Mrs. Gauri Deshapnde

Theory of Computation

Unit-I

Syllabus

Automata Theory

Basic terms: Symbols, String, Language, Automata, Finite automaton, transition and its properties,

FSM without output: Definition and types of finite automaton, Construction of DFA and NFA, Convert NFA to DFA, Equivalence of DFA and NFA, Minimization of FA.

FSM with output: Definition and Construction of Moore and Mealy Machines, Inter-conversion between Moore and Mealy Machines.

Formal Languages

Defining Grammar, Derivations, Languages generated by Grammar, Chomsky Classification of Grammar and Languages, Recursive Enumerable Sets, Operations on Languages, Languages and Automata

Introduction

Theory of computation is the branch that deals with how efficiently problems can be solved on a model of computations using an algorithm. Theory of computation field is divided into three branches.

1) Automata theory and languages
2) Computability Theory
3) Computational Complexity Theory

Automata theory -It is the study of abstract machines and automata [self-acting machine]

Computability theory- It is also known as recursion theory, is a branch of mathematical logic, of computer science, and of the theory of computation that originated in the 1930s with the study of computable functions and Turing degrees

Computational complexity- This theory focuses on classifying computational problems according to their inherent difficulty, and relating these classes to each other. A computational problem is a task solved by a computer. A computation problem is solvable by mechanical application of mathematical steps, such as an algorithm.

Basic Terms

Alphabets: An alphabet is any finite set of symbols.

Example: $\sum$ = {a, b, c, d} is an alphabet set where 'a', 'b', 'c', and 'd' are symbols.

String: A string is a finite sequence of symbols taken from $\sum$.

Example – 'cabcad' is a valid string on the alphabet set $\sum$ = {a, b, c, d}

Length of String :It is the number of symbols present in string.(denoted by |S|)

Example : if S='abcd' then |S|=4

if |S|=0 it is called as an empty string(denoted by ε)

Language: A language is a subset of $\sum^*$ for some alphabet $\sum$. It can be finite or infinite.

e.g : if the language takes all possible strings of length 2 over $\sum$ = {a,b}

then L={ab,aa,ba,bb}

Kleene Star: The Kleen star $\sum^*$ is a unary operator on a set of symbols or strings . $\sum$ gives infinite set of all possible strings of all possible length over $\sum$ including empty string.

Representation: $\sum^* = \sum_0 \cup \sum_1 \cup \sum_2 \cup$ .....where $\sum_p$ is the set of all possible strings of length of p.

Example: if $\sum$ = {a,b}, $\sum^*$ ={ε,a,b,aa,ab,ba,bb.....}

Kleene closure/Plus: The Kleen plus $\sum+$ is the infinite set of all possible strings of all possible length over $\sum$ excluding empty string.

Representation: $\sum+ = \sum 1 \cup \sum 2 \cup \sum 3 \cup$ .....where $\sum p$ is the set of all possible strings of length of p.

Example: if $\sum$ = {a,b}, $\sum+$ ={a,b,aa,ab,ba,bb.....}

Transition System and its Properties

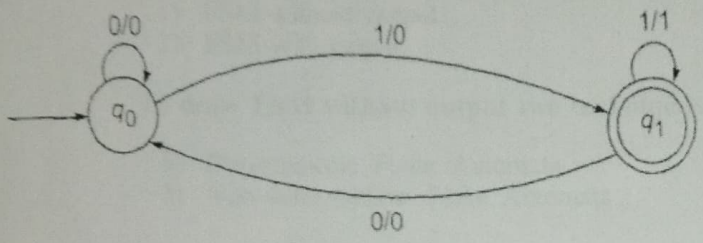
A **transition system** or **transition** graph is a finite directed labeled graph in which each vertex represents a state and the directed edges indicate of a state and the edges are labeled with input/output. This diagram shows a **transition system** and initial and final states.

Definition 3.2 A transition system is a 5-tuple $(Q, \Sigma, \delta, Q_0, F)$, where

(i) Q, Σ and F are the finite nonempty set of states, the input alphabet, and the set of final states, respectively, as in the case of finite automata;

(ii) $Q_0 \subseteq Q$, and Q_0 is nonempty; and

(iii) δ is a finite subset of $Q \times \Sigma^* \times Q$.



Property 1 $\delta(q, \Lambda) = q$ is a finite automaton. This means that the state of the system can be changed only by an input symbol.

Property 2 For all strings w and input symbols a,

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

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

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

It is mathematic model

Finite State Machine/Finite Automaton

- FSM is a calculation model that can be executed with the help of hardware otherwise software.
- This is used for creating sequential logic as well as a few computer programs.
- FSMs are used to solve the problems in fields like mathematics, games, linguistics, and artificial intelligence.
- In a system where specific inputs can cause specific changes in state that can be signified with the help of FSMs.
- **Definition** : An automaton with finite number of states is called a Finite Automaton(FA) or Finite State Machine(FSM).

An automaton can be represented by 5 tuple $(Q, \Sigma, \partial, q_0, F)$ where

Q is a finite set of states.

$\sum$ is finite set of symbols called the alphabet of the automaton.

∂ is transition function.

q_0 is the initial state from where any input is processed ($q_0 \in Q$)

F is a set of final state /states of Q (Fc Q)

- Finite State machine can be classified into two categories
  1) FSM without output
  2) FSM with output

To draw FSM without output two techniques are used:

1) Deterministic Finite Automata
2) Non deterministic Finite Automata

To draw FSM with output two machines are used:

1) Moore machine
2) Mealy machine

FSM without Output

Deterministic Finite Automata

A DFA can be represented by a 5 tuples $(Q, \sum, \partial, q_0, F)$ where

- Q is a finite set of states.

- $\sum$ is finite set of symbols called the alphabet of the automaton.

- ∂ is transition function.

- q_0 is the initial state from where any input is processed ($q_0 \in Q$)

- F is a set of final state /states of Q (Fc Q)

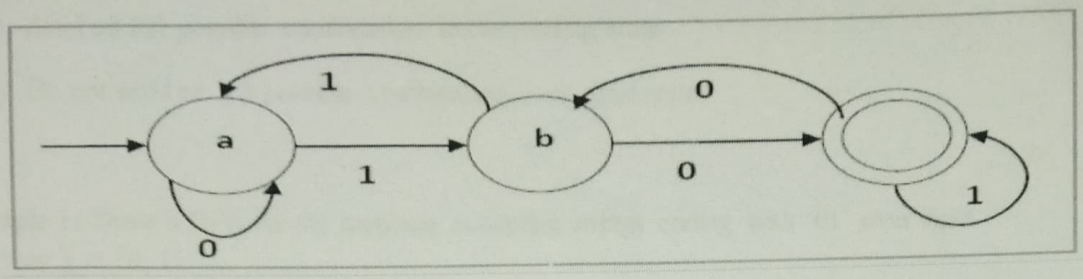
Example 1:

Let a deterministic finite automaton be

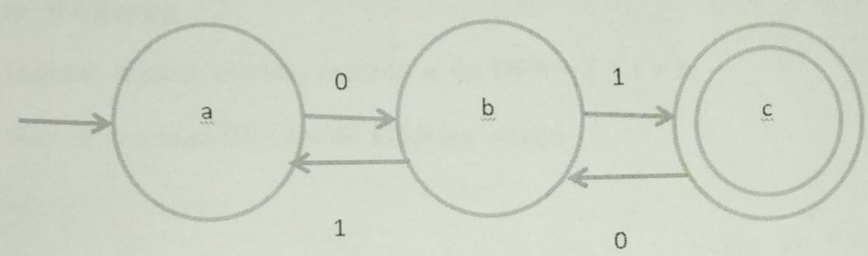
$Q = \{a, b, c\}$, $\sum = \{0, 1\}$, $q_0 = \{a\}$, $F = \{c\}$,

and Transition function δ as shown by the following table –

Present State	Next State for Input 0	Next State for Input 1
A	a	B
B	c	A
C	b	C



Example 2 : Generate Deterministic Finite Automaton transition table



Present State	Next State for Input 0	Next State for Input 1
A	b	-
B	-	c,a
C	b	-

- **How to construct DFA from given language?**

Problem 1: Languages consisting of strings ending with a particular substring.

Steps:

1) Calculate minimum number of DFA states. If length of substring is n then number of states are n+1
2) Decide the strings for which DFA will construct.
3) Always prefer existing path

4) Send all left possible combination to the starting state

5) Do not send all left possible combination over dead state.

Example 1: Draw a DFA for the language accepting strings ending with '01' over input alphabets $\sum = \{0, 1\}$

Solution:

Step 1:All strings of the language ends with substring "01".

So, length of substring = 2

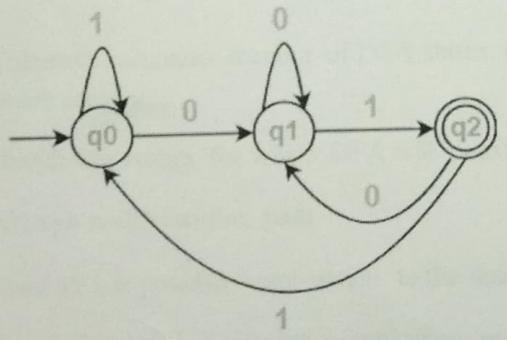
Thus, Minimum number of states required in the DFA = 2 + 1 = 3.

Step 2: We will construct DFA for the following strings-

01

001

0101



Example 2: Draw a DFA for the language accepting strings ending with 'abb' over input alphabets $\sum = \{a, b\}$

Solution:

Regular expression for the given language = (a + b)*abb

Step 1:All strings of the language ends with substring "abb".

So, length of substring = 3

Thus, Minimum number of states required in the DFA = 3 + 1 = 4.

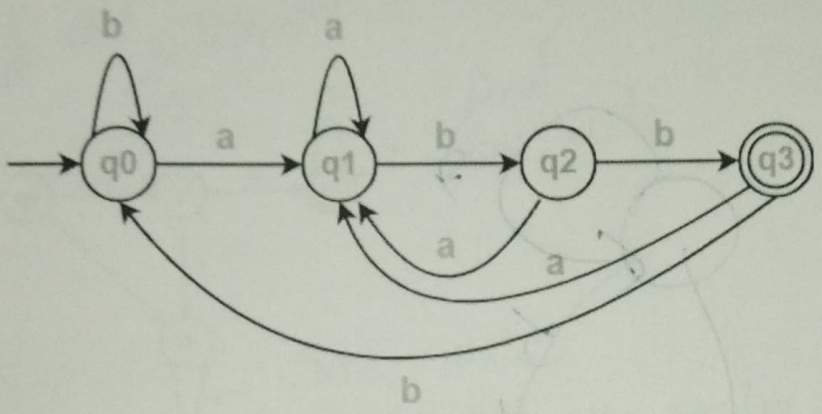
Step 2: We will construct DFA for the following strings-

abb

aabb

ababb

abbabb



- **How to construct DFA from given language?**

Problem 2: Languages consisting of strings starting with a particular substring.

Steps:

1) Calculate minimum number of DFA states. If length of substring is n then number of states are n+2
2) Decide the strings for which DFA will construct.
3) Always prefer existing path
4) Send all left possible combination to the dead state
5) Do not send all left possible combination over start state.

Example 1: Draw a DFA for the language accepting strings starting with 'ab' over input alphabets $\sum = \{a, b\}$

Solution:

Regular expression for the given language = $ab(a + b)^*$

Step 1:All strings of the language starts with substring "ab".

So, length of substring = 2.Thus, Minimum number of states required in the DFA = 2 + 2 = 4.

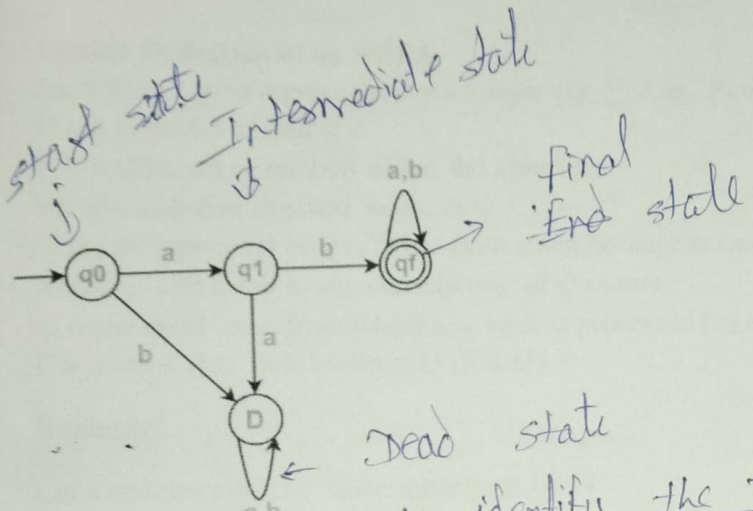
It suggests that minimized DFA will have 4 states.

Step 2: We will construct DFA for the following strings-

ab

aba

abab



start state
Intermediate state
final / End state
Dead state
to identify the regular grammer

Example 2: Draw a DFA for the language accepting strings starting with 'a' over input alphabets $\sum = \{a, b\}$

Deterministic finite automata
Combination of opesand and operators.

Solution:

Regular expression for the given language = $a(a + b)^*$ → generates zero or more

Step 1: All strings of the language starts with substring "a".

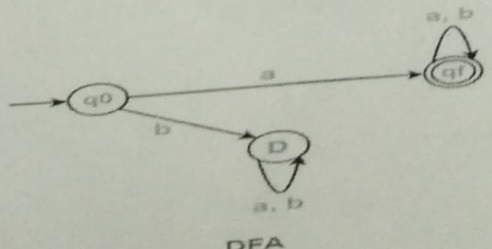
So, length of substring = 1. Thus, Minimum number of states required in the DFA = $1 + 2 = 3$.

It suggests that minimized DFA will have 3 states.

Step 2: We will construct DFA for the following strings-

a

aa

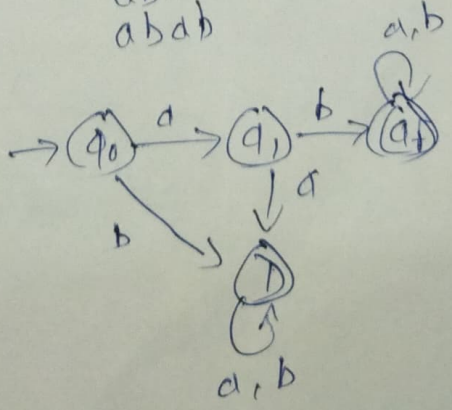


DFA

for ab
$ab(a+b)^*$
2 + 2 = 4.

abb
$abb(a+b)^*$ so, a, b

a
aba
abab



we don't know what is the next step of the automata.

II) Non Deterministic Finite Automata (NDFA/NFA)

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple $(Q, \sum, \delta, q_0, F)$ where –

Q is a finite set of states.

$\sum$ is a finite set of symbols called the alphabets.

δ is the transition function where $\delta: Q \times \sum \rightarrow 2^Q$

(Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)

q_0 is the initial state from where any input is processed ($q_0 \in Q$).

F is a set of final state/states of Q ($F \subseteq Q$).

Example

Let a non-deterministic finite automaton be →

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

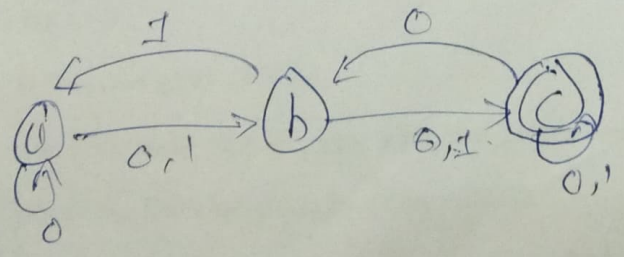
$\sum = \{0, 1\}$

$q_0 = \{a\}$

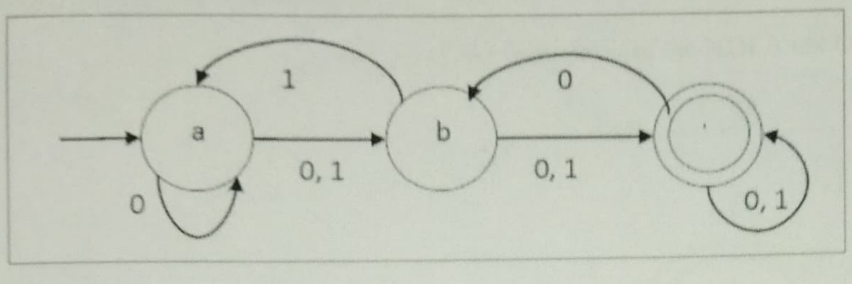
$F = \{c\}$

The transition function δ as shown below –

Present State	Next State for Input 0	Next State for Input 1
a	a, b	b
b	c	a, c
c	b, c	c



Its graphical representation would be as follows –



Difference between DFA and NDFA

DFA	NDFA
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*.
Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
Requires more space.	Requires less space.
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.

Convert NFA to DFA

Step 1 – Create state table from the given NDFA.

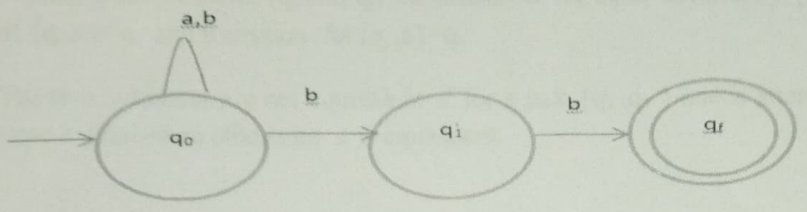
Step 2 – Create a blank state table under possible input alphabets for the equivalent DFA.

Step 3 – Mark the start state of the DFA by q0 (Same as the NDFA).

Step 4 – Find out the combination of States $\{Q_0, Q_1, \ldots, Q_n\}$ for each possible input alphabet.

Step 5 – Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

Step 6 – The states which contain any of the final states of the NDFA are the final states of the equivalent DFA

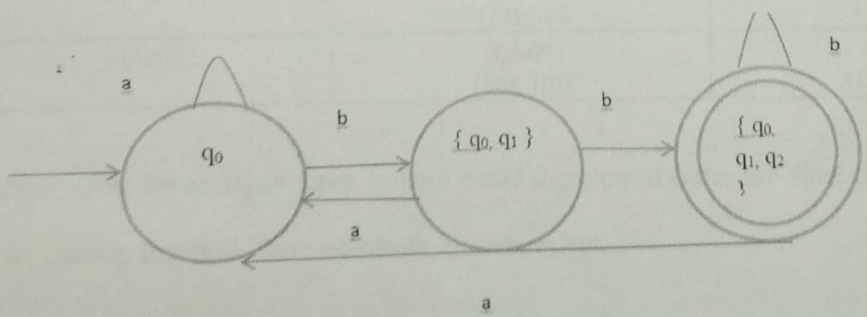


Step 1: Construct NFA transition table

Present States	a	b
q_0	q_0	{ q_0, q_1 }
q_1	-	q_f
q_f	-	-

Step2: Convert into DFA

Present States	a	B
q_0	q_0	{ q_0,q_1}
{ q_0,q_1}	q_0	{ q_0,q_1,q_2}
{ q_0,q_1,q_2}	q_0	{ q_0,q_1,q_2}

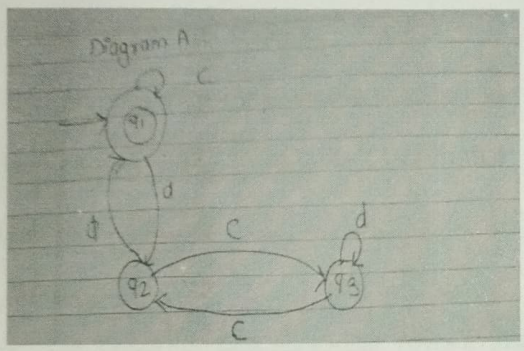


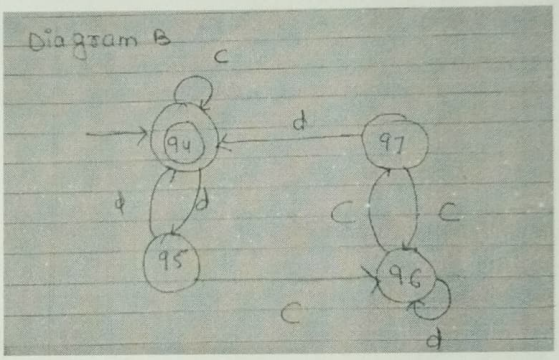
- **Equivalence of two finite automata**
  In finite automata there are three kind of states: 1) Initial state 2)Intermediate state 3)Final state

Steps to identify equivalence:

For any pairs of states (q_i and q_j) the transition for input defined by $\{q_a, q_b\}$ where transition of $\{q_i, a\}=q_a$ and transition for $\{q_j, a\}=q_b$

The two automata are not equivalent if for a pair $\{q_a, q_b\}$ one is intermediate state and other state is final state otherwise it is equivalent.





States	Input c	Input d
{q1,q4}	{q1,q4} (final,final)	{q2,q5} (Im,Im)
{q2,q5}	{q3,q6} (Im,Im)	{q1,q4} (Im,Im)
{q3,q6}	{q2,q7} (Im,Im)	{q3,q6} (Im,Im)
{q2,q7}	{q3,q6} (Im, Im)	{q1,q4} (final,final)

In above table for each pair input symbol c and d got equal states like final or intermediate.

So we can say that both finite automata are equivalent.

Minimization of DFA

The Minimization of DFA means reducing the number of states from given finite automata

Steps:

1) Remove all the states that are unreachable from initial states.
2) Draw the transition table for all pairs of states.
3) Split transition table into two tables T1 & T2

-T1 contains all final states

-T2 contains all non-final states

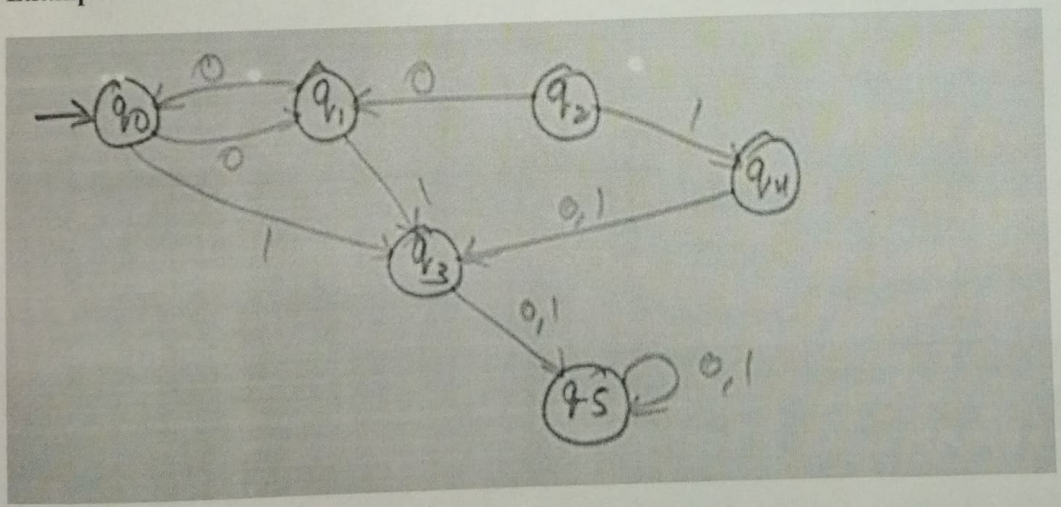
4) Find similar states from table T1 such that

$\partial(q, a)=p$

$\partial(, a)=p$

5) Repeat step 3 until we find no similar rows available at T1.
6) Repeat step3 and step 4 in Table 2
7) Now combine reduced T1 and T2 also

Example:



1) Remove q2 and q4 in Finite Automata
2) Draw transition table for the rest of the states.

State	0	1
q0	q1	q3
q1	q0	q3
q3	q5	q5
q5	q5	q5

3) i) Create Table T1 which start from non -final states

State	0	1
q0	q1	q3
q1	q0	q3

ii) Create Table T2 which start from final states

State	0	1
q3	q5	q5
q5	q5	q5

4) First table does not have any similar rows.

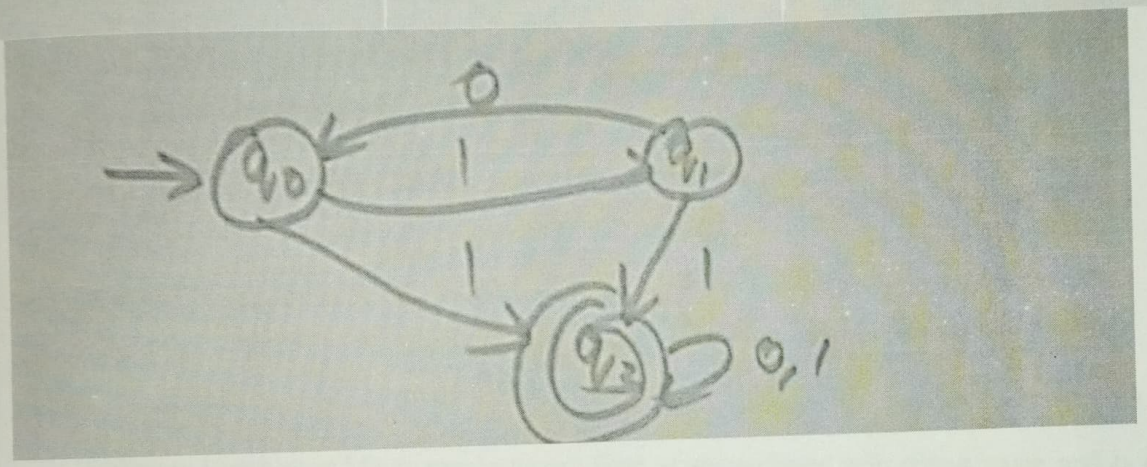
Second table has similar rows.

So skip q5 and replace q5 by q3

State	0	1
q3	q3	q3

5 combine T1 and T2

State	0	1
q0	q1	q3
q1	q0	q3
q3	q3	q3



FSM with Output

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

1) Mealy Machine

2) Moore machine

Mealy Machine

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

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

Q is a finite set of states.

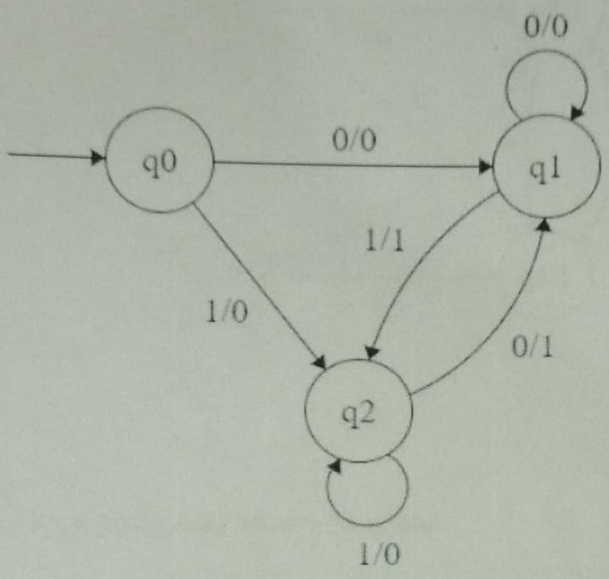
$\sum$ is a finite set of symbols called the input alphabet.

O is a finite set of symbols called the output alphabet.

δ is the input transition function where $\delta: Q \times \sum \rightarrow Q$

X is the output transition function where X: $Q \times \sum \rightarrow O$

q_0 is the initial state from where any input is processed ($q_0 \in Q$).



Moore Machines: Moore machines are finite state machines with output value and its output depends only on present state. It can be defined as (Q, q0, $\sum$, O, δ, λ) where:

Q is finite set of states.

q0 is the initial state.

$\sum$ is the input alphabet.

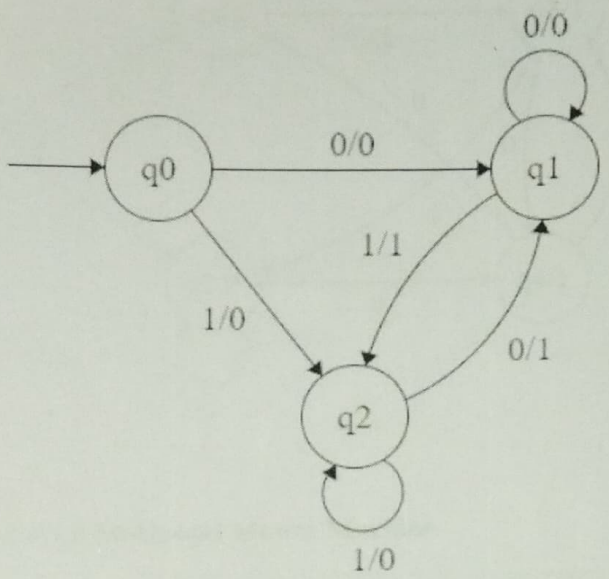
O is the output alphabet.

δ is transition function which maps $Q \times \sum \rightarrow Q$.

λ is the output function which maps $Q \rightarrow O$.

X is the output transition function where X: $Q \times \sum \rightarrow O$

$\mathbf{q_0}$ is the initial state from where any input is processed ($q_0 \in Q$).



Moore Machines: Moore machines are finite state machines with output value and its output depends only on present state. It can be defined as (Q, q0, $\sum$, O, δ, λ) where:

Q is finite set of states.

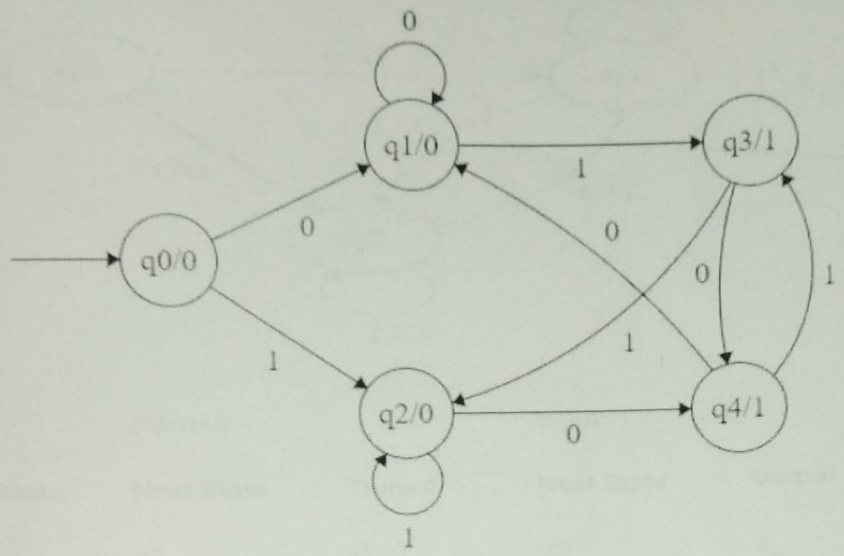
q0 is the initial state.

$\sum$ is the input alphabet.

O is the output alphabet.

δ is transition function which maps $Q \times \sum \rightarrow Q$.

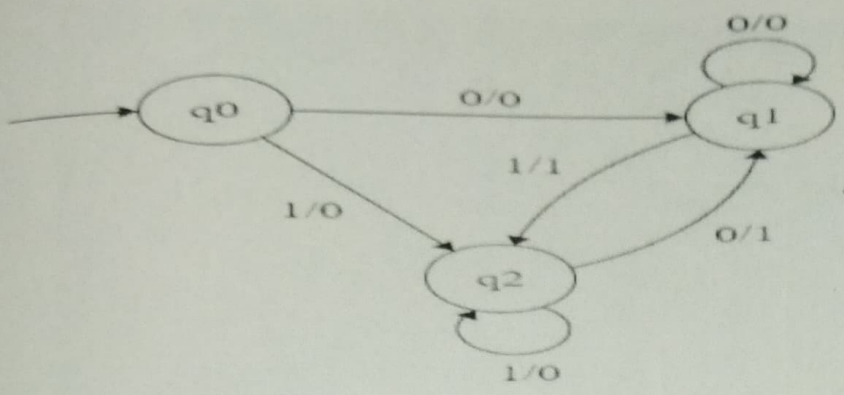
λ is the output function which maps $Q \rightarrow O$.



- **Difference between Mealy and Moore Machine**

Mealy Machine	Moore Machine
Output depends both upon the present state and the present input	Output depends only upon the present state.
Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
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.
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.

Construct Transition table from Mealy Diagram



Mealy

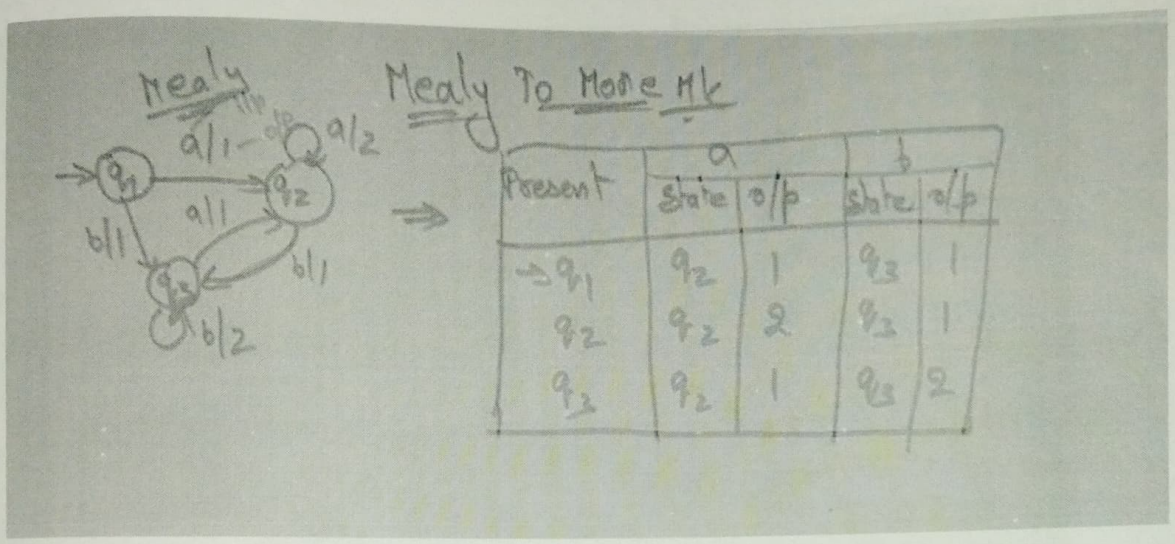
	Input=0		Input=1	
Present State	Next State	Output	Next State	Output
q0	q1	0	q2	0
q1	q1	0	q2	1
q2	q1	1	q2	0

Construct Transition table from Moore Diagram

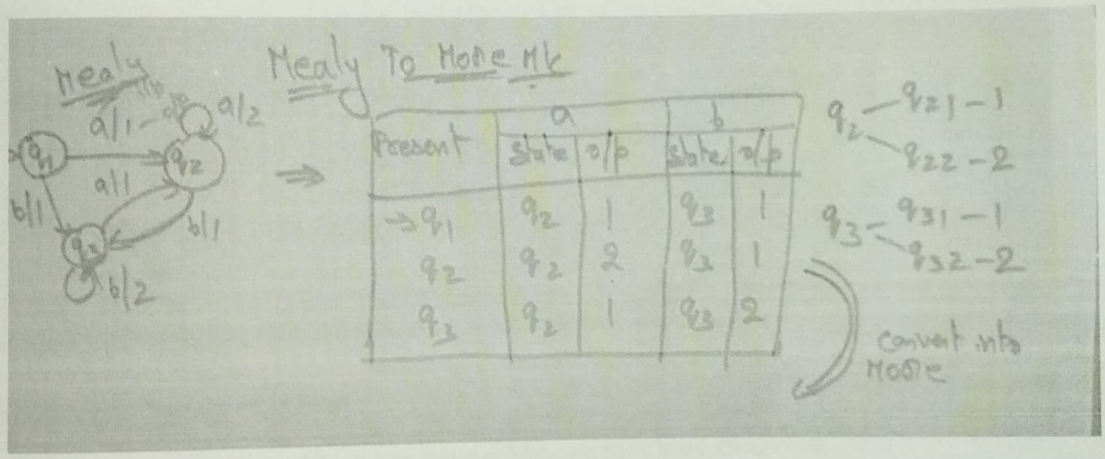
Current State	Next State (δ)		Output(λ)
	0	1	
q_0	q_1	q_2	1
q_1	q_2	q_1	1
q_2	q_2	q_0	0

Convert Mealy to Moore Machine

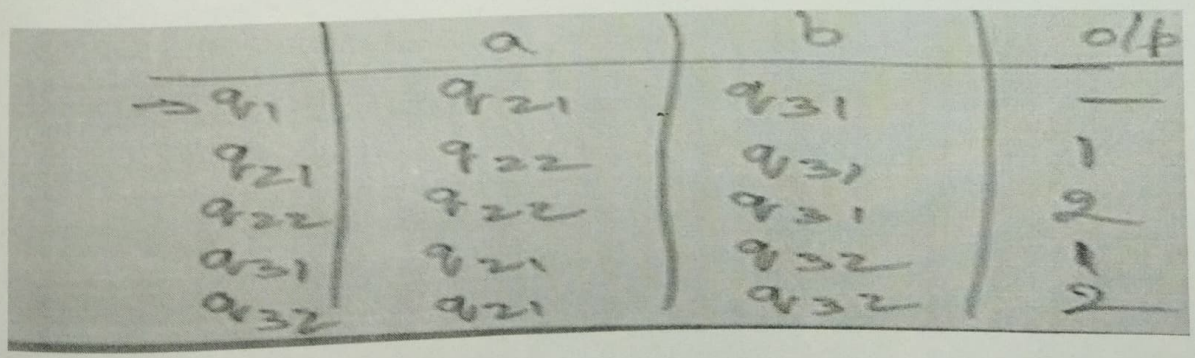
Step1: Construct Transition table from given Mealy machine diagram

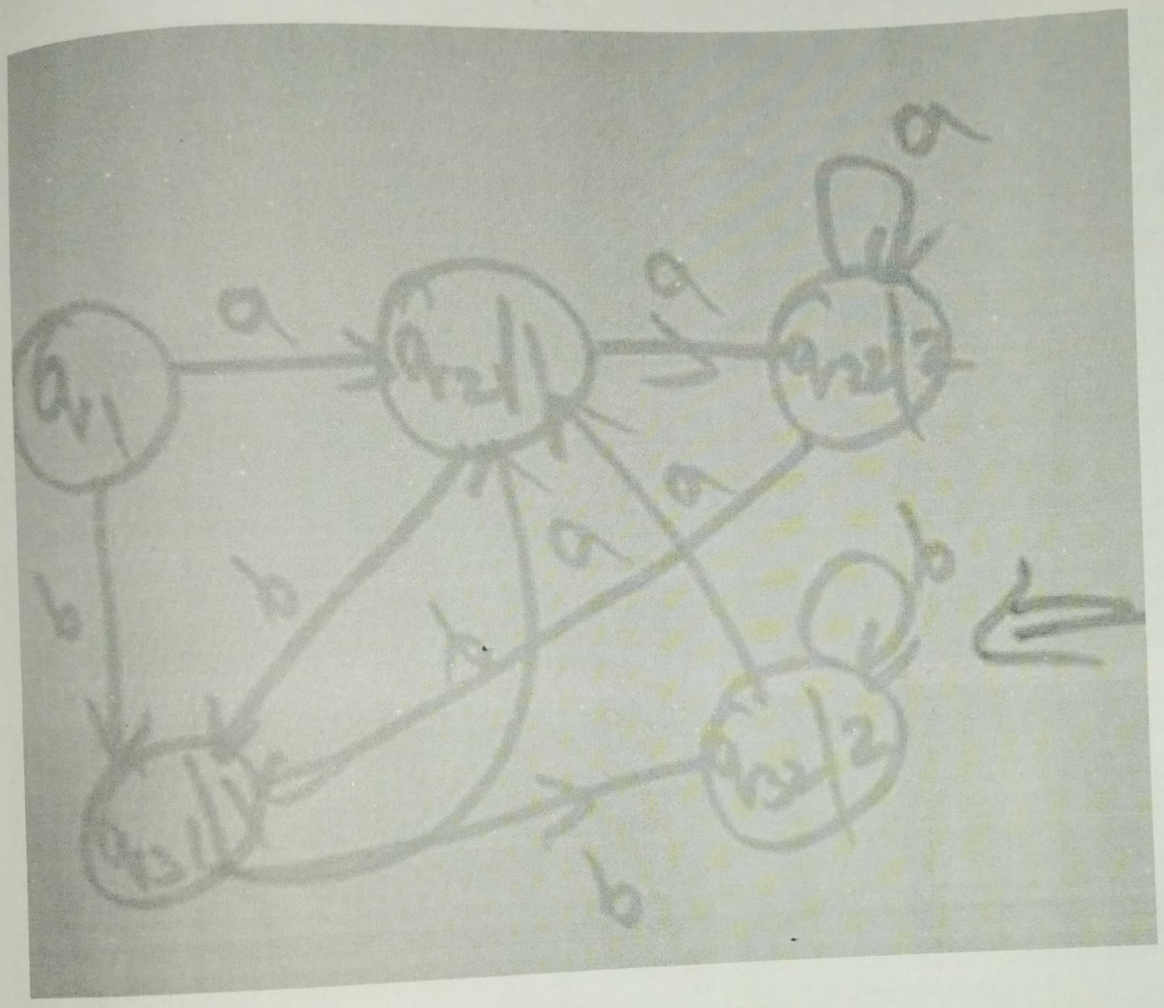


Step2: Split transition



Step3: Construct Moore Transition table

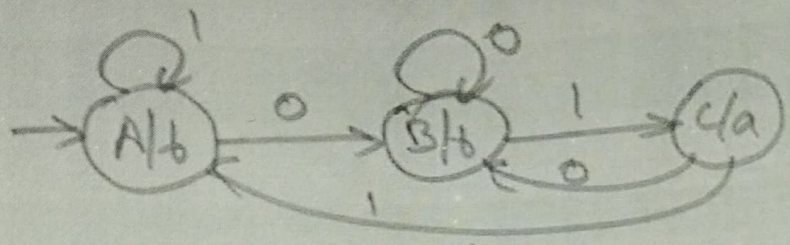




Convert Moore to Mealy Machine

Step1: Construct Transition table from given Moore machine diagram

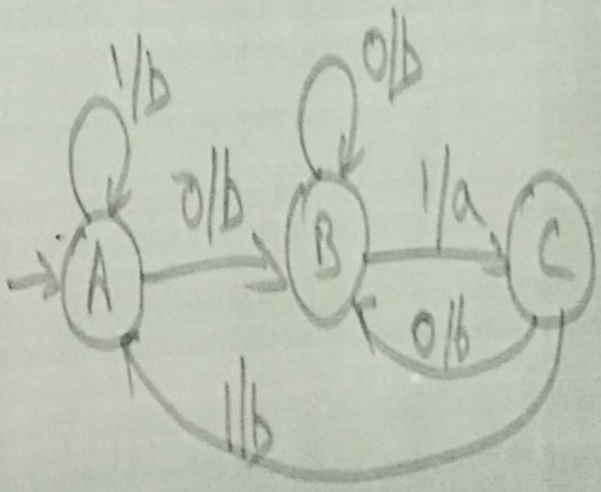
Example :



↓

state	0	1	o/p
→ A	B	A	b
B	B	C	b
C	B	A	a

Mealy tx dig



Formal Languages

Defining Grammar:

In the literary sense of the term, grammars denote syntactical rules for conversation in natural languages. Linguistics has attempted to define grammars since the inception of natural languages like English, Sanskrit, Mandarin, etc.

A grammar **G** can be formally written as a 4-tuple (N, T, S, P) where –

N or V_N is a set of variables or non-terminal symbols

T or $\sum$ is a set of Terminal symbols.

S is a special variable called the Start symbol, $S \in N$

P is Production rules for Terminals and Non-terminals. A production rule has the form $\alpha \rightarrow \beta$, where α and β are strings on $V_N \cup \sum$ and least one symbol of α belongs to V_N.

A Grammar is mainly composed of two basic elements-

1. Terminal symbols

2. Non-terminal symbols

1. Terminal Symbols- Terminal symbols are those which are the constituents of the sentence generated using a grammar. Terminal symbols are denoted by using small case letters such as a, b, c etc.

2. Non-Terminal Symbols- Non-Terminal symbols are those which take part in the generation of the sentence but are not part of it. Non-Terminal symbols are also called as **auxiliary symbols** or **variables**.
Non-Terminal symbols are denoted by using capital letters such as A, B, C etc.
Eg: S--> AB
A-->a

B-->b

This is a production

Now the starting symbol is **S** here

Non terminals are **S A B**

Terminals are **a b**

So we can say the from must be

G(V, T, P, S) = G((S, A, B),(a, b),(S-->AB,A-->a,B-->b),(S))

Derivations from a Grammar

Strings may be derived from other strings using the productions in a grammar. If a grammar **G** has a production $\alpha \rightarrow \beta$, we can say that **x α y** derives **x β y** in **G**.

This derivation is written as –

$x \alpha y \Rightarrow_G x \beta y$

Example:

Let us consider the grammar –

G2 = ({S, A}, {a, b}, S, {S → aAb, aA → aaAb, A → ε })

Some of the strings that can be derived are –

S ⇒ aAb using production S → aAb

⇒ aaAbb using production aA → aAb

⇒ aaaAbbb using production aA → aAb

⇒ aaabbb using production A → ε

Consider a grammar G = (V , T , P , S) where-

V = { S } // Set of Non-Terminal symbols

T = { a , b } // Set of Terminal symbols

P = { S → aSbS , S → bSaS , S → ∈ } // Set of production rules

S = { S } // Start symbol

This grammar generates the strings having equal number of a's and b's

Example:

Suppose we have the following grammar –

G: N = {S, A, B} T = {a, b} P = {S → AB, A → aA|a, B → bB|b}

The language generated by this grammar –

L(G) = {ab, a^2b, ab^2, a^2b^2, ...}

= {$a^m b^n$ | m ≥ 1 and n ≥ 1}

Construction of Grammar

Example 1:

Problem – Suppose, L (G) = $\{a^m b^n \mid m \geq 0 \text{ and } n > 0\}$. We have to find out the grammar **G** which produces **L(G)**.

Solution

Since $L(G) = \{a^m b^n \mid m \geq 0 \text{ and } n > 0\}$

the set of strings accepted can be rewritten as –

L(G) = {b, ab,bb, aab, abb, ...}

Here, the start symbol has to take at least one 'b' preceded by any number of 'a' including null.

To accept the string set {b, ab, bb, aab, abb, ...}, we have taken the productions –

$S \rightarrow aS$, $S \rightarrow B$, $B \rightarrow b$ and $B \rightarrow bB$

$S \rightarrow B \rightarrow b$ (Accepted)

$S \rightarrow B \rightarrow bB \rightarrow bb$ (Accepted)

$S \rightarrow aS \rightarrow aB \rightarrow ab$ (Accepted)

$S \rightarrow aS \rightarrow aaS \rightarrow aaB \rightarrow aab$(Accepted)

$S \rightarrow aS \rightarrow aB \rightarrow abB \rightarrow abb$ (Accepted)

Thus, we can prove every single string in L(G) is accepted by the language generated by the production set.

Hence the grammar –

G: ({S, A, B}, {a, b}, S, { $S \rightarrow aS \mid B$, $B \rightarrow b \mid bB$ })

Example 2:

Problem – Suppose, L (G) = $\{a^m b^n \mid m > 0 \text{ and } n \geq 0\}$. We have to find out the grammar G which produces L(G).

Solution –

Since $L(G) = \{a^m b^n \mid m > 0 \text{ and } n \geq 0\}$, the set of strings accepted can be rewritten as –

L(G) = {a, aa, ab, aaa, aab ,abb, ...}

Here, the start symbol has to take at least one 'a' followed by any number of 'b' including null.

To accept the string set {a, aa, ab, aaa, aab, abb, ...}, we have taken the productions –

$S \rightarrow aA$, $A \rightarrow aA$, $A \rightarrow B$, $B \rightarrow bB$,$B \rightarrow \lambda$

$S \rightarrow aA \rightarrow aB \rightarrow a\lambda \rightarrow a$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaB \rightarrow aa\lambda \rightarrow aa$ (Accepted)

$S \rightarrow aA \rightarrow aB \rightarrow abB \rightarrow ab\lambda \rightarrow ab$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaaA \rightarrow aaaB \rightarrow aaa\lambda \rightarrow aaa$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaB \rightarrow aabB \rightarrow aab\lambda \rightarrow aab$ (Accepted)

$S \rightarrow aA \rightarrow aB \rightarrow abB \rightarrow abbB \rightarrow abb\lambda \rightarrow abb$ (Accepted)

Thus, we can prove every single string in L(G) is accepted by the language generated by the production set.

Hence the grammar –

G: ({S, A, B}, {a, b}, S, {S → aA, A → aA | B, B → λ | bB })

Language generated by Grammar

The set of all strings that can be derived from a grammar is said to be the language generated from that grammar. A language generated by a grammar **G** is a subset formally defined by $L(G)=\{W|W \in \sum^*, S \Rightarrow_G W\}$

If **L(G1) = L(G2)**, the Grammar **G1** is equivalent to the Grammar **G2**.

Example

If there is a grammar

G: N = {S, A, B} T = {a, b} P = {S → AB, A → a, B → b}

Here **S** produces **AB**, and we can replace **A** by **a**, and **B** by **b**. Here, the only accepted string is **ab**, i.e.

L(G) = {ab}

Example

Suppose we have the following grammar –

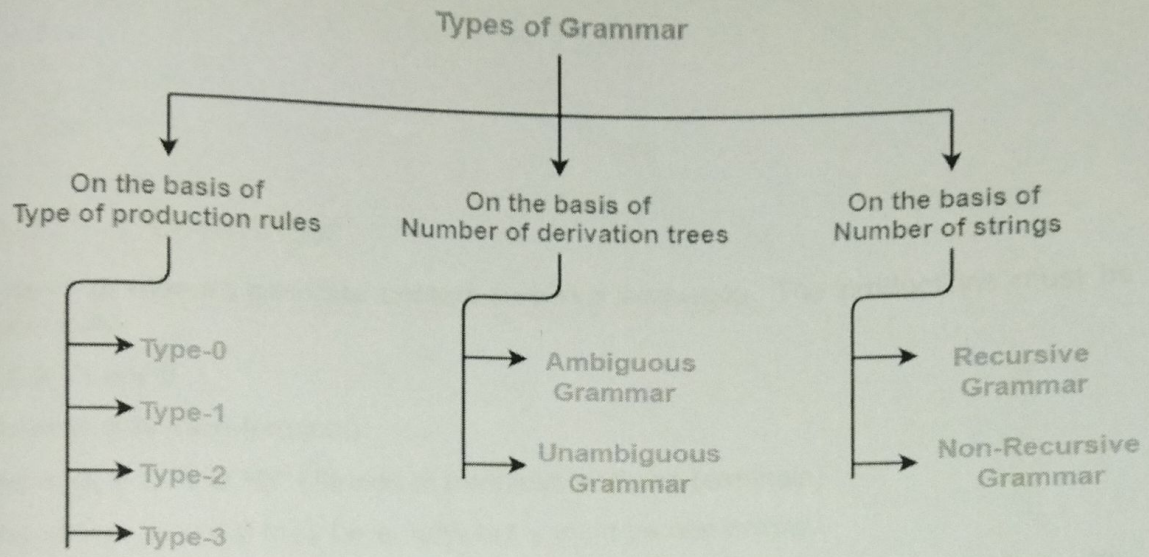
G: N = {S, A, B} T = {a, b} P = {S → AB, A → aA|a, B → bB|b}

The language generated by this grammar –

$L(G) = \{ab, a^2b, ab^2, a^2b^2, ...\}$

$= \{a^m b^n \mid m \geq 1 \text{ and } n \geq 1\}$

Types Grammar



(Chomsky Hierarchy)

Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

```
X → ε
X → a | aY
Y → b
```

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

S → X a
X → a
X → aX
X → abc
X → ε

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule **S → ε** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

AB → AbBc
A → bcA
B → b

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α → β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

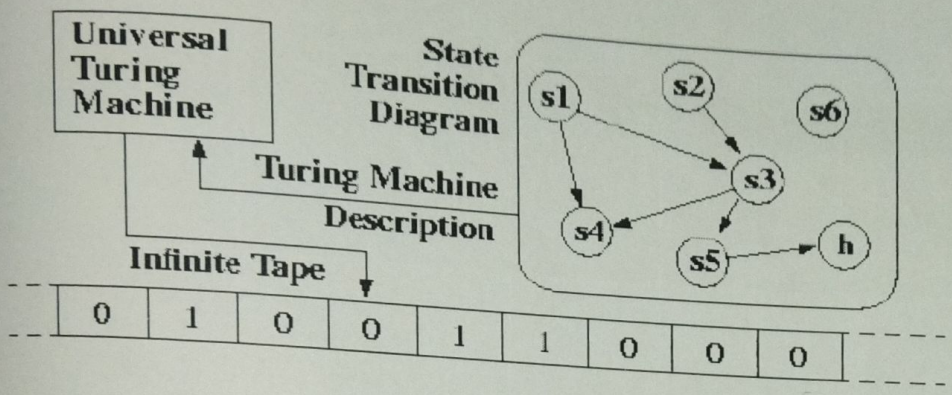
Example

S → ACaB
Bc → acB
CB → DB
aD → Db

Recursive Enumerable Set

The Turing machine may

1) Halt and accept the input
2) Halt and reject the input
3) Never Halt and loop



Recursively Enumerable Language: There is Turing machine for a language which accept every string otherwise not.

Recursively Language: There is Turing machine for a language which halt on every string.

Recursive Enumerable (RE) or Type -0 Language

RE languages or type-0 languages are generated by type-0 grammars. An RE language can be accepted or recognized by Turing machine which means it will enter into final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means TM can loop forever for the strings which are not a part of the language. RE languages are also called as Turing recognizable languages.

Recursive Language (REC)

A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language. e.g.; L= $\{a^nb^nc^n|n>=1\}$ is recursive because we can construct a turing machine which will move to final state if the string is of the form $a^nb^nc^n$ else move to non-final state. So the TM will always halt in this case. REC languages are also called as Turing decidable languages. The relationship between RE and REC languages can be shown in Figure 1.

Closure Properties of Recursive Languages

Union: If L1 and If L2 are two recursive languages, their union L1∪L2 will also be recursive because if TM halts for L1 and halts for L2, it will also halt for L1∪L2.

Concatenation: If L1 and If L2 are two recursive languages, their concatenation L1.L2 will also be recursive.

Kleene Closure: If L1is recursive, its kleene closure L1* will also be recursive

Intersection and complement: If L1 and If L2 are two recursive languages, their intersection L1 ∩ L2 will also be recursive

Operation on Languages

Certain operations on languages are common. This includes the standard set operations, such as union, intersection, and complement. Another class of operation is the element-wise application of string operations.

Examples: suppose L_1 and L_2 are languages over some common alphabet Σ.

- The concatenation $L_1 \cdot L_2$ consists of all strings of the form vw where v is a string from L_1 and w is a string from L_2.
- The intersection $L_1 \cap L_2$ of L_1 and L_2 consists of all strings that are contained in both languages
- The complement $\neg L_1$ of L_1 with respect to Σ consists of all strings over Σ that are not in L_1.
- The Kleene star: the language consisting of all words that are concatenations of zero or more words in the original language;