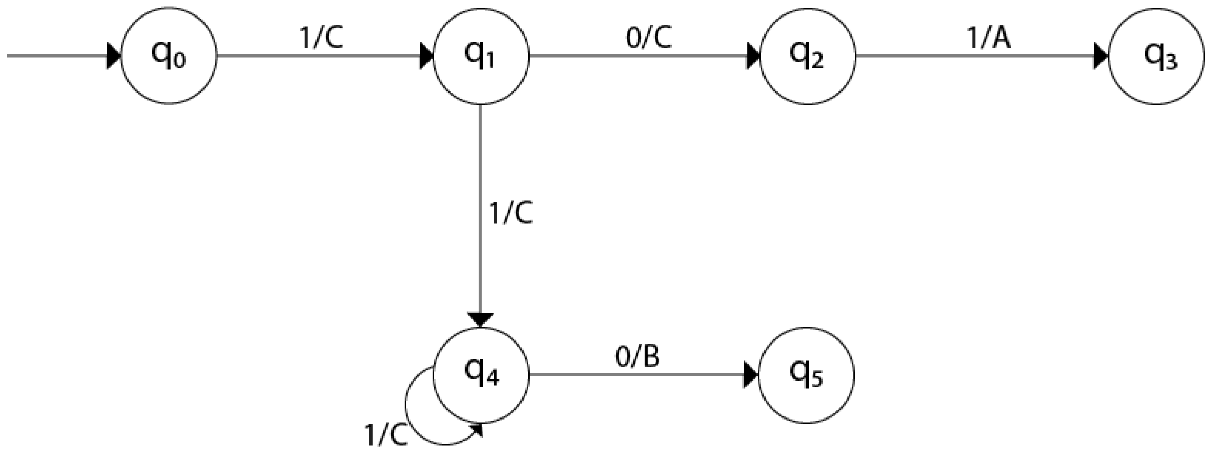
Ex: Design a Moore machine with the input alphabet {0, 1} and output alphabet {Y, N} which produces Y as output if input sequence contains 1010 as a substring otherwise, it produces N as output.



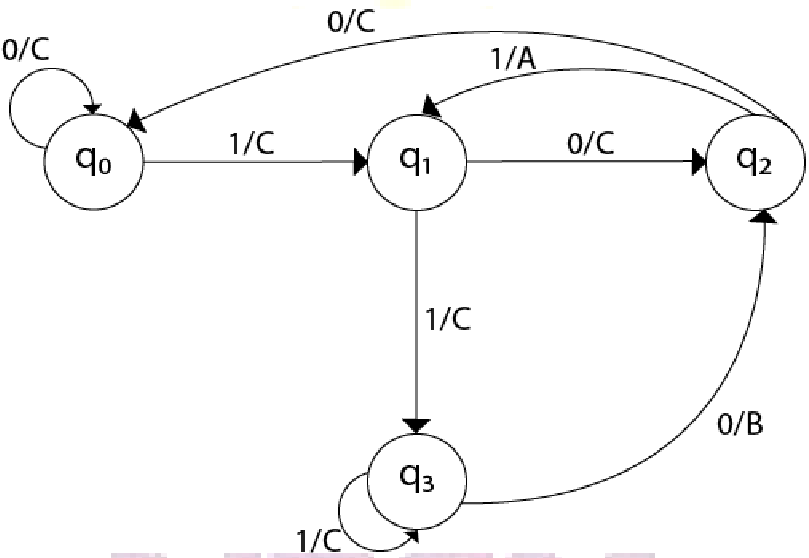
1.6.2. MEALY MACHINE

- A Mealy machine is a machine in which output symbol depends upon the present input symbol and present state of the machine. In the Mealy machine, the output is represented with each input symbol for each state separated by /.
  Def: The Mealy machine can be described by 6- tuple M= (Q, ∑, Δ, q0, δ, λ') where
- Q: finite set of states
- q0: initial state of machine
- ∑: finite set of input alphabet
- Δ: output alphabet
- δ: Q × ∑ → Q transition function
- λ: Q × ∑ →Δ output function

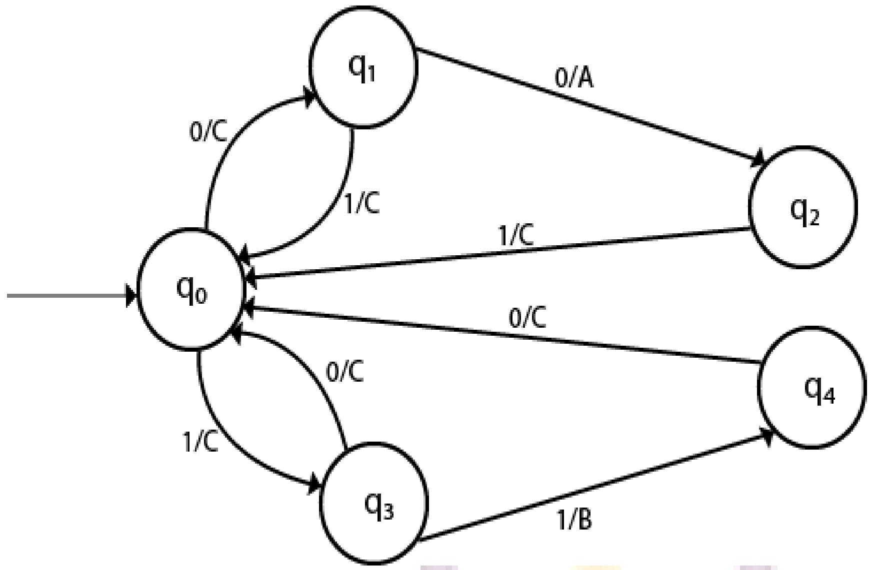
Ex: Design a Mealy machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.
Solution: For designing such a machine, we will check two conditions, and those are 101 and 110. If we get 101, the output will be A. If we recognize 110, the output will be B. For other strings the output will be C.
The partial diagram will be:



Now we will insert the possibilities of 0's and 1's for each state. Thus the Mealy machine becomes:



Ex: Design a mealy machine that scans sequence of input of 0 and 1 and generates output 'A' if the input string terminates in 00, output 'B' if the string terminates in 11, and output 'C' otherwise.



1.6.3. CONVERSION FROM MEALY MACHINE TO MOORE MACHINE:

In Moore machine, the output is associated with every state, and in Mealy machine, the output is given along the edge with input symbol. To convert Moore machine to Mealy machine, state output symbols are distributed to input symbol paths. But while converting the Mealy machine to Moore machine, we will create a separate state for every new output symbol and according to incoming and outgoing edges are distributed.

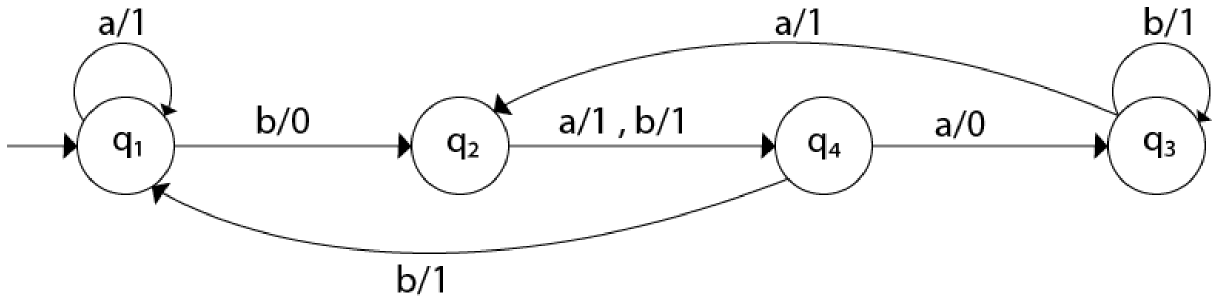
Mealy to Moore machine Conversion:

Step 1: For each state (Qi), calculate the number of different outputs that are available in the transition table of the Mealy machine.

Step 2: Copy state Qi, if all the outputs of Qi are the same. Break qi into n states as Qin, if it has n distinct outputs where n = 0, 1, 2....

Step 3: If the output of initial state is 0, insert a new initial state at the starting which gives ε output.

Ex: Convert the following Mealy machine into equivalent Moore machine.



Present State	Next State			
	a		b	
	State	O/P	State	O/P
q_1	q_1	1	q_2	0
q_2	q_4	1	q_4	1
q_3	q_2	1	q_3	1
q_4	q_3	0	q_1	1

- For state q1, there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.
- For state q2, there is 2 incident edges with output 0 and 1. So, we will split this state into two states q20(state with output 0) and q21(with output 1).
- For state q3, there is 2 incident edges with output 0 and 1. So, we will split this state into two states q30(state with output 0) and q31(state with output 1).
- For state q4, there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.

Input	0		1	
q1	q1	1	q21	1
q20	q4	1	q4	1
q21	q4	1	q4	1
q30	q21	1	q31	1
q31	q21	1	q31	1
q4	q30	0	q1	1

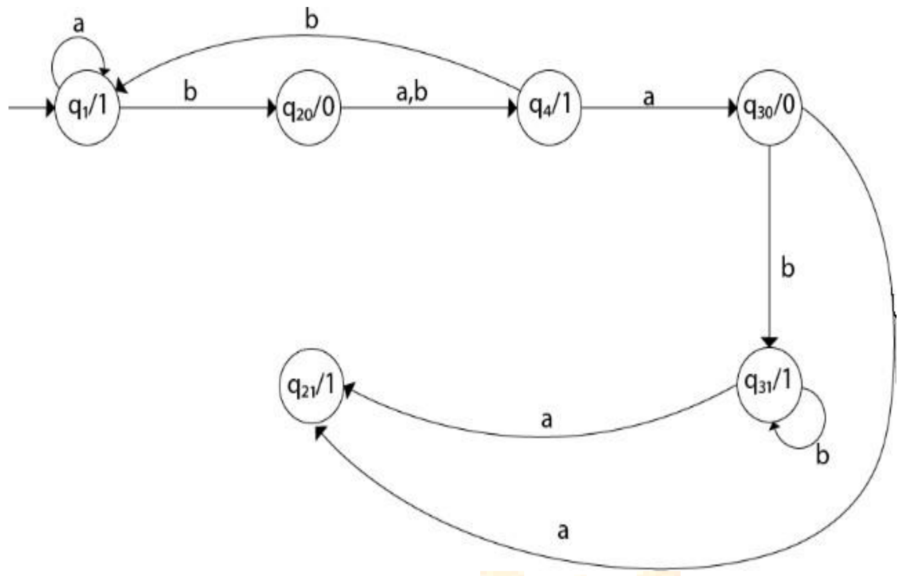
Input	0	1	Output
q1	q1	q21	1
q20	q4	q4	0
q21	q4	q4	1
q30	q21	q31	0
q31	q21	q31	1
q4	q30	q1	1

Transition table for Moore machine

	0	1	Output
q1	q1	q21	1
q20	q4	q4	0
q21	q4	q4	1
q30	q21	q31	0
q31	q21	q31	1
q4	q30	q1	1

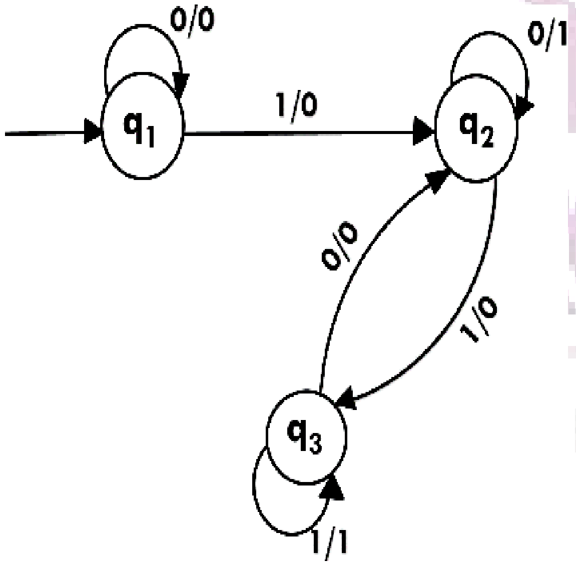
	0	1	Output
q0	q1	q21	1
q1	q1	q21	1
q20	q4	q4	0
q21	q4	q4	1
q30	q21	q31	0
q31	q21	q31	1
q4	q30	q1	1

Transition diagram for Moore machine :



Ex: Convert the following Mealy machine into equivalent Moore machine.

Transition Diagram:



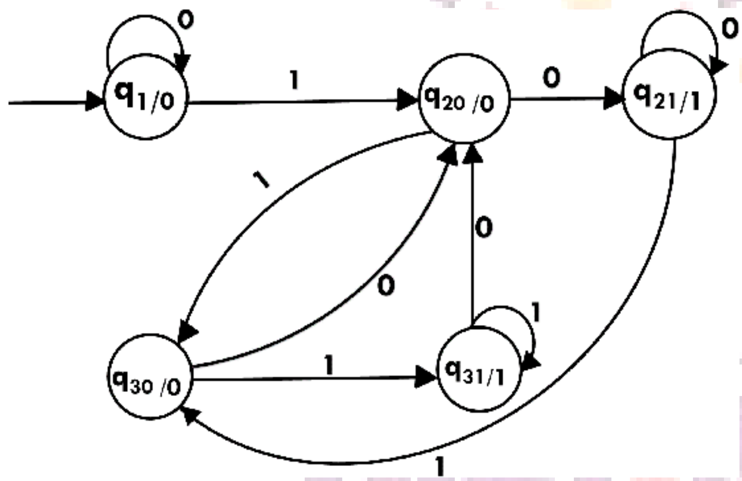
Present State	Next State 0		Next State 1	
	State	O/P	State	O/P
q_1	q_1	0	q_2	0
q_2	q_2	1	q_3	0
q_3	q_2	0	q_3	1

The state q1 has only one output. The state q2 and q3 have both output 0 and 1. So we will create two states for these states. For q2, two states will be q20(with output 0) and q21(with output 1). Similarly, for q3 two states will be q30(with output 0) and q31(with output 1).

Transition table for Moore machine will be:

Present State	Next State 0	Next State 1	0/P
q_1	q_1	q_{20}	0
q_{20}	q_{21}	q_{30}	0
q_{21}	q_{21}	q_{30}	1
q_{30}	q_{20}	q_{31}	0
q_{31}	q_{20}	q_{31}	1

Transition diagram for Moore machine will be:



1.6.4. CONVERSION FROM MOORE MACHINE TO MEALY MACHINE

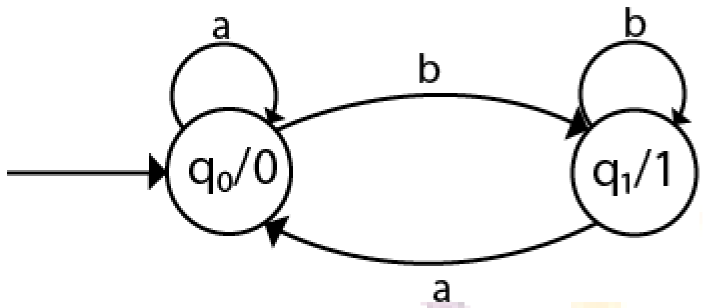
- In the Moore machine, the output is associated with every state, and in the mealy machine, the output is given along the edge with input symbol. The equivalence of the Moore machine and Mealy machine means both the machines generate the same output string for same input string.
- We cannot directly convert Moore machine to its equivalent Mealy machine because the length of the Moore machine is one longer than the Mealy machine for the given input. To convert Moore machine to Mealy machine, state output symbols are distributed into input

symbol paths. We are going to use the following method to convert the Moore machine to Mealy machine.

Method for conversion of Moore machine to Mealy machine

Let M = (Q, ∑, δ, λ, q0) be a Moore machine. The equivalent Mealy machine can be represented by M'
= (Q, ∑, δ, λ', q0). The output function λ' can be obtained as**: λ' (q, a) = λ(δ(q, a)) Ex:Convert the following Moore machine into its equivalent Mealy machine.**

Solution:



The transition table of given Moore machine is as follow.

Q	a	b	Output(λ)
q0	q0	q1	0
q1	q0	q1	1

The equivalent Mealy machine can be obtained as follows:

λ' (q0, a) = λ(δ(q0, a)) = λ(q0) = 0

λ' (q0, b) = λ(δ(q0, b)) = λ(q1) = 1 The λ for state q1 is as follows:
λ' (q1, a) = λ(δ(q1, a)) = λ(q0) = 0

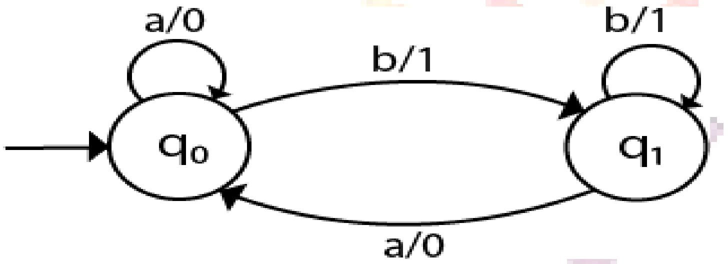
λ' (q1, b) = λ(δ(q1, b)) = λ(q1) = 1

Hence the transition table for the Mealy machine can be drawn as follows:

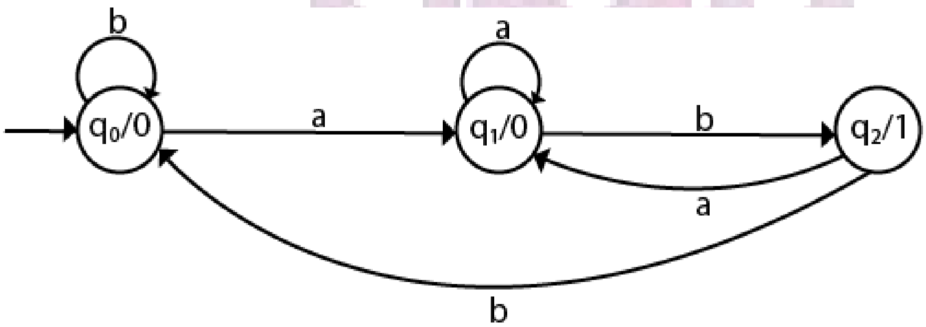
Q \ Σ	Input 0		Input 1	
	State	O/P	State	O/P
q_0	q_0	0	q_1	1
q_1	q_0	0	q_1	1

The equivalent Mealy machine will be

Note: The length of output sequence is 'n+1' in Moore machine and is 'n' in the Mealy machine



Ex: Convert the following Moore machine into its equivalent Mealy machine.



Q	**a**	**b**	**Output(λ)**
q0	q1	q0	0
q1	q1	q2	0
q2	q1	q0	1

The equivalent Mealy machine can be obtained as follows:

λ' (q0, a) = λ(δ(q0, a)) = λ(q1) = 0

λ' (q0, b) = λ(δ(q0, b)) = λ(q0) = 0 The λ for state q1 is as follows:

λ' (q1, a) = λ(δ(q1, a)) = λ(q1) = 0

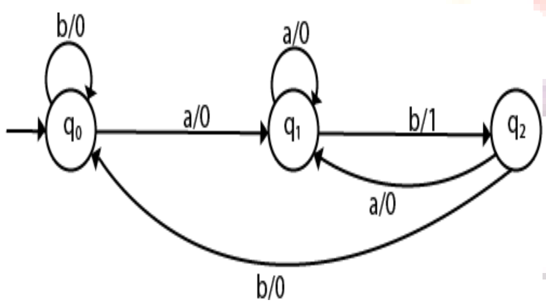
λ' (q1, b) = λ(δ(q1, b)) = λ(q2) = 1 The λ for state q2 is as follows:

λ' (q2, a) = λ(δ(q2, a)) = λ(q1) = 0

λ' (q2, b) = λ(δ(q2, b)) = λ(q0) = 0

Hence the transition table for the Mealy machine can be drawn as follows:

Q \ Σ	Input a		Input b	
	State	Output	State	Output
q_0	q_1	0	q_0	0
q_1	q_1	0	q_2	1
q_2	q_1	0	q_0	0



Ex:Convert the given Moore machine into its equivalent Mealy machine.

Q	**a**	**b**	**Output(λ)**
q0	q0	q1	0
q1	q2	q0	1
q2	q1	q2	2

Differences between Mealy and Moore Machines:

S.No	Mealy Machine	Moore Machine
1	Output depends both upon the present state and the present input	Output depends only upon the present state.
2	Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
3	The value of the output function is a function of the transitions and the changes, when the input logic on the present state is done.	The value of the output function is a function of the current state and the changes at the clock edges, whenever state changes occur.
4	Mealy machines react faster to inputs. They generally react in the same clock cycle.	In Moore machines, more logic is required to decode the outputs resulting in more circuit delays. They generally react one clock cycle later.

IMPORTANT QUESTIONS:

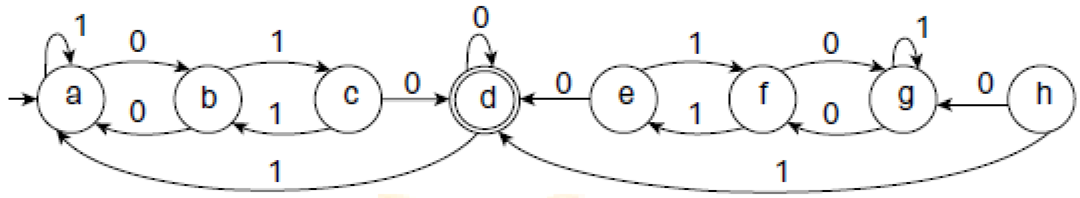
1. Define Star closure / Kleen Closure
2. Define Positive Closure
3. Define Language
4. Define DFA, NFA and epsilon NFA
5. Define epsilon closure
6. Define Moore and Mealy machines

PART-B

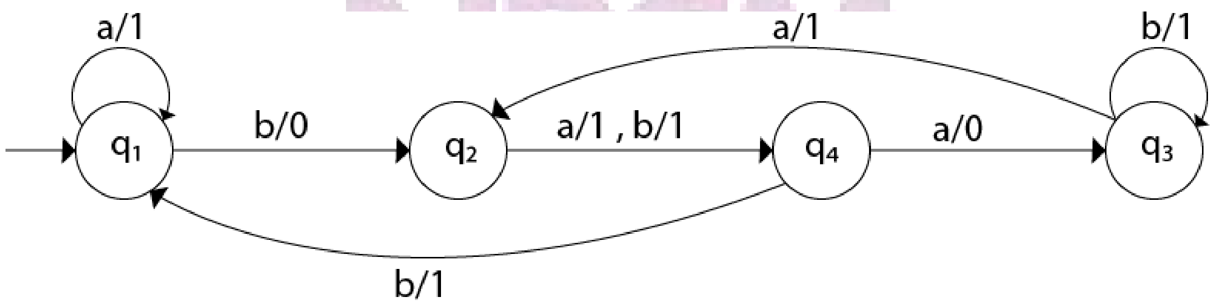
7. Draw the block diagram of Finite Automata and explain each component
8. Design FA which accepts i) even number of 0's and even number of 1's ii) even number of 0’s and odd number of 1’s iii) odd number of 0’s and even number of 1’s and iv) odd number of 0’s and odd number of 1’s over ∑ = {0, 1}
9. Design a DFA for L(M) = {w | w ε {0, 1}*} and W is a string that does not contain consecutive 1's}.
10. Obtain the DFA that recognizes the language L(M)={W/W is in $\{a, b\ c\}^*$ and W contains the pattern abac}
11. Design a FA that accepts the set of all strings that interpreted as binary representation of an unsigned decimal number i) which is divisible by 2 ii) divisible by 4 iii) which is divisible

by 5.

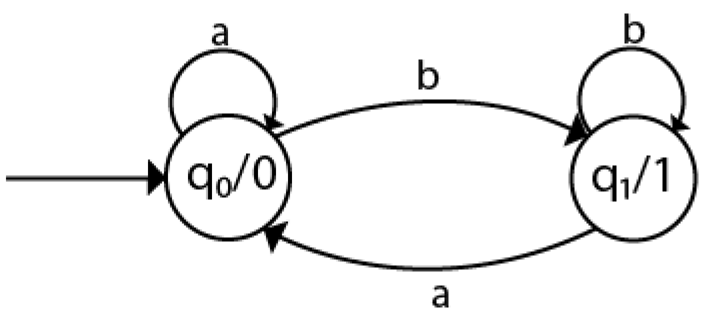
12. Design an NFA with ∑ = {0, 1} in which double '1' is followed by double '0'.
13. Design an NFA with ∑ = {0, 1} accepts all string in which the third symbol from the right end is always 0.
14. What are the differences between DFA, NFA
15. Write the algorithm to convert i) NFA to DFA ii) epsilon NFA to NFA and iii) epsilon NFA to DFA. Explain by taking an example for each conversion.
16. Find minimum-state automaton equivalent to the transition diagram



17. Design a Moore machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C
18. Construct a Moore machine that determines whether an input string contains an even or odd number of 1's. The machine should give 1 as output if an even number of 1's are in the string and 0 otherwise.
19. Design a Mealy machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.
20. Design a mealy machine that scans sequence of input of 0 and 1 and generates output 'A' if the input string terminates in 00, output 'B' if the string terminates in 11, and output 'C' otherwise.
21. Convert the following Mealy machine into equivalent Moore machine



22. Convert the following Moore machine into its equivalent Mealy machine



UNIT-2

Regular Expressions: Finite Automata and Regular Expressions, Applications of Regular Expressions, Algebraic Laws for Regular Expressions, Conversion of Finite Automata to Regular Expressions. Pumping Lemma for Regular Languages: Statement of the pumping lemma, Applications of the Pumping Lemma. Closure Properties of Regular Languages: Closure properties of Regular languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata.

2.1. REGULAR EXPRESSION

- The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
- The languages accepted by some regular expression are referred to as Regular languages.
- A regular expression can also be described as a sequence of pattern that defines a string.
- Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.
- Regular Set: sets which are accepted by FA
- Ex: L={a,aa,aaa,…}

Regular Expression: Let ∑ be an I/P alphabet . The RE over ∑ can be defined as follows:

- ¢ is a regular expression.
- € is a regular expression.
- For any a in ∑, a is a regular expression.
- If r1 and r2 are regular expressions, then
- (r1 + r2) is a regular expression.
- (r1 .r2) is a regular expression.
- (r1*) is a regular expression.
- ($r1^+$) is a regular expression.

WRITE RES FOR THE FOLLOWING LANGUAGES:

- Accepting all combinations of a’s over the set □={a} Ans: a^*
- Accepting all combinations of a’s over the set □={a} except null string Ans: a^+

- Accepting any no of a’s and b’s

Ans: (a+b)* or (a/b)*

- Strings ending with 00 over the set {0,1} Ans: (0+1)*00
- Strings starts with 1 and ends with 0 over the set {0,1} Ans: 1(0/1)*1

- Any no of a's followed by any no of b's then followed by any no of c's
  Ans: a*b*c*
- starting and ending with a and having any combination of b's in between.
  Ans: a b* b
- Starting with a but not having consecutive b's.

Ans: L = {a, aba, aab, aba, aaa, abab…}
R = a{b + ab}*

- The language accepting all the string in which any number of a's is followed by any number of b's is followed by any number of c's.
  Ans: R = a* b* c*
- The language over ∑ = {0} having even length of the string.
  Ans: R = (00)*
- For the language L over ∑ = {0, 1} such that all the string do not contain the substring 01.
  Ans: The Language is as follows: L = {ε, 0, 1, 00, 11, 10, 100}
  R = (1* 0*)
- For the language containing the string over {0, 1} in which there are at least two occurrences of 1's between any two occurrences of 1's between any two occurrences of 0's.
  Ans: (0111*0)*.
  Similarly, if there is no occurrence of 0's, then any number of 1's are also allowed. Hence the r.e. for required language is:
  R = (1 + (0111*0))*
- The regular expression for the language containing the string in which every 0 is immediately followed by 11.
  Ans:R = (011 + 1)*
- String which should have at least one 0 and at least one 1.
  Ans: R = [(0 + 1)* 0 (0 + 1)* 1 (0 + 1)*] + [(0 + 1)* 1 (0 + 1)* 0 (0 + 1)*]
- Describe the language denoted by following regular expression (b* (aaa)* b*)*
  Ans: The language consists of the string in which a's appear triples, there is no restriction on the
  number of b's.

2.1.1 ALGEBRAIC LAWS FOR REGULAR EXPRESSIONS:

- Given R, P, L, Q as regular expressions, the following identities hold:
- ∅* = ε, ε* = ε
- RR* = R*R=R^+
- (R*)* = R*
- (PQ)*P =P(QP)*

- $(P+Q)^* = (P^*Q^*)^* = (P^*+Q^*)^*$
- $R + \emptyset = \emptyset + R = R$ (The identity for union)
- $R\,\varepsilon = \varepsilon R = R$ (The identity for concatenation)
- $\emptyset R = R\,\emptyset = \emptyset$ (The annihilator for concatenation)
- $R + R = R$ (Idempotent law)
- $P(Q+R) = PQ+PR$ (Left distributive law)
- $(Q+R)\,P = QP+RP$ (Right distributive law)
- $\varepsilon + RR^* = \varepsilon + R^*R = R^*$

2.1.2. ARDEN'S THEOREM:

Statement: Let B and C are two regular expressions. If **C** does not contain null string, then **A=B+AC** has a unique solution **A= BC***

Proof: Given that B and C are two regular expressions and **C** does not contain null string Case(i):

Let us verify whether A=BC* is a solution of A=B+AC

Substitute A =BC* in the above equation A=B+AC

$A=B+BC^*C=B(\varepsilon +C^*C)=BC^*$ since $\varepsilon+RR^*=R^*$ $BC^*=BC^*$

LHS=RHS==>

Therefore A=BC* is a solution of A=B+AC

Case (ii): Let us PT A=BC* is a unique solution of A=B+AC A=B+AC

$=B+(B+AC)C=B+BC+AC^2$

$=B+BC+(B+AC)C= B+BC+BC^2 +AC^3$

$= B+BC+BC^2 +BC^3 +AC^4$

$=B(\varepsilon+C+C^2 +C^3 +\ldots\ldots)$

$= BC^*$

Therefore A=BC* is a unique solution

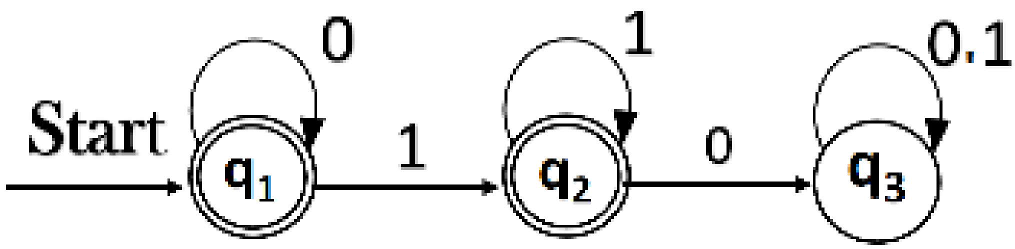
Note: Assumptions for Applying Arden's Theorem

- The transition diagram must not have NULL transitions
- It must have only one initial state.

Using Arden's theorem to construct RE from FA:

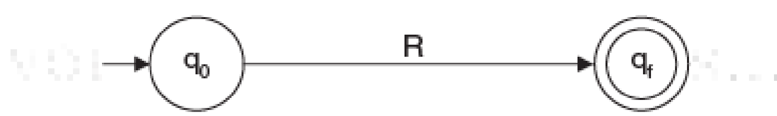
- If there are n number of states in the FA then we will get n number of equations.
- The equations are constructed in the following way:
- State name= state name from which inputs are coming. Input symbol .i.e., αji represents the transition from qj to qi then $qi = \alpha ji \,.\, qj$
- If qj is a start state then we have:
- $qi = \alpha ji * qj + \varepsilon$
- Solve the above equations to obtain final state which contains input symbols only.

Ex: Construct the regular expression for the given DFA

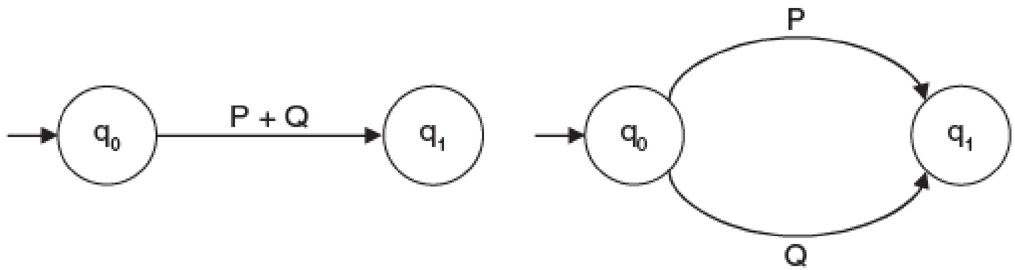


Solution:
Let us write down the equations q1 = q1 0 + ε

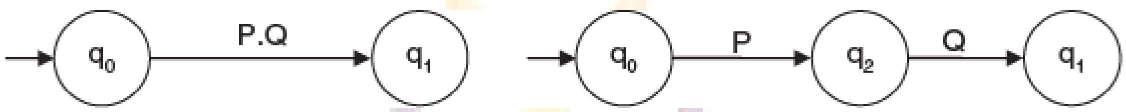
Since q1 is the start state, so ε will be added, and the input 0 is coming to q1 from q1 hence we write State = source state of input × input coming to it Similarly, q2 = q1 1 + q2 1 q3 = q2 0 + q3 (0+1)
Since the final states are q1 and q2, we are interested in solving q1 and q2 only. Let us see q1
first q1 = q1 0 + ε
We can re-write it as q1 = ε + q1 0
Which is similar to R = Q + RP, and gets reduced to R = OP*.
Assuming R = q1, Q = ε, P = 0 We get q1 = ε.(0)* q1 = 0* (ε.R*= R*)
Substituting the value into q2, we will get
q2 = 0* 1 + q2 1 q2 = 0* 1 (1)* (R = Q + RP → Q P*)
The regular expression is given by
r = q1 + q2 = $0^* + 0^*\ 1.1^*$ r = $0^* + 0^*\ 1^+\ (1.1^* = 1^+)$
Construction of FA from RE: There are two methods to construct FA from RE. They are i) Top down approach and ii) Bottom up approach.
Top down Approach:
This is divided into several steps as given in the following.
Step-1: Take two states, one is the beginning state and another is the final state. Make a transition from the beginning state to the final state and place the RE in between the beginning and final states



Step-2:If in the RE there is a + (union) sign, then there are parallel paths between the two states



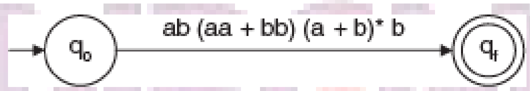
Step-3: If in the RE there is a .(dot) sign, then one extra state is added between the two states.



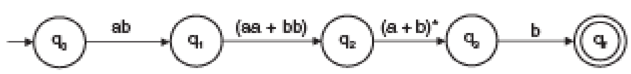
Step-4: If in the RE there is a ‘*’ (closure) sign, then a new state is added in between. A loop is added on the new state and the label Λ is put between the first to new and new to last.

Ex: Construct Finite Automata equivalent to the Regular Expression L = ab(aa + bb)(a + b)* b.

Step I: Take a beginning state q0 and a final state qf. Between the beginning and final state place the regular expression.



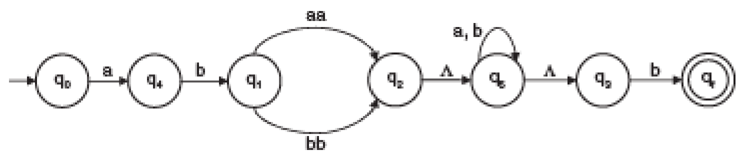
Step II: There are three dots (.) between ab, (aa + bb), (a + b)*, and b. Three extra states are added between q0 and qf.



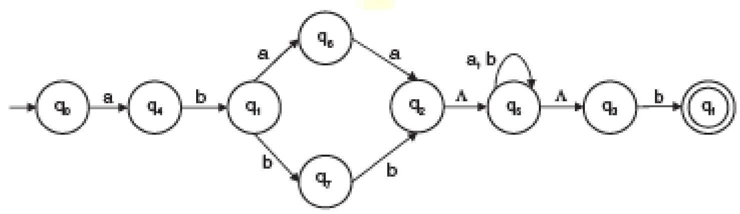
Step III: Between ‘a’ and ‘b’ there is a dot (.), so extra state is added

Step IV: In aa + bb there is a +, therefore there is parallel edges between q1 and q2.

Between q2 and q3 there is (a + b)*. So, extra state q5 is added between q2 and q3. Loop with label a, b is placed on q5 and Λ transition is made between q2, q5 and q5, q3.



Step V: In aa and bb there are dots (.). Thus two extra states are added between q1 and q2 (one for aa and another bb). The final finite automata for the given regular expression is given below.



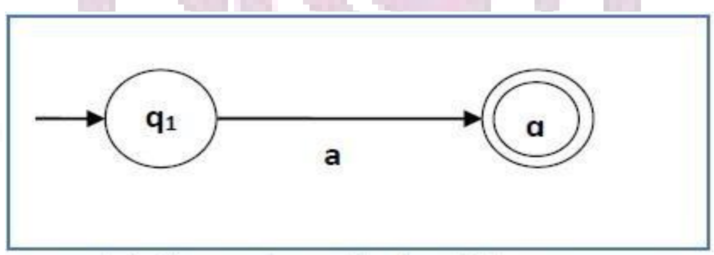
(e)

Ex: Construct an FA equivalent to the RE: L = (a + b)*(aa + bb)(a + b)*.

Ex: Construct an FA equivalent to the RE: L = ab + (aa + bb)(a + b)* b.

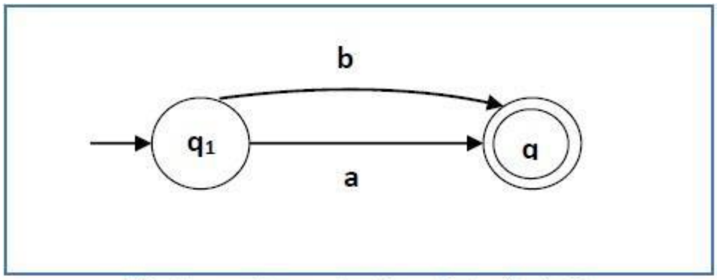
2.1.4. BOTTOM-UP APPROACH (THOMSON CONSTRUCTION):

Step-1: For input a ∈ ∑, the transition diagram is



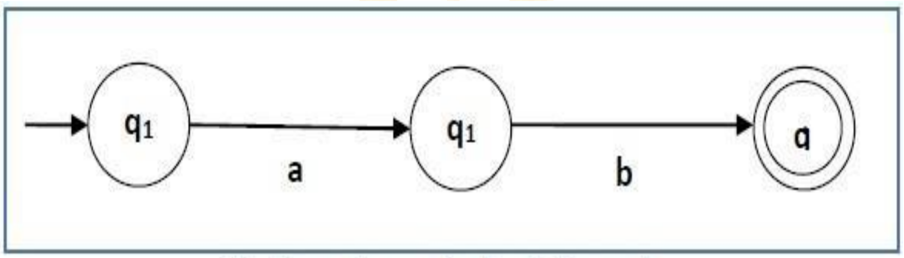
Finite automata for RE = a

Step-2: If r1 and r2 are two RES then the transition diagram for the RE r1 + r 2 is



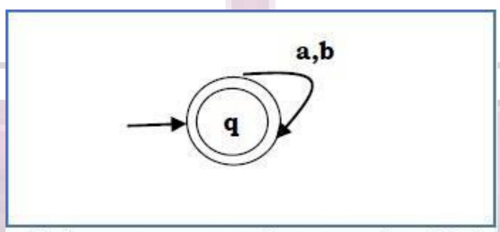
Finite automata for RE= (a+b)

Step-3: If r1 and r2 are two RES then the transition diagram for the RE r1 . r 2 is



Finite automata for RE = ab

Step-4: If r is a RE then the transition diagram for r* is

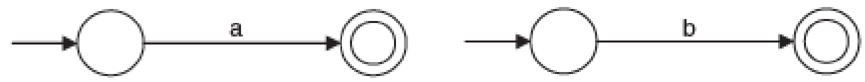


Finite automata for RE= (a+b)*

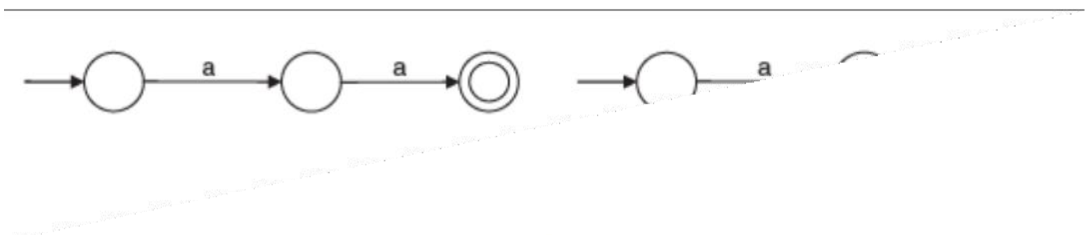
Ex: Construct Finite Automata equivalent to the Regular Expression L = ab(aa + bb)(a + b)*a.

Solution:

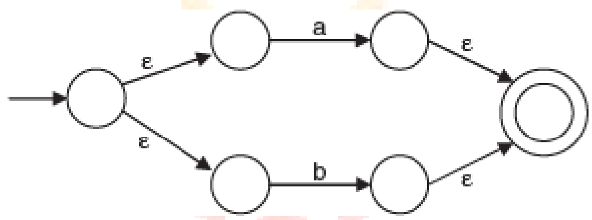
Step I: The terminal symbols in L are ‘a’ and ‘b’. The transition diagrams for ‘a’ and ‘b’ are given below:



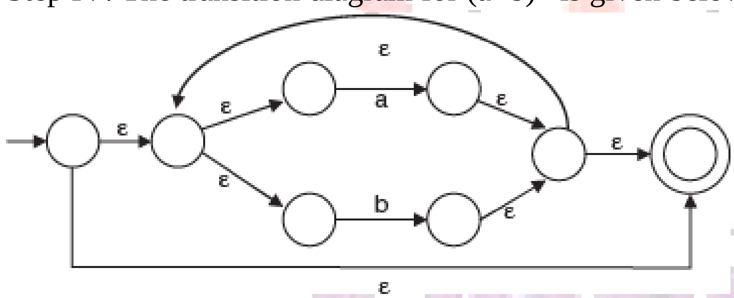
Step II: The transition diagrams for ‘aa’, ‘ab’, ‘bb’ are given below



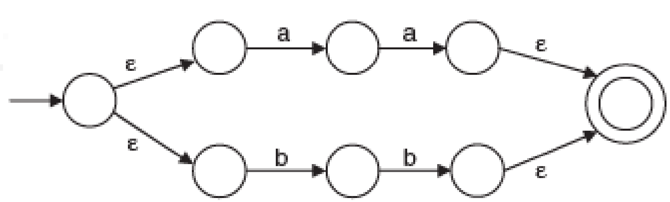
Step III: The transition diagram for (a+b) is given below



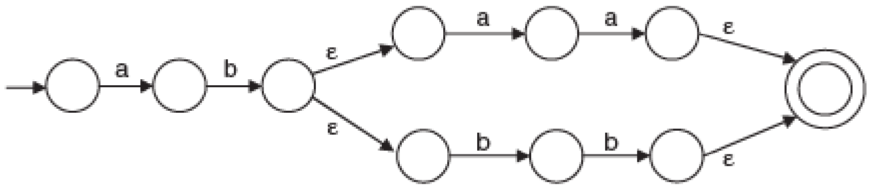
Step IV: The transition diagram for (a+b)* is given below



Step V: For (aa+bb) the transitional diagram is given below



Step VI: The constructed transitional diagram for ab(aa+bb) is given below



Step VII: The constructed transitional diagram for ab(aa + bb)(a + b)*a is given below

Language Acceptance: Start with the start symbol, at every step, and replace the non-terminal by the right-hand side (RHS) of the rule. Continue this until a string of terminals is derived. The string of terminals gives the language accepted by grammar.

Types of Grammars–Chomsky Hierarchy:

Linguist Noam Chomsky defined a hierarchy of languages, in terms of complexity. This four-level hierarchy, called the Chomsky hierarchy, corresponds to four classes of machines. Each higher level in the hierarchy incorporates the lower levels, that is, anything that can be computed by a machine at the lowest level can also be computed by a machine at the next highest level.

The Chomsky hierarchy classifies grammar according to the form of their productions into the following levels:

2.1.5. REGULAR GRAMMAR

- A regular grammar is a mathematical object, G, with four components, G = (N, T, P, S), where
- N is a nonempty, finite set of nonterminal symbols,
- T is a finite set of terminal symbols , or alphabet, symbols,
- P is a set of grammar rules, each of one having one of the forms $A \rightarrow aB$ $A \rightarrow a$ $A \rightarrow \varepsilon$, for A, B
  $\in$ N, a $\in \Sigma$, and ε the empty string, and
- S $\in$ N is the start symbol.

2.2. PUMPING LEMMA FOR RLS:

- The pumping lemma is generally used to prove certain languages are not regular
- Language is said to be regular: If a DFA,NFA or epsilon NFA can be constructed to exactly accept a language
- If a RE can be constructed to exactly generate the strings in a language.
  Formal Definition of Pumping Lemma:
- if L is a regular language represented with automaton with maximum of n states , then there is a word in L such that the length |Z|>=n, we may write Z=UVW in such a way that |UV|<+n,
  |V|>=1, and for all i>=0, $UV^i W$ is in L.

- Ex: Prove that $L = \{a^i b^i \mid i \geq 0\}$ is not regular.
  At first, we assume that **L** is regular and n is the number of states. Let z= aabb=uvw
  Where u=a, v= ab, w=b Whein i=0, uv^iw=uw=ab is in L
  When i=1, uv^iw=uvw=aabb is in l
  When i=2, $uv^iw=uv^2w$=aababbis not in L Hence L is not Regular
  Ex: State whether L = {a2n| n > 0} is regular.
  Ex: State whether $L = \{0^n \mid n \text{ is a prime}\}$ is regular Ex: State whether $L = \{a^n \mid n \geq 0\}$ is regular
  Ex: State whether $L = \{a^n b^m \mid n, m \geq 0\}$ is regular

2.2.1. CLOSURE PROPERTIES OF RLS

1) Context-free languages are closed under

Union: Let L1 and L2 be two context-free languages. Then L1 ∪ L2 is also context free.

Example

- Let L1 = { $a^n b^n$, n > 0}. Corresponding grammar G1 will have P: S1 → aAb|ab
- Let L2 = { $c^m d^m$, m ≥ 0}. Corresponding grammar G2 will have P: S2 → cBb| ε
- Union of L1 and L2, L = L1 ∪ L2 = { $a^n b^n$ } ∪ { $c^m d^m$ }
- The corresponding grammar G will have the additional production S → S1 | S2

Concatenation: If L1 and L2 are context free languages, then L1L2 is also context free.

Example: Union of the languages L1 and L2, L = L1L2 = { $a^n b^n c^m d^m$ }

The corresponding grammar G will have the additional production S → S1 S2

Kleene Star: If L is a context free language, then L* is also context free.

Example

- Let L = { $a^n b^n$, n ≥ 0}. Corresponding grammar G will have P: S → aAb| ε
- Kleene Star L1 = { $a^n b^n$ }*
- The corresponding grammar G1 will have additional productions S1 → SS1 | ε

Context-free languages are not closed under Intersection: If L1 and L2 are context free languages, then

L1 ∩ L2 is not necessarily context free.

Intersection with Regular Language − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.

Complement − If L1 is a context free language, then L1' may not be context free

2.2.2. Decision Properties of Regular Languages:

- A property is a yes/no question about one or more languages.
- Some examples:

❖Is *L* empty?

❖Is *L* finite?

❖Are *L*1 and *L*2 equivalent?

❖*Is 'w' a member in L?*

- A property is a decision property for regular languages if an algorithm exists that can answer the question (for regular languages).

2.2.3. MINIMIZATION OF DFA: REDUCTION OF NO OF STATES IN FA:

Any DFA defines a unique language but the converse is not true i.e., for any language there is a unique DFA is not always true.

INDISTINGUISHABLE AND DISTINGUISHABLE STATES:

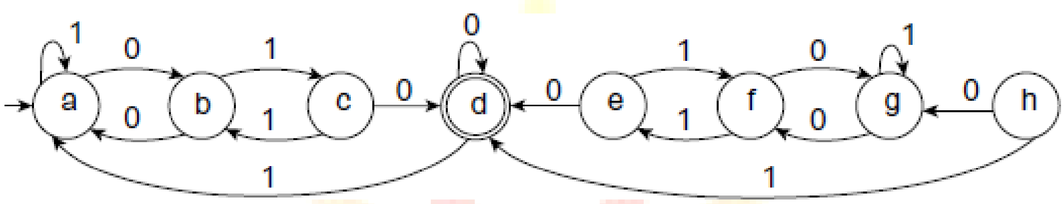
Two states p and q of a DFA are **indistinguishable** if δ(p,w) is in F => δ(q,w) is in F and δ(p,w) is not in F => δ(q,w) is not in F

Two states p and q of a DFA are **distinguishable** if δ(p,w) is in F and δ(q,w) is not in F or vice versa.

DFA MINIMIZATION: MYHILLNERODE THEOREM

Algorithm:
Input − DFA, Output − Minimized DFA
Step 1 :For each pair [p,q] where p is in F and q is in Q-F, mark[p,q]=X
Step 2 :For each pair of distinct state [p,q] in FXF or (Q-F)X(Q-F) do
- if for some input symbol a, δ([p,q],a)=[r,s], if [r,s]=X then
- ❑ mark[p,q]=X
- ❑ Recursively mark all unmarked pairs which lead to [p,q] on input for all a is in ∑
- else
- ❑ For all input symbols a do

put [p.q] on the list for δ([p,q],a) unless δ([p,q],a)=[r,r]
Step 3: For each pair [p,q] which is unmarked are the states which are equivalent
Ex: Find minimum-state automaton equivalent to the transition diagram



Transition Table:

	0	1
a	b	a
b	a	c
c	d	b
d	d	a
e	d	f
f	g	e
g	f	g
h	g	D

Q={a,b,c,d,e,f,g,h}F={d}NF={a,b,c,e,f,g,h}
Step1: FXNF={(d,a), (d,b), (d,c),(d,e),(d,f),(d,g),(d,h)}
Mark the above states as one is final and other is non final.

b							
c							
d	X	X	X				
e				X			
f				X			
g				X			
h				X			
	a	b	C	d	e	f	g

NFX NF={(a,b),(a,c),(a,e),(a,f),(a,g),(a,h),(b,c),(b,e),(b,f),(b,g),(b,h),(e,f),(e,g),(e,h),(f,g),(f,h),(g,h)}

Step 2:

(a)Find the states that are distinguishable with a

δ([a, b], 0) = [b, a] δ([a, b], 1) = [a, c]

δ([a, c], 0) = [b, d] δ([a, c], 1) = [a, b] since [b,d]=X mark [a,c]=X since [a,c]=X mark [a,b]=X

δ([a, e], 0) = [b, d] δ([a, e], 1) = [a, f] since[b,d]=X mark[a,e]=X

δ([a, f], 0) = [b, g] δ([a, f], D) = [a, e] since [a,e]=X mark [a,f]=X

δ([a, g], 0) = [b, f] δ([a, g], D) = [a, g]

δ([a, h], 0) = [b, g] δ([a, h], D) = [a, d] since[a, d]=X mark[a,h]=X

(b)Find the states that are distinguishable with b

δ([b, c], 0) = [a,d] δ([b, c], 1) = [c, b] since [a,d]=X mark [b,c]=X

δ([b, e], 0) = [a, d] δ([b, e], 1) = [c, f] since[a,d]=X mark[b,e]=X

δ([b, f], 0) = [a, g] δ([b, f], 1) = [c, e]

δ([b, g], 0) = [a, f] δ([b, g], 1) = [c, g] since [a,f]=X mark[b,g]=X

δ([b, h], 0) = [a, g] δ([b, h], 1) = [c, d] since[c, d]=X mark[b,h]=X (c)Find the states that are distinguishable with c

δ([c, e], 0) = [d, d] δ([c, e], 1) = [b, f]

δ([c, f], 0) = [d, g] δ([c, f], 1) = [b, e] since [d,g]=X mark[c,f]=X

δ([c, g], 0) = [d, f] δ([c, g], 1) = [b, g] since [d,f]=X mark[c,g]=X

δ([c, h], 0) = [d, g] δ([c, h], 1) = [b, d] since[d,g]=X mark[c,h]=X

(e)Find the states that are distinguishable with e

δ([e, f], 0) = [d, g] δ([e, f], 1) = [f, e] since [d,g]=X mark[e,f]=X

δ([e, g], 0) = [d, f] δ([e, g], 1) = [f, g] since [d,f]=X mark[e,g]=X

δ([e, h], 0) = [d, g] δ([e, h], 1) = [f, d] since[d,g]=X mark[e,h]=X

(f) Find the states that are distinguishable with f

δ([f, g], 0) = [g, f] δ([f, g], 1) = [e, g] since [e,g]=X mark[f,g]=X

δ([f, h], 0) = [g, g] δ([f, h], 1) = [e, d] since[e,d]=X mark[f,h]=X

(g) Find the states that are distinguishable with g

δ([g, h], 0) = [f, g] δ([g, h], 1) = [g, d] since[g,d]=X mark[g,h]=X

b	X						
c	X	X					
d	X	X	X				
e	X	X		X			
f	X		X	X	X		
g		X	X	X	X	X	
h	X	X	X	X	X	X	X
	A	b	c	D	e	f	g

	0	1
a	b	a
b	a	c
c	d	b
d	d	a
e	d	f
f	g	e
g	f	g
h	g	d

	0	1
a	b	a
b	a	c
c	d	b
d	d	a
e=c	d	f=b
f=b	g=a	e=c
g= a	f=b	g=a
h	g=a	d

	0	1
a	b	a
b	a	c
c	d	b
d	d	a
c	d	b
b	a	C
a	b	a
h	a	d

In the above table, [a,g], [b,f] and [c,e] are equivalent states. Hence a==g, b==f, and c==e

Simplified DFA

	0	1
A	b	A
B	a	C
C	d	B
D	d	A
H	a	D

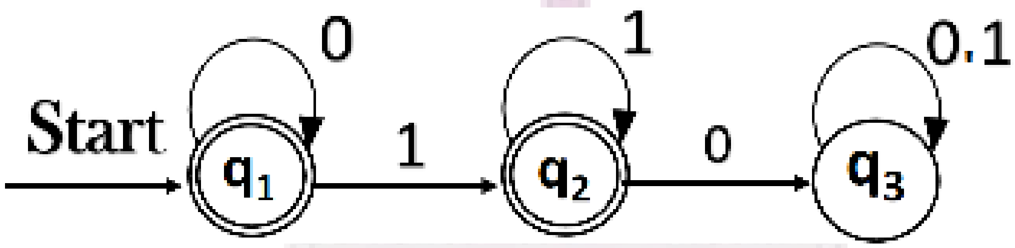
IMPORTANT UESTIONS:

Part-A

1. Define Regular Expression?
2. Write a regular expression for the language accepting all the strings in which any number of a's is followed by any number of b's is followed by any number of c's.
3. State ARDEN'S THEOREM
4. State and prove ARDEN'S theorem
5. Define Regular Grammar
6. State pumping lemma for CFL

Part-B

7. Construct the regular expression for the given DFA



8. Construct an FA equivalent to the RE: L = (a + b)*(aa + bb)(a + b)*.
9. Construct Finite Automata equivalent to the Regular Expression L = ab(aa + bb)(a + b)*a using bottom-up approach.
10. Construct Regular grammar for the RE a*(a+b)b*
11. Applications of pumping lemma
12. Closure Properties of CFLs

UNIT-3

Context-Free Grammars: Definition of Context- Free Grammars, Derivations Using a Grammar, Left most and Right most Derivations, the Language of a Grammar, Sentential Forms, Parse Tress, Applications of Context-Free Grammars, Ambiguity in Grammars and Languages. Push Down Automata: Definition of the Push down Automaton, the Languages of a PDA, Equivalence of PDA's and CFG's, Acceptance by final state, Acceptance by empty stack, Deterministic Pushdown Automata, From CFG to PDA, From PDA to CFG.

3.1. Context-Free Grammar:

Def: A grammar G=(V,T,P,S) is said to be CFG if all productions in Pare of the form $\alpha \longrightarrow \beta$ Where α is in V , i.e., set of non-terminals and $|\alpha| = 1$, i.e., there will be only one non-terminal at the left hand side (LHS) and β is in V U Σ, i.e., β is a combination of non-terminals and terminals.

Ex: Construct a CFG for the language $L = \{WCW^R \mid W \in (a, b)^*\}$ Ans: $S \longrightarrow aSa/bSb/C$

Ex: Construct a CFG for the regular expression (0 + 1)* 0 1*. Ans:

$S \longrightarrow ASB/0$ $A \longrightarrow 0A/1A/\varepsilon$ $B \longrightarrow 1B/\varepsilon$

Ex:Construct a CFG for the regular expression (011 + 1)* (01)*.

Ans:

$S \longrightarrow BC$ $B \longrightarrow AB/\varepsilon$ $A \longrightarrow 011/1$ $C \longrightarrow DC/\varepsilon$ $D \longrightarrow 01$

Ex: Construct CFG for defining palindrome over {a , b}.

Ans: $S \rightarrow aSa/bSb/a/b/\varepsilon$

Ex: Construct CFG for the set of strings with equal number of a’s and b’s.

Ans: $S \rightarrow SaSbS /SbSaS/\varepsilon$

Ex: Write the language generated by the grammar $S \rightarrow SaSbS /SbSaS/\varepsilon$

Ex: Write the language generated by the grammar $S \rightarrow aSa/bSb/a/b/\varepsilon$

Ex: Write the language generated by the grammar $S \longrightarrow aSa/bSb/C$

3.1.1. DERIVATION AND PARSE TREE:

- **Derivation:** The process of generating a language from the given production rules of a grammar. The non-terminals are replaced by the corresponding strings of the right hand side (RHS) of the production. But if there are more than one non-terminal, then which of the ones will be replaced must be determined. Depending on this selection, the derivation is divided into two parts:
- **Leftmost derivation**: A derivation is called a leftmost derivation if we replace only the leftmost non-terminal by some production rule at each step of the generating process of the language from the grammar.
- **Rightmost derivation:** A derivation is called a rightmost derivation if we replace only the right- most non-terminal by some production rule at each step of the generating process of

the language from the grammar.

Ex: Derive a^4 from by grammar S → aS/ ε

S ⇒aS⇒aaS⇒aaaS⇒aaaaS⇒aaaaε = aaaa

The language has the strings {ε, a, aa, aaa, …….}.

Ex: Derive a^2 from by grammarS → SS/ a/ ε

Ans: S ⇒ SS⇒Sa⇒aa (or)

S ⇒ SS⇒ SSS⇒SSa⇒SSSa⇒SaSa⇒εaSa⇒εa εa = aa

Ex: Find L(G) and derive the string abbab for the following grammar?

S → aS/bS/a/b Solution:

S ⇒aS⇒abS⇒abbS⇒abbaS⇒abbab

Context free language generated by the grammar is (a + b)+.

Ex: Find the language and derive abbaaba from the following grammar: S → XaaX X → aX | bX |ε

Solution:

CFL is (a + b)*aa(a + b)*.

We can derive abbaaba as follows:

S ⇒XaaX⇒aXaaX⇒abXaaX⇒abbXaaX⇒abbεaaX = abbaaX⇒abbaabX⇒abbaabaX ⇒abbaabae⇒abbaaba

Ex: Give the language defined by grammarG = {{S}, {a}, {S → SS}, S}

Ans: L(G) = Φ. Since there is no terminal that is derived from S.

Ex: Give the language defined by grammar

G = {{S, C}, {a, b}, P, S} where P is given byS → aCa, C → aCa | b,

Ans:S ⇒aCa⇒aaCaa⇒aaaCaaa L(G) = {$a^n ba^n$ / n ≥ 1}.

Ex: Give the language defined by grammarG = {{S}, {0, 1}, P, S} where P is given by S → 0S1 | |ε

Ans: S ⇒ 0S1 ⇒ 00S11 ⇒ 0011.

L(G) = {$0^n 1^n$ / n ≥ 0}.

- Construct the string 0100110 from the following grammar by using (i)Leftmost derivation (ii) Rightmost derivation

  S ⟶ 0S/1AA ,A ⟶ 0/1A/0B , B ⟶ 1/0BB,

  Ans: **Leftmost Derivation**

  $S \Rightarrow 0\underline{S} \Rightarrow 01\underline{A}A \Rightarrow 010\underline{B}A \Rightarrow 0100\underline{B}BA \Rightarrow 01001\underline{B}A \Rightarrow 010011\underline{A} \Rightarrow 0100110$

  (The non-terminals that are replaced are underlined.)

  Rightmost Derivation

  $S \Rightarrow 0\underline{S} \Rightarrow 01A\underline{A} \Rightarrow 01\underline{A}0 \Rightarrow 010\underline{B}0 \Rightarrow 0100B\underline{B}0 \Rightarrow 0100\underline{B}10 \Rightarrow 0100110$

  (The non-terminals that are replaced are underlined.)

  Ex: Consider the CFG ({S, X}, {a, b), P, S) where productions are S → baXaS | ab, X → Xab|aa. Find LMD and RMD for string w = baaaababaab.

  Solution: The following is a LMD:

  S ⇒baXaS {as S →baXaS}

  ⇒baXabaS {as X →Xab}

  ⇒baXababaS {as X →Xab}

⇒baaaababaS {as X →aa}
⇒baaaababaab {as S →ab} The following is a RMD:
S ⇒baXaS {as S →baXaS}
⇒baXaab {as S →ab}
⇒baXabaab {as X →Xab}
⇒baXababaab {as X →Xab}
⇒baaaababaab {as X →aa}
Any word that can be generated by a given CFG can have LMD|RMD.
Ex: Consider the CFG:S → aB | bA, A → a | aS | bAA, B → b | bS | aBB. Find LMD and RMD for (the string) w = aabbabba.
Ans: The following is a LMD:
S ⇒aB⇒aaBB⇒aabSB⇒aabbAB⇒aabbaB⇒aabbabS⇒aabbabbA⇒aabbabba
The following is a RMD:
S ⇒aB⇒aaBB⇒aaBbS⇒aaBbbA⇒aaBbba⇒aabSbba⇒aabbAbba⇒aabbabba

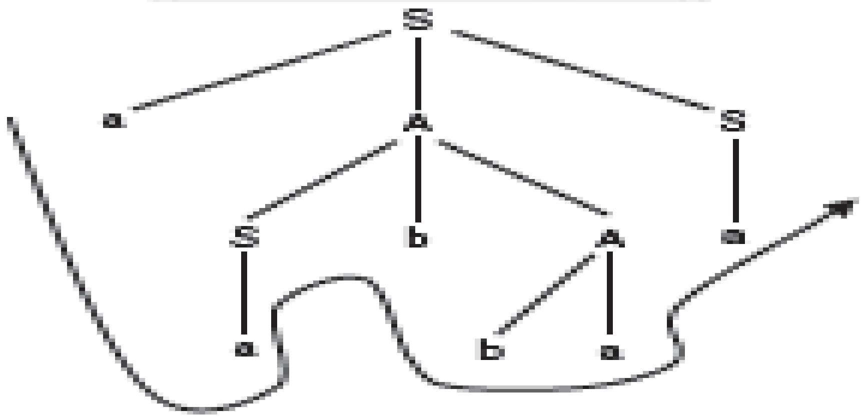
3.1.2. PARSE TREE:

- A parse tree is the tree representation of deriving a CFL from a given context free grammar. These types of trees are sometimes called as derivation trees.
- A parse tree is an ordered tree in which the LHS of a production represents a parent node and the RHS of a production represents a children node.
- **Note:** The parse tree construction is possible only for CFG.

Procedure to Construct Parse Tree:

- Each vertex of the tree must have a label. The label is a non-terminal or terminal or null (ε).
- The root of the tree is the start symbol, i.e., S.
- The label of the internal vertices is a non-terminal symbol.
- If there is a production A ⟶ $X_1X_2 \ldots X_K$, then for a vertex label A, the children of the node will be $X_1, X_2, \ldots X_K$.
  - A vertex n is called a leaf of the parse tree if its label is a terminal symbol or null (ε). **Ex:Find the parse tree for generating the string 0100110 from the following grammar.** S ⟶ 0S/1AAA ⟶ 0/1A/0B B ⟶ 1/0BB

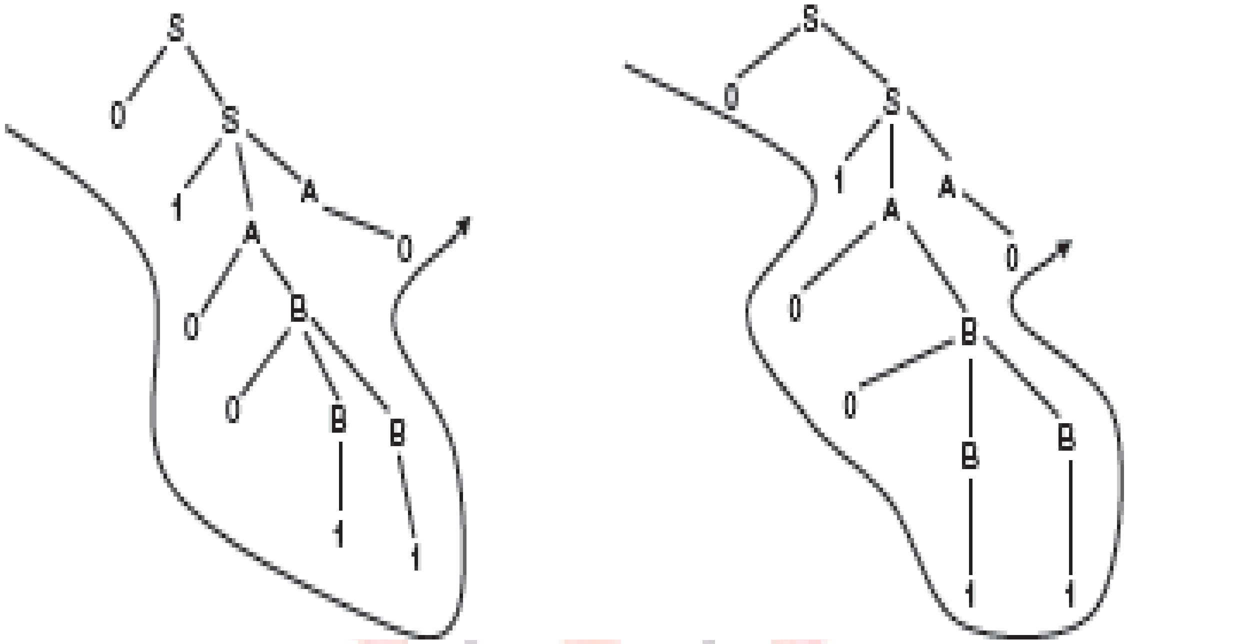
For generating the string 0100110 from the given CFG



The Left Most Derivation (LMD) will be S ⟶ 0S ⟶ 01AA ⟶ 010BA ⟶B⟶ 01001BA ⟶ 010011A ⟶ 0100110 and the derivation tree is called Left Most Derivation Tree(LMD Tree)

The Right Most Derivation (RMD) will be S → 0S → 01AA → 01A0 → → 0100BB0 → 0100B10 → 0100110 and the derivation tree is called Right Most Derivation Tree(RMD Tree).

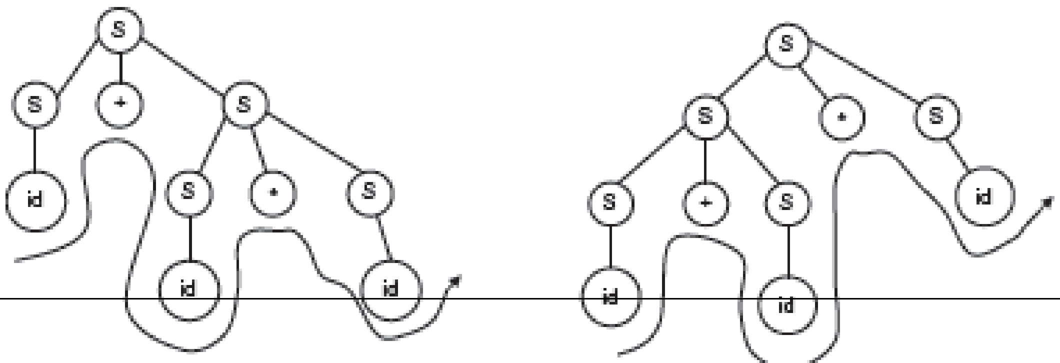
LMD AND RMD TREES:
Find the parse tree for generating the string 0100110 from the following grammar.



Left Most Derivation Tree Right Most Derivation Tree **Ex: Construct a parse tree for the string aabbaa from the following grammar.** S → a/aAS, A → SS/SbA/ba

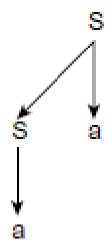
3.1.3. AMBIGUOUS GRAMMAR:

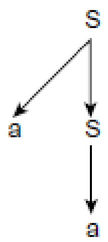
The different parse trees generated from the different derivations may be the same or may be different.
A grammar of a language is called ambiguous if any of the cases for generating a particular string, more than one parse tree(LMD Tree. RMD Tree) can be generated.
Procedure to test ambiguous Grammar: Grammar will be given.Consider a string which produces two derivation trees to prove that the grammar is ambiguous.
Ex: Prove that the following grammar is ambiguous.
P: S → E + E/E * E/id
Let us take a string id + id*id.
The string can be generated in the following ways.
Derivation (i):S=> S + S=>S + S*S => id + S*S => id + id*S=> id + id*id Derivation (ii):



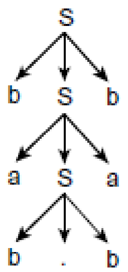
S => S*S => S + S*S => id + S*S => id + id*S => id + id*id The parse trees for derivation (i) and (ii) are shown below.

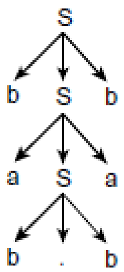
Ex: Consider the Grammar G with productions: S → aS | Sa | a.Show that G is ambiguous.
Ans: Consider the string w=aa





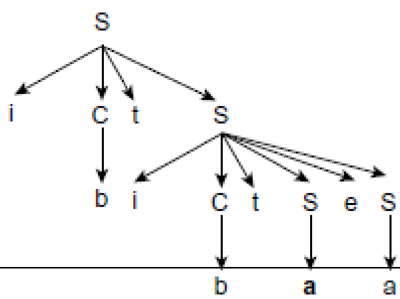
LMD Tree RMD Tree
LMD Tree!=RMD Tree. Hence the grammar is ambiguous
Ex: The grammar G for PALINDROMES isS → aSa | bSb | a | b |ε. Check if G is ambiguous.
Ans: Consider the string w=babbab.

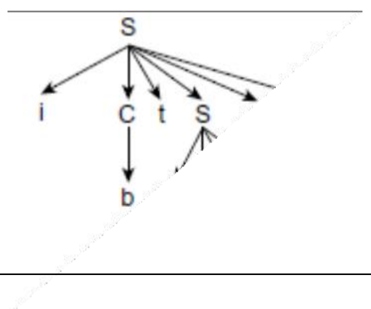




LMD Tree RMD Tree
LMD Tree=RMD Tree. Hence the grammar is unambiguous

Ex: Check whether the following grammar is ambiguous or not. S → i C t S | i C t S e S | a, C → b
Ans: Consider the string w=ibtibtaea

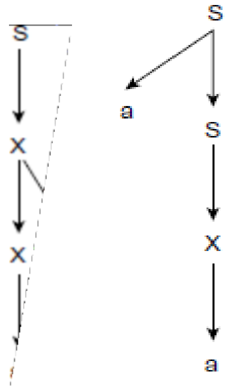




LMD Tree **RMD Tree**

LMD Tree!=RMD Tree. Hence the grammar is ambiguous Ex: Consider the Grammar G with productions: S → aS | aSb | X, X → Xa | a Show that G is ambiguous.

Ans: Consider the string w=aa



LMD Tree RMD Tree

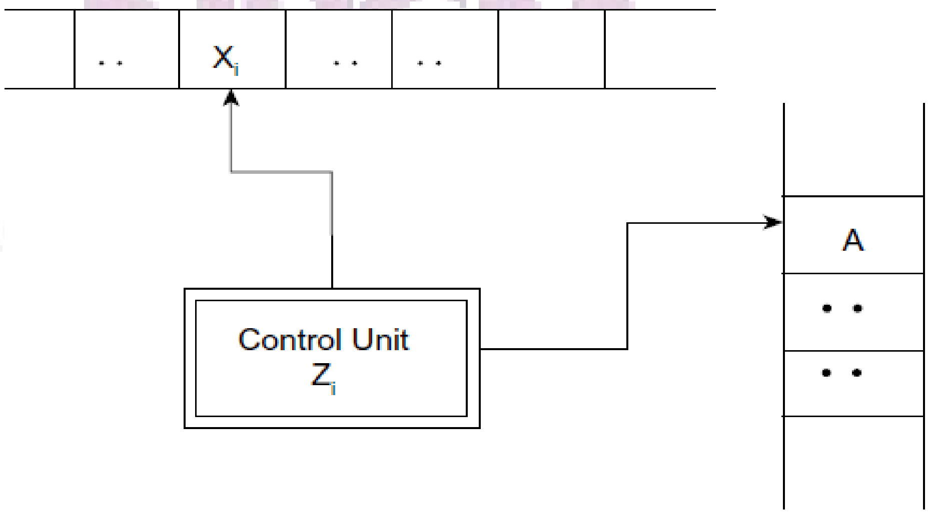
LMD Tree!=RMD Tree. Hence the grammar is ambiguous

3.2. PUSH-DOWN AUTOMATA:

Limitations of FA:

- The memory capability of Finite Automata is very limited.
- It can memorize the current input symbol.
- It cannot memorize previously processed symbols.
- Hence, by adding memory concept to FA, we will get Push down Automata.
- PDA is the same as Finite Automata with the attachment of an auxiliary amount of storage as a stack.

Block Diagram of PDA:



A PDA consists of four components:

1) An input tape, 2) a reading head, 3) a finite control and 4) a stack**.**

- **Input tape:** The input tape contains the input symbols. The tape is divided into a number of squares. Each square contains a single input character. The string placed in the input tape is traversed from left to right. The two end sides of the input string contain an infinite number of blank symbols.

- **Reading head:** The head scans each square in the input tape and reads the input from the tape. The head moves from left to right. The input scanned by the reading head is sent to the finite control of the PDA.
- **Finite control:** The finite control can be considered as a control unit of a PDA. An automaton always resides in a state. The reading head scans the input from the input tape and sends it to the finite control. A two-way head is also added with the finite control to the stack top. Depending on the input taken from the input tape and the input from the stack top, the finite control decides in which state the PDA will move and which stack symbol it will push to the stack or pop from the stack or do nothing on the stack.
- Stack: A stack is a temporary storage of stack symbols. Every move of the PDA indicates one of the following to the stack
- Push: One stack symbol may be added to the stack
- Pop: One stack symbol may be deleted from the top of the stack. In the stack, there is always a symbol z0 which denotes the bottom of the stack.

Def: Push Down Automata

A PDA consists of a 7-tuple M = (Q, Σ, G, δ, q0, z0, F), Where

Q: Finite set of states.

Σ: Finite set of input symbols.

Γ: Finite set of stack symbols.

δ: $Q\ X(\Sigma\ U\ \{\varepsilon\})\ X\Gamma^* \rightarrow QX\Gamma^*$ is a Transition function q0: Initial state of the PDA.

z0: Stack bottom symbol. F: Final state of the PDA.

PDA has 2 alphabets:

- a) An input alphabet ∑
- b) A stack alphabet Γ

Moves on PDA: A move on PDA may indicate:

- An element may be added to the stack (q, a, b) = (q, ab)
- An element may be deleted from the stack: (q, a, b) = (q, ε) and
- There may or may not be a change of state.
- δ(q, a, b) = (q, ab) indicates that in the state q on seeing a, a is pushed onto the stack. There is no change of state.
- δ(q, a, b) = (q, ε) indicates that in the state q on seeing a the current top symbol b is deleted from the stack.
- δ(q, a, b) = (q1, ab) indicates that a is pushed onto the stack and the state is changed to q1.

3.2.1. GRAPHICAL REPRESENTATION OF PDA:

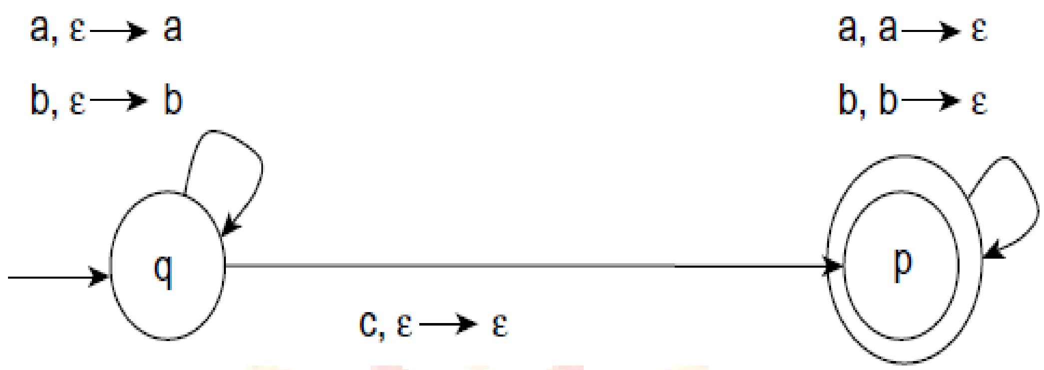
Let M = (Q, ∑, Γ, δ, q0, Z0, F) be a PDA where Q = {p, q}, ∑ = {a, b, c}, Γ = {a, b}, q0 = q, F
= {p}, and δ is given by the following equations:

δ(q, a, z0) = {(q, az0)} /*Push*/

δ(q, b, z0) = {(q, bz0)} /*Push*/

δ(q, a, a) = {(q, aa)}

δ(q, b, a) = {(q, ba)}
δ(q, a , b) = {(q, ab)}
δ(q, b, b) = {(q, bb)}
δ(q, c, z0) = {(p, z0)} /* Neither Push nor Pop*/
δ(q, c,a) = {(p, a)}
δ(q, c, b) = {(p,b)}
δ(p, a, a) = {(p, ε)}   /*Pop*/
δ(p, b, b) = {(p, ε)}   /*pop*/



3.2.2. INSTANTANEOUS DESCRIPTION OF PDA:

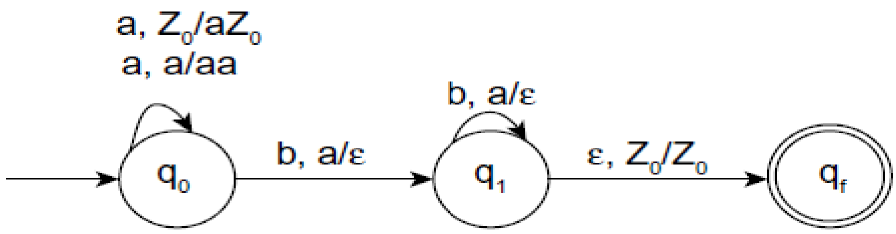
- During processing, the PDA moves from one configuration to another configuration. At any given instance, the configuration of PDA is expressed by the current state, the input symbol, and the content of stack.
- The configuration is expressed as a triple (q, x, y), where q- current state.
  x - input string to be processed.
  y- is the content of the stack where the leftmost symbol corresponds to top of stack, and the rightmost is the bottom element.
  Ex: When string ababcbcb is processed, the instantaneous description is as shown below.
  δ(q, ababcbab, z0)
  ⇒δ (q, babcbab, az0)
  ⇒δ(q, abcbab, baz0)
  ⇒δ(q, bcbab, abaz0)
  ⇒δ(q, cbab, babaz0)
  ⇒δ(p, bab, babaz0)
  ⇒δ(p, ab, abaz0)
  ⇒δ(p, b, baz0)
  ⇒ (p, ε, az0)

LANGUAGE ACCEPTANCE BY PDA:

A language can be accepted by a PDA using two approaches:

1. Acceptance by final state: The PDA accepts its input by consuming it and finally it enters the final state.
2. Acceptance by empty stack: On reading the input string from the initial configuration for some PDA, the stack of PDA becomes empty.
   Design a PDA which accepts the language L={$a^n b^n$/n>=1}

- Transition Diagram



Transition functions

δ(q0, a, Z0) = {(q0, aZ0)} /*Push a*/

δ(q0, a, a) = {(q0, aa)} /*Push a*/

δ(q0, b, a) = {(q1, ε)} /* Pop a and change the state*/

δ(q1, b, a) = {(q1, ε)} /*Pop a*/

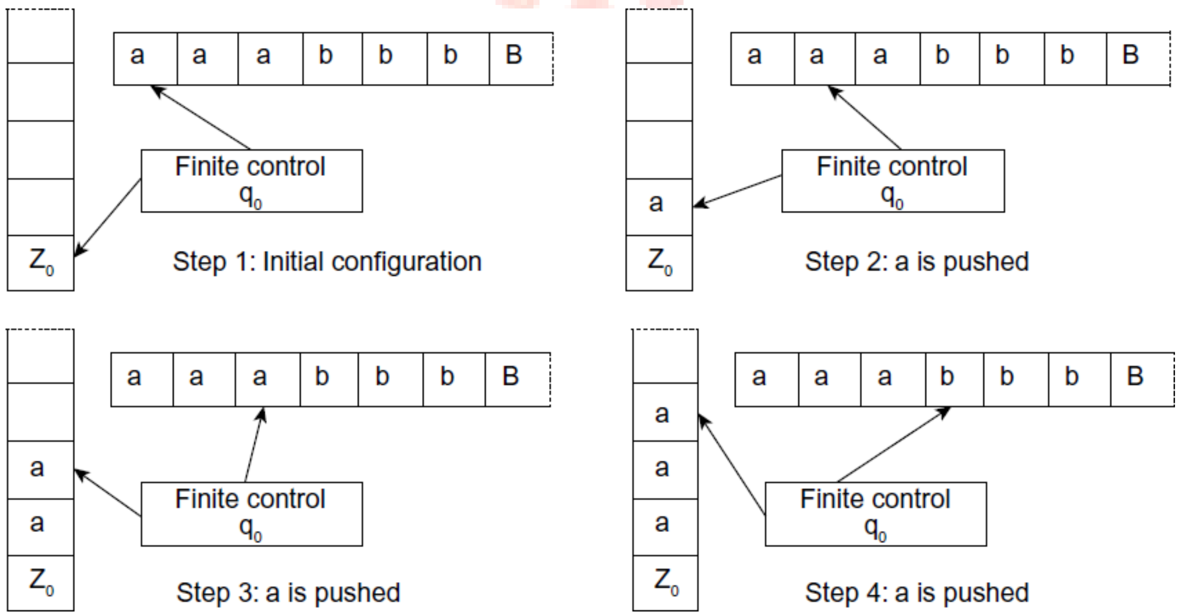
δ(q1, ε, Z0) = {(qf, Z0)} /*change to final state and halt*/

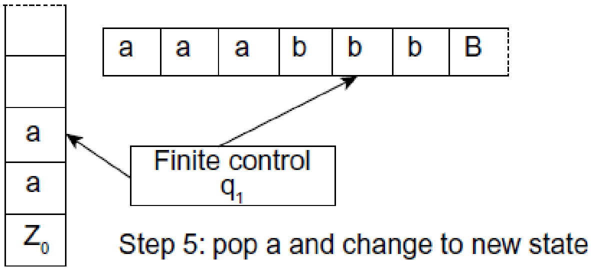
3.2.3. LANGUAGE ACCEPTANCE BY PDA:

Test whether the string aaabbb is accepted or not using

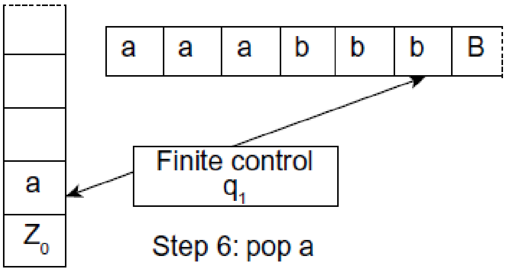
(a) Stack Empty Method (b) Final State Method

Stack Empty Method:

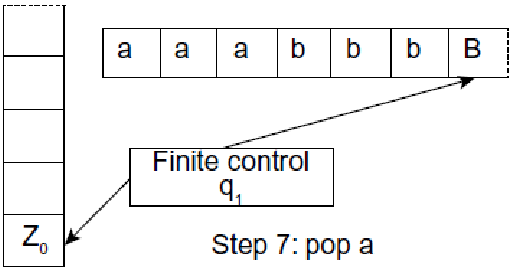




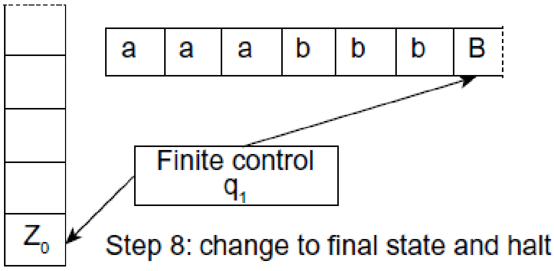
Step 5: pop a and change to new state



Step 6: pop a



Step 7: pop a



Step 8: change to final state and halt

LANGUAGE ACCEPTANCE BY PDA:

<u>Final State Method</u>

δ(q0, aaabbb, Z0)
⇒δ (q0, aabbb, aZ0)
⇒δ(q0, abbb, aaZ0)
⇒δ(q0, bbb, aaaZ0)
⇒δ(q1, bb, aaZ0)
⇒δ(q1, b, aZ0)
⇒δ(q1, ε , Z0)
⇒δ(qf, ε, Z0)
⇒string is accepted as PDA reached to final state and string is empty.

Ex: Design a PDA which accepts equal number of a's and b's over Σ = {a, b}.

$\delta(q_0, a, Z_0) = (q_0, aZ_0)$

$\delta(q_0, b, Z_0) = (q_0, bZ_0)$

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

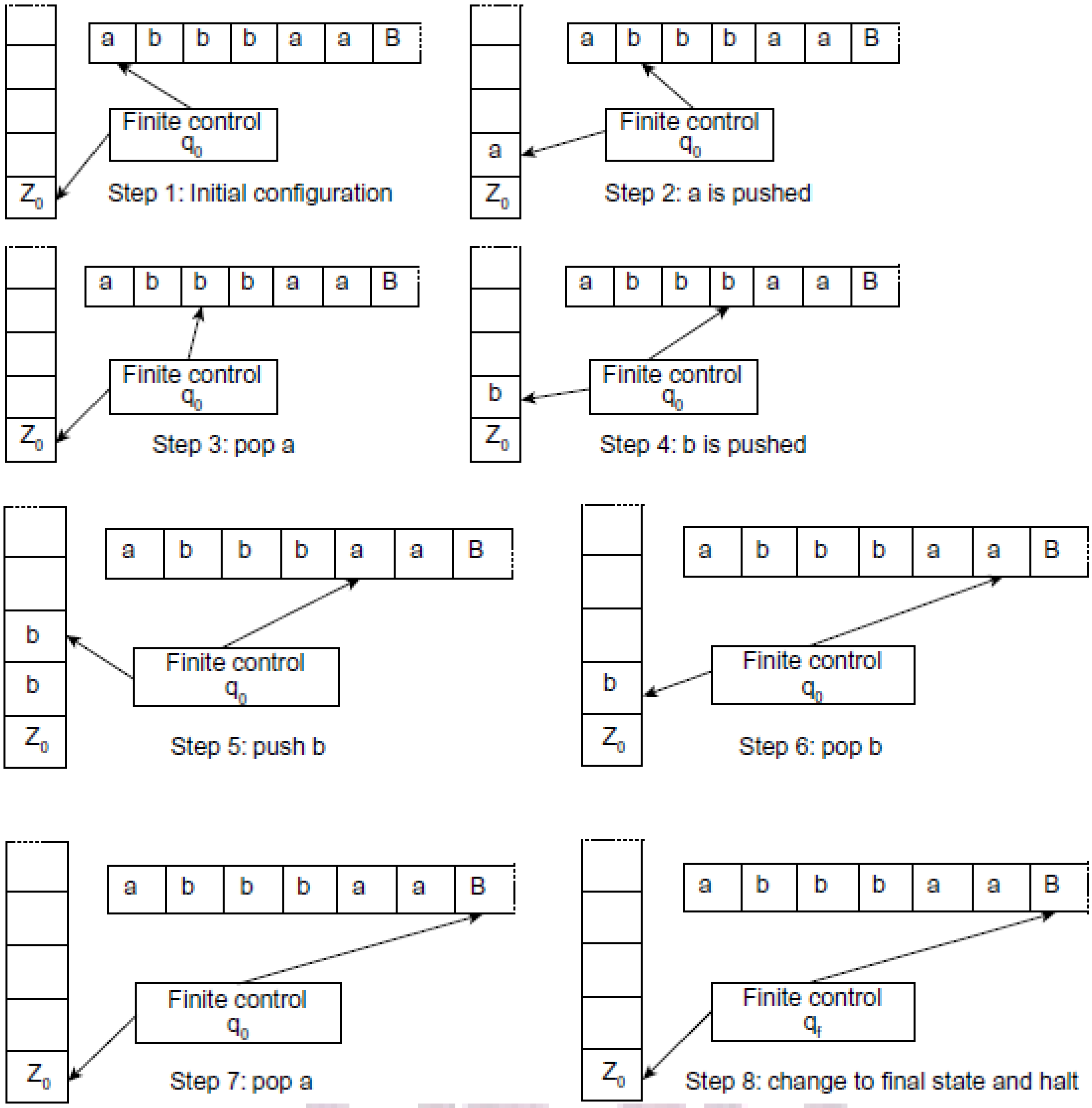
$\delta(q_0, b, b) = (q_0, bb)$

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

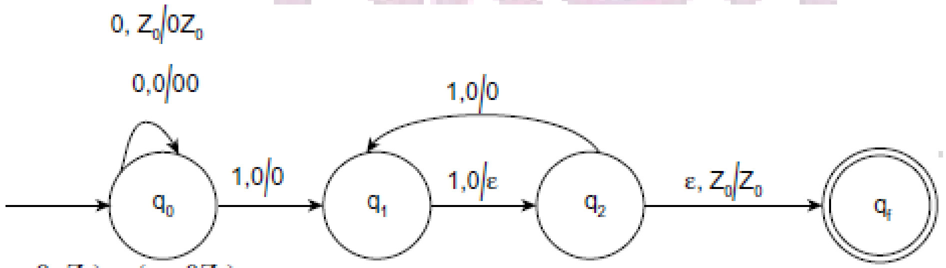
$\delta(q_0, b, a) = (q_0, \varepsilon)$

$\delta(q_0, \varepsilon, Z_0) = (q_f, Z_0)$

Consider a string abbbaa



Ex: Design a PDA that accepts $L=\{0^n1^{2n}/n>=1\}$



$\delta(q_0, 0, Z_0) = (q_0, 0Z_0)$

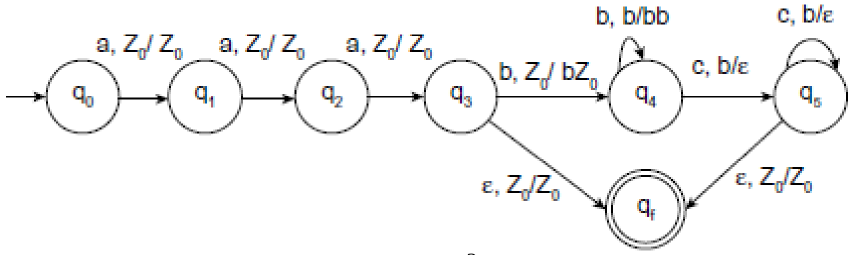
$\delta(q_0, 0, 0) = (q_0, 00)$

$\delta(q_0, 1, 0) = (q_1, 0)$

$\delta(q_1, 1, 0) = (q_2, \varepsilon)$

$\delta(q_2, 1, 0) = (q_1, 0)$

$\delta(q_2, \varepsilon, Z_0) = (q_f, Z_0)$



Ex: Design a PDA that accepts L={$a^3b^nc^n$/n>=0}

$\delta(q_0, a, Z_0) = (q_1, Z_0)$

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

$\delta(q_2, a, Z_0) = (q_3, Z_0)$

$\delta(q_3, \varepsilon, Z_0) = (q_f, Z_0)$

$\delta(q_3, b, Z_0) = (q_4, bZ_0)$

$\delta(q_4, b, b) = (q_4, bb)$

$\delta(q_4, c, b) = (q_5, \varepsilon)$

$\delta(q_5, c, b) = (q_5, \varepsilon)$

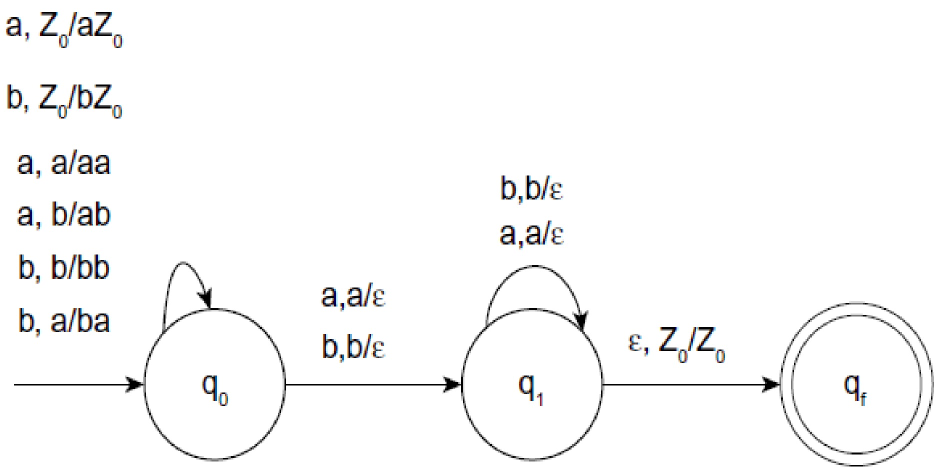
$\delta(q_5, \varepsilon, Z_0) = (q_f, Z_0)$

Ex: Design a PDA that accepts L={wcw^r / w is in $(a+b)^*$}

3.2.4. TYPES OF PDA:

- There are two types of PDA.
- Deterministic PDA (DPDA)
- Non-Deterministic PDA (NPDA)
- Deterministic PDA (DPDA): A PDA that has at most one choice of move in any state is called a deterministic PDA.
- Non-Deterministic PDA (NPDA) provides non-determinism in the moves defined.
- Deterministic PDAs (DPDAs) are very useful in programming languages. For example, parsers used in Yet Another Compiler Compiler (YACC) are deterministic PDA's (DPDA).
- A PDA M = (Q, Σ, G, δ, q0, z0, F), is (i) deterministic if and only if δ(q, a, X) has at most one move.

  (ii) Non-Deterministic if and only if δ(q, a, X) has one or more moves.Ex: Design a PDA which accepts L={WW^R|W is in (a+b)*} Transition Diagram



Transition Functions

- δ(q0, a, Z0) = (q0, aZ0)
- δ(q0, b, Z0) = (q0, bZ0)
- δ(q0, a, a) = (q0, aa)
- δ(q0, a, a)=(q1, ε)
- δ(q0, b, b) = (q0, bb)
- δ(q0, b, b)=(ql, ε)
- δ(q0, a, b) = (q0, ab)
- δ(q0, b, a) = (q0, ba)
- δ(q1, a, a) = (ql, ε)

  δ(ql, b, b) = (q1, ε), δ(q1, ε, Z0) = (qf, Z0)

3.2.5. CONSTRUCTION OF PDA FROM CFG:

- Step 1 − Convert the productions of the CFG into GNF.
- Step 2 − The PDA will have only one state {q}.
- Step 3 − the start symbol of CFG will be the start symbol in the PDA.
- Step 4 − All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.
- Step 5 − For each production in the form A → aX make a transition δ (q, a, A)=(q,X).
- Step 6- For each production in the form A → a make a transition δ (q, a, A)=(q, ε).

  <u>Ex:</u> Convert the following CFG in to PDA S→aAA, A→aS/bS/a

  Sol: The grammar is in GNF For S→aAA: δ (q, a, S)=(q,AA).

  For A→aS : δ (q, a, A)=(q,S).

  For A→bS : δ (q, b, A)=(q,S)

  For A→a :δ (q, a, A)=(q, ε).

  For A → aX :δ (q, a, A)=(q,X).

  For A → a :δ (q, a, A)=(q, ε)

  The Equivalent PDA:

  δ (q, a, S)=(q,AA).

  δ (q, a, A)=(q,S).

  δ (q, b, A)=(q,S)

  δ (q, a, A)=(q, ε)

3.2.6. CONSTRUCTING CFG FOR GIVEN PDA

- To convert the PDA to CFG, we use the following three rules:
- R1: The productions for start symbol S are given by S ➔ [qO, ZO, q] for each state q in Q.
- R2: Each move that pops a symbol from stack with transition as δ(*q*, a, Zi) = (*q*1, ε) induces a production as [*q*, Zi, *q*1] ➔ a for *q*1in Q.
- R3: Each move that does not pop symbol from stack with transition as
- δ (*q*, a, ZO) = (*q* , Z1Z2 Z3Z4…..) induces a production as [*q*, ZO, *q*m] → a[*q*l, Zl q2] [q2, Z2 q3] [q3, Z3 q4] [q4, Z4 q5]…[qm-l, Zm*q*m] for each *q*i in Q, where l <i< m.
- After defining all the rules, apply simplification of grammar to get reduced grammar

  Ex: Give the equivalent CFG for the following PDA M = {{q0 , q1 },{a, b}, {Z, ZO}, δ, qO, ZO} where δ is defined by

δ(qO, b, ZO) = (qO, ZZO)δ(qO, ε, ZO) = (qO, ε)δ(qO, b, Z) = (qO, ZZ)
δ(qO, a, Z) = (q1, Z)δ(q1, b, Z) = (q1, ε)δ(q1, a, ZO) = (qO, ZO)
Solution: The states are qO andql, and the stack symbols are Z and ZO.
The states are {S, [qO, ZO, qO], [qO, ZO, q1], [q1, ZO, qO], [q1, ZO, ql], [qO, Z, qO], [qO, Z, ql], [ql, Z, qO], [q1, Z, q1]}. S- Productions are given by Rule 1
S → [qO, ZO, qO] | [qO, ZO, q1]

(1) The CFG for δ(qO, b, ZO) = (qO, ZZO) is obtained by rule 3 [qO, ZO, qO] → b [qO, Z, qO] [qO, ZO, qO]
[qO, ZO, qO] → b [qO, Z, q1] [q1, ZO, qO]
[qO, ZO, q1] → b [qO, Z, qO] [qO, ZO, q1]
[qO, ZO, q1] →b [qO, Z, q1] [q1, ZO, q1]

(2) The CFG for δ(qO, ε, ZO) = (qO, ε) is obtained by rule 2 [qO, ZO, qO] →ε

(3) The CFG for δ(qO, b, Z) = (qO, ZZ) is obtained by rule 3 [qO, Z, qO] → b [qO, Z, qO] [qO, Z, qO]
[qO, Z, qO] → b [qO, Z, q1] [q1, Z, qO]
[qO, Z, q1] → b [qO, Z, qO] [qO, Z, q1]
[qO, Z, q1] → b [qO, Z, q1] [q1, Z, q1]

(4) The CFG for δ(qO, a, Z) = (q1, Z) is obtained by rule 3 [qO, Z, qO] → a [q1, Z, qO]
[qO, Z, q1] → a[q1, Z, q1]

(5) The CFG for δ(q1, b, Z) = (q1, ε) is obtained by rule 2 [q1, Z, q1] → b

(6) The CFG for δ(q1, a, ZO) = (qO, ZO) is obtained by rule 2 [ql, ZO, qO] → a [qO, ZO, qO]
[q1, ZO, q1] →a[qO, ZO, q1]

Simplifying grammar: In the above grammar, first identify the non-terminals that are not defined and eliminate the productions that refer to these productions. Similarly, use the procedure of eliminating the useless symbols and useless productions. Hence the complete grammar is as follows

S ➔ [qO, ZO, qO]

[qO, ZO, qO] ➔b [qO, Z, q1] [q1, ZO, qO]

[qO, ZO, qO] ➔ε

[qO, Z, q1] ➔b [qO, Z, q1] [q1, Z, q1] [qO, Z, q1] ➔a[q1, Z, q1]

[q1, Z, q1] ➔b

[q1, ZO, qO] ➔a [qO, ZO, qO]

3.2.7. APPLICATIONS OF CONTEXT-FREE GRAMMAR

- The compiler is a program that takes a program written is the source language as input and translates it into an equivalent program in the target language.
- Syntax analysis in an important phase in the compiler design.
- In this phase, mainly grammatical errors called syntax errors are checked.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by context-free grammar or not.
- If it satisfies, the parser creates the parse tree of that program. Otherwise, the parser gives the error messages

- CFGs are used in speech recognition and also in processing spoken words.

IMPORTANT QUESTIONS:

PART-A

- What are the limitations of FA
- Draw the block diagram of PDA
- Define PDA
- Define instantaneous description of PDA.

PART-B

- Design a PDA which accepts the language L={$a^n b^n$/n>=1}
- Design a PDA which accepts L={WW^R|W is in (a+b)*}
- Convert the following CFG in to PDA S→aAA, A→aS/bS/a
- Give the equivalent CFG for the following PDA M = {{q0 , q1 },{a, b}, {Z, ZO}, δ, qO, ZO} where δ is defined by δ(qO, b, ZO) = (qO, ZZO) δ(qO, ε, ZO) = (qO, ε) δ(qO, b, Z) = (qO, ZZ) δ(qO, a, Z) = (q1, Z) δ(q1, b, Z) = (q1, ε)δ(q1, a, ZO) = (qO, ZO)
- Explain two-Stack PDA and construct two-Stack PDA L={$a^n b^n c^n$: n>=1}
- Write instantaneous description for the string ababcbcb

UNIT-4

Normal Forms for Context-Free Grammars: Eliminating useless symbols, Eliminating €-Productions.Chomsky Normal form, Griebech Normal form. Pumping Lemma for Context-Free Languages: Statement of pumping lemma, Applications. Closure Properties of Context-Free Languages: Closure properties of CFL's, Decision Properties of CFL's Turing Machines: Introduction to Turing Machine, Formal Description, Instantaneous description,The language of a Turing machine

4.1. NORMAL FORM:

- For a grammar, the RHS of a production can be any string of terminals and non- terminals
- A grammar is said to be in normal form when every production of the grammar has some septic form.
- That means, instead of allowing any no of terminals and non-terminals on the RHS of the production, we permit only specific members on the RHS of the production.
- Two types of normal forms: (a) CNF (Chomsky Normal Form) and (b) GNF (Greibach Normal Form)

4.1.2. CNF: CHOMSKY NORMAL FORM

- A CFG is said to be in CNF if all the productions of the grammar are in the following form.
- Non-terminal ⟶ String of exactly two non-terminals
    - Non-terminal ⟶ Single terminal Ex: A ⟶ BC, B ⟶ b, C ⟶ c

PROCEDURE TO CONVERT CFG IN TO CNF:

1. Eliminate null productions and unit productions. i.e., simplify the grammar
2. Include productions of the form A → BC / a as it is.
3. Eliminate strings of terminals on the right-hand side of production if they exceed one as follows: Suppose we have the production S → a1a2a3 where a1,a2,a3 are terminals then introduce non-terminal Cai for terminal ai asCa1 → a1, Ca2 → a2, Ca3 → a3
4. To restrict the number of variables on the right-hand side, introduce new variables and separate them as follows:

Suppose we have the production with n non-terminals as shown below with 5 non-terminals

Y → X1 X2 X3 X4 X5

Add n-2 new productions using n-2 new non-terminals and modify the production as in the following:

Y → X1 R1 R1 → X2 R2 R2 → X3 R3

R3 → X4 X5 where the Ri are new non-terminals.

The language generated by the new CFG is the same as that generated by the original CFG.

Ex: Convert the following grammar into CNF. S $\longrightarrow$ bA/aB, A $\longrightarrow$ bAA/aS/a, B $\longrightarrow$ aBB/bS/a

Step1: The Grammar is minimized.

Step2: The productions A $\longrightarrow$ a and B $\longrightarrow$ a are in CNF. Hence leave the productions as it is.

Step 3: The productions S $\longrightarrow$ bA, S $\longrightarrow$ aB, A $\longrightarrow$ bAA, A $\longrightarrow$ aS, B $\longrightarrow$ aBB, B $\longrightarrow$ bS are not in CNF. So, we have to convert these into CNF.

Let us replace terminal 'a' by a non-terminal C_a and terminal 'b' by a non-terminal C_b.

Hence, two new productions $C_a \longrightarrow$ a and $C_b \longrightarrow$ b will be added to the grammar

By replacing a and b by new non-terminals and including the two productions, the modified grammar will be

S $\longrightarrow C_bA/C_aB$, A $\longrightarrow C_bAA/C_aS/a$, B $\longrightarrow C_aBB/C_bS/a$, $C_a \longrightarrow$ a, $C_b \longrightarrow$ b In the modified grammar, all the productions are not in CNF.

The productions A $\longrightarrow C_bAA$ and B $\longrightarrow C_aBB$ are not in CNF, because they contain more than two non-terminals at the RHS.

Let us replace AA by a new non-terminal D and BB by another new non-terminal E. Hence, two new productions D $\longrightarrow$ AA and E $\longrightarrow$ BB will be added to the grammar

So, the new modified grammar will be S $\longrightarrow C_bA/C_aB$

A $\longrightarrow C_bD/C_aS/a$ B $\longrightarrow C_aE/C_bS/a$ D $\longrightarrow$ AA

E $\longrightarrow$ BB

$C_a \longrightarrow$ a $C_b \longrightarrow$ b

It is in CNF

Ex: Convert following CFG to CNF:

S → AB | aB A → aab | ε B → bbA

Ex: Convert following CFG to CNF.

S → bA | aB A → bAA | aS | a

B → aBB | bS | b

Ex: Convert following CFG to CNF. S → ASB | ε

A → aAS | a

B → SbS | A | bb

4.1.3. LEFT RECURSION AND LEFT FACTORING:

- **Left Recursion**: A context-free grammar is called left recursive if a non-terminal 'A' as a leftmost symbol on the right side of a production. A $\longrightarrow$ Aa
- In other words, a grammar is left recursive if it has a non-terminal 'A' such that there is a derivation.
- A =>Aa for some string a
- There are two types of left recursion
- i) Direct Left Recursion ii) Indirect Left Recursion

DIRECT LEFT RECURSION:

Let the grammar be A $\longrightarrow$ Aα/β, where α and β consists of terminal and/or non-terminals but β does not start with A.

Elimination of Left Recursion:

For the production A $\longrightarrow$ Aα/β, the equivalent grammar after removing the left recursion is $A \longrightarrow \beta A^1$, $A^1 \longrightarrow \alpha A^1/\varepsilon$

In general, for a grammar in the form

$A \longrightarrow A\alpha 1 / A\alpha 2 / \ldots\ldots/A\alpha n / \beta 1 / \beta 2 / \ldots\ldots/\beta n$ The equivalent productions are

$A \longrightarrow \beta 1 A^1 / \beta 2\ A^1 / \ldots\ldots/ \beta n\ A^1\ A^1 \longrightarrow \alpha 1\ A^1 / \alpha 2\ A^1 / \ldots\ldots/\alpha n\ A^1/\varepsilon$

Ex: Remove the left recursion from the following grammar.

$E \longrightarrow E + T \mid T$, $T \longrightarrow T*F \mid F$, $F \longrightarrow id \mid (E)$

In the grammar there are two immediate left recursions $E \longrightarrow E + T$ and $T \longrightarrow T*F$. For $E \longrightarrow E + T$, the equivalent productions are $E \longrightarrow TE^1$ and $E^1 \longrightarrow +TE^1/\varepsilon$ For $T \longrightarrow T * F$, the equivalent productions are $T \longrightarrow FT^1$, $T^1 \longrightarrow *FT^1/\varepsilon$

The CFG after removing the left recursion becomes $E \Rightarrow TE^1$

$E^1 \Rightarrow +TE^1/\varepsilon$ $T \longrightarrow FT^1$

$T^1 \longrightarrow *FT^1|\varepsilon$ $F \longrightarrow id \mid (E)$

INDIRECT LEFT RECURSION:

A grammar of the form $A1 \longrightarrow A2a/b$, $A2 \longrightarrow A1c/d$ is called indirect left recursion.

Convert indirect left recursion in to direct left recursion and then apply the elimination of direct left recursion.

Consider $A2 \longrightarrow A1c/d$

Then $A2 \longrightarrow A2ac/bc/d$. it is in the direct left recursion. Eliminate Direct left recursion

$A2 \longrightarrow bcA2^1 /dA2^1$, $A2^1 \longrightarrow ac\ A2^1 /\varepsilon$

LEFT FACTORING:

A production rule of the form $A \longrightarrow \alpha\beta 1/\alpha\beta 2/\ldots/\alpha\beta n$ is called left factoring.

After left factoring, the previous grammar is transformed into: $A \longrightarrow \alpha A1$, $A1 \longrightarrow \beta 1/\beta 2/\ldots/\beta n$

Ex: Left Factor the following grammar. $A \longrightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$

The left factored grammar is $A \longrightarrow aA^1 /cdA^2$

$A^1 \longrightarrow bB/B$ $A^2 \longrightarrow g/eB/fB$

4.1.4. GREIBACH NORMAL FORM

- A grammar is said to be in GNF if every production of the grammar is of the form
- Non-terminal $\longrightarrow$ (single terminal)(non-terminal)*i.e. terminal followed by any combination of NTs including null.

Lemma I: Substitution Rule:

- Let G be a CFG.
- If $A \longrightarrow Ba$ and $B \longrightarrow b1/b2/\ldots/bn$ belongs to the production rules (P) of G, then a new grammar will $A \longrightarrow b1a/b2a/\ldots/bna$

Lemma II: Elimination of Left Recursion

- Let G be a CFG.
- If $A \longrightarrow Aa1/Aa2/\ldots/Aam/b1/b2/\ldots/bn$ belongs to P of G, then equivalent grammar is

$A \longrightarrow b1\ A^1/b2\ A^1/\ldots/bnA^1\ / b1/b2/\ldots/bn\ A^1 \longrightarrow a1A^1/a2\ A^1/\ldots/am\ A^1 /a1/a2/\ldots/am$

PROCESS FOR CONVERSION OF A CFG INTO GNF

- **Step I:** The given grammar is in CNF
- **Step II:** Rename the non-terminals as A1, A2 ……An with A1=S

- **Step III:** we need productions must be in the form that the RHS of productions must start with a terminal or with higher indexed variable. For each production $A_i \longrightarrow A_j\ a$

(ii) ifi<j leave the production as it is.

(iii) ifi=j then apply lemma2 (Elimination of Left Recursion)

(iv) ifi>j then apply lemma1. (Apply Substitution Rule)

- **Step IV:** For each production $A_i \longrightarrow A_j\ a$ where i<j apply substitution rule The resulting productions of the modified grammar will come into GNF. **Ex:Convert the following grammar in to GNF. S ⟶ AA/a, A ⟶ SS/b**

- **Step I:** There are no unit productions and no null production in the grammar. The given grammar is in CNF.
- **Step II:** In the grammar, there are two non-terminals S and A. Rename the non- terminals as A1 and A2 respectively. The modified grammar will be A1 ⟶ A2A2/a, A2 ⟶ A1A1/b
- **Step III:** In the grammar, A2 ⟶ A1A1 is not in the form $A_i \longrightarrow A_j\ a$ where i<j

Apply substitutionrule Therefore, A2 ⟶ A2A2A1/aA1/b

On the above production apply Lemma II, A2 ⟶ aA1X/bX/aA1/b, X ⟶ A2A1X/A2A1

The modified grammar A1 ⟶ A2A2/a, A2 ⟶ aA1X/bX/aA1/b, X ⟶ A2A1X/A2A1

- **Step IV:** apply substitution rule on A1 ⟶ A2A2/a Therefore, A1 ⟶ aA1XA2/bXA2/aA1A2/bA2/a

Apply substitution rule on X ⟶ A2A1X/A2A1

Therefore, X ⟶ aA1XA1X/bXA1X/aA1A1X/bA1X/ aA1XA1/bXA1/aA1A1/bA1 The modified grammar is

A1 ⟶ aA1XA2/bXA2/aA1A2/bA2/a A2 ⟶ aA1X/bX/aA1/b

X ⟶ aA1XA1X/bXA1X/aA1A1X/bA1X/ aA1XA1/bXA1/aA1A1/bA1 The above grammar is in GNF

Ex: Convert the following grammar in to GNF.

S → XA | BB

B → b | SB X → b

A → a

Ex: Convert the CFG to GNF

S → AB A

A → aA | εB → bB | ε

4.2. CLOSURE PROPERTIES OF CONTEXT-FREE LANGUAGES

- A set is closed (under an operation) if and only if the operation on two elements of the set produces another element of the set. If an element outside the set is produced, then the operation is not closed.
- CFL are closed under Union.
- If L1 and If L2 are two context free languages, their union L1 ∪ L2 will also be context free.
- Ex: L1 = { $a^nb^nc^m$ | m >= 0 and n >= 0 } and L2 = { $a^nb^mc^m$ | n >= 0 and m >= 0 } L1 ∪ L2 = { $a^nb^nc^m \cup a^nb^mc^m$ | n >= 0, m >= 0 } is also context free. L1 says

number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their union says either of two conditions to be true. So it is also context free language.

- **CFL are closed under Concatenation**
  If L1 and If L2 are two context free languages, their concatenation L1.L2 will also be context free.
  Ex: L1 = { a^nb^n | n >= 0 } and L2 = { c^md^m | m >= 0 } L3 = L1.L2 =
- L1 says number of a's should be equal to number of b's and L2 says number of c's should be equal to number of d's. Their concatenation says first number of a's should be equal to number of b's, then number of c's should be equal to number of d's. So, we can create a PDA which will first push for a's, pop for b's, push for c's then pop for d's. So it can be accepted by pushdown automata, hence context free.
- **CFL are closed under Kleen Closure**
  If L1 is context free, its Kleene closure L1* will also be context free. For example,
  L1 = { a^nb^n | n >= 0 }
  L1* = { a^nb^n | n >= 0 }* is also context free.
- **CFL are not closed under Intersection**
  Consider two languages L1= {$a^{n+1}b^{n+1}c^n$, where n, l>= 0} and L2 == {$a^nb^nc^{n+k}$, where n, k >= 0}.
  Consider L = L1 ∩ L2
  So, L = $a^{n+1}b^{n+1}c^n$ Ç $a^nb^nc^{n+k}$ = $a^nb^nc^n$, where n >= 0.
  $a^nb^nc^n$ is a context sensitive language not a context free. As one instance is proved not to be context free then we can decide that context free languages are not closed under intersection.

  CFL are not closed under Intersection and Complementation.
  From the set theory, we can prove L1 Ç L2 = L1 UL2. (D' Morgan's Law) If the union of the complements of L1 and L2 are closed, i.e., also context free, then the LHS will also be context free. But we have proved that L1 Ç L2 is not context free. We are getting a contradiction here. So, CFLs are not closed under complementation.

4.2. PUMPING LEMMA FOR CFL

Let L be a CFL. Then, we can find a natural number n such that 1) Every z ∈ L where|z|>=n and z can be written as z = uvwxy, for some strings u,v,w,x,yi) | vx | >=1 ii) | vwx | <= n and uv^iwx^iy∈ L for all i>= 0

Note: Method to test a language is CFL or not.

- **Step I:** Assume that L is context free. Find a natural number such that | z | >=n.
- **Step II:** So, we can write z = uvwxy for some strings u, v, w, x, y.
- **Step III:** Find a suitable k such that uv^iwx^iy is not in L. This is a contradiction, and so L is not context free.

Ex: Using Pumping Lemma, Show that L = {$a^nb^nc^n$ where n >=1} is not CFL
The given language is L = {$a^nb^nc^n$ where n >=1} L={ab, aabbcc, aaabbbccc,….}
Let z=aabbcc=uvwxy

Where u=a,v=a,w=b,x=b,y=cc

When i=0, uv^iwx^iy=uwy=abcc is not in L, Therefore L is not a CFL

Ex: ST L={a^p:p is a prime number} is not CFL

Ex: Prove that the language L = {a^{i2}/i ≥ 1 } is not context free.

4.3. Turing Machine:

Limitations of Finite State Machine/Finite Automata:

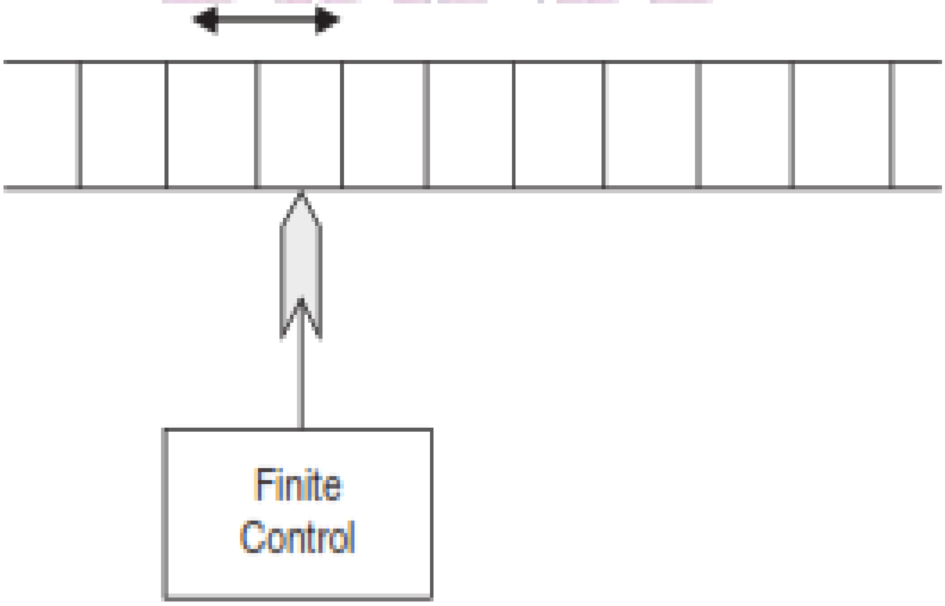
- Can remember only current symbol
- Cannot remember previous long sequence of input

Limitations of Pushdown Automata:

- It uses stack to remember any long input sequence
- Accepts a larger class of languages than that of FA, Computation power is limited
- To overcome the above limitations, Alan Turing has proposed a model called a Turing Machine(TM) with a two-way infinite tape. The tape is divided into cells, each of which can hold only one symbol. The input of the machine is a string w=w1w2w...wn initially written on the left most portion of the tape, followed by an infinite sequence of blanks B.
- The machine is able to move a read/write head left and right over the tape as it performs computation. It can read and write symbols on the tape as it pleases.

	B	B .	W 1.	W 2			W n	B	B	

BLOCK DIAGRAM OF TURING MACHINE



It is a simple mathematical model of a general purpose computer. It is capable of performing any calculation which can be performed by any computing machine. Hence this model is popularly known as “Turing Machine”.

FEATURES OF TM:

- It has external memory which remembers arbitrarily long sequence of input.
- It has unlimited memory capability.
- The model has a facility by which the input at left or right on the tape can be read easily.
- The machine can produce certain output based on its input. In this machine there is no distinction between input and output symbols.
- The TM can be thought of as a finite state automata connected to a read or write head,
- It has one tape which is divided into a number of cells. Each cell can store only one symbol.
- The read or write head can examine one cell at a time.
- In one move the machine examine the present symbol under the head on the tape and present state of an automaton to determine:
- A new symbol should be written on the tape in the cell under the head
- The head moves one cell either left(L) or right(R), The next state of the automata.
- Whether to halt or not.

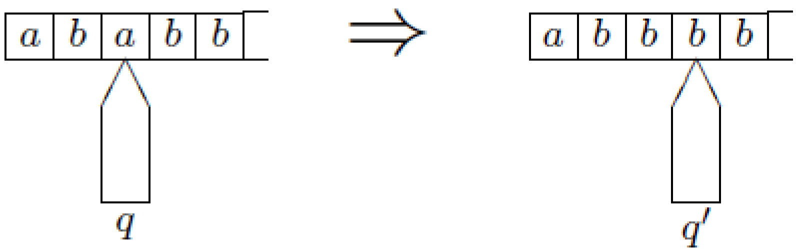
DEF: TURING MACHINE:

- A TM is expressed as a 7-tuple (Q, T, B,∑, δ, q0, F) where:
- **Q-**finite set of states
- **T** -tape alphabet (symbols which can be written on Tape)
- **B**∈ T -blank symbol (every cell is filled with B except input alphabet initially)
- ∑ -the input alphabet (symbols which are part of input alphabet)
- **δ :**Q × T → Q × T × {L,R} transition function which maps.
- **q0** -the initial state
- **F** -the set of final states.

INSTANTANEOUS DESCRIPTION OF TM

- ID of TM is (l,q,r) where
- l- tape contents left to the head of TM
- r- tape contents right to the head of TM including the symbol under head and
- q- current state

- Ex:



- Where l=ab   l=abb

  r=abb   r=bb

Current state=q current state=q^1

Moves: At any given time the move of TM depends on i) Current state and ii) input symbol i.e., (q,a). the o/p of move would be (q1, b, L) Where q1 = next state, b= symbol to be replaced by a and L= move left one symbol.

Ex: δ(qi,a)= (qj,b,L) i.e., in the state qi on receiving a symbol a , then change to a new state qj , replace a by b and the move left.

Acceptance or Rejection by TM:

- Let us assume the final Configuration of TM is (u,q,w)
- Accept: If q ∈ F
- Reject: If q ∉ F and /or next moves are not defined/loops
- If either accept or reject then TM halts(Stops)

TM as Language Accepter:

- M accepts w iff the execution of M on w terminating and ends in the accepting state
- M rejects w iff the execution of M on w terminating and ends in the non accepting state
- M does not accept w iff M rejects or M loops on w,

Ex: Write IDs for the following TM

δ(q0,a)= (q0,X,R), δ(q0,b)= (q0,b,R), δ(q0,B)= (q1,B,L), δ(q1,b)= (q1,Y,L) , δ(q1,X)= (q1,X,L), δ(
q1,B)= (q2,B,H) and string w=abba.

B	B	B	a	b	b	a	B	B	B

Current state= q0

B	B	B	X	b	b	a	B	B	B

Current state= q0

B	B	B	X	b	b	a	B	B	B

Current state= q0

B	B	B	X	b	b	a	B	B	B

Current state= q0

B	B	B	X	b	b	X	B	B	B

Current state= q0

B	B	B	X	b	b	X	B	B	B

Current state= q1

B	B	B	X	b	b	X	B	B	B

Current state= q1

B	B	B	X	b	Y	X	B	B	B

Current state= q1

B	B	B	X	Y	Y	X	B	B	B

Current state= q1

B	B	B	X	Y	Y	X	B	B	B

Current state= q1

δ(q1,B)= (q2,B,H)

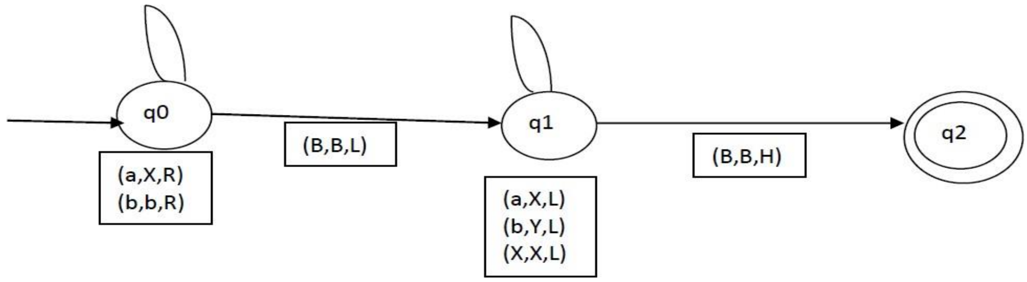
4.3.1. REPRESENTATION OF TM:

- **Representation of TM**: A TM can be represented by means of Transition Table and Transition diagram.
- **Representation of TM using Transition Table:** The Transition table for the above TM is as given below.

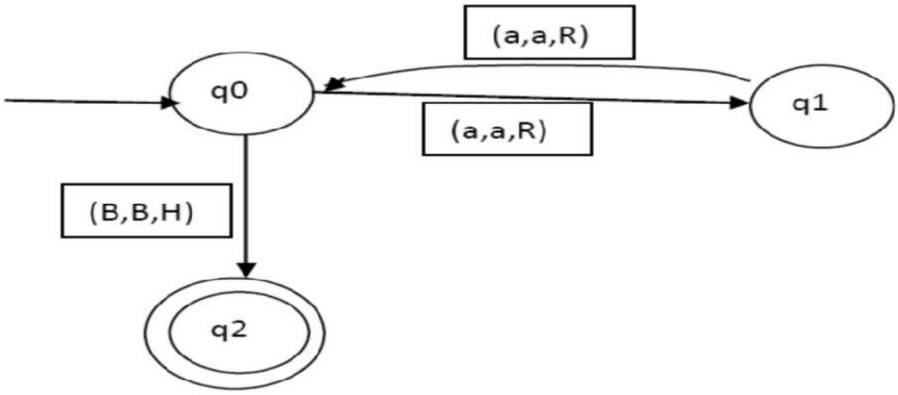
δ	**A**	**b**	**X**	**Y**	**B**
q0	(q0,X,R)	(q0,b,R)	--	-	(q1,B,L)
q1	--	(q1,Y,L)	(q1,X,L)	-	(q2,B,H)

Representation of TM using Transition Diagram: The states are represented by vertices

and transitions are represented by directed edges. The edges are labeled in the form of (α ,β, γ) or α ⟶ β, γ where α(∈T) is the current input symbol, β(∈T) is the symbol to be replaced wh
α and γ={L,R}. The TM for the above example is as



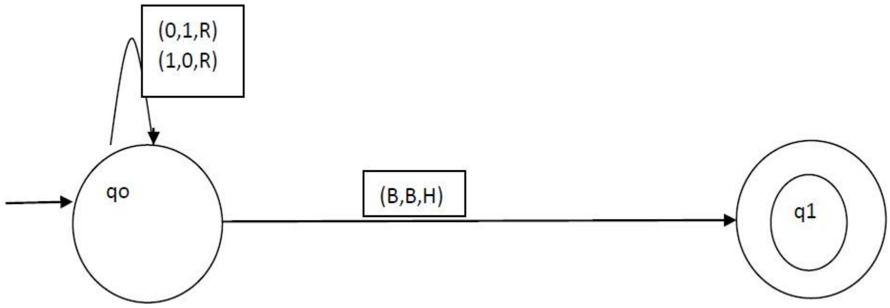
Ex: Design a TM to recognize all strings consisting of even no of a's defined over {a}
Transition Diagram



Transition Table

δ	a	B
q0	(q1,a,R)	(q2,B,H)
q1	(q0,a,R)	--
q2	--	--

Ex: Design a TM for finding 1's Complement of a given binary number
Transition Diagram



δ	0	1	B
q0	(q0,1,R)	(q0,0,R)	(q1,B,H)

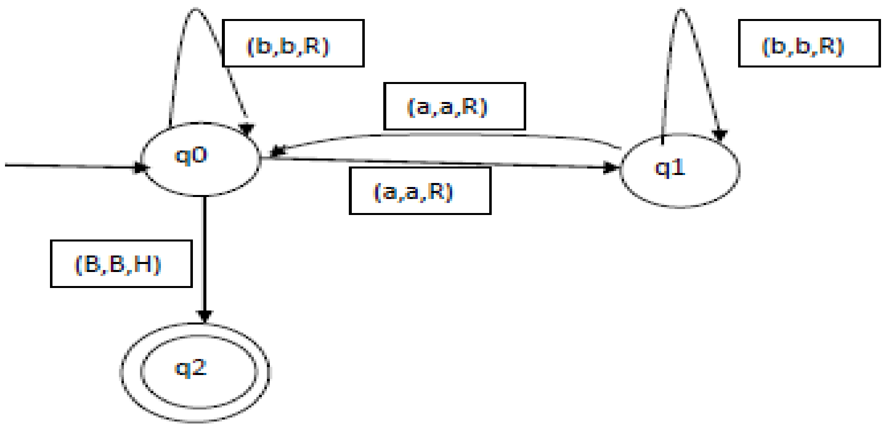
Ex: Design a TM for finding 2's Complement of a given binary number

Transition Diagram

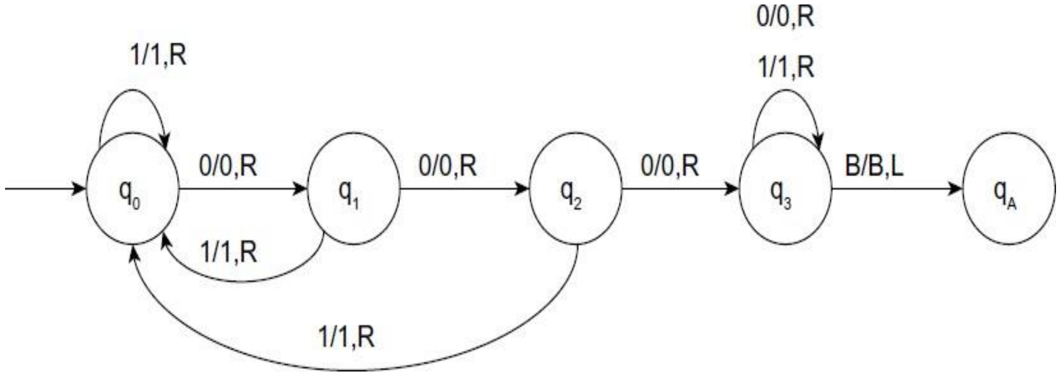
Transition Table

δ	0	1	B
q0	(q0,0,R)	(q0,1,R)	(q1,B,L)
q1	(q1,0,L)	(q2,1,L)	--
q2	(q2,1,L)	(q2,0,L)	(q3,B,H)
q3	--	--	--

Ex: Construct a TM for language consisting of strings having any no of b's and even no of a's defined over {a,b}.



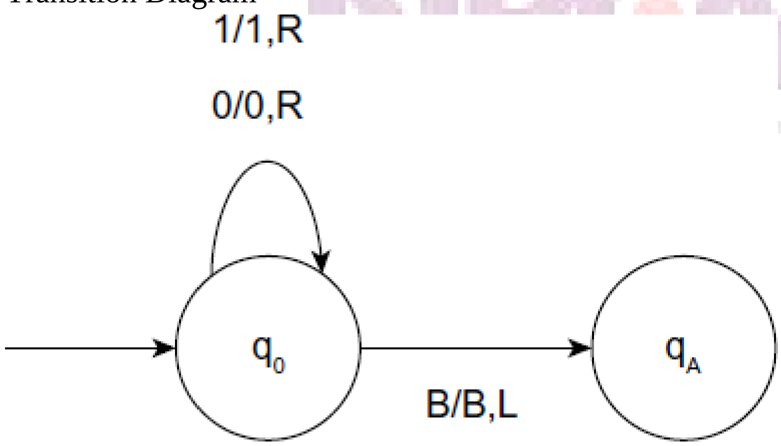
Design a TM to accept strings formed with 0 and 1 and having substring 000



	0	1	B
$\rightarrow q_0$	$(q_1, 0, R)$	$(q_0, 1, R)$	---
q_1	$(q_2, 0, R)$	$(q_0, 1, R)$	---
q_2	$(q_3, 0, R)$	$(q_0, 1, R)$	---
q_3	$(q_3, 0, R)$	$(q_3, 1, R)$	(q_A, B, R)
q_A	---	---	---

Ex: Design TM to accept strings belonging to the language $(0+1)^*$

Transition Diagram

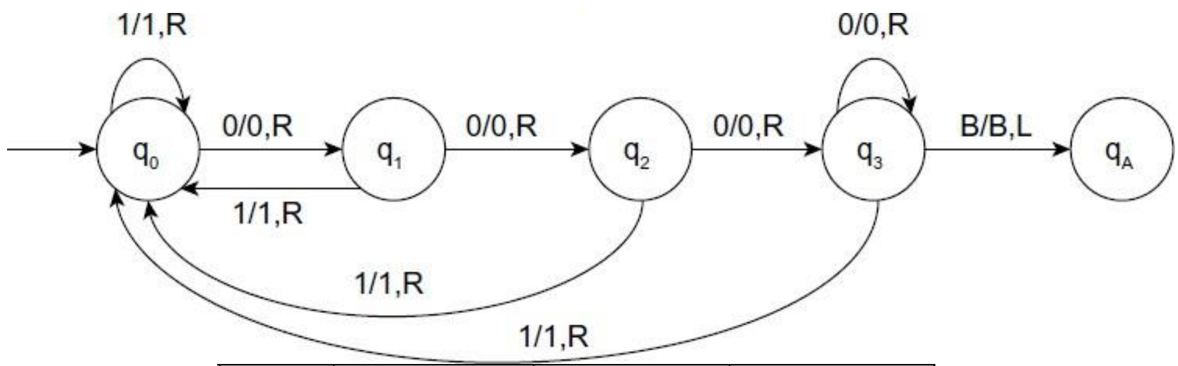


Transition Table

	a = 0	a = 1	a = B
$\rightarrow q_0$	$(q_0, 0, R)$	$(q_0, 0, R)$	(q_A, B, L)
q_1	----	----	----

Ex: Design a TM to accept strings formed on {0,1} and ending with 000

Transition Diagram

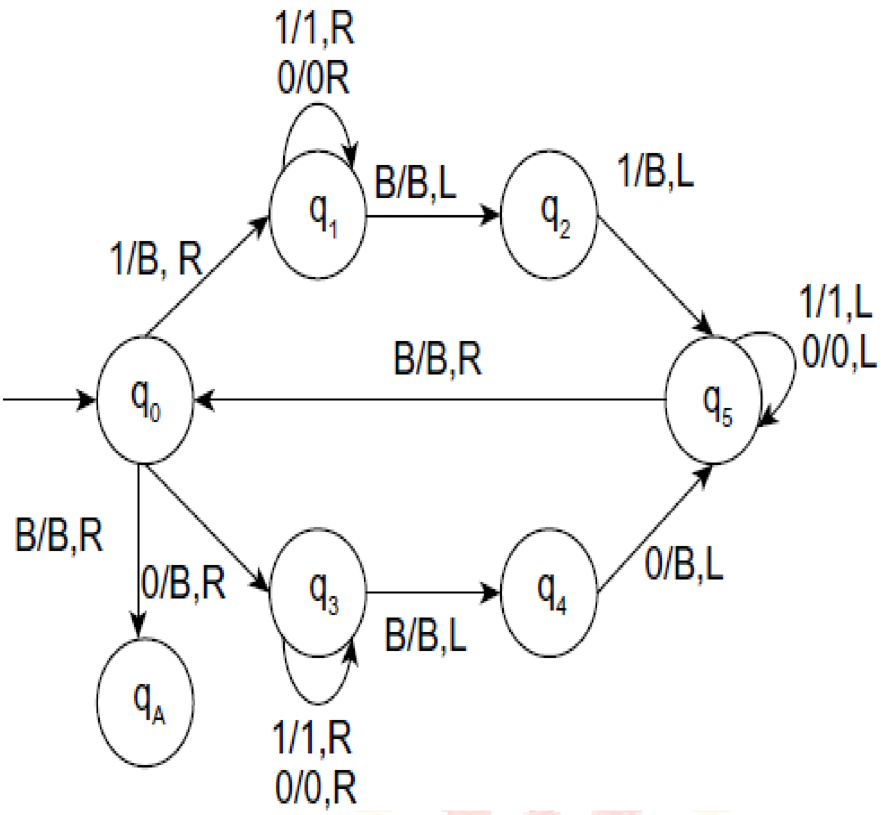


Transition Table:

	0	1	B
$\rightarrow q_0$	$(q_1, 0, R)$	$(q_0, 1, R)$	---
q_1	$(q_2, 0, R)$	$(q_0, 1, R)$	---
q_2	$(q_3, 0, R)$	$(q_0, 1, R)$	---
q_3	$(q_3, 0, R)$	$(q_0, 1, R)$	(q_A, B, R)
q_A	---	---	---

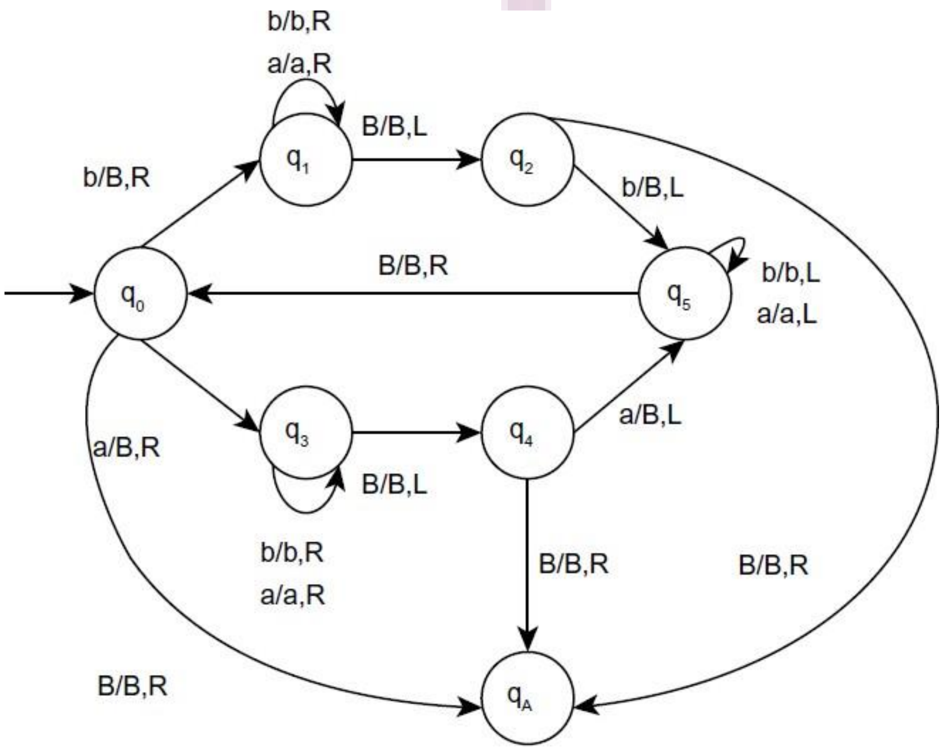
Ex: Design a TM for accepting strings of the language L={ww^r: w ∈(0+1)* }

Transition Diagram


1/1,R
0/0R
B/B,L
1/B,L
1/B, R
1/1,L
0/0,L
B/B,R
q_1
q_2
q_0
q_5
B/B,R
0/B,R
q_3
q_4
0/B,L
B/B,L
q_A
1/1,R
0/0,R

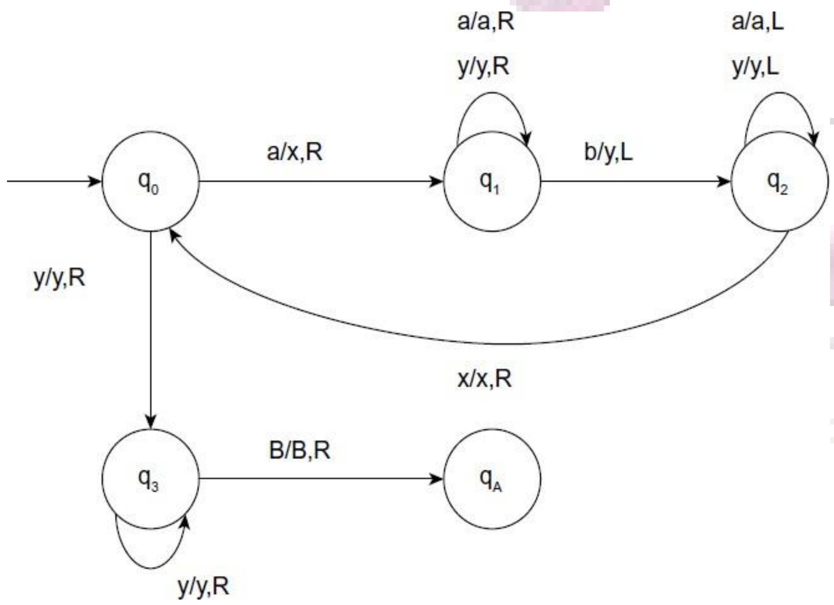
	0	1	B
$\rightarrow q_0$	(q_1, B, R)	(q_3, B, R)	(q_A, B, R)
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, B, L)
q_2	---	(q_5, B, L)	---
q_3	$(q_3, 0, R)$	$(q_3, 1, R)$	(q_4, B, L)
q_4	(q_5, B, L)	---	---
q_5	$(q_5, 0, L)$	$(q_5, 1, L)$	(q_0, B, R)
q_A	---	---	---

Ex: Design a TM for palindrome strings over {a,b} Transition Diagram



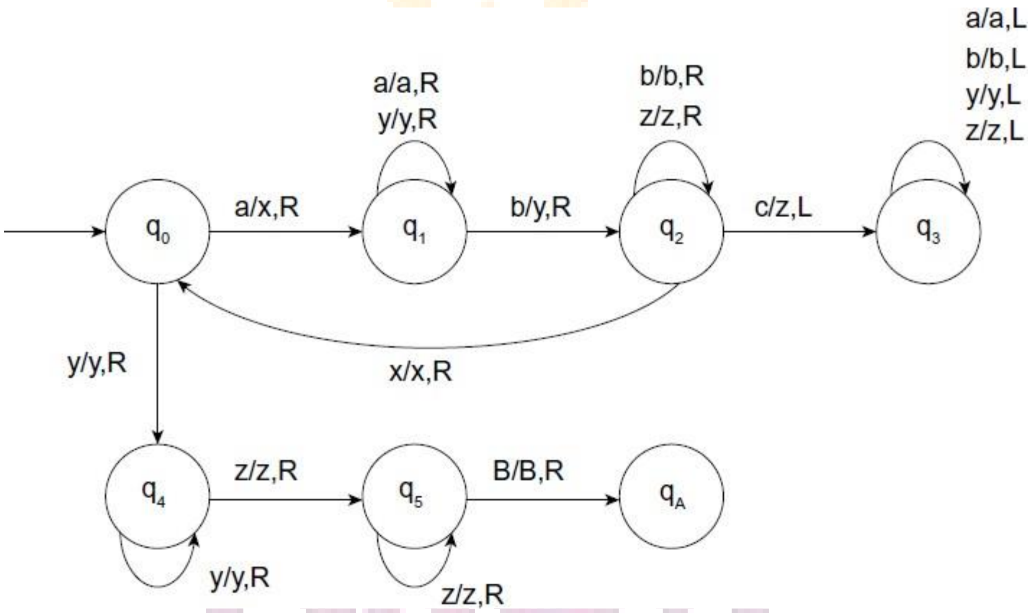
	a	b	B
$\rightarrow q_0$	(q_1, B, R)	(q_3, B, R)	(q_A, B, R)
q_1	(q_1, a, R)	(q_1, b, R)	(q_2, B, L)
q_2	---	(q_5, B, L)	(q_A, B, R)
q_3	(q_3, a, R)	(q_3, b, R)	(q_4, B, L)
q_4	(q_5, B, L)	---	(q_A, B, R)
q_5	(q_5, a, L)	(q_5, b, L)	(q_0, B, R)
q_A	---	---	---

Ex: Design a TM which accepts $L=\{a^n b^n: n>=1\}$



	a	b	X	Y	B
$\rightarrow q_0$	(q_1, x, R)	---	---	(q_3, y, R)	---
q_1	(q_1, a, R)	(q_2, y, L)	---	(q_1, y, R)	---
q_2	(q_2, a, L)	---	(q_0, x, R)	(q_2, y, L)	---
q_3	---	---	---	(q_3, y, R)	(q_A, B, R)
q_A	---	---	---	---	---

Ex:Design a TM which accepts L={$a^n b^n c^n$: n>=1}



	a	b	c	x	Y	z	B
$\rightarrow q_0$	(q_1, x, R)	---	---	---	(q_4, y, R)	---	---
q_1	(q_1, a, R)	(q_2, y, R)	---	---	(q_1, y, R)	---	---
q_2	---	(q_2, b, R)	(q_3, z, L)	---	---	(q_2, z, R)	---
q_3	(q_3, a, L)	(q_3, b, L)	---	(q_4, y, R)	(q_3, y, L)	(q_3, z, L)	---
q_4	---	---	---	---	(q_4, y, R)	(q_5, z, R)	---
q_4	---	---	---	---	---	(q_5, z, R)	(q_A, B, R)
q_A	---	---	---	---	---	---	---

TM AS INTEGER FUNCTION:

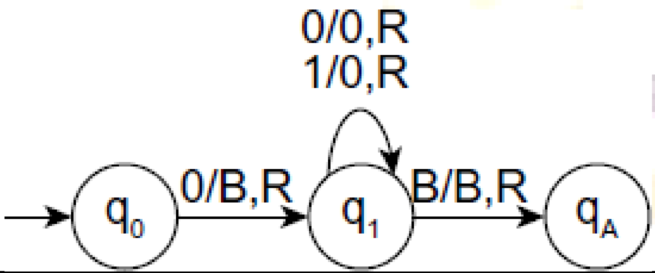
A Turing machine M computes a function f if, when given input w in the domain of f, the machine halts in its accept state with f(w) written (leftmost) on the tape. To use TM as a computational machine, it is required to place the integer numbers as 0m.

Suppose it is required to add two numbers; that is, f(m, n) = m + n, then the numbers m and n are to be placed on the tape as 0m10n where 1 is a separator for the numbers m and n. Once processing is completed and the TM halts, the tape would have the contents as 0(m+n), which is the required result of the computation.

Ex: Design a TM to add two numbers a and b.

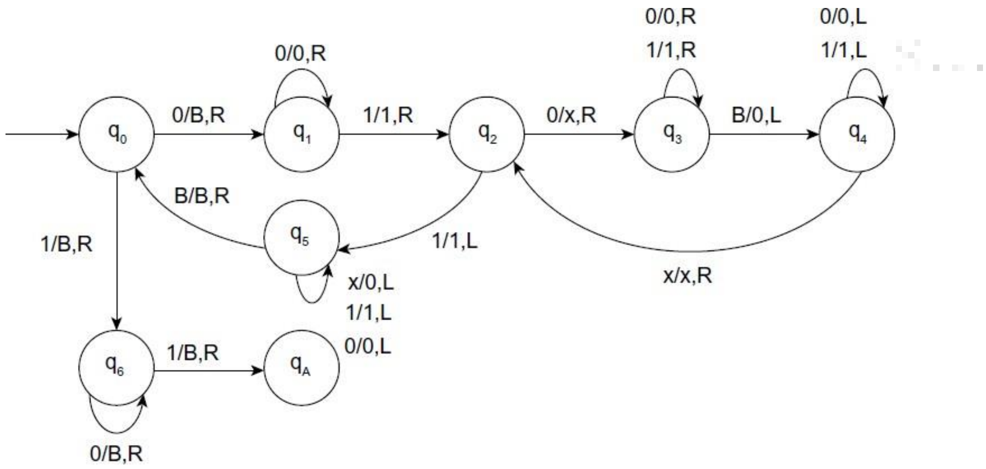
Sol: Let the numbers be 2 and 3. The addition of these numbers using simple logic is explained. The numbers are placed as $B0^2 10^3 B$.

After processing, the tape content would be $B0^5B$. The simple logic that can be used is: to replace the occurrence of 0 by B and move to right and replace 1 to 0, so that it is in required form as $B0^5B$.



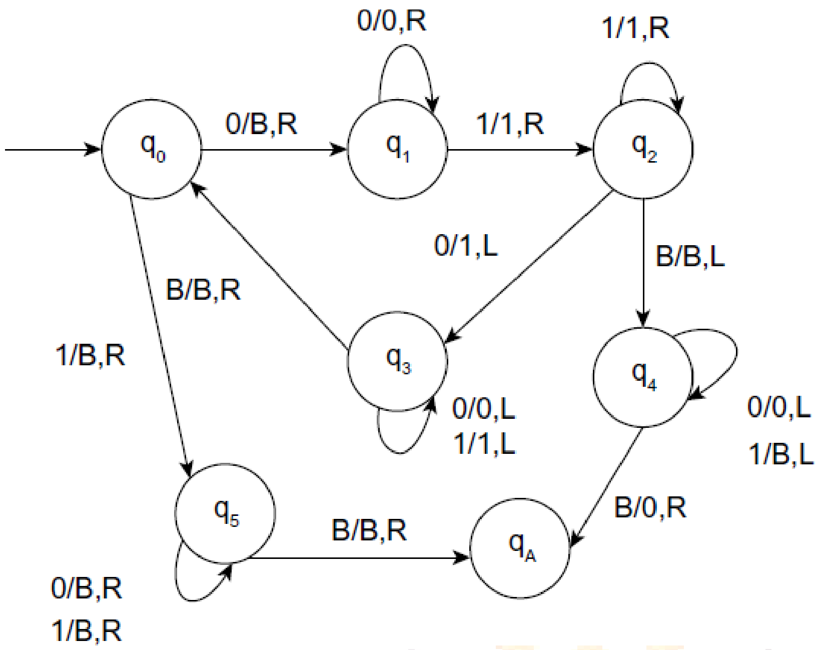
	0	1	B
$\rightarrow q_0$	(q_1, B, R)	---	---
q_1	$(q_1, 0, R)$	$(q_1, 0, R)$	(q_A, B, R)
q_A	---	---	---

Ex: Design TM for Multiplication of two integers



Ex: Design TM for f(m,n)=m-n, m>=n

	0	1	x	B
$\rightarrow q_0$	(q_1, B, R)	(q_6, B, R)	---	---
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	---	---
q_2	(q_3, x, R)	$(q_5, 1, L)$	---	---
q_3	$(q_3, 0, R)$	$(q_3, 1, R)$	---	$(q_4, 0, L)$
q_4	$(q_4, 0, L)$	$(q_4, 1, L)$	(q_2, x, R)	---
q_5	$(q_5, 0, L)$	$(q_5, 1, L)$	$(q_5, 0, L)$	(q_0, B, R)
q_6	(q_3, B, R)	(q_A, B, R)	---	---
q_A	---	---	---	---



	0	1	x	B
q_0	(q_1, B, R)	(q_5, B, R)	---	---
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	---	---
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	---	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	---	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	---	$(q_A, 0, R)$
q_5	(q_5, B, R)	(q_A, B, R)	(q_5, B, R)	---
q_6	(q_6, x, L)	(q_6, x, L)	(q_6, x, L)	(q_5, B, R)
q_A	---	---	---	---

CONVERSION OF REGULAR EXPRESSION TO TM

- **Step1:** Convert the RE to an equivalent Automaton without epsilon transitions
- **Step2:** Change both the +-initial and final states of the Automata to an intermediate state
- **Step3:** insert a new initial state with a transition (B,B,R) to the Automata's initial state
- **Step4:** convert the transitions with label a to (a,a,R)
- **Step5:** insert a new final state with a transition (B,B,R) from Automata's final state to the new final state.

Ex: Construct a TM for the RE $(a+b)^*(aa+bb)\ (a+b)^*$

IMPORTANT QUESTIONS:

Part-A

1. Define Turing Machine?
2. Design a TM for finding 2's Complement of a given binary number TM as Integer Function
3. Draw the block diagram of PDA
4. Define PDA
5. Define instantaneous description of PDA.

PART-B

6. Design a PDA which accepts the language L={$a^n b^n$/n>=1}
7. Design a PDA which accepts L={WW^R|W is in (a+b)*}
8. Convert the following CFG in to PDA S→aAA, A→aS/bS/a
9. Give the equivalent CFG for the following PDA M = {{q0 , q1 },{a, b}, {Z, ZO}, δ, qO, ZO} where δ is defined by δ(qO, b, ZO) = (qO, ZZO) δ(qO, ε, ZO) = (qO, ε) δ(qO, b, Z) = (qO, ZZ) δ(qO, a, Z) = (q1, Z) δ(q1, b, Z) = (q1, ε)δ(q1, a, ZO) = (qO, ZO)
10. Construct a TM for language consisting of strings having any no of b's and even no of a's defined over {a,b}.
11. Design a TM to accept strings formed with 0 and 1 and having substring 000
12. Design a TM for accepting strings of the language L={wwr : w ∈ (0+1)* }
13. Design a TM for palindrome strings over {a,b}
14. Design a TM which accepts L={anbn: n>=1}
15. Design a TM which accepts L={anbncn : n>=1}
16. Design a TM to add two numbers a and b
17. Design TM for the Multiplication of two integers
18. Construct a TM for the RE (a+b)*(aa+bb) (a+b)*

UNIT-5

Types of Turing machine: Turing machines and halting Undecidability: Undecidability, A Language that is Not Recursively Enumerable, An Undecidable Problem That is RE, Undecidable Problems about Turing Machines, Recursive languages, Properties of recursive languages, Post's Correspondence Problem, Modified Post Correspondence problem,Other Undecidable Problems, Counter machines.

5.1. VARIATIONS OF THE TM:

1. **Multi Tape Turing Machine:**

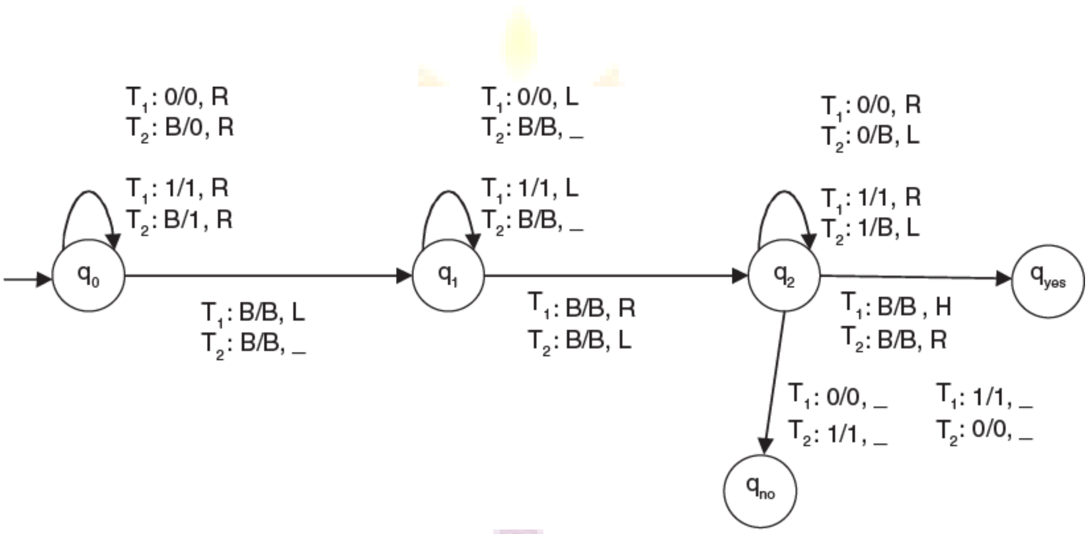
- Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.
- Depending on the present state and i/p symbol scanned by each of the head, the TM can Change its state.
- Write a new symbol on the respective cell of the respective tape from where the i/ps were scanned, Move the head one left/right.

Def: A Multi-tape Turing machine can be formally described as a 7-tuple (Q, T, ∑, δ, q0,B, F) where

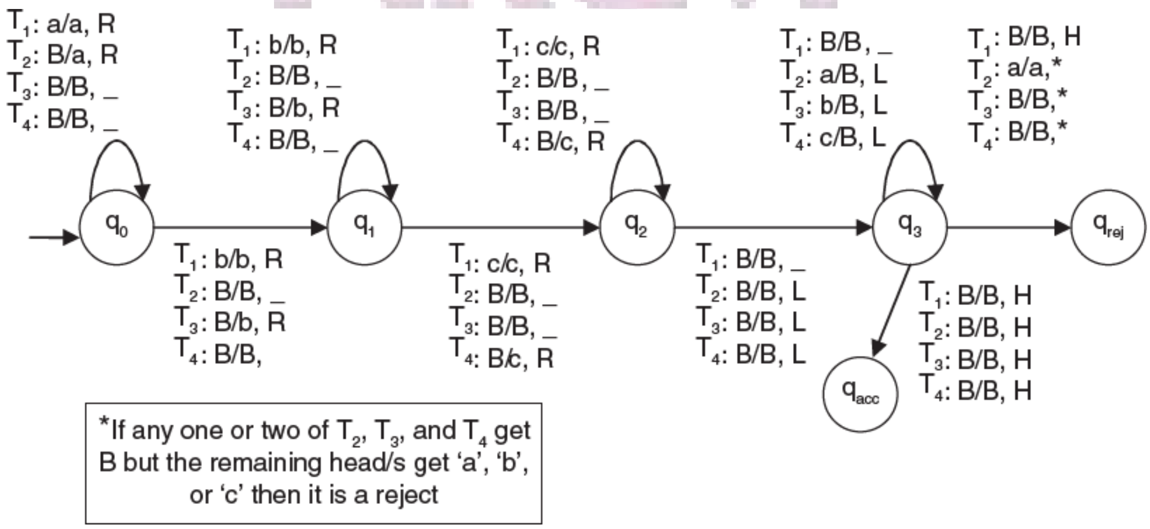
Q is a finite set of states
T is the tape alphabet
∑ is the input alphabet
δ: $QXT^k \longrightarrow QXT^kX\{L,R\}$ is a transition function

q0 is the initial state
B is a blank symbol
F is the set of final states

Ex: Design a Multi tape TM for checking whether a binary string is a palindrome or not Sol:
Consider a TM with two tapes. The i/p is written on the first tape.
The machine works by the following process:
Copy the i/p from the first tape to the second tape by traversing the first tape from left to right.
Traverse from the first tape again from right to left and point the head to the first symbol of i/p on tape 1.
Moves the two heads pointing on the two tapes in opposite directions checking that the two symbols are identical and erasing the copy in tape2 at the same time.

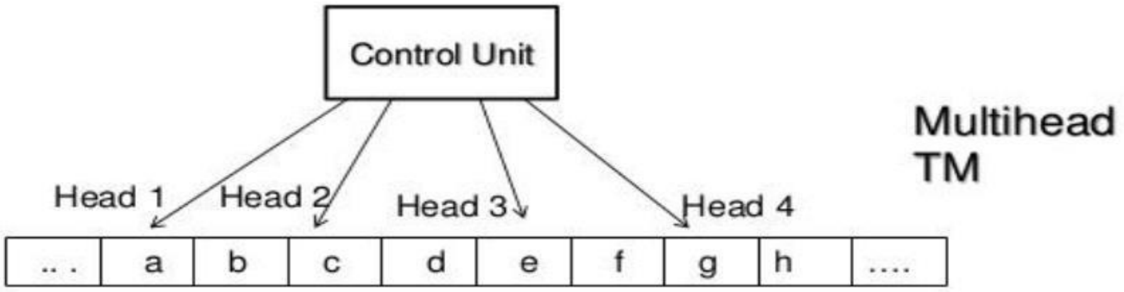


Ex: Design a multi-tape TM for $L=a^nb^nc^n$

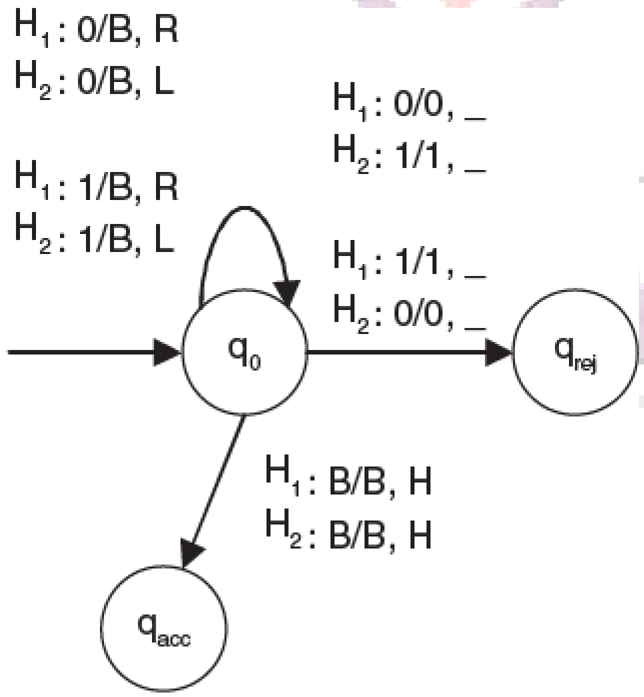


2. Multi-head Turing Machine:

A multi-head Turing machine contains 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.



- Design a multi-head TM for checking whether a binary string is a palindrome or not.
- Sol: Consider a TM with two heads. The heads are pointing to the two ends of the string on the tape. Both the heads traverse the string in the opposite direction. The head1 has the priority over head2.
- If both of the heads gets the same symbols, then it traverses the next input right or left by replacing the present symbol by B.
- If both heads gets B, then halt and declare the string as a palindrome.



TWO-WAY INFINITE TAPE TURING MACHINE

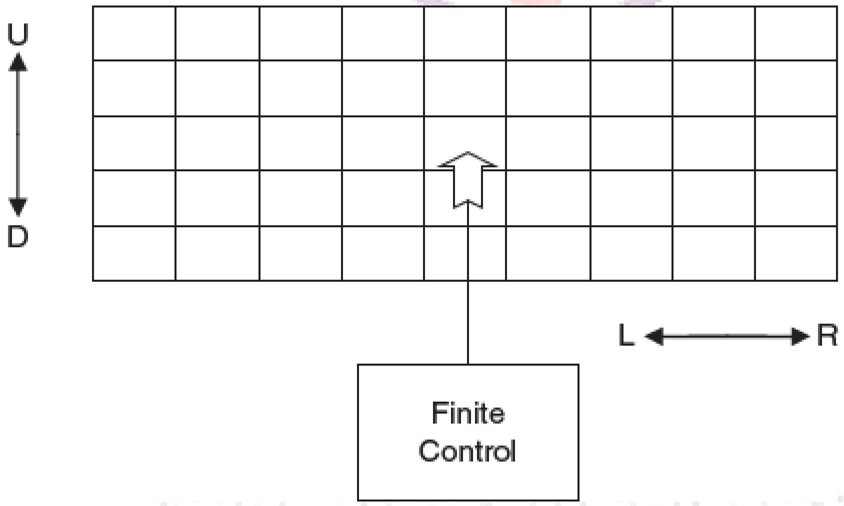
- In general in a TM, there is a left boundary. If the head crosses that boundary and wants to go left, then the situation is called **a crash condition**. But the head may traverse the right side up to infinity. In this sense, the i/p tape of the general TM can be treated as a one-way infinite tape.

- A TM where there is infinite number of sequences of blank on both sides of the tape is called a two-way infinite tape TM. A typical diagram of the i/p tape of a two-way infinite TM is:

- -	- -	- -	B	B	a	a	b	b	B	B	- -	- -

MULTI-DIMENSIONAL (K=2) TAPE TURING MACHINE

- It has multi-dimensional tape where head can move any direction that is left, right, up or down.
- Multi-dimensional tape Turing machine can be simulated by one-dimensional Turing machine.
- The i/p tape of 1-D TM is extended to infinite in both sides, but in one direction. If the i/p tape can be extended infinitely in more than one dimension, then the TM is called a multi-dimensional TM.
- In general case, consider k=2, which means that the i/p tape is extended to infinitely in right and down directions. For this case, the read/write head can move in the left, right, up and down directions.
- The transition function for a K-dimensional TM is **δ:QX∑ ⟶ QXTX{L,R,U,D,H}** where L=Left, R-Right, U-Up and D-Down.



NON-DETERMINISTIC TURING MACHINE

A non-deterministic Turing machine has a single, one way infinite tape.For a given state and input symbol has at least 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

Def: A non-Deterministic TM is expressed as a 7-tuple (Q, T, B,∑, δ, q0, F) where:

Q-finite set of states

T -tape alphabet (symbols which can be written on Tape)

B∈ T -blank symbol (every cell is filled with B except input alphabet initially)

$\sum$ -the input alphabet (symbols which are part of input alphabet)
δ :$Q \times T \rightarrow \mathbf{2}^{Q} \times T \times \{L,R\}$ transition function which maps.
q0 -the initial state
F -the set of final states.

Ex: Construct a TM over {a,b} which contains a substring abb.

Ex: Design a TM for 0^n1^m, where m,n>=0 and n may not be equal to m

Enumerator: It is a type of TM which is attached a printer. It has a work tape and an o/p tape. The work tape is a write only once tape. At each step, the machine chooses a symbol to write from the output alphabet on the output tape.

After writing a symbol on the output tape, the head placed on the output moves right by one position. The enumerator has a special state, say q_p , entering which the output tape is erased and the tape head moves to the leftmost position and finally the string is printed. A string w is printed as o/p by the enumerator if the o/p tape contains w at the time the machine enters in to q_p.

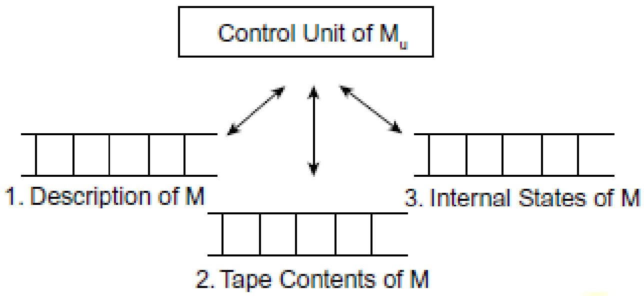
The transition function of enumerator is δ: $Q X \sum X T \longrightarrow Q X \sum X \{L,R\} X T X \{L,R\}$

UNIVERSAL TM:

- A universal Turing machine (UTM) is a Turing machine that simulates an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input to that machine from its own tape.
- To design UTM, add the following to the TM:
- Increase the no of read-write heads (like multiple heads TM)
- Increase the no of input tapes (multiple tape TM)
- Increase the dimension of moving the read-write head (K-Dimensional TM)
- Add special purpose memory like stack.
- A UTM, M_U is an automaton that, given as input the description of any TM and a string w, can simulate the computation of M for input w. To construct such an M_U we first choose a standard way of describing TMs.
- We may, without loss of generality, assume that M=(Q, {0,1}, {0,1,B}, δ, q_0,B,q_f) where q_f is a single final state. The alphabet {0,1,B}∈T are represented as a_1,a_2, and a_3. The direction left and right are represented as D_1 and D_2 respectively. The
  transitions of TM are encoded in a special binary representation where each symbol is separated by 1.
  Ex: if there is a transition δ(q_i, a_j)= (q_k, a_l,D_m) then the binary representation for the transition is as given as $0^i10^j10^k10^l10^m$.
- The binary code for the Turing machine is M which has transitions t_1, t_2, t_3, ... t_n is represented as 111$t_1$11$t_2$11$t_3$11. ....11t_n111.
- Note: The transitions need not be in any particular order.
- If a string has to be verified then the problem is represented as a tuple <M,w> where M is the definition of TM and w is the input string.
- Ex: Let M=({q1,q2,q3}, {0,1}, {0.1.B}, δ, q1, B,{q2}) have moves defined as δ(q1, 1)=(q3,
  0,R), δ(q3, 0)= (q1, 1,R), δ(q3, 1)= (q2, 0,R), δ(q3, B)= (q3, 1,L).
- Give the problem representation for the string w=1011
- Sol: Let binary representation for states{q1,q2,q3}be {0,00,000}, alphabet {0,1,B} be {0,00,000} and direction {R,L} be {0,00}. The transitions are represented as follows:

Transition	Binary Representation
δ(q1, 1)= (q3, 0,R)	010010001010
δ(q3, 0)= (q1, 1,R)	000101010010
δ(q3, 1)= (q2, 0,R)	0001001001010

δ(q3, B)= (q3, 1,L)	00010001000100100



The problem instance <M,1011> is represented as 111 010010001010 11 000101010010 11 0001001001010 11 00010001000100100 111 1011

- For any input M and w, Tape 1 will keep an encoded definition of M, Tape 2 will contain the tape contents of M and Tape 3, the internal state of M. Mu looks first at the contents of Tapes 2 and 3 to determine the configuration of M. The behavior of the M is as follows.
- 1. Check the format of Tape 1 for the validations of the TM model.

a. No two transitions should begin with Oi1Oj1 for the same i and j.

b. Check that if $O^i1O^j1O^k1O^l1O^m$ represents a transition, then $1 < j < 3$, $1 < l < 3$, and $1 < m < 3$.

- 2. Initialize Tape 2 to contain w. Initialize Tape 3 to hold a single O representing initial state q1. For all the tapes, the tape heads are positioned at the left end and these symbols are marked to identify the starting position.
- 3. When Tape 3 holds OO, it is said to reach the final state, and the machine can halt.
- 4. Let at any given time aJ be the symbol currently scanned by tape head 2 and let O^i, the contents of Tape 3 (which indicates state). Scan Tape 1 from the left end to the second 111 looking for a substring beginning with $11O^i1O^j1$.

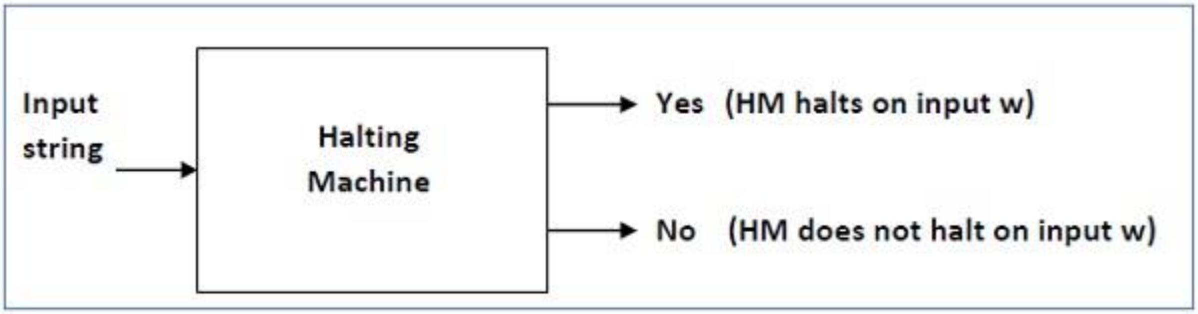
a. if no such string is found, then halt and reject.

b. if found, then let the suffix be $O^k1O^l1O^m11$. Put O^k on Tape 3, print a on the tape cell scanned by head 2 and move the head in direction Dm.

- It is clear that Mu accepts<M, w> if and only if M accepts w. It is also true that if M runs forever on w, Mu runs forever on <M, w> and if M halts on w without accepting, Mu also halts on w without accepting.

5.1.1. HALTING PROBLEM:

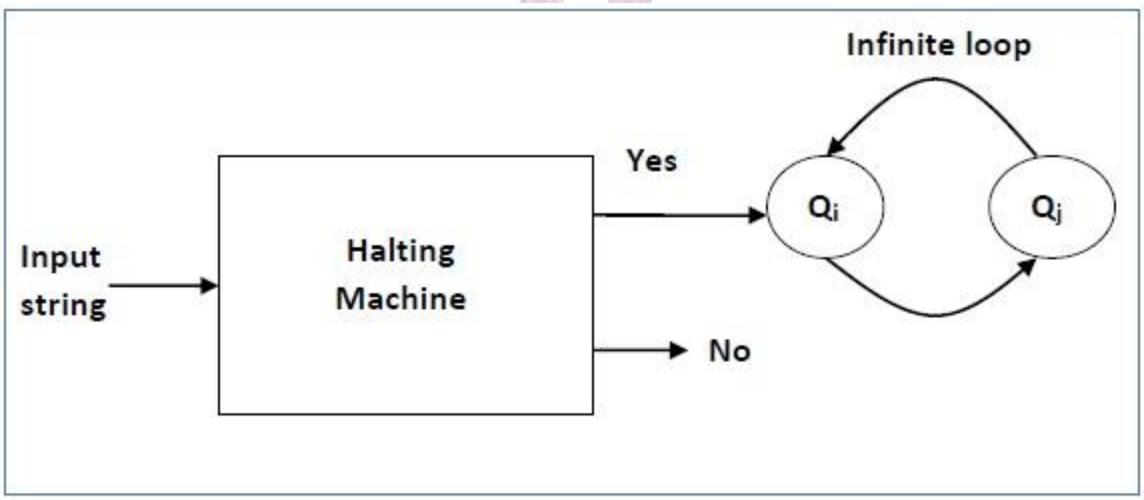
Input − A Turing machine and an input string **w**.
Problem − Does the Turing machine finish computing of the string **w** in a finite number of steps? The answer must be either yes or no.
Proof − At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine −



Now we will design an **inverted halting machine (HM)'** as −

- If **H** returns YES, then loop forever.
- If **H** returns NO, then halt.

The following is the block diagram of an 'Inverted halting machine' −



Further, a machine **$(HM)_2$** which input itself is constructed as follows −

- If $(HM)_2$ halts on input, loop forever.
- Else, halt.

Here, we have got a contradiction. Hence, the halting problem is **undecidable**.

Types of Grammars–Chomsky Hierarchy:

Linguist Noam Chomsky defined a hierarchy of languages, in terms of complexity. This four-level hierarchy, called the Chomsky hierarchy, corresponds to four classes of machines. Each higher level in the hierarchy incorporates the lower levels, that is, anything that can be computed by a machine at the lowest level can also be computed by a machine at the next highest level.

The Chomsky hierarchy classifies grammar according to the form of their productions into the following levels:

- **Type 0 grammars–unrestricted grammars**: These grammars include all formal grammars. In unrestricted grammars (URGs), all the productions are of the form $\alpha \rightarrow \beta$ where α and β may have any number of terminals and non-terminals, that is, no restrictions on either side of productions. Every grammar is included in it if it has at least one non-terminal on the left-hand side (LHS). They generate exactly all languages that can be recognized by a Turing machine. The language that is recognized by a Turing machine is defined as a set of all the strings on which it halts. These languages are also known as recursively enumerable languages.
  Ex:
  $aA \rightarrow abCB$ $aA \rightarrow bAA$ $bA \rightarrow a$
  $S \rightarrow aAb \mid \varepsilon$
- **Type 1 grammars–context-sensitive grammars**: These grammars define the context-sensitive languages. In context-sensitive grammar (CSG), all the productions of the form $\alpha \rightarrow \beta$ where the length of α is less than or equal to the length of β i.e. $|\alpha| \leq |\beta|$, α and β may have any number of terminals and non-terminals.
  These grammars can have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with A as non-terminal and α, β, and γ are strings of terminals and non-terminals. We can replace A by γ where A lies between α and β. Hence the name CSG. The strings α and β may be empty, but γ must be non-empty. It cannot include the rule $S \rightarrow \varepsilon$. These languages are exactly all languages that can be recognized by linear bound automata.
  Ex: $a\underline{A}bcD \rightarrow a\underline{bcDbc}D$
- **Type 2 grammars – context-free grammars:** These grammars define context-free languages. These are defined by rules of the form $\alpha \rightarrow \beta$ with $|\alpha| \leq |\ \beta$ where $|\alpha| = 1$ and is a non-terminal and β is a string of terminals and non-terminals. We can replace α by β regardless of where it appears. Hence the name context-free grammar (CFG). These languages are exactly those languages that can be recognized by a pushdown automaton. Context-free languages define the syntax of all programming languages.

Ex:

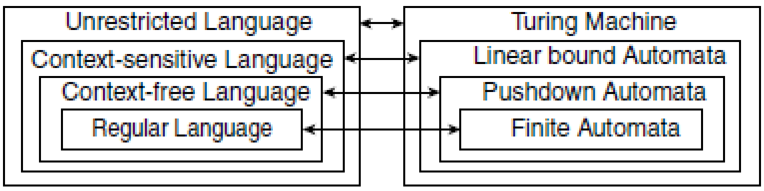
1. S → aS |S a| a

2. S→ aAA |bBB|ε

- **Type 3 grammars – regular grammars:** These grammars generate regular languages. Such a grammar restricts its rules to a single non-terminal on the LHS. The RHS consists of either a single terminal or a string of terminals with a single nonterminal on the left or right end. Here rules can be of the form A→ a B | a or A →Ba | a.
  The rule S → ε is also allowed here. These languages are exactly those languages that can be recognized by a finite state automaton. This family of formal languages can be obtained by regular expressions also. Regular languages are used to define search patterns and the lexical structure of programming languages.
  Right linear grammar: A → a A | a Left linear grammar: A → A a | a
  Table 1.1 Chomsky's hierarchy

Grammar	Languages	Automaton	Production rules
Type 0	Recursively enumerable	Turing machine	α → β No restrictions on b, a should have At least one non-terminal
Type 1	Context-sensitive	Linear bounded automata	α → β, |α| ≤ |β|
Type 2	Context-free	Pushdown automaton	α → β, |α| ≤ |β|, |α| =1
Type 3	Regular	Finite state automaton	α → β, α = {V} and β = V{T}* or {T}*V or T*

The hierarchy of languages and the machine that can recognize the same is shown below.

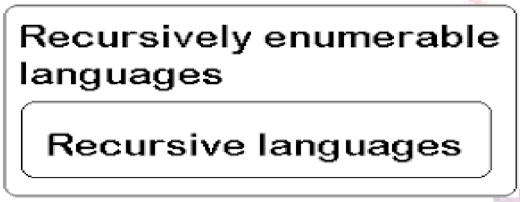


Every RG is context-free, every CFG is context-sensitive and every CSG is unrestricted.

So the family of regular languages can be recognized by any machine. CFLs are recognized by pushdown automata, linear bound automata, and Turing machines. CSLs are recognized by linear bound automata and Turing machines. Unrestricted languages are recognized by only Turing machines.

5.2. RECURSIVE AND RECURSIVELY ENUMERABLE LANGUAGES:

- There are three possible outcomes of executing a TM over a given input. The TM may halt and accept the input Halt and Reject the input or Never Halt.
- **Recursive Language**: A language is recursive if there exists a TM that accepts every string of the language and rejects every string (over the same alphabet) that is not in the language.
- **Note:** If a language L is recursive, then its complement L^1 must also be recursive.
- **Recursively Enumerable Language:** A language is recursively enumerable if there exists a TM that accepts every string of the language and does not accept the strings that are not in the language (i.e., strings may be rejected or may cause the TM to go into an infinite loop).
- **Note:** Every recursive language is also recursively enumerable but the converse need not be true.



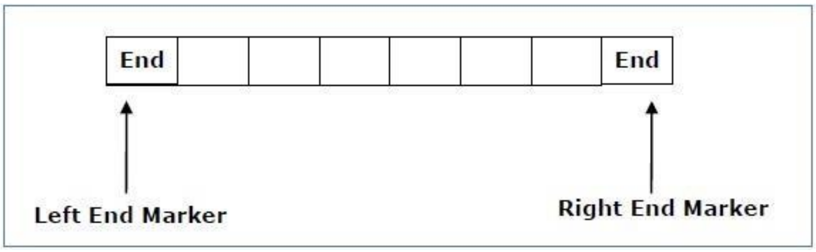
5.3. Closure Properties of Recursive and Recursively enumerable languages

- **Union**: If L1 and If L2 are two recursive languages, their union L1∪L2 will also be recursive because if TM halts for L1 and halts for L2, it will also halt for L1∪L2.
- **Concatenation:** If L1 and If L2 are two recursive languages, their concatenation L1.L2 will also be recursive.
- Ex: L1= $\{a^n b^n c^n | n>=0\}$
- L2= $\{d^m e^m f^m | m>=0\}$
- L3= L1.L2= $\{ a^n b^n c^n d^m e^m f^m : m>=0 \text{ and } n>=0\}$ is also recursive.
- **Kleene Closure:** If L1is recursive, its kleene closure L1* will also be recursive. For Example: L1= $\{a^n b^n c^n | n>=0\}$ $L1^*= (\{a^n b^n c^n | n>=0\})^*$ is also recursive
- **Intersection and complement**: If L1 and If L2 are two recursive languages, their intersection L1 ∩ L2 will also be recursive. Similarly, complement of recursive language L1 which is ∑*-L1, will also be recursive.
- The complement of a recursive language is recursive.

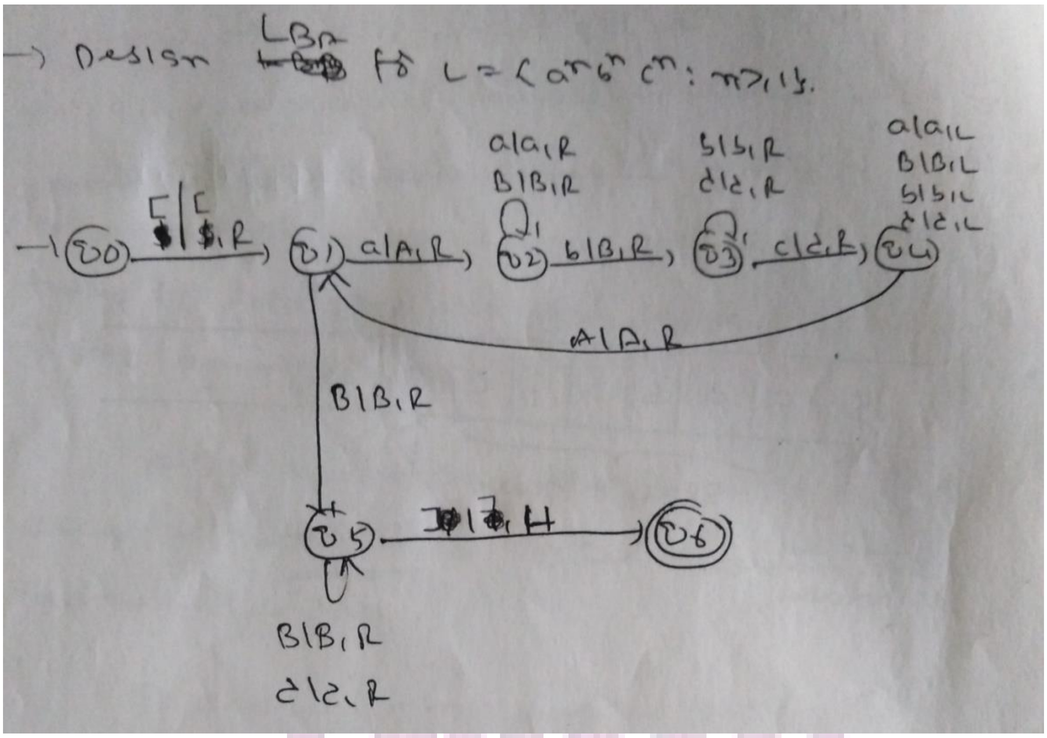
5.4. Linear Bounded Automata (LBA)

- A NDTM is called Linear Bound Automata (LBA) if
- Its input alphabet includes two special symbols[and] as left and right end markers .
- It has no moves beyond these end markers. i.e., no left move from [and

no right move from]. It never changes the symbols [and].



- **Def:** A LBA is defined using 8-tuples as M=(Q, Σ, Γ, δ,q0,[,],F)
- Where Q, Σ, Γ, δ,q0 and ,F are same as for NDTM, [and] are left and right end markers.
- **Ex:** Design LBA for L={$a^n b^n c^n$: n>=1}



5.5. INTRODUCTION TO UNDECIDABILITY

- In the theory of computation, we often come across such problems that are answered either 'yes' or 'no'. The class of problems which can be answered as 'yes' are called solvable or decidable. Otherwise, the class of problems is said to be unsolvable or undecidable.
- **Decidable:** A decision problem that can be solved by an algorithm is called decidable. All the languages recognized by TM are decidable.
- Ex: Given two numbers x and y, does x evenly divides y?
- **Decidable:** A decision problem A is called **decidable** or **effectively solvable** if A is a recursive set.
- **Partially decidable:** A problem is called **partially decidable/semi decidable/ solvable/**

provable if A is a recursively enumerable set.

- **Undecidable:** Partially decidable problems and any other problems that are not decidable are called undecidable.
- Undecidability of a problem means that there is no particular algorithm that can determine whether a given problem has a solution or not.

Post Correspondence Problem (PCP): It is an undecidable decision problem. Let us define the PCP.

- "The Post's correspondence problem consists of two lists of strings that are of equal length over the input. The two lists are A = $w_1, w_2, w_3, \ldots\ldots\ldots\ldots, w_n$ and B = $x_1, x_2, x_3, \ldots x_n$ then there exists a non-empty set of integers $i_1, i_2, i_3, \ldots\ldots\ldots\ldots, i_n$ n>=1 such that, w_1, $w_2, w_3, \ldots. w_n = x_1, x_2, x_3, \ldots\ldots\ldots x_n$"
- To solve the post correspondence problem we try all the combinations of $i_1, i_2, i_3, \ldots\ldots\ldots\ldots\ldots, i_n$ to find the $w_i = x_i$ then we say that PCP has a solution and is decidable otherwise PCP is undecidable.
- Consider the following sequence and find whether it has a solution (decidable) or not.

I	List A	List B
1	1	111
2	10111	10
3	10	0

Sol:

- If we take 3, first character in A is 1 and first character in B is 0. So we will not get same strings.
- If we take 1 then A starts with 1 and B also starts with 1, but for the next two characters in A, there is no matching sequence.
- So we starts with 2. i.e., i=2 Therefore

I	2	
Wi	1011 1	
Xi	10	

- Length of first string >second string
- Next consider B which starts with 1. We have 1 and 2. If we consider 2 next symbol is 0 and does not match. Hence, consider 1.

I	2	1	String
Wi	10111	1	101111
Xi	10	11 1	10111

- Still Length of first string >second string. Again choose 1

i	2	1	1	String
Wi	10111	1	1	1011111
xi	10	11 1	111	10111111

- Length of first string <second string. Consider 3

i	2	1	1	3	String
W i	10111	1	1	10	101111110
xi	10	11 1	111	0	101111110

- Length of first string = second string, hence stop the procedure and declare the sequence 2113 as a solution. Therefore, it is decidable.

5.5 MPCP: MODIFIED VERSION OF PCP:

- MPCP is decidable <==> PCP is decidable.
- In MPCP, there is the additional requirement on a solution that the first pair on the list X and Y must be the first pair in the solution.
- More formally, an instance of MPCP is two lists
- X=w1,w2,w3,....wk and Y=x1,x2,x3,.............xk
- and a solution is a list of 0 or more integers i1,i2,i3,. ip such that w1,wi1,wi2,...wip=x1,xi1,xi2,. xip.
- The difference between the MPCP and PCP is that in the MPCP, a solution required to start with the first string on each list.
- If we have a problem instance represented in MPCP then it can be reduced to PCP. If there is a solution for PCP instance then there exists a solution for MPCP instance.
- **Procedure to convert MPCP to PCP or Reduction of MPCP to PCP:**
- Let the list G and H be the given instance of MPCP
- Let Σ be the smallest alphabet containing all the symbols in the list G and H.
- Consider two special symbols {θ, $} not present in Σ and find two new lists X from G and Y from H using the following rules.

- x_i of list X is obtained from g_i by inserting \$ symbol after each character of g_i.
- y_i of list Y is obtained from h_i by inserting \$ symbol before each character of h_i.
- Create new words as follows. $x_0=\$g_1$, $y_0=h_1$, $x_{k+1}=\theta$, $y_{k+1}=\$\theta$.

Consider the following MPCP instance and find whether it has a solution.

I	g_i	h_i
1	100	1
2	0	100
3	1	00

- Sol: Total strings in PCP is 3 where as in MPCP total strings is 5(0^{th} and 4^{th}). 1 This problem can be converted to MPCP by applying the above procedure
- Remaining process is same as PCP. In PCP first string is not fixed. We can start with any arbitrary sequence where as in MPCP we need to start with first string.

i	xi	yi
0	\$1\$0\$0\$	\$1
1	1\$0\$0\$	\$1
2	0\$	\$1\$0\$0
3	1\$	\$0\$0
4	Θ	\$θ

Step 1:

Solution sequence	0	
Xi	\$1\$0\$0\$	
Yi	\$1	

i	xi	yi
0	\$1\$0\$0\$	\$1
1	1\$0\$0\$	\$1
2	0\$	\$1\$0\$0
3	1\$	\$0\$0
4	θ	\$θ

Step 2: identify string in yi starts with 0

Solution sequence	0	3	
Xi	\$1\$0\$0\$	1\$	
Yi	\$1	\$0\$0	

Step 3: identify string in yi starts with 1(i.e., 2 or 1). Select 2

Solution sequence	0	3	2	
Xi	\$1\$0\$0\$	1\$	0\$	
Yi	\$1	\$0\$0	\$1\$0\$0	

Step 4:identify string in xi starts with 0(i.e., 2). Select 2

i	xi	yi
0	$1$0$0$	$1
1	1$0$0$	$1
2	0$	$1$0$0
3	1$	$0$0
4	θ	$θ

Solution sequence	0	3	2	2	
xi	$1$0$0$	1$	0$	0$	
yi	$1	$0$0	$1$0$0	$1$0$0	

Step 5: identify string in xi starts with 1(i.e., 1 or 3). Select 1

Solution sequence	0	3	2	2	1	
xi	$1$0$0$	1$	0$	0$	1$0$0$	
yi	$1	$0$0	$1$0$0	$1$0$0	$1	

Step 6: identify string in xi starts with 1(i.e., 1 or 3). Select 1

Solution sequence	0	3	2	2	1	1
xi	$1$0$0	1$	0$	0$	1$0$0$	1$0$0$

	$					
yi	$1	$0$0	$1$0$0	$1$0$0	$1	$1

String xi and yi are not matching. Hence select 3 instead of 1

i	xi	yi
0	$1$0$0$	$1
1	1$0$0$	$1
2	0$	$1$0$0
3	1$	$0$0
4	θ	$θ

Solution sequence	0	3	2	2	1	3	
xi	$1$0$0$	1$	0$	0$	1$0$0$	1$	
yi	$1	$0$0	$1$0$0	$1$0$0	$1	$0$0	

Step 7: identify string in xi starts with 0(i.e., 2). Select 2

Solution sequence	0	3	2	2	1	3	2	
xi	$1$0$0$	1$	0$	0$	1$0$0$	1$	0$	
yi	$1	$0$0	$1$0$0	$1$0$0	$1	$0$0	$1$0$0	

Step 8: identify string in xi starts with 0(i.e., 2). Select 2

Solution sequence	0	3	2	2	1	3	2	2
xi	$1$0$	1$	0$	0$	1$0$0	1$	0$	0$

	0$				$			
yi	$1	$0$0	$1$0$0	$1$0$0	$1	$0$0	$1$0$0$0	$1$0$0$0

It is in the loop. Hence select 1 in step 3

Solution sequence	0	3	1
xi	$1$0$0$	1$	1$0$0$
yi	$1	$0$0	$1

i	xi	yi
0	$1$0$0$	$1
1	1$0$0$	$1
2	0$	$1$0$0
3	1$	$0$0
4	θ	$θ

Step 9: identify string in yi starts with 1(i.e., 1). Select 1

Solution sequence	0	3	1	1
xi	$1$0$0$	1$	1$0$0$	1$0$0$
yi	$1	$0$0	$1	$1

Step 10: identify string in yi starts with 0(i.e., 3). Select 3

Solution sequence	0	3	1	1	3
xi	$1$0$0	1$	1$0$0$	1$0$0$	1$

	$				
yi	$1	$0$0	$1	$1	$0$0

Step 11: identify string in yi starts with 1(i.e., 2). Select 2

Solution sequence	0	3	1	1	3	2
xi	$1$0$0$	1$	1$0$0$	1$0$0$	1$	0$
yi	$1	$0$0	$1	$1	$0$0	$1$0$0

i	xi	yi
0	$1$0$0$	$1
1	1$0$0$	$1
2	0$	$1$0$0
3	1$	$0$0
4	θ	$θ

Step 12: identify string in yi starts with 1(i.e., 2 or 1). Select 2

Solution sequence	0	3	1	1	3	2	2
xi	$1$0$0$	1$	1$0$0$	1$0$0$	1$	0$	0$
yi	$1	$0$0	$1	$1	$0$0	$1$0$0	$1$0$0

Step 13: Both are same. Then select 4

Solution sequence	0	3	1	1	3	2	2	4
xi	$1$0$	1$	1$0$0	1$0$0	1$	0$	0$	θ

	0$		$	$				
yi	$1	$0$0	$1	$1	$0$0	$1$0$0	$1$0$0	$θ

String xi= $1$0$0 1$ 1$0$0$1$0$0$1$0$0$θ String yi= $1$0$0 1$ 1$0$0$1$0$0$1$0$0$θ MPCP Solution Sequence:0,3,1,1,3,2,2,4 PCP SolutionSequence:3,1,1,3,2,2,4

P Problems: As the name says these problems can be solved in polynomial time, i.e.; O(n), O(n2) or O(n^k), where k is a constant.

NP (Non-Polynomial or Non-deterministic Polynomial-time) Problems: The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search. For example, the Sudoku game.

NP-Hard Problems: A problem is said to be NP-Hard when an algorithm for solving NP-Hard can be translated to solve any NP problem. Then we can say, this problem is at least as hard as any NP problem, but it could be much harder or more complex.

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

NP-Complete Problems: NP-Complete (NPC) problems are problems that are present in both the NP and NP-Hard classes. That is NP-Complete problems can be verified in polynomial time and any NP problem can be reduced to this problem in polynomial time.

Examples of NP-Complete problems where no polynomial time algorithm is known are as
follows −

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfactory, etc.

5.6. Counter Machines:

Counter Machine may also be regarded as a restricted multi-stack machine. The restrictions are as follows,
a)There are only two stack symbols,which we shall refer to as Z0(the bottom of stack marker) and X.

b)Z0 is initially on each stack.
c)We may replace Z0 only by a string of the form X^iz0 for some i >=0.
d)We may replace X only by X^i for some i >= 0. That's Z0 appears only on the bottom of each stack and all other stack symbols if any are X.

IMPORTANT QUESTIONS:

- Define Recursive Languages
- Define recursively enumerable languages
- Define LBA
- State PCP and MPCP
- Explain Variations of the TM
- Construct a TM over {a,b} which contains a substring abb
- Write a note on Universal Turing Machine
- Closure properties of Recursive and Recursively enumerable languages
- Design LBA for L={$a^n b^n c^n$: n>=1}
- Consider the following sequence and find whether it has a solution (decidable) or not.

i	List A	List B
1	1	111
2	10111	10
3	10	0

- Write the Procedure to convert MPCP to PCP or Reduction of MPCP to PCP
- Consider the following MPCP instance and find whether it has a solution

i	gi	hi
1	100	1
2	0	100
3	1	00