MCA-20204 FORMAL LANGUAGES & AUTOMATA THEORY

UNIT-I

Finite Automata and Regular Expressions: Basic Concepts of Finite State Systems, Chomsky Hierarchy of Languages, Deterministic and Non-Deterministic Finite Automata, Finite Automata with є-moves, Regular Expressions.

Regular sets &Regular Grammars: Basic Definitions of Formal Languages and Grammars, Regular Sets and Regular Grammars, Closure Properties of Regular Sets, Pumping Lemma for Regular Sets, Decision Algorithm for Regular Sets, Minimization of Finite Automata.

UNIT-II

Context Free Grammars and Languages: Context Free Grammars and Languages, Derivation Trees, simplification of Context Free Grammars, Normal Forms, Pumping Lemma for CFL, Closure properties of CFL's.

Push down Automata: Informal Description, Definitions, Push-Down Automata and Context free Languages, Parsing and Push-Down Automata

UNIT-III

Turing Machines: The Definition of Turing Machine, Design and Techniques for Construction of Turing Machines, Combining Turing Machines.

Universal Turing Machines and Undecidability: Universal Turing Machines. The Halting Problem, Decidable & Undecidable Problems - Post Correspondence Problem.

UNIT-IV

The Propositional calculus: The Prepositional Calculus : Introduction – Syntax of the Prepositional Calculus – Truth-Assignments – Validity and Satisfiability – Equivalence and Normal Forms – resolution in Prepositional Calculus.

The Predicate calculus: Syntax of the Predicate Calculate Calculus – Structures and Satisfiability – Equivalence – Un-solvability and NP-Completeness.

- ## Theory of Automata

Theory of automata is a theoretical branch of computer science and mathematical. It is the study of abstract machines and the computation problems that can be solved using these machines. The abstract machine is called the automata. The main motivation behind developing the automata theory was to develop methods to describe and analyse the dynamic behaviour of discrete systems.

This automaton consists of states and transitions. The **State** is represented by **circles**, and the **Transitions** is represented by **arrows**.

Automata is the kind of machine which takes some string as input and this input goes through a finite number of states and may enter in the final state.

Formal language

In mathematics, computer science, and linguistics, a formal language is a set of strings of symbols that may be constrained by rules that are specific to it. The alphabet of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed; frequently it is required to be finite

A formal language is often defined by means of a formal grammar such as a regular grammar or context-free grammar, also called its formation rule.

Alphabet

- **Definition** − An **alphabet** is any finite set of symbols.
- **Example** − ∑ = {a, b, c, d} is an **alphabet set** where ‘a’, ‘b’, ‘c’, and ‘d’ are **symbols**.

String

- **Definition** − A **string** is a finite sequence of symbols taken from ∑.
- **Example** − ‘cabcad’ is a valid string on the alphabet set ∑ = {a, b, c, d}

Length of a String

- **Definition** − It is the number of symbols present in a string. (Denoted by |**S**|).
- **Examples** −
    - If S = ‘cabcad’, |S|= 6
    - If |S|= 0, it is called an **empty string** (Denoted by **λ** or **ε**)

Kleene Star

- **Definition** − The Kleene star, ∑*, is a unary operator on a set of symbols or strings, ∑, that gives the infinite set of all possible strings of all possible lengths over ∑ including **λ**.
- **Representation** − $\sum^* = \sum_0 \cup \sum_1 \cup \sum_2 \cup \ldots\ldots$ where $\sum_p$ is the set of all possible stringsof length p.
- **Example** − If ∑ = {a, b}, ∑* = {λ, a, b, aa, ab, ba, bb,..............}

Kleene Closure / Plus

- **Definition** − The set $\sum^+$ is the infinite set of all possible strings of all possible lengths over ∑ excluding λ.
- **Representation** − $\sum^+ = \sum_1 \cup \sum_2 \cup \sum_3 \cup$…….

  $\sum^+ = \sum^* - \{ \lambda \}$
- **Example** − If ∑ = { a, b } , $\sum^+$ = { a, b, aa, ab, ba, bb,............... }

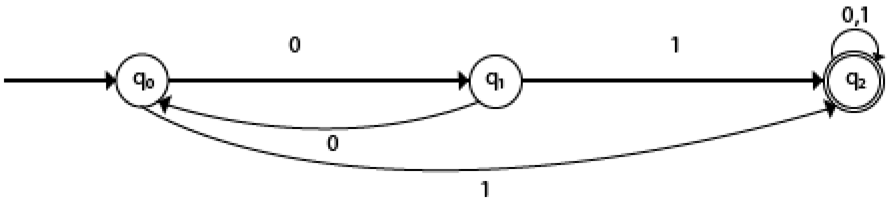
Language

- **Definition** − A language is a subset of ∑* for some alphabet ∑. It can be finite or infinite.
- **Example** − If the language takes all possible strings of length 2 over ∑ = {a, b}, then L = { ab, aa, ba, bb }

  Transition Table

  The transition table is basically a tabular representation of the transition function. It takes two arguments (a state and a symbol) and returns a state (the "next state").

Example 1:



Solution:

Transition table of given DFA is as follows:

Present State	Next state for Input 0	Next State of Input 1
→q0	q1	q2
q1	q0	q2
*q2	q2	q2

An automaton with a finite number of states is called a **Finite Automaton** (FA) or **Finite State Machine** (FSM).

Formal definition of a Finite Automaton

An automaton can be represented by a 5-tuple (Q, ∑, δ, q_0, F), where −

- **Q** is a finite set of states.

- ∑ is a finite set of symbols, called the **alphabet** of the automaton.
- **δ** is the 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 ($F \subseteq Q$).

Finite Automata Model:

Finite automata can be represented by input tape and finite control.

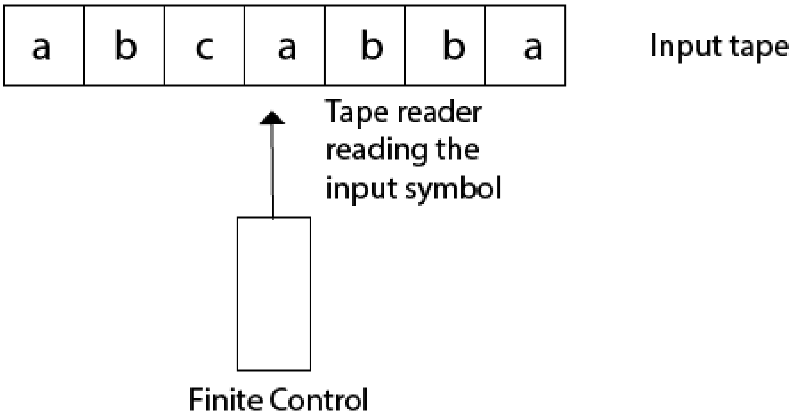


Fig :- Finite automata model

Input tape: It is a linear tape having some number of cells. Each input symbol is placed in each cell.

Finite control: The finite control decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right, and at a time only one input symbol is read.

- Finite Automaton can be classified into two types −

- Deterministic Finite Automaton (DFA)
- Non-deterministic Finite Automaton (NDFA / NFA)

Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton.**

Formal Definition of a DFA

A DFA can be represented by a 5-tuple (Q, ∑, δ, q_0, 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
- **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

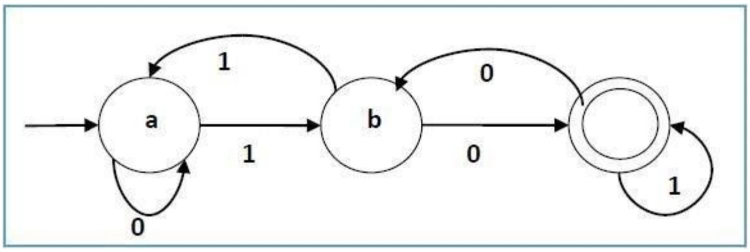
Let a deterministic finite automaton be →

- Q = {a, b, c},
- ∑ = {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

Its graphical representation would be as follows –



Example 1:

Design a FA with ∑ = {0, 1} accepts the only input 101.

Solution:

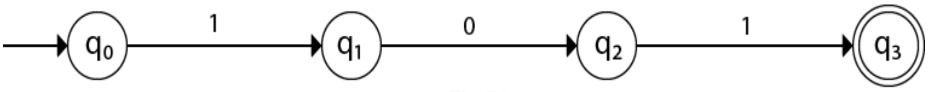


Fig: FA

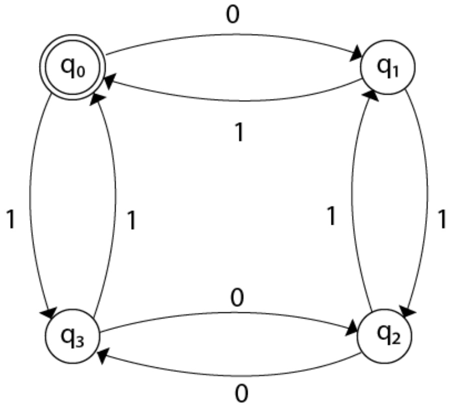
In the given solution, we can see that only input 101 will be accepted. Hence, for input 101, there is no other path shown for other input.

Example 2:

Design FA with ∑ = {0, 1} accepts even number of 0's and even number of 1's.

Solution:

This FA will consider four different stages for input 0 and input 1. The stages could be:

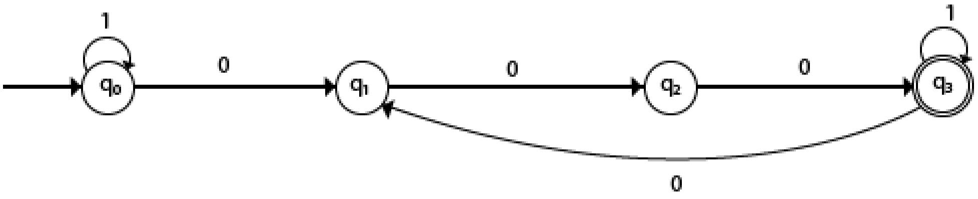


Example 3:

Design FA with ∑ = {0, 1} accepts the set of all strings with three consecutive 0's.

Solution:

The strings that will be generated for this particular languages are 000, 0001, 1000, 10001, ... in which 0 always appears in a clump of 3. The transition graph is as follows:



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**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where −

- **Q** is a finite set of states.
- ∑ is a finite set of symbols called the alphabets.
- **δ** is the transition function where δ: Q × ∑ → 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

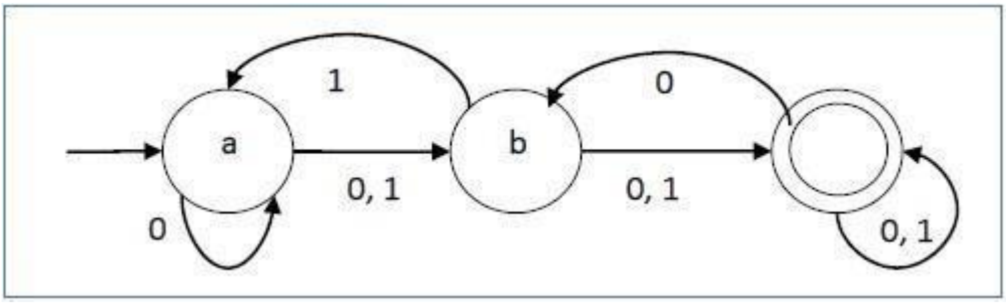
Let a non-deterministic finite automaton be →

- Q = {a, b, c}
- ∑ = {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 −



Acceptors, Classifiers, and Transducers

Acceptor (Recognizer)

An automaton that computes a Boolean function is called an **acceptor**. All the states of an acceptor is either accepting or rejecting the inputs given to it.

Classifier

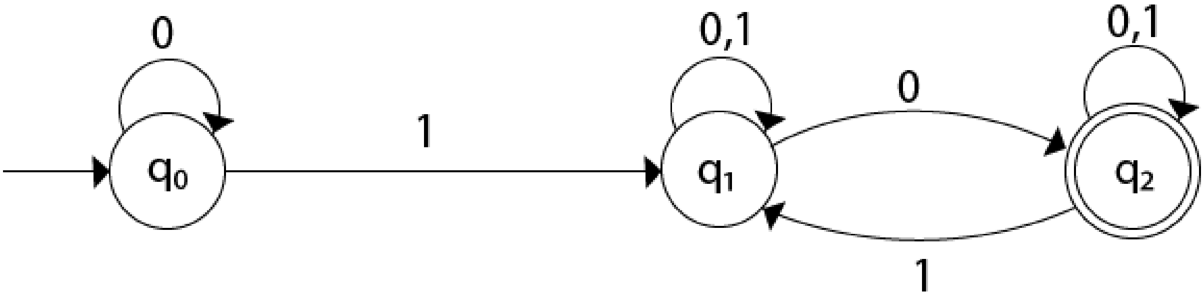
A **classifier** has more than two final states and it gives a single output when it terminates.

Transducer

An automaton that produces outputs based on current input and/or previous state is called a **transducer**. Transducers can be of two types −

- **Mealy Machine** − The output depends both on the current state and the current input.
- **Moore Machine** − The output depends only on the current state.

Conversion from NFA to DFA



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

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

Now we will obtain δ' transition for state q0.

1. δ'([q0], 0) = [q0]
2. δ'([q0], 1) = [q1]

The δ' transition for state q1 is obtained as:

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

The δ' transition for state q2 is obtained as:

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

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

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

= {q1, q2} ∪ {q2}

= [q1, q2]

2. δ'([q1, q2], 1) = δ(q1, 1) ∪ δ(q2, 1)

= {q1} ∪ {q1, q2}

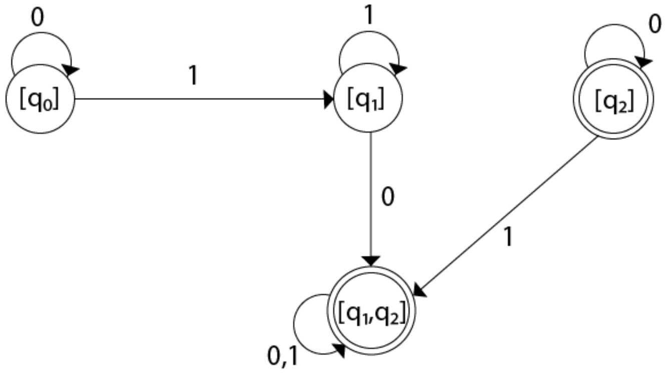
= {q1, q2}

= [q1, q2]

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

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

The Transition diagram will be:



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

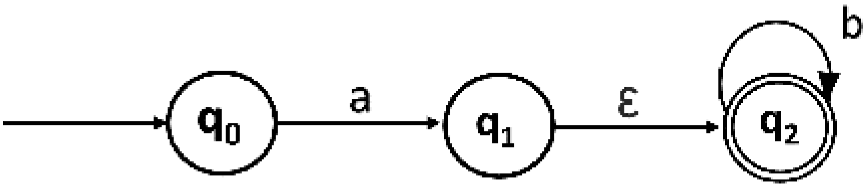
Eliminating ε Transitions

NFA with ε can be converted to NFA without ε, and this NFA without ε can be converted to DFA. To do this, we will use a method, which can remove all the ε transition from given NFA. The method will be:

1. Find out all the ε transitions from each state from Q. That will be called as ε-closure{q1} where qi ∈ Q.
2. Then δ' transitions can be obtained. The δ' transitions mean a ε-closure on δ moves.
3. Repeat Step-2 for each input symbol and each state of given NFA.
4. Using the resultant states, the transition table for equivalent NFA without ε can be built.

Example:

Convert the following NFA with ε to NFA without ε.



Solutions: We will first obtain ε-closures of q0, q1 and q2 as follows:

1. ε-closure(q0) = {q0}
2. ε-closure(q1) = {q1, q2}
3. ε-closure(q2) = {q2}

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

1. 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}

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

Now the δ' transition on q1 is obtained as:

1. δ'(q1, a) = ε-closure(δ(δ^(q1, ε),a))
   = ε-closure(δ(ε-closure(q1),a))
   = ε-closure(δ(q1, q2), a)

= ε-closure(δ(q1, a) ∪ δ(q2, a))

= ε-closure(Φ ∪ Φ)

= Φ

2. δ'(q1, b) = ε-closure(δ(δ^(q1, ε),b))

   = ε-closure(δ(ε-closure(q1),b))

   = ε-closure(δ(q1, q2), b)

   = ε-closure(δ(q1, b) ∪ δ(q2, b))

   = ε-closure(Φ ∪ q2)

   = {q2}

The δ' transition on q2 is obtained as:

1. 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

2. 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:

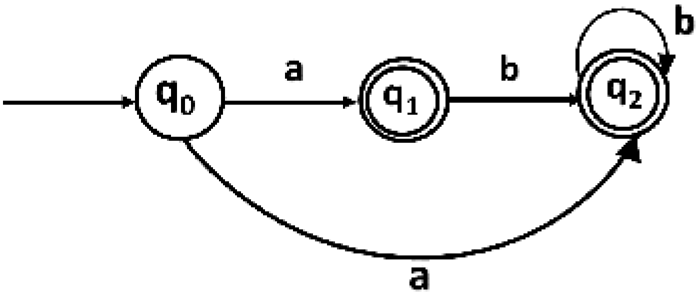
1. δ'(q0, a) = {q0, q1}
2. δ'(q0, b) = Φ
3. δ'(q1, a) = Φ
4. δ'(q1, b) = {q2}
5. δ'(q2, a) = Φ
6. δ'(q2, b) = {q2}

The transition table can be:

States	**a**	**b**
→q0	{q1, q2}	Φ
*q1	Φ	{q2}

*q2	Φ	{q2}

State q1 and q2 become the final state as ε-closure of q1 and q2 contain the final state q2. The NFA can be shown by the following transition diagram:



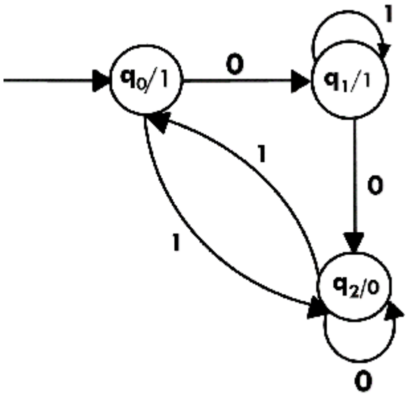
Moore Machine

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. Moore machine can be described by 6 tuples (Q, q0, ∑, O, δ, λ) where,

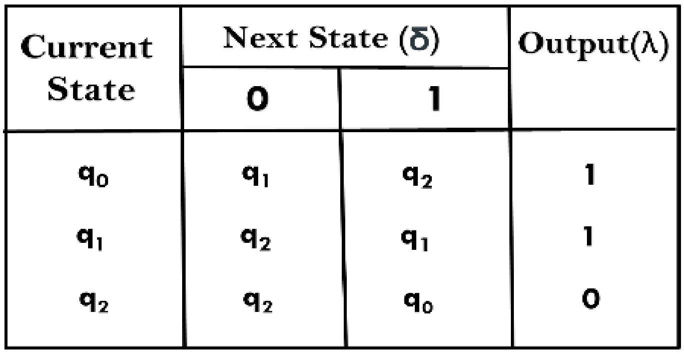
1. Q: finite set of states
2. q0: initial state of machine
3. ∑: finite set of input symbols
4. O: output alphabet
5. δ: transition function where Q × ∑ → Q
6. λ: output function where Q → O

Example 1:

The state diagram for Moore Machine is



Transition table for Moore Machine is:



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

Mealy Machine

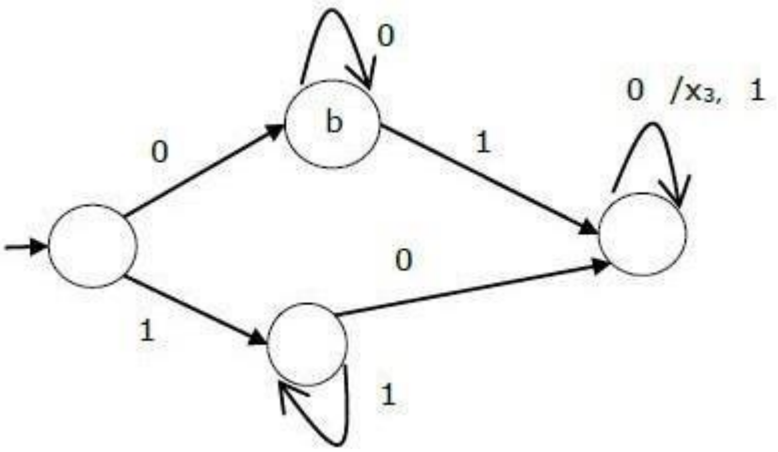
A Mealy machine is a machine in which output symbol depends upon the present input symbol and present state of the machine. In the Mealy machine, the output is represented with each input symbol for each state separated by /. The Mealy machine can be described by 6 tuples (Q, q0, ∑, O, δ, λ') where

1. Q: finite set of states
2. q0: initial state of machine
3. ∑: finite set of input alphabet
4. O: output alphabet
5. δ: transition function where Q × ∑ → Q
6. λ': output function where Q × ∑ →O

The state table of a Mealy Machine is shown below −

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

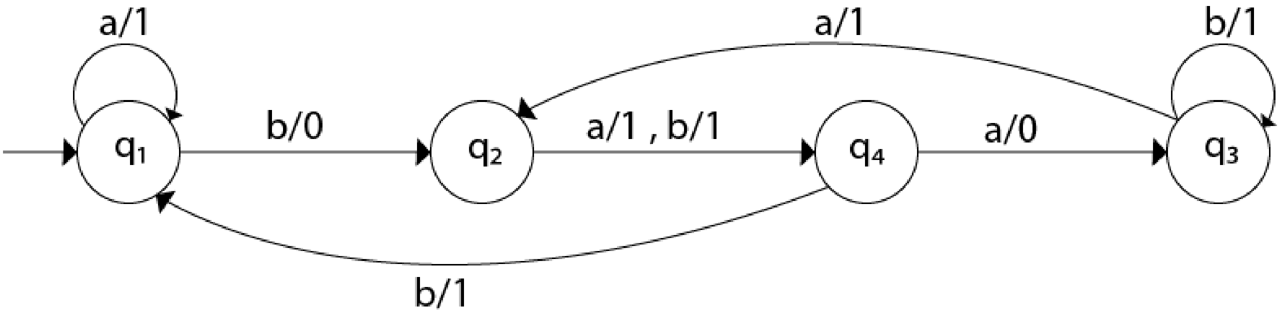
The state diagram of the above Mealy Machine is –



Conversion from Mealy machine to Moore Machine

Example 1:

Convert the following Mealy machine into equivalent Moore machine.



Solution:

Transition table for above Mealy machine is as follows:

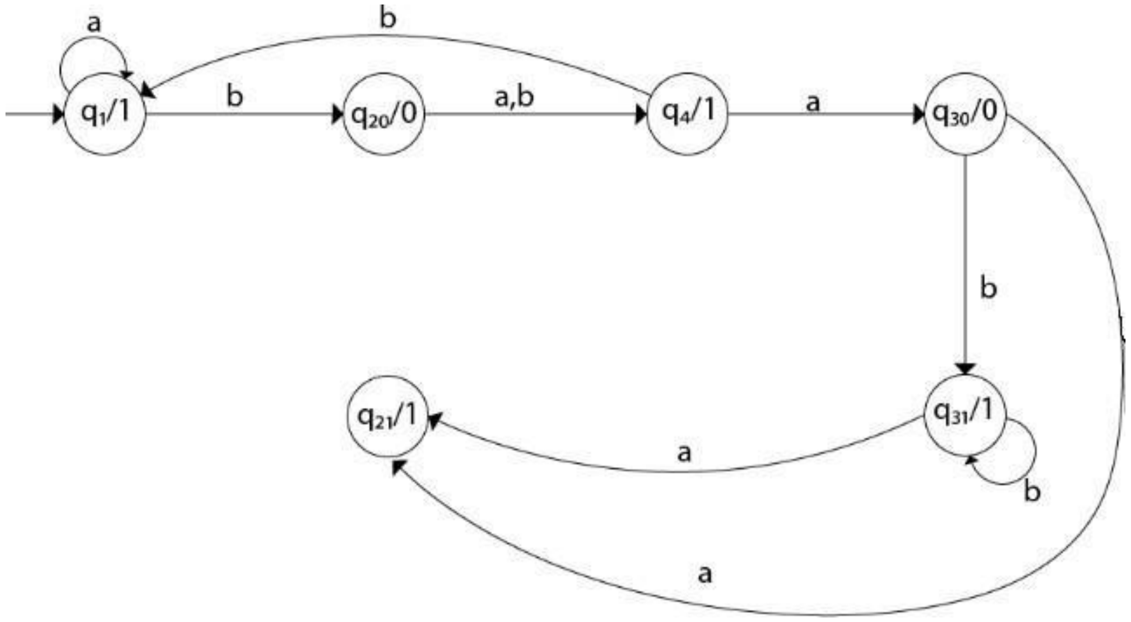
Present State	Next State			
	a		b	
	State	O/P	State	O/P
q_1	q_1	1	q_2	0
q_2	q_4	1	q_4	1
q_3	q_2	1	q_3	1
q_4	q_3	0	q_1	1

- 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 edge 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 edge 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.

Transition table for Moore machine will be:

Present State	Next State		Output
	a=0	a=1	
q_1	q_1	q_2	1
q_{20}	q_4	q_4	0
q_{21}	∅	∅	1
q_{30}	q_{21}	q_{31}	0
q_{31}	q_{21}	q_{31}	1
q_4	q_3	q_4	1

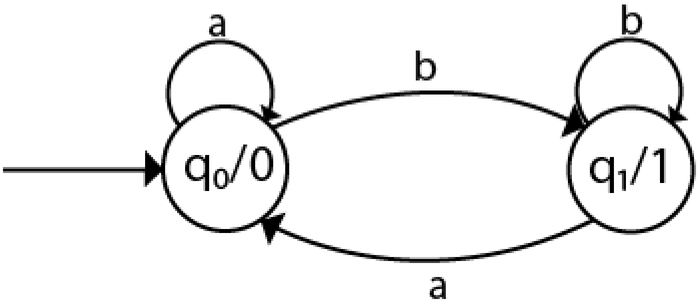
Transition diagram for Moore machine will be:



Conversion from Moore machine to Mealy Machine

Example 1:

Convert the following Moore machine into its equivalent Mealy machine.



solution:

The transition table of given Moore machine is as follows:

Q	a	b	Output(λ)
q0	q0	q1	0
q1	q0	q1	1

The equivalent Mealy machine can be obtained as follows:

1. λ' (q0, a) = λ(δ(q0, a))
   = λ(q0)
   = 0

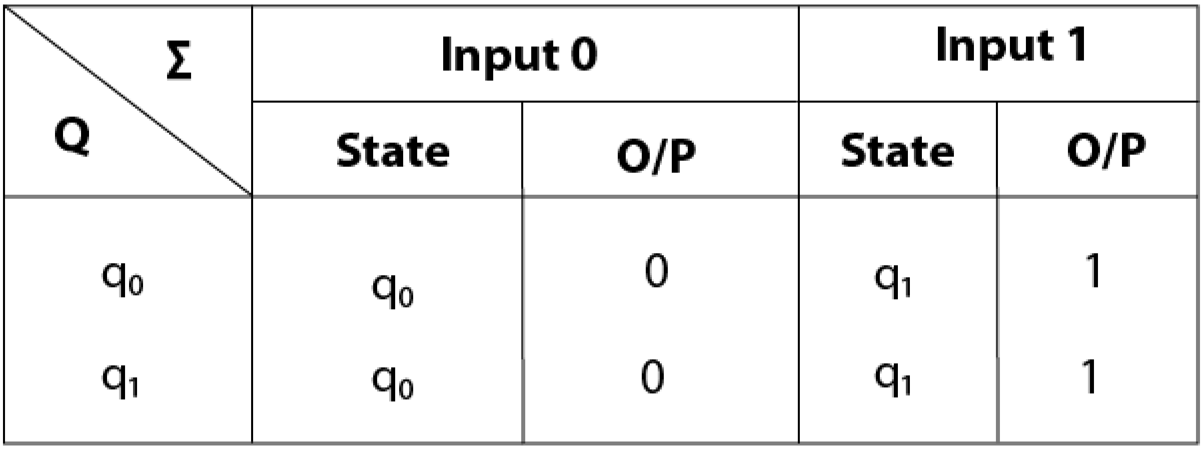
2. λ' (q0, b) = λ(δ(q0, b))
   = λ(q1)
   = 1

The λ for state q1 is as follows:

1. λ' (q1, a) = λ(δ(q1, a))
   = λ(q0)
   = 0
2. λ' (q1, b) = λ(δ(q1, b))
   = λ(q1)

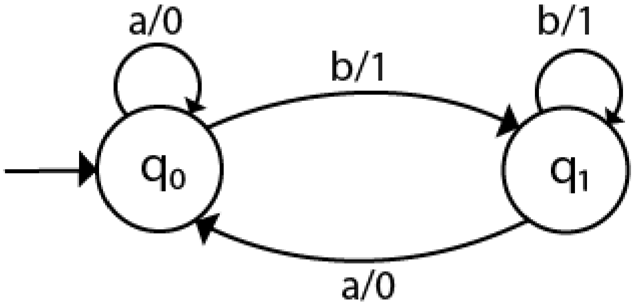
= 1

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,



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.

Operations on Regular Language

The various operations on regular language are:

Union: If L and M are two regular languages then their union L U M is also a union.

1. 1. L U M = {s | s is in L or s is in M}

Intersection: If L and M are two regular languages then their intersection is also an intersection.

1. 1. L ∩ M = {st | s is in L and t is in M}

Kleen closure: If L is a regular language then its Kleen closure L1* will also be a regular language.

1. 1. L* = Zero or more occurrence of language L.

Example 1:

Write the regular expression for the language accepting all combinations of a's, over the set ∑ = {a}

Solution:

All combinations of a's means a may be zero, single, double and so on. If a is appearing zero times, that means a null string. That is we expect the set of {ε, a, aa, aaa, ... }. So we give a regular expression for this as:

1. R = a*

That is Kleen closure of a.

Example 2:

Write the regular expression for the language accepting all combinations of a's except the null string, over the set ∑ = {a}

Solution:

The regular expression has to be built for the language

1. L = {a, aa, aaa, ... }

This set indicates that there is no null string. So we can denote regular expression as:

R = a^+

Example 3:

Write the regular expression for the language accepting all the string containing any number of a's and b's.

Solution:

The regular expression will be:

1. r.e. = (a + b)*

This will give the set as L = {ε, a, aa, b, bb, ab, ba, aba, bab,.......}, any combination of a and b.

The (a + b)* shows any combination with a and b even a null string.

Example 1:

Write the regular expression for the language accepting all the string which are starting with 1 and ending with 0, over ∑ = {0, 1}.

Solution:

In a regular expression, the first symbol should be 1, and the last symbol should be 0. The r.e. is as follows:

1. R = 1 (0+1)* 0

Example 2:

Write the regular expression for the language starting and ending with a and having any having any combination of b's in between.

Solution:

The regular expression will be:

1. R = a b* b

Example 3:

Write the regular expression for the language starting with a but not having consecutive b's.

Solution: The regular expression has to be built for the language:

1. L = {a, aba, aab, aba, aaa, abab, ... }

The regular expression for the above language is:

1. R = {a + ab}*

Example 4:

Write the regular expression for 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.

Solution: As we know, any number of a's means a* any number of b's means b*, any number of c's means c*. Since as given in problem statement, b's appear after a's and c's appear after b's. So the regular expression could be:

1. R = a* b* c*

Example 5:

Write the regular expression for the language over ∑ = {0} having even length of the string.

Solution:

The regular expression has to be built for the language:

1. L = {ε, 00, 0000, 000000, ... }

The regular expression for the above language is:

1. R = (00)*

Example 6:

Write the regular expression for the language having a string which should have atleast one 0 and alteast one 1.

Solution:

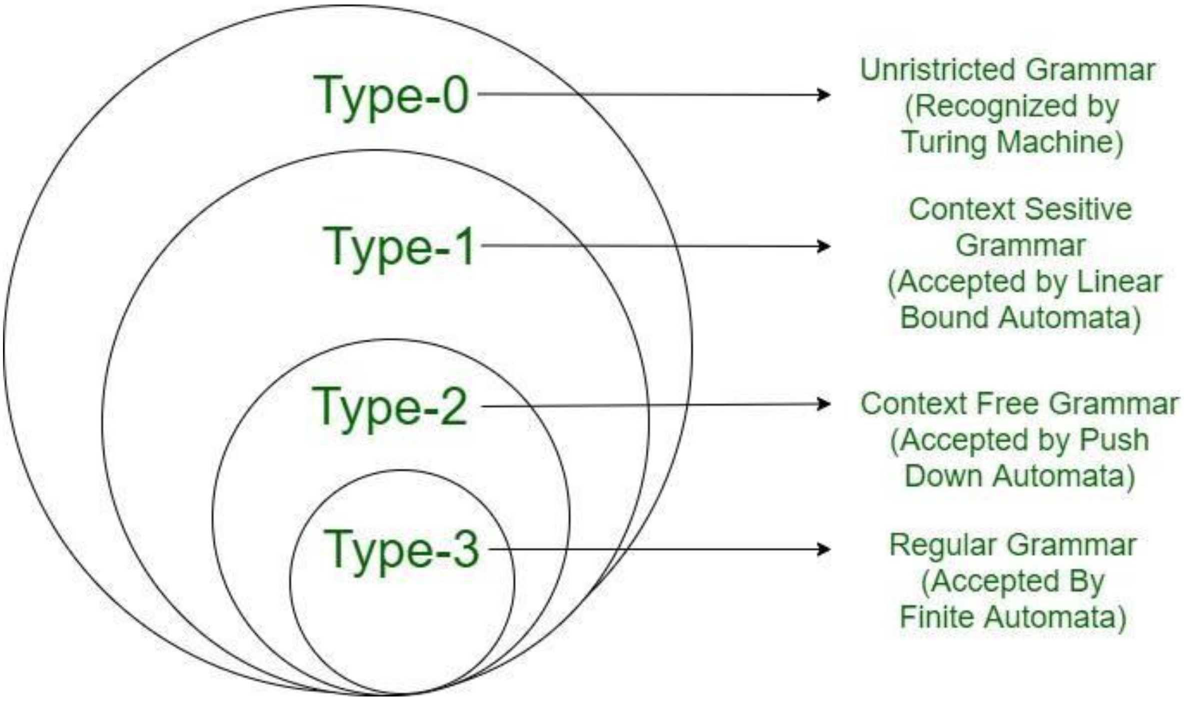
The regular expression will be:

1. R = [(0 + 1)* 0 (0 + 1)* 1 (0 + 1)*] + [(0 + 1)* 1 (0 + 1)* 0 (0 + 1)*]

- ## Chomsky Hierarchy of Languages

According to Noam Chomosky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other −

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton


Type-0
Type-1
Type-2
Type-3
Unristricted Grammar (Recognized by Turing Machine)
Context Sesitive Grammar (Accepted by Linear Bound Automata)
Context Free Grammar (Accepted by Push Down Automata)
Regular Grammar (Accepted By Finite Automata)

Type 0: Unrestricted Grammar:
In Type 0
Type-0 grammars include all formal grammars. Type 0 grammar language are recognized by turing machine. These languages are also known as the Recursively Enumerable languages

Grammar Production in the form of

$\alpha \rightarrow \beta$

where

α is (V + T)* V (V + T)*
V : Variables
T : Terminals.

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

For example,

Sab –> ba
A –> S.

Here, Variables are S, A and Terminals a, b.

Type 1: Context Sensitive Grammar)
Type-1 grammars generate the context-sensitive languages. The language generated by the grammar are recognized by the Linear Bound Automata
In Type 1
I. First of all Type 1 grammar should be Type 0.
II. Grammar Production in the form of

$\alpha \rightarrow \beta$

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

i.e count of symbol in α is less than or equal to β

For Example,
S –> AB
AB –> abc
B –> b

Type 2: Context Free Grammar:
Type-2 grammars generate the context-free languages. The language generated by the grammar is recognized by a Pushdown automata.
In Type 2,
1. First of all it should be Type 1.
2. Left hand side of production can have only one variable.
$|\alpha| = 1$.

Their is no restriction on β .

For example,
S –> AB
A –> a
B –> b

Type 3: Regular Grammar:
Type-3 grammars generate regular languages. These languages are exactly all languages that can be accepted by a finite state automaton.
Type 3 is most restricted form of grammar.
Type 3 should be in the given form only :

V –> VT / T (left-regular grammar)
or)
V –> TV /T (right-regular grammar)
for example:

S –> a

The above form is called as strictly regular grammar.

There is another form of regular grammar called extended regular grammar. In this form :

V –> VT* / T*. (extended left-regular grammar)
(or)
V –> T*V /T* (extended right-regular grammar)

for example :
S –> ab.

- **Regular Sets:** Any set that represents the value of the Regular Expression is called a **Regular Set.**

Properties of Regular Sets

Property 1. *The union of two regular set is regular.*

Proof −

Let us take two regular expressions

RE_1 = a(aa)* and RE_2 = (aa)*

So, L_1 = {a, aaa, aaaaa,.....} (Strings of odd length excluding Null)

and L_2 ={ ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

$L_1 \cup L_2$ = { ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,........}

(Strings of all possible lengths including Null)

RE ($L_1 \cup L_2$) = a* (which is a regular expression itself)

Hence, proved.

Property 2. *The intersection of two regular set is regular.*

Proof −

Let us take two regular expressions

RE_1 = a(a*) and RE_2 = (aa)*

So, L_1 = { a,aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,........} (Strings of even length including Null)

$L_1 \cap L_2$ = { aa, aaaa, aaaaaa,.......} (Strings of even length excluding Null)

RE ($L_1 \cap L_2$) = aa(aa)* which is a regular expression itself.

Hence, proved.

Property 3. *The complement of a regular set is regular.*

Proof −

Let us take a regular expression −

RE = (aa)*

So, L = {ε, aa, aaaa, aaaaaa,........} (Strings of even length including Null)

Complement of **L** is all the strings that is not in **L**.

So, L' = {a, aaa, aaaaa,......} (Strings of odd length excluding Null)

RE (L') = a(aa)* which is a regular expression itself.

Hence, proved.

Property 4. *The difference of two regular set is regular.*

Proof −

Let us take two regular expressions −

RE_1 = a (a*) and RE_2 = (aa)*

So, L_1 = {a, aa, aaa, aaaa, ... } (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,........ } (Strings of even length including Null)

$L_1 - L_2$ = {a, aaa, aaaaa, aaaaaaa,......}

(Strings of all odd lengths excluding Null)

RE ($L_1 - L_2$) = a (aa)* which is a regular expression.

Hence, proved.

Property 5. *The reversal of a regular set is regular.*

Proof −

We have to prove $\mathbf{L^R}$ is also regular if **L** is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

L^R = {10, 01, 11, 01}

RE (L^R) = 01 + 10 + 11 + 10 which is regular

Hence, proved.

Property 6. *The closure of a regular set is regular.*

Proof −

If L = {a, aaa, aaaaa,........ } (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)*

L* = {a, aa, aaa, aaaa , aaaaa,..................... } (Strings of all lengths excluding Null)

RE (L*) = a (a)*

Hence, proved.

Property 7. *The concatenation of two regular sets is regular.*

Proof −

Let RE_1 = (0+1)*0 and RE_2 = 01(0+1)*

Here, L_1 = {0, 00, 10, 000, 010,....... } (Set of strings ending in 0)

and L_2 = {01, 010,011,......} (Set of strings beginning with 01)

Then, L_1 L_2 = {001,0010,0011,0001,00010,00011,1001,10010,.................}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*

Hence, proved.

- **Pumping Lemma for Regular Sets:**

Let L be a regular language. Then there exists a constant **'c'** such that for every string **w** in **L** −

$|w| \geq c$

We can break **w** into three strings, **w = xyz**, such that −

- $|y| > 0$
- $|xy| \leq c$
- For all $k \geq 0$, the string xy^kz is also in L.

Applications of Pumping Lemma

Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

- If **L** is regular, it satisfies Pumping Lemma.
- If **L** does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular

- At first, we have to assume that **L** is regular.
- So, the pumping lemma should hold for **L**.
- Use the pumping lemma to obtain a contradiction −
  - Select **w** such that **$|w| \geq c$**
  - Select **y** such that **$|y| \geq 1$**
  - Select **x** such that **$|xy| \leq c$**
  - Assign the remaining string to **z.**
  - Select **k** such that the resulting string is not in **L.**

Hence L is not regular.

Problem

Prove that **$L = \{a^ib^i \mid i \geq 0\}$** is not regular.

Solution −

- At first, we assume that **L** is regular and n is the number of states.
- Let w = a^nb^n. Thus $|w| = 2n \geq n$.
- By pumping lemma, let w = xyz, where $|xy| \leq n$.
- Let $x = a^p$, $y = a^q$, and $z = a^rb^n$, where $p + q + r = n$, $p \neq 0$, $q \neq 0$, $r \neq 0$. Thus $|y| \neq 0$.
- Let k = 2. Then $xy^2z = a^pa^{2q}a^rb^n$.
- Number of as = $(p + 2q + r) = (p + q + r) + q = n + q$
- Hence, $xy^2z = a^{n+q}\, b^n$. Since $q \neq 0$, xy^2z is not of the form a^nb^n.
- Thus, xy^2z is not in L. Hence L is not regular.

- ## Minimization of Finite Automata

Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM(finite state machine) with redundant states after minimizing the FSM.

We have to follow the various steps to minimize the DFA. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. 1. δ (q, a) = p
2. 2. δ (r, a) = p

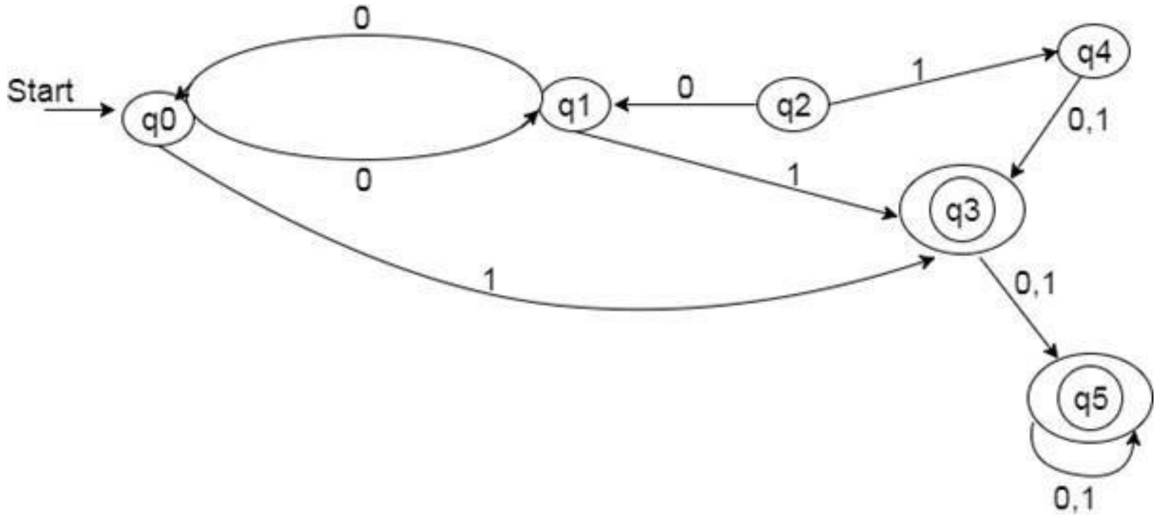
That means, find the two states which have the same value of a and b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Example:



Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states.

State	0
→q0	q1
q1	q0
*q3	q5
*q5	q5

Step 3: Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

State	0
q0	q1
q1	q0

2. Another set contains those rows, which starts from final states.

State	0
q3	q5
q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

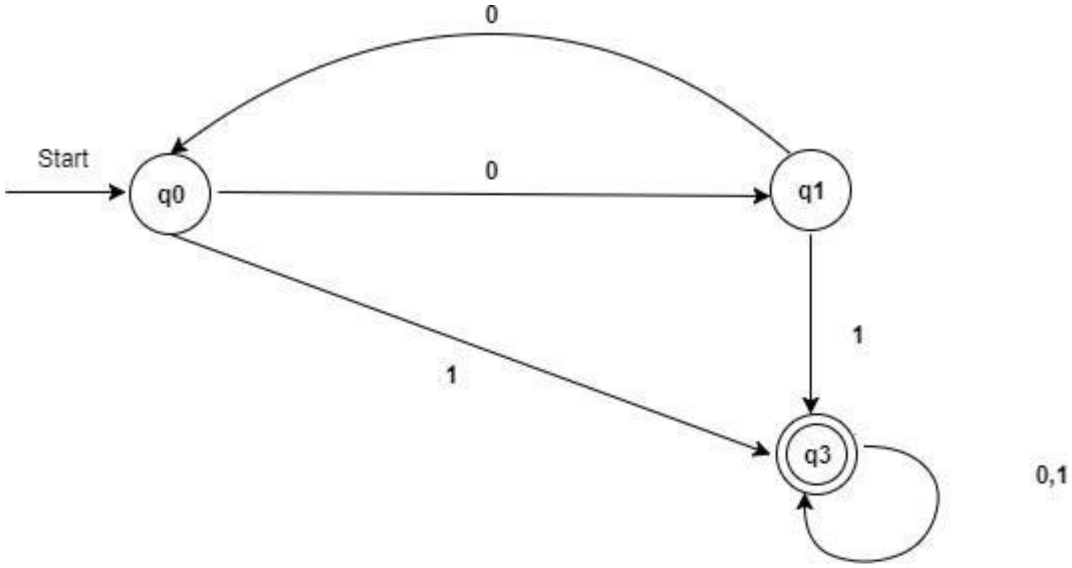
Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0
q3	q3

Step 6: Now combine set 1 and set 2 as:

State	0
→q0	q1
q1	q0
*q3	q3

Now it is the transition table of minimized DFA.



- **Decision Algorithm for Regular Sets:**

  Approximately all the properties are decidable in case of finite automaton.

(i) Emptiness
(ii) Non-emptiness
(iii) Finiteness
(iv) Infiniteness
(v) Membership
(vi) Equality
These are explained as following below.

(i) Emptiness and Non-emptiness:

- **Step-1:** select the state that cannot be reached from the initial states & delete them (remove unreachable states).
- **Step 2:** if the resulting machine contains at least one final states, so then the finite automata accepts the non-empty language.
- **Step 3:** if the resulting machine is free from final state, then finite automata accepts empty language.

(ii) Finiteness and Infiniteness:

- **Step-1:** select the state that cannot be reached from the initial state & delete them (remove unreachable states).
- **Step-2:** select the state from which we cannot reach the final state & delete them (remove dead states).
- **Step-3:** if the resulting machine contains loops or cycles then the finite automata accepts infinite language.
- **Step-4:** if the resulting machine do not contain loops or cycles then the finite automata accepts infinite language.

(iii) Membership:

Membership is a property to verify an arbitrary string is accepted by a finite automaton or not i.e. it is a member of the language or not.
Let M is a finite automata that accepts some strings over an alphabet, and let ‘w’ be any string defined over the alphabet, if there exist a transition path in M, which starts at initial state & ends in anyone of the final state, then string ‘w’ is a member of M, otherwise ‘w’ is not a member of M.

(iv) Equality:

Two finite state automata M1 & M2 is said to be equal if and only if, they accept the same language. Minimise the finite state automata and the minimal DFA will be unique.

UNIT-II

Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

Where,

G describes the grammar

T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols

P describes a set of production rules

S is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

Capabilities of CFG

There are the various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficient parser can be constructed automatically.
- Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

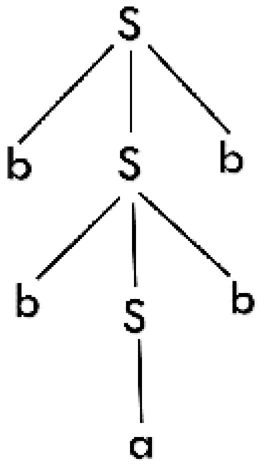
We have two options to decide which non-terminal to be replaced with production rule.

Draw a derivation tree for the string "bab" from the CFG given by

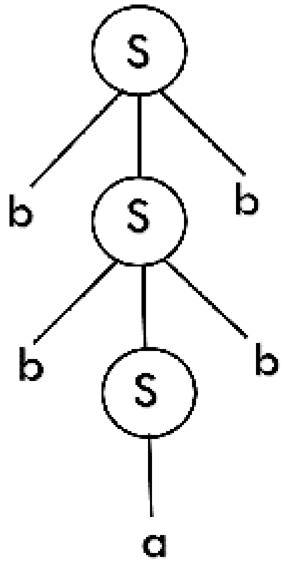
1. S → bSb | a | b

Solution:

Now, the derivation tree for the string "bbabb" is as follows:



The above tree is a derivation tree drawn for deriving a string bbabb. By simply reading the leaf nodes, we can obtain the desired string. The same tree can also be denoted by,



Left-most Derivation

In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Construct a derivation tree for the string aabbabba for the CFG given by,

1. S → aB | bA
2. A → a | aS | bAA
3. B → b | bS | aBB

Solution:

To draw a tree, we will first try to obtain derivation for the string aabbabba

S

aB

$a\ \boxed{aBB}$

$aa\ \boxed{bS}\ B$

$aab\ \boxed{bA}\ B$

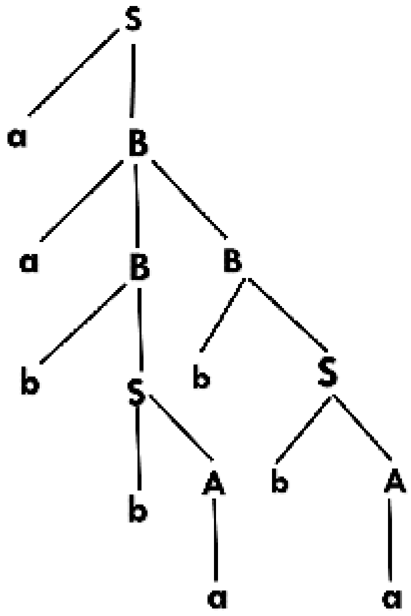
$aabb\ \boxed{a}\ B$

$aabba\ \boxed{bS}$

$aabbab\ \boxed{bA}$

$aabbabb\ \boxed{a}$

Now, the derivation tree is as follows:



Right-most Derivation

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

Derive the string "aabbabba" for leftmost derivation and rightmost derivation using a CFG given by,

1. S → aB | bA
2. A → a | aS | bAA
3. B → b | bS | aBB

Solution:

Leftmost derivation:

1. S
2. aB S → aB
3. aaBB B → aBB
4. aabB B → b
5. aabbS B → bS
6. aabbaB S → aB
7. aabbabS B → bS
8. aabbabbA S → bA
9. aabbabba A → a

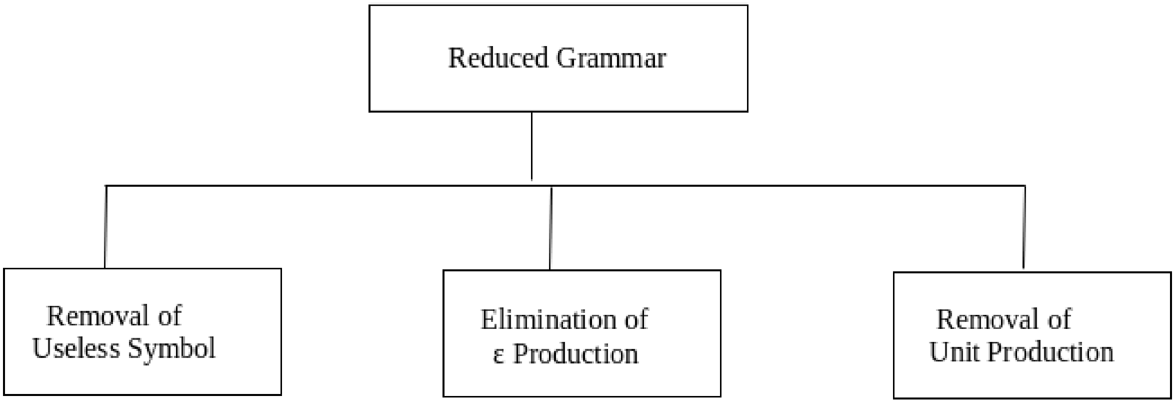
Rightmost derivation:

1. S
2. aB S → aB
3. aaBB B → aBB
4. aaBbS B → bS
5. aaBbbA S → bA
6. aaBbba A → a
7. aabSbba B → bS
8. aabbAbba S → bA
9. aabbabba A → a

Simplification of CFG

As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal). Having extra symbols, unnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below:

1. Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as X → Y where X and Y are non-terminal.
3. If ε is not in the language L then there need not to be the production X → ε.


Reduced Grammar
Removal of Useless Symbol
Elimination of ε Production
Removal of Unit Production

Removal of Useless Symbols

A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

1. T → aaB | abA | aaT
2. A → aA
3. B → ab | b
4. C → ad

In the above example, the variable 'C' will never occur in the derivation of any string, so the production C → ad is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production A → aA is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production A → aA, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

Elimination of ε Production

The productions of type S → ε are called ε productions. These type of productions can only be removed from those grammars that do not generate ε.

Step 1: First find out all nullable non-terminal variable which derives ε.

Step 2: For each production A → a, construct all production A → x, where x is obtained from a by removing one or more non-terminal from step 1.

Step 3: Now combine the result of step 2 with the original production and remove ε productions.

Example:

Remove the production from the following CFG by preserving the meaning of it.

1. S → XYX
2. X → 0X | ε
3. Y → 1Y | ε

Solution:

Now, while removing ε production, we are deleting the rule X → ε and Y → ε. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared.

Let us take

1. S → XYX

If the first X at right-hand side is ε. Then

1. S → YX

Similarly if the last X in R.H.S. = ε. Then

1. S → XY

If Y = ε then

1. S → XX

If Y and X are ε then,

1. S → X

If both X are replaced by ε

1. S → Y

Now,

1. S → XY | YX | XX | X | Y

Now let us consider

1. X → 0X

If we place ε at right-hand side for X then,

1. X → 0
2. X → 0X | 0

Similarly Y → 1Y | 1

Collectively we can rewrite the CFG with removed ε production as

1. S → XY | YX | XX | X | Y
2. X → 0X | 0
3. Y → 1Y | 1

Removing Unit Productions

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

Step 1: To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

Step 2: Now delete X → Y from the grammar.

Step 3: Repeat step 1 and step 2 until all unit productions are removed.

For example:

1. S → 0A | 1B | C
2. A → 0S | 00

3. B → 1 | A
4. C → 01

Solution:

S → C is a unit production. But while removing S → C we have to consider what C gives. So, we can add a rule to S.

1. S → 0A | 1B | 01

Similarly, B → A is also a unit production so we can modify it as

1. B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

1. S → 0A | 1B | 01
2. A → 0S | 00
3. B → 1 | 0S | 00
4. C → 01

Chomsky's Normal Form (CNF)

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

For example:

1. G1 = {S → AB, S → c, A → a, B → b}
2. G2 = {S → aA, A → a, B → c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S → aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

1. S1 → S

Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

1. S → RA
2. R → a

Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as:

1. S → RS
2. R → AS

Example:

Convert the given CFG to CNF. Consider the given grammar G1:

1. S → a | aA | B
2. A → aBB | ε
3. B → Aa | b

Solution:

Step 1: We will create a new production S1 → S, as the start symbol S appears on the RHS. The grammar will be:

1. S1 → S
2. S → a | aA | B
3. A → aBB | ε
4. B → Aa | b

Step 2: As grammar G1 contains A → ε null production, its removal from the grammar yields:

1. S1 → S
2. S → a | aA | B
3. A → aBB
4. B → Aa | b | a

Now, as grammar G1 contains Unit production S → B, its removal yield:

1. S1 → S
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Also remove the unit production S1 → S, its removal from the grammar yields:

1. S0 → a | aA | Aa | b
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Step 3: In the production rule S0 → aA | Aa, S → aA | Aa, A → aBB and B → Aa, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → XBB
4. B → AX | b | a
5. X → a

Step 4: In the production rule A → XBB, RHS has more than two symbols, removing it from grammar yield:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → RB
4. B → AX | b | a
5. X → a
6. R → XB

Hence, for the given grammar, this is the required CNF.

Greibach Normal Form (GNF)

GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF(Greibach normal form) if all the production rules satisfy one of the following conditions:

- A start symbol generating ε. For example, S → ε.
- A non-terminal generating a terminal. For example, A → a.

- A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.

For example:

1. G1 = {S → aAB | aB, A → aA| a, B → bB | b}
2. G2 = {S → aAB | aB, A → aA | ε, B → bB | ε}

The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF. However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as A → ε and B → ε contains ε(only start symbol can generate ε). So the grammar G2 is not in GNF.

Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

Step 2: If the grammar exists left recursion, eliminate it.

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

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

If any production rule in the grammar is not in GNF form, convert it.

Example:

1. S → XB | AA
2. A → a | SA
3. B → b
4. X → a

Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.

The production rule A → SA is not in GNF, so we substitute S → XB | AA in the production rule A → SA as:

1. S → XB | AA
2. A → a | XBA | AAA

3. B → b
4. X → a

The production rule S → XB and B → XBA is not in GNF, so we substitute X → a in the production rule S → XB and B → XBA as:

1. S → aB | AA
2. A → a | aBA | AAA
3. B → b
4. X → a

Now we will remove left recursion (A → AAA), we get:

1. S → aB | AA
2. A → aC | aBAC
3. C → AAC | ε
4. B → b
5. X → a

Now we will remove null production C → ε, we get:

1. S → aB | AA
2. A → aC | aBAC | a | aBA
3. C → AAC | AA
4. B → b
5. X → a

The production rule S → AA is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule S → AA as:

1. S → aB | aCA | aBACA | aA | aBAA
2. A → aC | aBAC | a | aBA
3. C → AAC
4. C → aCA | aBACA | aA | aBAA
5. B → b
6. X → a

The production rule C → AAC is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule C → AAC as:

1. S → aB | aCA | aBACA | aA | aBAA
2. A → aC | aBAC | a | aBA
3. C → aCAC | aBACAC | aAC | aBAAC
4. C → aCA | aBACA | aA | aBAA

5. B → b
6. X → a

Hence, this is the GNF form for the grammar G.

Pumping Lemma for Context-free Languages (CFL)

Pumping Lemma for CFL states that for any Context Free Language L, it is possible to find two substrings that can be 'pumped' any number of times and still be in the same language. For any language L, we break its strings into five parts and pump second and fourth substring.

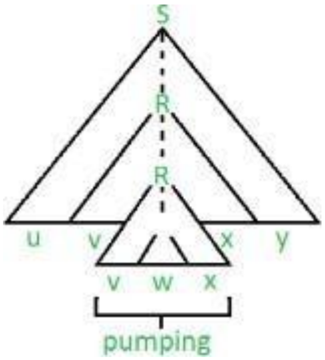
Pumping Lemma, here also, is used as a tool to prove that a language is not CFL. Because, if any one string does not satisfy its conditions, then the language is not CFL.

Thus, if L is a CFL, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists $u, v, w, x, y \in \Sigma*$, such that $x = uvwxy$, and

(1) $|vwx| \leq n$

(2) $|vx| \geq 1$

(3) for all $i \geq 0$: $uv^iwx^iy \in L$



For above example, 0^n1^n is CFL, as any string can be the result of pumping at two places, one for 0 and other for 1.

Let us prove, $L_{012} = \{0^n1^n2^n \mid n \geq 0\}$ is not Context-free.

Let us assume that L is Context-free, then by Pumping Lemma, the above given rules follow.

Now, let $x \in L$ and $|x| \geq n$. So, by Pumping Lemma, there exists u, v, w, x, y such that (1) – (3) hold.

We show that for all u, v, w, x, y (1) – (3) do not hold.

If (1) and (2) hold then $x = 0^n1^n2^n = uvwxy$ with $|vwx| \leq n$ and $|vx| \geq 1$.

(1) tells us that vwx does not contain both 0 and 2. Thus, either vwx has no 0's, or vwx has no 2's. Thus, we have two cases to consider.

Suppose vwx has no 0's. By (2), vx contains a 1 or a 2. Thus uwy has 'n' 0's and uwy either has less than 'n' 1's or has less than 'n' 2's.

But (3) tells us that $uwy = uv^0wx^0y \in L$.

So, uwy has an equal number of 0's, 1's and 2's gives us a contradiction. The case where vwx has no 2's is similar and also gives us a contradiction. Thus L is not context-free.

Closure Properties of CFL:

Context-free languages are **closed** under −

- Union
- Concatenation
- Kleene Star operation

Union

Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n , n > 0\}$. Corresponding grammar G_1 will have P: S1 → aAb|ab

Let $L_2 = \{ c^m d^m , m \geq 0\}$. Corresponding grammar G_2 will have P: S2 → cBb| ε

Union of L_1 and L_2, $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

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

Concatenation

If L_1 and L_2 are context free languages, then L_1L_2 is also context free.

Example

Union of the languages L_1 and L_2, $L = L_1L_2 = \{ a^n b^n c^m d^m \}$

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 $L = \{ a^n b^n , n \geq 0\}$. Corresponding grammar G will have P: S → aAb| ε

Kleene Star $L_1 = \{ a^n b^n \}^*$

The corresponding grammar G_1 will have additional productions S1 → SS_1 | 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.

Push down Automata

Basic Structure of PDA

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is −

"Finite state machine" + "a stack"

A pushdown automaton has three components −

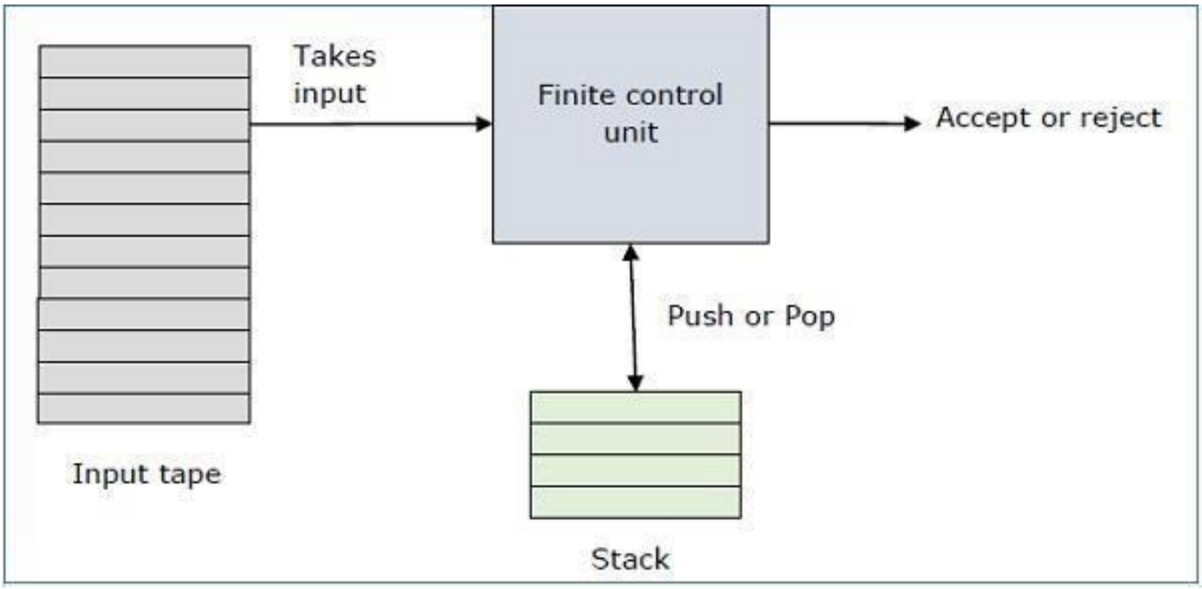
- an input tape,
- a control unit, and
- a stack with infinite size.

The stack head scans the top symbol of the stack.

A stack does two operations −

- **Push** − a new symbol is added at the top.
- **Pop** − the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.



PDA can be formally described as a 7-tuple (Q, ∑, S, δ, q_0, I, F) −

- **Q** is the finite number of states
- ∑ is input alphabet
- **S** is stack symbols
- **δ** is the transition function: Q × (∑ U {ε}) × S × Q × S*
- **q_0** is the initial state (q_0 ∈ Q)
- **I** is the initial stack top symbol (I ∈ S)
- **F** is a set of accepting states (F ∈ Q)

Instantaneous Description (ID)

ID is an informal notation of how a PDA computes an input string and make a decision that string is accepted or rejected

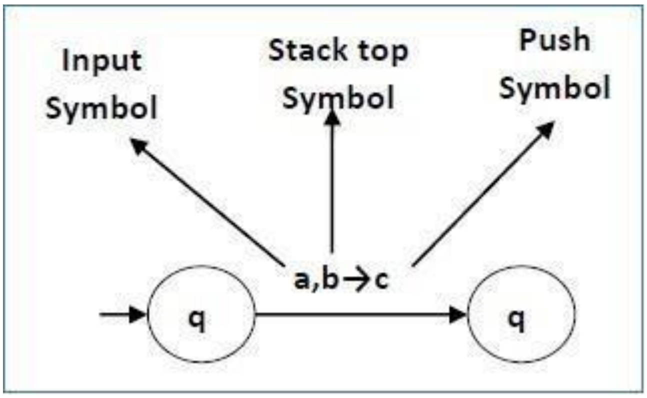
An instantaneous description is a triple (q, w, α) where:

q describes the current state.

w describes the remaining input.

α describes the stack contents, top at the left.

The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as a,b → c −



This means at state **q_1**, if we encounter an input string **‘a’** and top symbol of the stack is **‘b’**, then we pop **‘b’**, push **‘c’** on top of the stack and move to state **q_2**.

There are two different ways to define PDA acceptability.

Final State Acceptability

In final state acceptability, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

For a PDA (Q, ∑, S, δ, q_0, I, F), the language accepted by the set of final states F is −

L(PDA) = $\{w \mid (q_0, w, I) \vdash^* (q, \varepsilon, x), q \in F\}$

for any input stack string **x**.

Empty Stack Acceptability

Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

For a PDA (Q, ∑, S, δ, q_0, I, F), the language accepted by the empty stack is −

L(PDA) = $\{w \mid (q_0, w, I) \vdash^* (q, \varepsilon, \varepsilon), q \in Q\}$

Push-Down Automata and Context free Languages

If a grammar **G** is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar **G**. A parser can be built for the grammar **G**.

Also, if **P** is a pushdown automaton, an equivalent context-free grammar G can be constructed where

L(G) = L(P)

Algorithm to find PDA corresponding to a given CFG

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F)

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** where a is terminal and **A, X** are combination of terminal and non-terminals, make a transition **δ (q, a, A)**.

Example 1:

Convert the following grammar to a PDA that accepts the same language.

1. S → 0S1 | A
2. A → 1A0 | S | ε

Solution:

The CFG can be first simplified by eliminating unit productions:

1. S → 0S1 | 1S0 | ε

Now we will convert this CFG to GNF:

1. S → 0SX | 1SY | ε
2. X → 1
3. Y → 0

The PDA can be:

R1: δ(q, ε, S) = {(q, 0SX) | (q, 1SY) | (q, ε)}
R2: δ(q, ε, X) = {(q, 1)}
R3: δ(q, ε, Y) = {(q, 0)}
R4: δ(q, 0, 0) = {(q, ε)}
R5: δ(q, 1, 1) = {(q, ε)}

Algorithm to find CFG corresponding to a given PDA

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F) such that the non- terminals of the grammar G will be {X_{wx} | w,x ∈ Q} and the start state will be $A_{q0,F}$.

Step 1 − For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m) contains (x, ε), add the production rule X_{wx} → a X_{yz}b in grammar G.

Step 2 − For every w, x, y, z ∈ Q, add the production rule X_{wx} → $X_{wy}X_{yx}$ in grammar G.

Step 3 − For w ∈ Q, add the production rule X_{ww} → ε in grammar G.

Parsing and Push-Down Automata

Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. Compiler is used to check whether or not a string is syntactically correct. A parser takes the inputs and builds a parse tree.

A parser can be of two types −

- **Top-Down Parser** − Top-down parsing starts from the top with the start-symbol and derives a string using a parse tree.
- **Bottom-Up Parser** − Bottom-up parsing starts from the bottom with the string and comes to the start symbol using a parse tree.

Design of Top-Down Parser

For top-down parsing, a PDA has the following four types of transitions −

- Pop the non-terminal on the left hand side of the production at the top of the stack and push its right-hand side string.
- If the top symbol of the stack matches with the input symbol being read, pop it.
- Push the start symbol 'S' into the stack.
- If the input string is fully read and the stack is empty, go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is $(Q, \sum, S, \delta, q_0, I, F)$, then the top-down parsing is −

(x+y*z, I) ⊢(x +y*z, SI) ⊢ (x+y*z, S+XI) ⊢(x+y*z, X+XI)

⊢(x+y*z, Y+X I) ⊢(x+y*z, x+XI) ⊢(+y*z, +XI) ⊢ (y*z, XI)

⊢(y*z, X*YI) ⊢(y*z, y*YI) ⊢(*z,*YI) ⊢(z, YI) ⊢(z, zI) ⊢(ε, I)

Design of a Bottom-Up Parser

For bottom-up parsing, a PDA has the following four types of transitions −

- Push the current input symbol into the stack.

- Replace the right-hand side of a production at the top of the stack with its left-hand side.
- If the top of the stack element matches with the current input symbol, pop it.
- If the input string is fully read and only if the start symbol ‘S’ remains in the stack, pop it and go to the final state ‘F’.

Example

- Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −
- P: S → S+X | X, X → X*Y | Y, Y → (S) | id
- ***Solution***
- If the PDA is (Q, ∑, S, δ, q_0, I, F), then the bottom-up parsing is −
- (x+y*z, I) ⊢ (+y*z, xI) ⊢ (+y*z, YI) ⊢ (+y*z, XI) ⊢ (+y*z, SI)
- ⊢(y*z, +SI) ⊢ (*z, y+SI) ⊢ (*z, Y+SI) ⊢ (*z, X+SI) ⊢ (z, *X+SI)
- ⊢ (ε, z*X+SI) ⊢ (ε, Y*X+SI) ⊢ (ε, X+SI) ⊢ (ε, SI)

****

UNIT-III

Turing Machines

Turing Machine was invented by Alan Turing in 1936 and it is used to accept Recursive Enumerable Languages (generated by Type-0 Grammar)

Definition

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

There are various features of the Turing machine:

1. It has an external memory which remembers arbitrary long sequence of input.
2. It has unlimited memory capability.
3. The model has a facility by which the input at left or right on the tape can be read easily.
4. The machine can produce a certain output based on its input. Sometimes it may be required that the same input has to be used to generate the output. So in this machine, the distinction between input and output has been removed. Thus a common set of alphabets can be used for the Turing machine.

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

- **Q** is a finite set of states
- **T** is the tape alphabet (symbols which can be written on Tape)
- **B** is blank symbol (every cell is filled with B except input alphabet initially)
- ∑ is the input alphabet (symbols which are part of input alphabet)
- **δ** is a transition function which maps Q × T → Q × T × {L,R}. Depending on its present state and present tape alphabet (pointed by

head pointer), it will move to new state, change the tape symbol (may or may not) and move head pointer to either left or right.

- **q0** is the initial state
- **F** is the set of final states. If any state of F is reached, input string is accepted.

Example of Turing machine

Turing machine M = (Q, X, ∑, δ, q_0, B, F) with

- Q = {q_0, q_1, q_2, q_f}
- X = {a, b}
- ∑ = {1}
- q_0 = {q_0}
- B = blank symbol
- F = {q_f }

δ is given by −

Tape alphabet symbol	Present State 'q_0'	Present State 'q_1'	Present State 'q_2'
a	$1Rq_1$	$1Lq_0$	$1Lq_f$
b	$1Lq_2$	$1Rq_1$	$1Rq_f$

Here the transition $1Rq_1$ implies that the write symbol is 1, the tape moves right, and the next state is q_1. Similarly, the transition $1Lq_2$ implies that the write symbol is 1, the tape moves left, and the next state is q_2.

Time and Space Complexity of a Turing Machine

For a Turing machine, the time complexity refers to the measure of the number of times the tape moves when the machine is initialized for some input symbols and the space complexity is the number of cells of the tape written.

Time complexity all reasonable functions −

T(n) = O(n log n)

TM's space complexity −

S(n) = O(n)

A TM accepts a language if it enters into a final state for any input string w. A language is recursively enumerable (generated by Type-0 grammar) if it is accepted by a Turing machine.

A TM decides a language if it accepts it and enters into a rejecting state for any input not in the language. A language is recursive if it is decided by a Turing machine.

There may be some cases where a TM does not stop. Such TM accepts the language, but it does not decide it.

Designing a Turing Machine

The basic guidelines of designing a Turing machine have been explained below with the help of a couple of examples.

Example 1

Design a TM to recognize all strings consisting of an odd number of α's.

Solution

The Turing machine **M** can be constructed by the following moves −

- Let **q_1** be the initial state.
- If **M** is in **q_1**; on scanning α, it enters the state **q_2** and writes **B** (blank).
- If **M** is in **q_2**; on scanning α, it enters the state **q_1** and writes **B** (blank).
- From the above moves, we can see that **M** enters the state **q_1** if it scans an even number of α's, and it enters the state **q_2** if it scans an odd number of α's. Hence **q_2** is the only accepting state.

Hence,

$M = \{\{q_1, q_2\}, \{1\}, \{1, B\}, \delta, q_1, B, \{q_2\}\}$

where δ is given by −

Tape alphabet symbol	Present State 'q_1'	Present State 'q_2'
α	BRq_2	BRq_1

Example 2

Design a Turing Machine that reads a string representing a binary number and erases all leading 0's in the string. However, if the string comprises of only 0's, it keeps one 0.

Solution

Let us assume that the input string is terminated by a blank symbol, B, at each end of the string.

The Turing Machine, **M**, can be constructed by the following moves −

- Let $\mathbf{q_0}$ be the initial state.
- If **M** is in $\mathbf{q_0}$, on reading 0, it moves right, enters the state $\mathbf{q_1}$ and erases 0. On reading 1, it enters the state $\mathbf{q_2}$ and moves right.
- If **M** is in $\mathbf{q_1}$, on reading 0, it moves right and erases 0, i.e., it replaces 0's by B's. On reaching the leftmost 1, it enters $\mathbf{q_2}$ and moves right. If it reaches B, i.e., the string comprises of only 0's, it moves left and enters the state $\mathbf{q_3}$.
- If **M** is in $\mathbf{q_2}$, on reading either 0 or 1, it moves right. On reaching B, it moves left and enters the state $\mathbf{q_4}$. This validates that the string comprises only of 0's and 1's.
- If **M** is in $\mathbf{q_3}$, it replaces B by 0, moves left and reaches the final state $\mathbf{q_f}$.
- If **M** is in $\mathbf{q_4}$, on reading either 0 or 1, it moves left. On reaching the beginning of the string, i.e., when it reads B, it reaches the final state $\mathbf{q_f}$.

Hence,

$M = \{\{q_0, q_1, q_2, q_3, q_4, q_f\}, \{0,1, B\}, \{1, B\}, \delta, q_0, B, \{q_f\}\}$

where δ is given by −

Tape alphabet symbol	Present State 'q_0'	Present State 'q_1'	Present State 'q_2'	Present State 'q_3'	Present State 'q_4'
0	BRq_1	BRq_1	ORq_2	-	OLq_4
1	$1Rq_2$	$1Rq_2$	$1Rq_2$	-	$1Lq_4$
B	BRq_1	BLq_3	BLq_4	OLq_f	BRq_f

Construction of Turing Machines

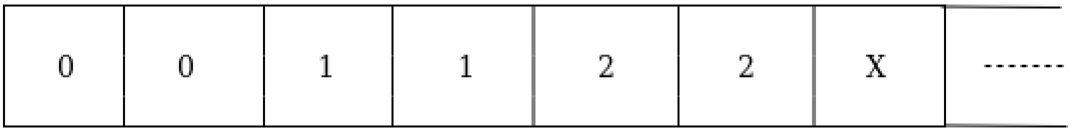
Example 1:

Construct a TM for the language L = {$0^n1^n2^n$} where n≥1

Solution:

L = {$0^n1^n2^n$ | n≥1} represents language where we use only 3 character, i.e., 0, 1 and 2. In this, some number of 0's followed by an equal number of 1's and then followed by an equal number of 2's. Any type of string which falls in this category will be accepted by this language.

The simulation for 001122 can be shown as below:



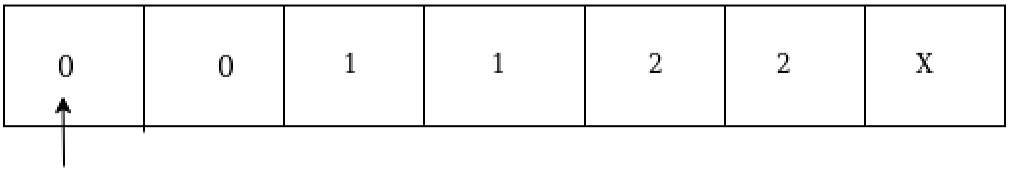
6.8M

98

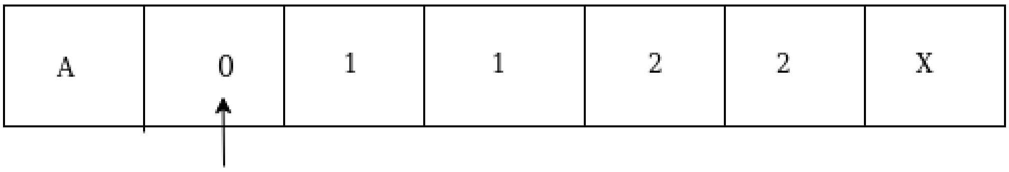
SQL CREATE TABLE

Next

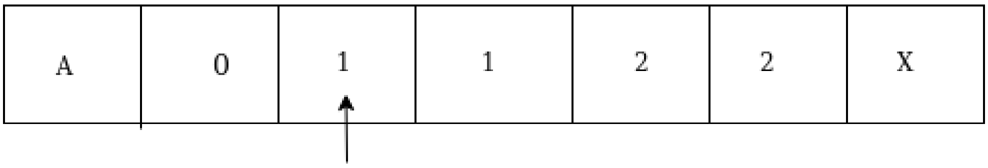
Now, we will see how this Turing machine will work for 001122. Initially, state is q0 and head points to 0 as:



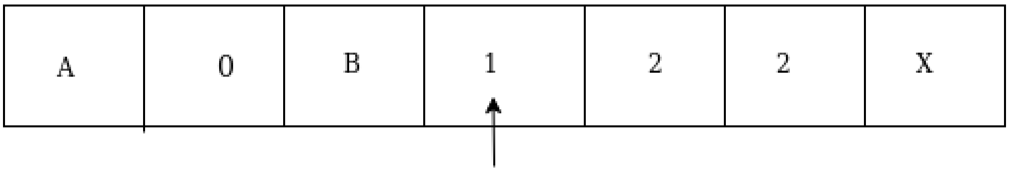
The move will be δ(q0, 0) = δ(q1, A, R) which means it will go to state q1, replaced 0 by A and head will move to the right as:



The move will be δ(q1, 0) = δ(q1, 0, R) which means it will not change any symbol, remain in the same state and move to the right as:

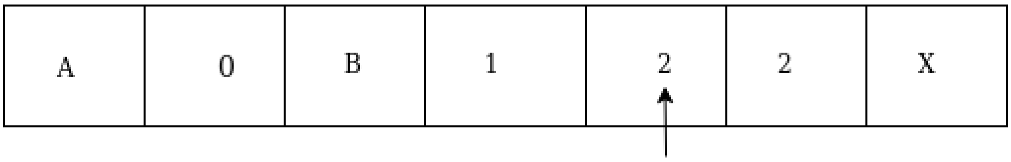


The move will be δ(q1, 1) = δ(q2, B, R) which means it will go to state q2, replaced 1 by B and head will move to right as:



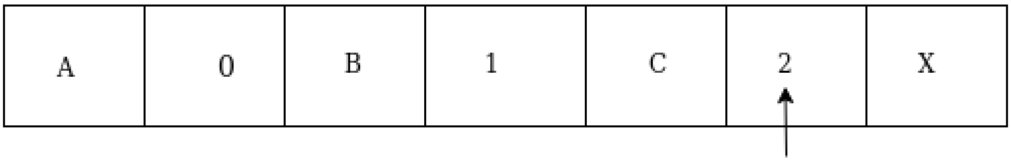
A	0	B	1	2	2	X

The move will be δ(q2, 1) = δ(q2, 1, R) which means it will not change any symbol, remain in the same state and move to right as:



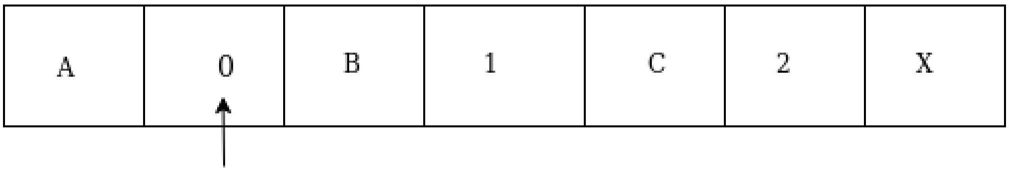
A	0	B	1	2	2	X

The move will be δ(q2, 2) = δ(q3, C, R) which means it will go to state q3, replaced 2 by C and head will move to right as:



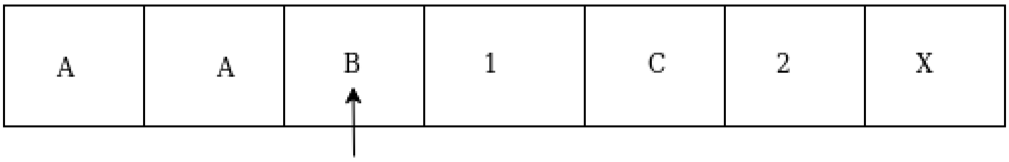
A	0	B	1	C	2	X

Now move δ(q3, 2) = δ(q3, 2, L) and δ(q3, C) = δ(q3, C, L) and δ(q3, 1) = δ(q3, 1, L) and δ(q3, B) = δ(q3, B, L) and δ(q3, 0) = δ(q3, 0, L), and then move δ(q3, A) = δ(q0, A, R), it means will go to state q0, replaced A by A and head will move to right as:



A	0	B	1	C	2	X

The move will be δ(q0, 0) = δ(q1, A, R) which means it will go to state q1, replaced 0 by A, and head will move to right as:



A	A	B	1	C	2	X

The move will be δ(q1, B) = δ(q1, B, R) which means it will not change any symbol, remain in the same state and move to right as:

A	A	B	1	C	2	X
			↑			

The move will be δ(q1, 1) = δ(q2, B, R) which means it will go to state q2, replaced 1 by B and head will move to right as:

A	A	B	B	C	2	X
				↑		

The move will be δ(q2, C) = δ(q2, C, R) which means it will not change any symbol, remain in the same state and move to right as:

A	A	B	B	C	2	X
					↑	

The move will be δ(q2, 2) = δ(q3, C, L) which means it will go to state q3, replaced 2 by C and head will move to left until we reached A as:

A	A	B	B	C	C	X
	↑					

immediately before B is A that means all the 0's are market by A. So we will move right to ensure that no 1 or 2 is present. The move will be δ(q2, B) = (q4, B, R) which means it will go to state q4, will not change any symbol, and move to right as:

A	A	B	B	C	C	X
		↑				

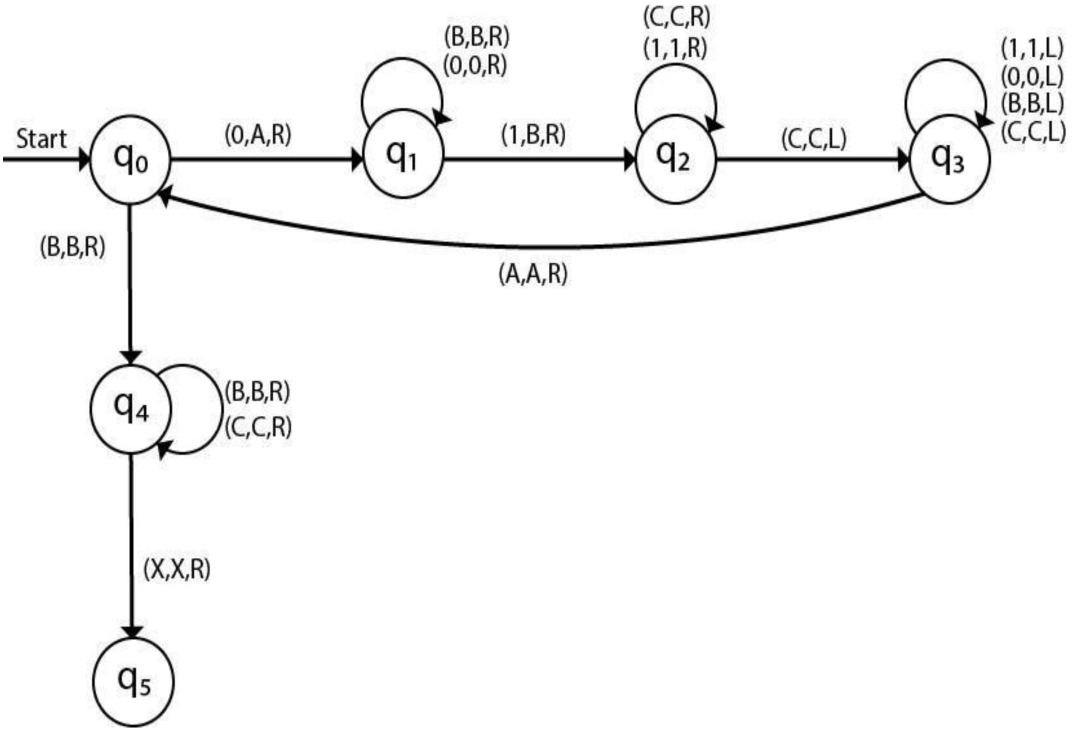
The move will be (q4, B) = δ(q4, B, R) and (q4, C) = δ(q4, C, R) which means it will not change any symbol, remain in the same state and move to right as:

A	A	B	B	C	C	X
						↑

The move δ(q4, X) = (q5, X, R) which means it will go to state q5 which is the HALT state and HALT state is always an accept state for any TM.

A	A	B	B	C	C	X
						↑

The same TM can be represented by Transition Diagram:



Example 2:

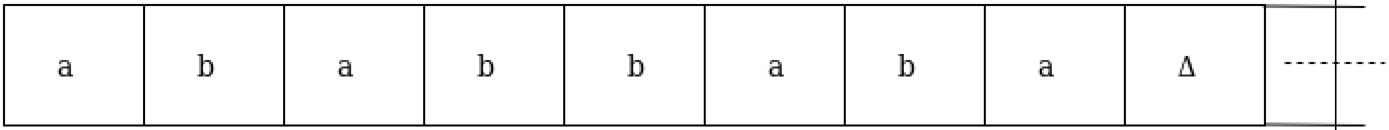
Construct a TM machine for checking the palindrome of the string of even length.

Solution:

Firstly we read the first symbol from the left and then we compare it with the first symbol from right to check whether it is the same.

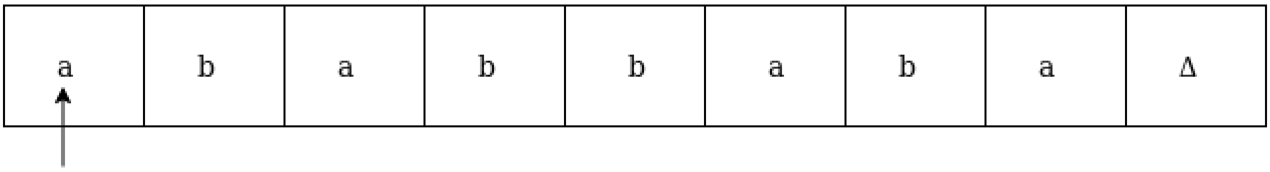
Again we compare the second symbol from left with the second symbol from right. We repeat this process for all the symbols. If we found any symbol not matching, we cannot lead the machine to HALT state.

Suppose the string is ababbabaΔ. The simulation for ababbabaΔ can be shown as follows:



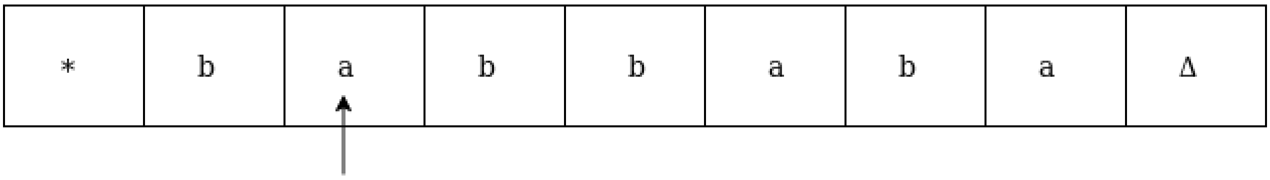
a	b	a	b	b	a	b	a	Δ

Now, we will see how this Turing machine will work for ababbabaΔ. Initially, state is q0 and head points to a as:



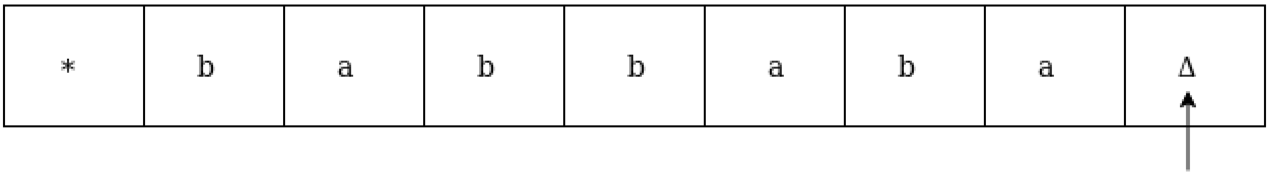
a	b	a	b	b	a	b	a	Δ

We will mark it by * and move to right end in search of a as:



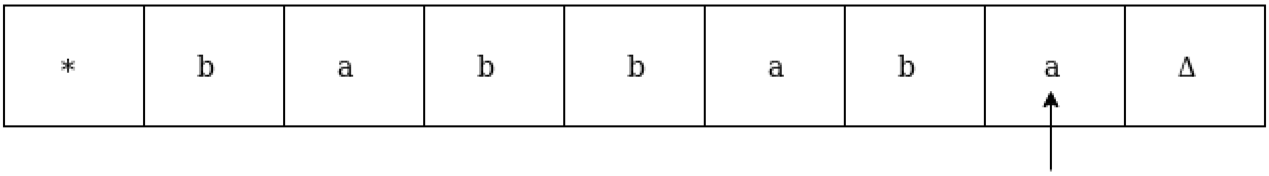
*	b	a	b	b	a	b	a	Δ

We will move right up to Δ as:



*	b	a	b	b	a	b	a	Δ

We will move left and check if it is a:



*	b	a	b	b	a	b	a	Δ

It is 'a' so replace it by Δ and move left as:

*	b	a	b	b	a	b	Δ
							↑

Now move to left up to * as:

*	b	a	b	b	a	b	Δ
↑							

Move right and read it

*	b	a	b	b	a	b	Δ
	↑						

Now convert b by * and move right as:

*	*	a	b	b	a	b	Δ
		↑					

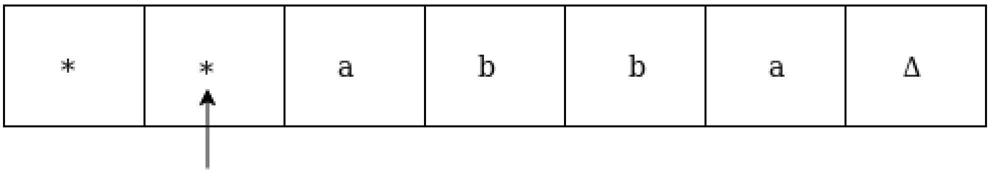
Move right up to Δ in search of b as:

*	*	a	b	b	a	b	Δ
							↑

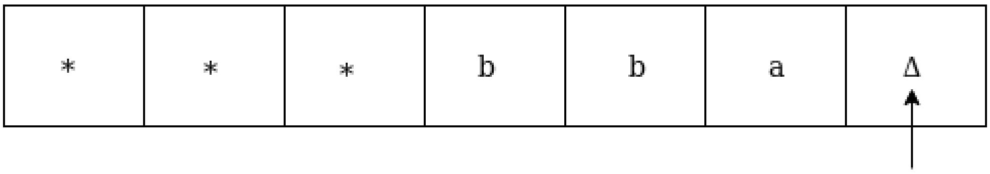
Move left, if the symbol is b then convert it into Δ as:

*	*	a	b	b	a	Δ
						↑

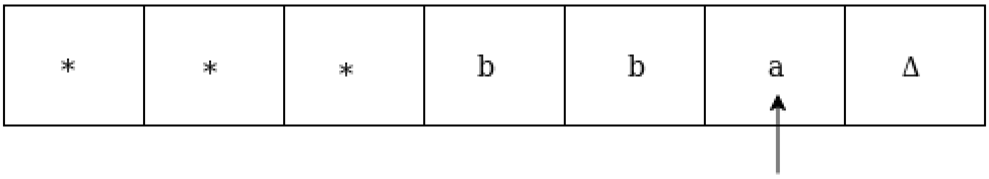
Now move left until * as:



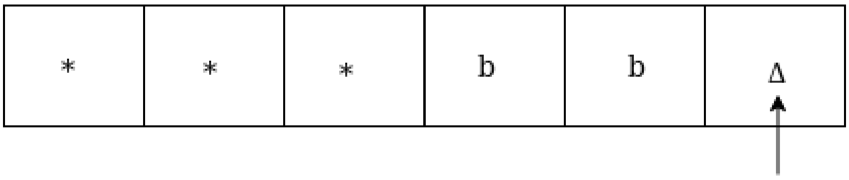
Replace a by * and move right up to Δ as:



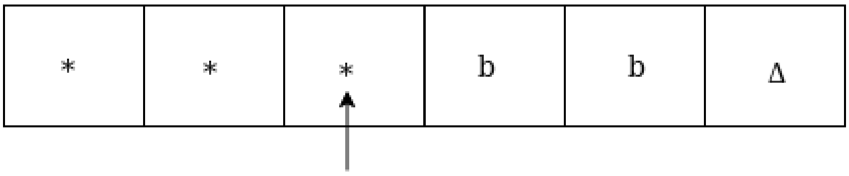
We will move left and check if it is a, then replace it by Δ as:



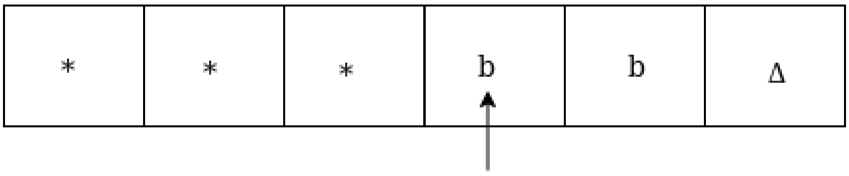
It is 'a' so replace it by Δ as:



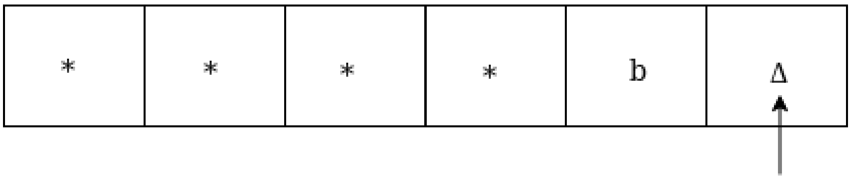
Now move left until *



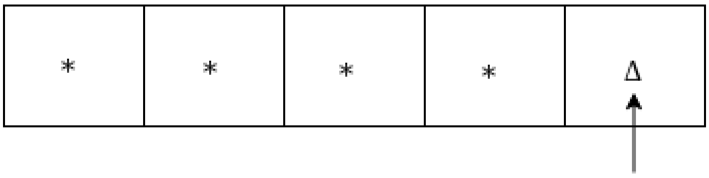
Now move right as:



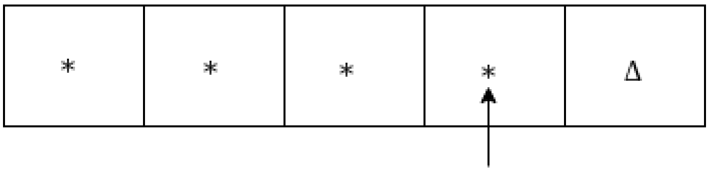
Replace b by * and move right up to Δ as:



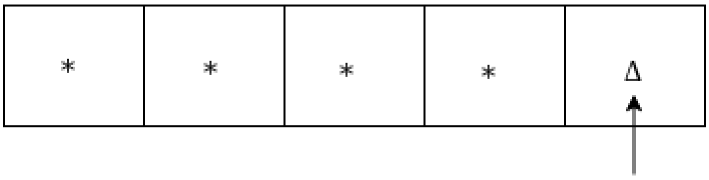
Move left, if the left symbol is b, replace it by Δ as:



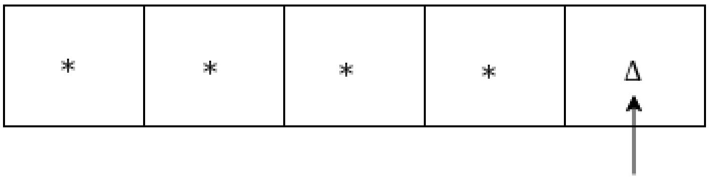
Move left till *



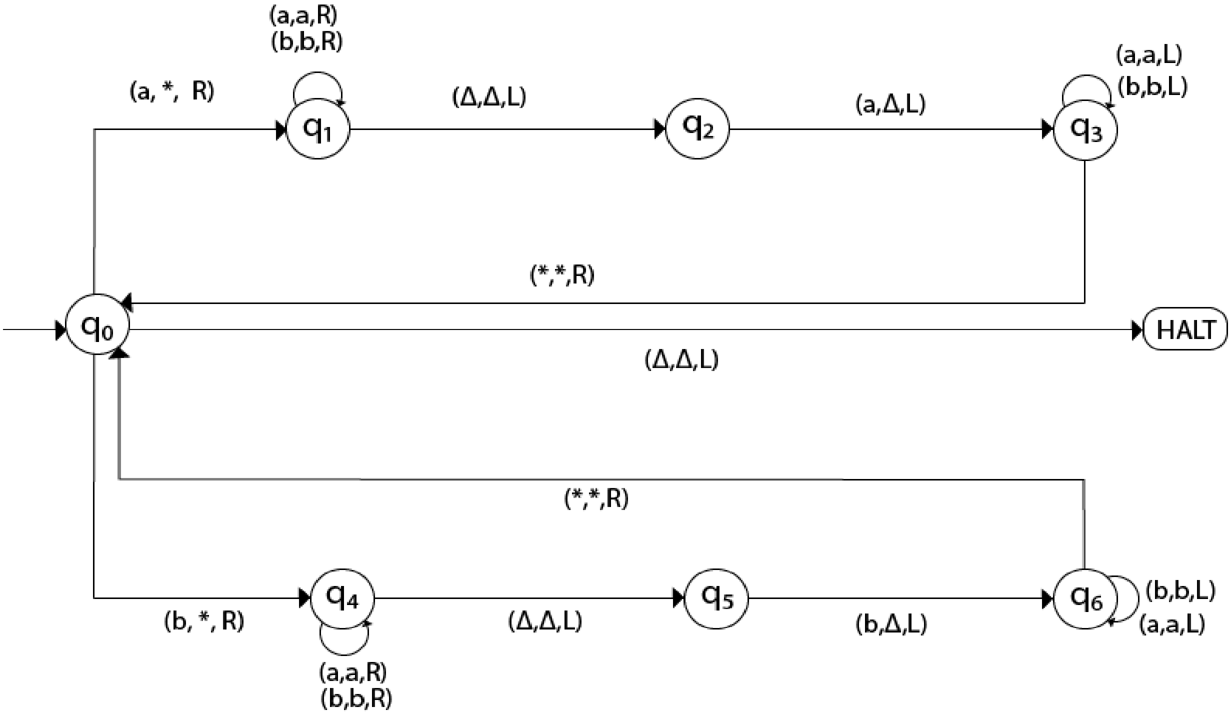
Move right and check whether it is Δ



Go to HALT state



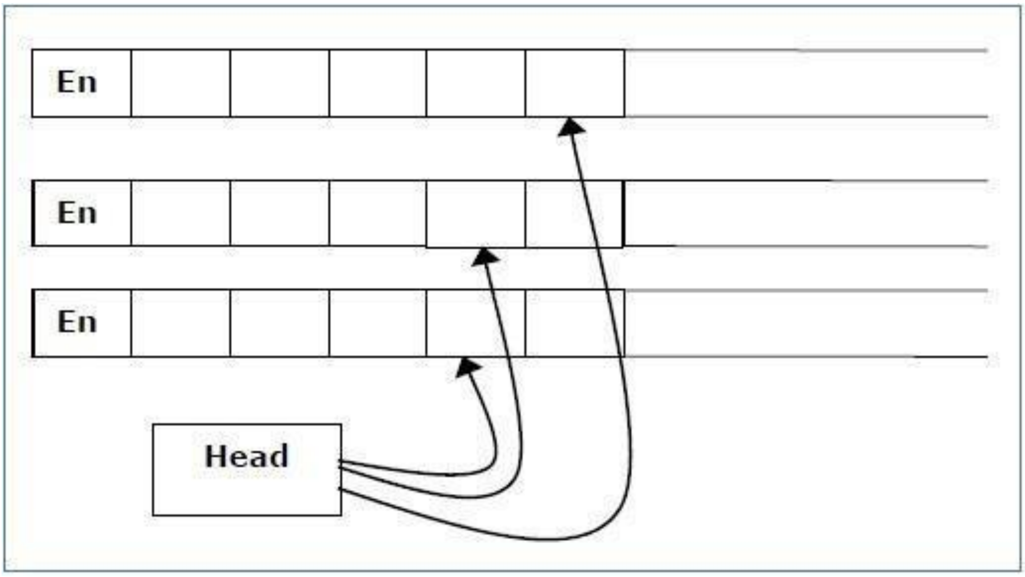
The same TM can be represented by Transition Diagram:



Types of Turing Machines

1. Multi-tape Turing Machines:

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.



A Multi-tape Turing machine can be formally described as a 6-tuple (Q, X, B, δ, q_0, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- **B** is the blank symbol
- **δ** is a relation on states and symbols where

  $\delta: Q \times X^k \rightarrow Q \times (X \times \{\text{Left\_shift, Right\_shift, No\_shift }\})^k$

  where there is **k** number of tapes
- **q_0** is the initial state
- **F** is the set of final states

Note − Every Multi-tape Turing machine has an equivalent single-tape Turing machine.

2. Multi-track Turing machines :

Multi-track Turing machines, a specific type of Multi-tape Turing machine, contain multiple tracks but just one tape head reads and writes on all tracks. Here, a single tape head reads n symbols from **n** tracks at one step. It accepts recursively enumerable languages like a normal single-track single-tape Turing Machine accepts.

A Multi-track Turing machine can be formally described as a 6-tuple (Q, X, ∑, δ, q_0, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet
- **δ** is a relation on states and symbols where

  $\delta(Q_i, [a_1, a_2, a_3, \ldots.]) = (Q_j, [b_1, b_2, b_3, \ldots], \text{Left\_shift or Right\_shift})$
- **q_0** is the initial state
- **F** is the set of final states

Note − For every single-track Turing Machine **S**, there is an equivalent multi-track Turing Machine **M** such that **L(S) = L(M)**.

3. Non-Deterministic Turing Machine :

In a Non-Deterministic Turing Machine, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic.

The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is calleda **Decider** and if for some input, all branches are rejected, the input is also rejected.

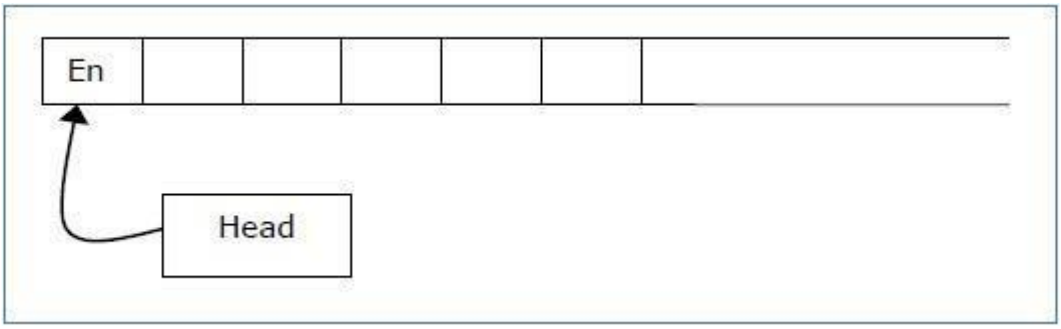
A non-deterministic Turing machine can be formally defined as a 6-tuple (Q, X, ∑, δ, q_0, B, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet
- **δ** is a transition function;

  $\delta : Q \times X \rightarrow P(Q \times X \times \{Left\_shift, Right\_shift\})$.
- $\mathbf{q_0}$ is the initial state
- **B** is the blank symbol
- **F** is the set of final states

4. Turing Machine with a semi-infinite tape :

A Turing Machine with a semi-infinite tape has a left end but no right end. The left end is limited with an end marker.



It is a two-track tape −

- **Upper track** − It represents the cells to the right of the initial head position.
- **Lower track** − It represents the cells to the left of the initial head position in reverse order.

The infinite length input string is initially written on the tape in contiguous tape cells.

The machine starts from the initial state $\mathbf{q_0}$ and the head scans from the left end marker 'End'. In each step, it reads the symbol on the tape under its head. It writes a new symbol on that tape cell and then it moves the head either into left or right one tape cell. A transition function determines the actions to be taken.

It has two special states called **accept state** and **reject state**. If at any point of time it enters into the accepted state, the input is accepted and if it enters into the reject state, the input is rejected by the TM. In some cases, it continues to run infinitely without being accepted or rejected for some certain input symbols.

Note − Turing machines with semi-infinite tape are equivalent to standard Turing machines.

Universal Turing Machines

The Turing Machine (TM) is the machine level equivalent to a digital computer.

It was suggested by the mathematician Turing in the year 1930 and has become the most widely used model of computation in computability and complexity theory.

The model consists of an input and output. The input is given in binary format form on to the machine's tape and the output consists of the contents of the tape when the machine halts

The problem with the Turing machine is that a different machine must be constructed for every new computation to be performed for every input output relation.

This is the reason the Universal Turing machine was introduced which along with input on the tape takes the description of a machine M.

The Universal Turing machine can go on then to simulate M on the rest of the content of the input tape.

A Universal Turing machine can thus simulate any other machine.

The idea of connecting multiple Turing machine gave an idea to Turing −

- Can a Universal machine be created that can 'simulate' other machines?
- This machine is called as Universal Turing Machine

This machine would have three bits of information for the machine it is simulating

- A basic description of the machine.
- The contents of machine tape.

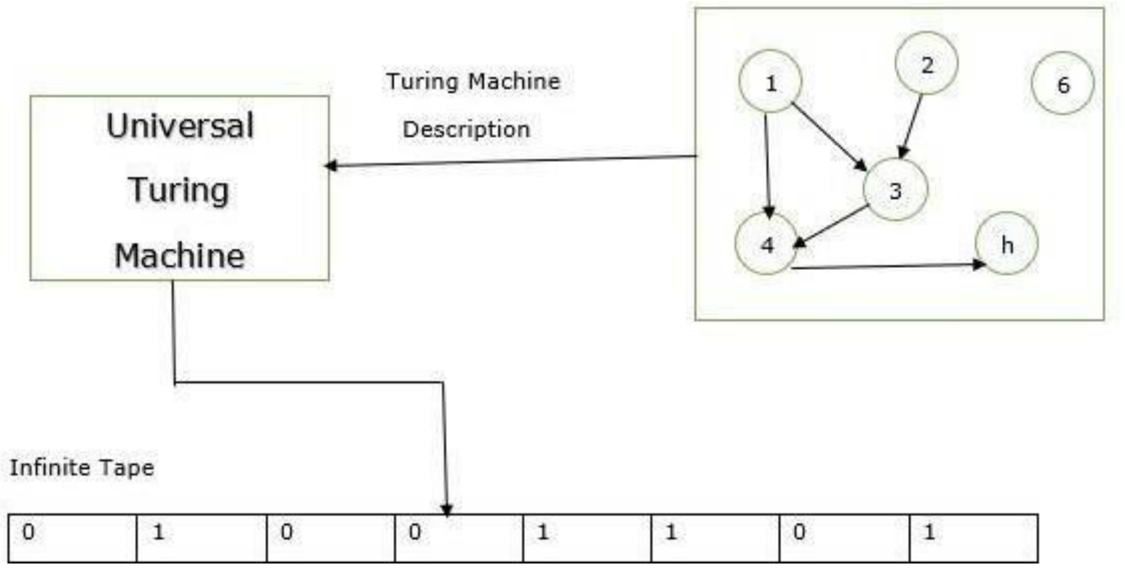
- The internal state of the machine.

The Universal machine would simulate the machine by looking at the input on the tape and the state of the machine.

It would control the machine by changing its state based on the input. This leads to the idea of a “computer running another computer”.

It would control the machine by changing its state based on the input. This leads to the idea of a “computer running another computer”.

The schematic diagram of the Universal Turing Machine is as follows −

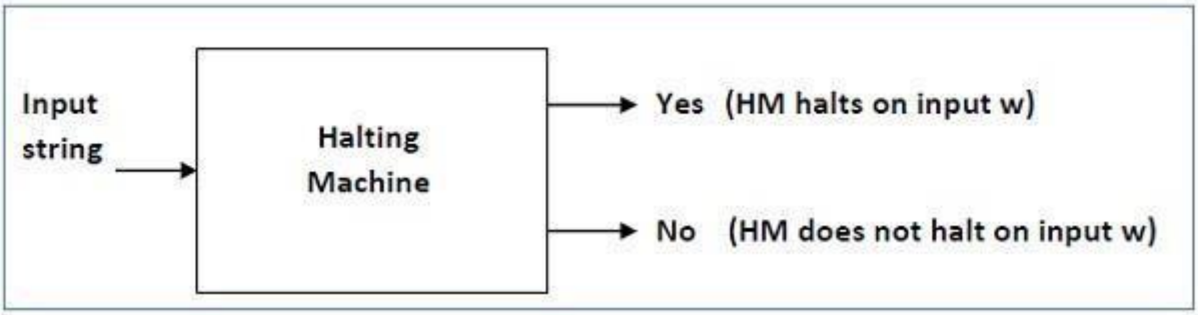


The Halting Problem

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.

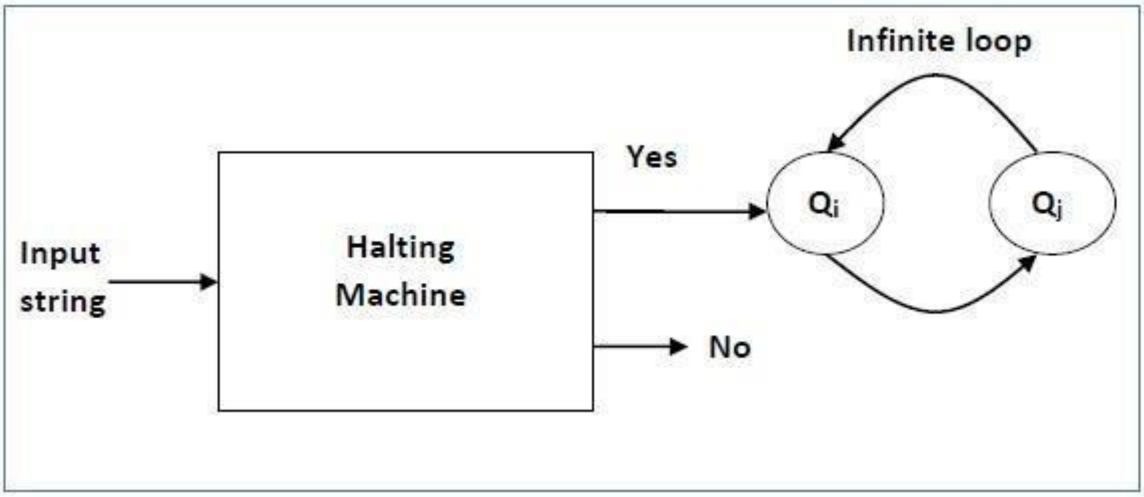
Proof − At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a ‘yes’ or ‘no’ in a finite amount of time. If the halting machine finishes in a finite amount 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’ −



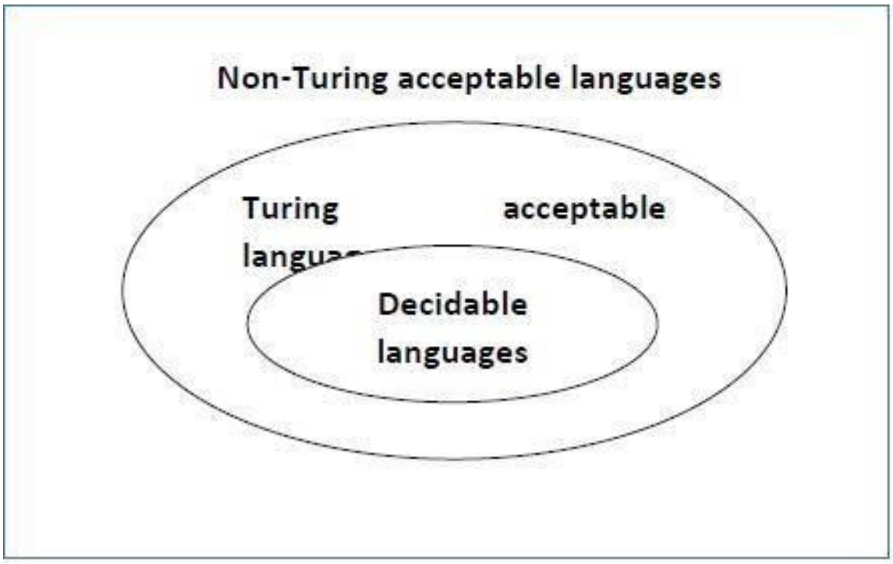
Further, a machine **(HM)$_2$** which input itself is constructed as follows −

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

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

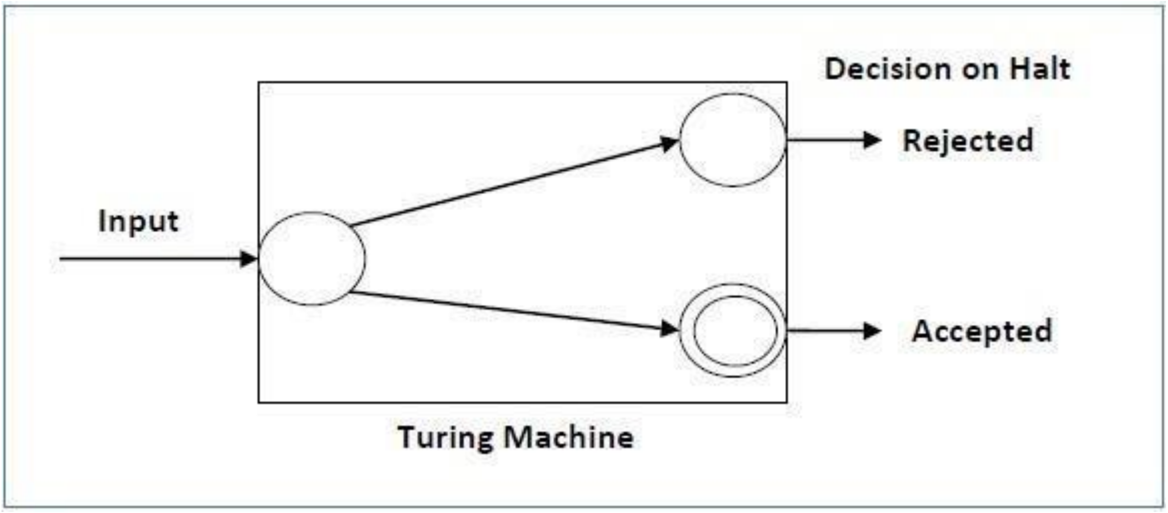
Decidable & Undecidable Problems

A language is called **Decidable** or **Recursive** if there is a Turing machine which accepts and halts on every input string **w**. Every decidable language is Turing-Acceptable.



A decision problem **P** is decidable if the language **L** of all yes instances to **P** is decidable.

For a decidable language, for each input string, the TM halts either at the accept or the reject state as depicted in the following diagram −



Example 1

Find out whether the following problem is decidable or not −

Is a number 'm' prime?

Solution

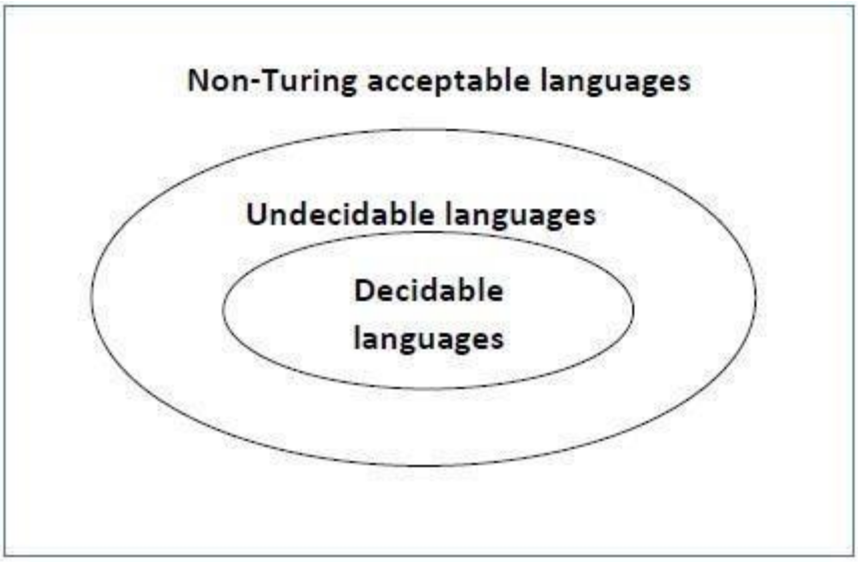
Prime numbers = {2, 3, 5, 7, 11, 13,..................}

Divide the number **'m'** by all the numbers between '2' and '√m' starting from '2'.

If any of these numbers produce a remainder zero, then it goes to the “Rejected state”, otherwise it goes to the “Accepted state”. So, here the answer could be made by ‘Yes’ or ‘No’.

Hence, it is a decidable problem.

For an undecidable language, there is no Turing Machine which accepts the language and makes a decision for every input string **w** (TM can make decision for some input string though). A decision problem **P** is called “undecidable” if the language **L** of all yes instances to **P** is not decidable. Undecidable languages are not recursive languages, but sometimes, they may be recursively enumerable languages.



Example

- The halting problem of Turing machine
- The mortality problem
- The mortal matrix problem
- The Post correspondence problem, etc.

The Post correspondence problem

The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem. The PCP problem over an alphabet ∑ is stated as follows −

- Given the following two lists, **M** and **N** of non-empty strings over ∑ −
- M = (x_1, x_2, x_3,………, x_n)
- N = (y_1, y_2, y_3,………, y_n)

- We can say that there is a Post Correspondence Solution, if for some $i_1, i_2, \ldots\ldots\ldots\ldots i_k$, where $1 \le i_j \le n$, the condition $x_{i1} \ldots\ldots. x_{ik} = y_{i1} \ldots\ldots. y_{ik}$ satisfies.
- Example 1
- Find whether the lists
- M = (abb, aa, aaa) and N = (bba, aaa, aa)
- have a Post Correspondence Solution?
- **Solution**

	x_1	x_2	x_3
M	Abb	aa	aaa
N	Bba	aaa	aa

- Here,
- $x_2x_1x_3$ = **'aaabbaaa'**
- and $y_2y_1y_3$ = **'aaabbaaa'**
- We can see that
- $x_2x_1x_3 = y_2y_1y_3$
- Hence, the solution is **i = 2, j = 1, and k = 3.**
- **Example 2**
- Find whether the lists **M = (ab, bab, bbaaa)** and **N = (a, ba, bab)** have a Post Correspondence Solution?
- **Solution**

	x_1	x_2	x_3
M	ab	bab	bbaaa
N	a	ba	bab

- In this case, there is no solution because −

- $| \mathbf{x_2x_1x_3} | \neq | \mathbf{y_2y_1y_3} |$ (Lengths are not same)
- Hence, it can be said that this Post Correspondence Problem is **undecidable**.

UNIT-IV

Propositional calculus

A proposition is the basic building block of logic. It is defined as a declarative sentence that is either True or False, but not both. The **Truth Value** of a proposition is True(denoted as T) if it is a true statement, and False(denoted as F) if it is a false statement.

To represent propositions, **propositional variables** are used. By Convention, these variables are represented by small alphabets such as .
The area of logic which deals with propositions is called **propositional calculus** or **propositional logic**.
It also includes producing new propositions using existing ones. Propositions constructed using one or more propositions are called **compound propositions**. The propositions are combined together using **Logical Connectives** or **Logical Operators**.

Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

a. **Atomic Propositions**

b. **Compound propositions**

- **Atomic Proposition:** Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.
- **Example:**

1. a) 2+2 is 4, it is an atomic proposition as it is a **true** fact.
2. b) "The Sun is cold" is also a proposition as it is a **false** fact.

- **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

1. a) "It is raining today, and street is wet."
2. b) "Ankit is a doctor, and his clinic is in Mumbai."

Truth-Assignments

In propositional logic generally we use five connectives which are −

- OR (∨∨)
- AND (AA)
- Negation/ NOT (¬¬)
- Implication / if-then (→→)
- If and only if (⇔⇔).

OR (∨∨**)** − The OR operation of two propositions A and B (written as A∨BA∨B) is true if at least any of the propositional variable A or B is true.

The truth table is as follows −

A	**B**	**A** ∨ **B**
True	True	True
True	False	True
False	True	True
False	False	False

AND (AA**)** − The AND operation of two propositions A and B (written as AABAAB) is true if both the propositional variable A and B is true.

The truth table is as follows −

A	**B**	**A** A **B**
True	True	True
True	False	False
False	True	False

False	False	False

Negation (¬¬) − The negation of a proposition A (written as ¬A¬A) is false when A is true and is true when A is false.

The truth table is as follows −

A	**¬ A**
True	False
False	True

Implication / if-then (→→) − An implication A→B A→B is the proposition “if A, then B”. It is false if A is true and B is false. The rest cases are true.

The truth table is as follows −

A	**B**	**A → B**
True	True	True
True	False	False
False	True	True
False	False	True

If and only if (⇔⇔) − A⇔BA⇔B is bi-conditional logical connective which is true when p and q are same, i.e. both are false or both are true.

The truth table is as follows −

A	**B**	**A ⇔ B**
True	True	True
True	False	False

False	True	False
False	False	True

Tautologies

A Tautology is a formula which is always true for every value of its propositional variables.

Example − Prove [(A→B)ɅA]→B[(A→B)ɅA]→B is a tautology

The truth table is as follows −

A	**B**	**A → B**	**(A → B)** Ʌ **A**	**[(A → B)** Ʌ **A] → B**
True	True	True	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	True

As we can see every value of [(A→B)ɅA]→B[(A→B)ɅA]→B is "True", it is a tautology.

Contradictions

A Contradiction is a formula which is always false for every value of its propositional variables.

Example − Prove (A∨B)Ʌ[(¬A)Ʌ(¬B)](A∨B)Ʌ[(¬A)Ʌ(¬B)] is a contradiction

The truth table is as follows –

A	**B**	**A ∨ B**	**¬ A**	**¬ B**	**(¬ A) A (¬ B)**	**(A ∨ B) A [(¬ A) A (¬ B)]**
True	True	True	False	False	False	False
True	False	True	False	True	False	False
False	True	True	True	False	False	False
False	False	False	True	True	True	False

As we can see every value of (A∨B)A[(¬A)A(¬B)](A∨B)A[(¬A)A(¬B)] is “False”, it is a contradiction.

Contingency

A Contingency is a formula which has both some true and some false values for every value of its propositional variables.

Example − Prove (A∨B)A(¬A)(A∨B)A(¬A) a contingency

The truth table is as follows −

A	**B**	**A ∨ B**	**¬ A**	**(A ∨ B) A (¬ A)**
True	True	True	False	False
True	False	True	False	False
False	True	True	True	True
False	False	False	True	False

A	B	A ∨ B	¬ (A ∨ B)	¬ A	¬ B	[(¬ A) A (¬ B)]
True	True	True	False	False	False	False
True	False	True	False	False	True	False
False	True	True	False	True	False	False
False	False	False	True	True	True	True

As we can see every value of (A∨B)A(¬A)(A∨B)A(¬A) has both "True" and "False", it is a contingency.

Propositional Equivalences

Two statements X and Y are logically equivalent if any of the following two conditions hold −

- The truth tables of each statement have the same truth values.
- The bi-conditional statement X⇔YX⇔Y is a tautology.

Example – Prove ¬(A∨B)and[(¬A)A(¬B)]¬(A∨B)and[(¬A)A(¬B)] are equivalent

Testing by 1st method (Matching truth table)

Here, we can see the truth values of ¬(A∨B)and[(¬A)A(¬B)]¬(A∨B)and[(¬A)A(¬B)] are same, hence the statements are equivalent.

A	**B**	**¬ (A ∨ B)**	**[(¬ A) A (¬ B)]**	**[¬ (A ∨ B)] ⇔ [(¬ A) A (¬ B)]**
True	True	False	False	True
True	False	False	False	True
False	True	False	False	True
False	False	True	True	True

Testing by 2nd method (Bi-conditionality)

As [¬(A∨B)]⇔[(¬A)A(¬B)][¬(A∨B)]⇔[(¬A)A(¬B)] is a tautology, the statements are equivalent.

Inverse, Converse, and Contra-positive

Implication / if-then (→)(→) is also called a conditional statement. It has two parts −

- Hypothesis, p
- Conclusion, q

As mentioned earlier, it is denoted as p→qp→q.

Example of Conditional Statement − "If you do your homework, you will not be punished." Here, "you do your homework" is the hypothesis, p, and "you will not be punished" is the conclusion, q.

Inverse − An inverse of the conditional statement is the negation of both the hypothesis and the conclusion. If the statement is "If p, then q", the inverse will be "If not p, then not q". Thus the inverse of p→qp→q is ¬p→¬q¬p→¬q.

Example − The inverse of "If you do your homework, you will not be punished" is "If you do not do your homework, you will be punished."

Converse − The converse of the conditional statement is computed by interchanging the hypothesis and the conclusion. If the statement is “If p, then q”, the converse will be “If q, then p”. The converse of p→qp→q is q→pq→p.

Example − The converse of "If you do your homework, you will not be punished" is "If you will not be punished, you do your homework”.

Contra-positive − The contra-positive of the conditional is computed by interchanging the hypothesis and the conclusion of the inverse statement. If the statement is “If p, then q”, the contra-positive will be “If not q, then not p”. The contra-positive of p→qp→q is ¬q→¬p¬q→¬p.

Example − The Contra-positive of " If you do your homework, you will not be punished” is "If you are punished, you did not do your homework”.

Duality Principle

Duality principle states that for any true statement, the dual statement obtained by interchanging unions into intersections (and vice versa) and interchanging Universal set into Null set (and vice versa) is also true. If dual of any statement is the statement itself, it is said **self-dual** statement.

Example − The dual of (A∩B)UC(A∩B)UC is (AUB)∩C(AUB)∩C

Normal Forms

We can convert any proposition in two normal forms −

- Conjunctive normal form
- Disjunctive normal form

Conjunctive Normal Form

A compound statement is in conjunctive normal form if it is obtained by operating AND among variables (negation of variables included) connected with ORs. In terms of set operations, it is a compound statement obtained by Intersection among variables connected with Unions.

Examples

- (A∨B)A(A∨C)A(B∨C∨D)(A∨B)A(A∨C)A(B∨C∨D)
- (PUQ)∩(QUR)(PUQ)∩(QUR)

Disjunctive Normal Form

A compound statement is in disjunctive normal form if it is obtained by operating OR among variables (negation of variables included) connected with ANDs. In terms of set operations, it is a compound statement obtained by Union among variables connected with Intersections.

Examples

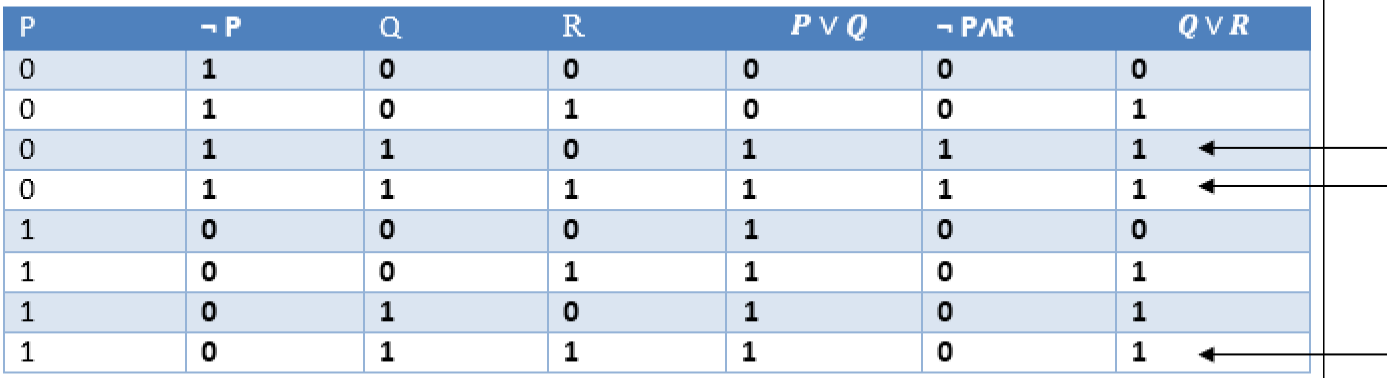
- (AAB)∨(AAC)∨(BACAD)(AAB)∨(AAC)∨(BACAD)
- (P∩Q)∪(Q∩R)

Resolution:

The Resolution rule state that if P∨Q and ¬ PAR is true, then Q∨R will also be true. **It can be represented as**

Notation of Resolution $\dfrac{P\vee Q,\ \neg P\wedge R}{Q\vee R}$

Proof by Truth-Table:



P	¬ P	Q	R	P ∨ Q	¬ P∧R	Q ∨ R
0	1	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	1
1	0	1	0	1	0	1
1	0	1	1	1	0	1

Resolution in FOL

Resolution

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. It was invented by a Mathematician John Alan Robinson in the year 1965.

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form**.

Clause: Disjunction of literals (an atomic sentence) is called a **clause**. It is also known as a unit clause.

Conjunctive Normal Form: A sentence represented as a conjunction of clauses is said to be **conjunctive normal form** or **CNF**.

Example:

We can resolve two clauses which are given below:

[Animal (g(x) V Loves (f(x), x)] and [¬ Loves(a, b) V ¬Kills(a, b)]

Where two complimentary literals are: **Loves (f(x), x) and ¬ Loves (a, b)**

These literals can be unified with unifier **θ= [a/f(x), and b/x]** , and it will generate a resolvent clause:

[Animal (g(x) V ¬ Kills(f(x), x)].

Steps for Resolution:

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

To better understand all the above steps, we will take an example in which we will apply resolution.

Example:

a. **John likes all kind of food.**

b. **Apple and vegetable are food**

c. **Anything anyone eats and not killed is food.**

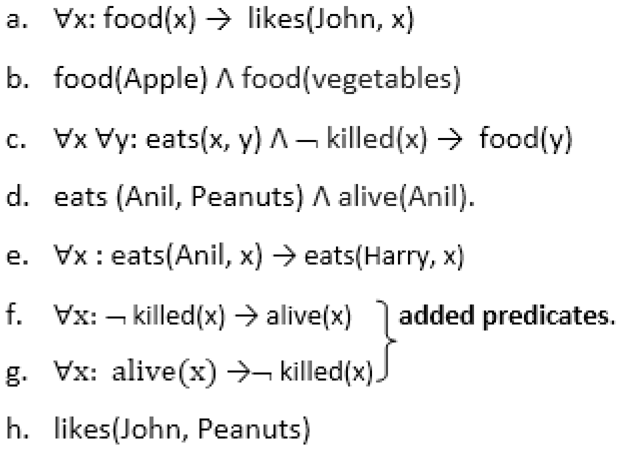
d. **Anil eats peanuts and still alive**

e. **Harry eats everything that Anil eats.**
   Prove by resolution that:

f. **John likes peanuts.**

Step-1: Conversion of Facts into FOL

In the first step we will convert all the given statements into its first order logic.



a. ∀x: food(x) → likes(John, x)

b. food(Apple) Λ food(vegetables)

c. ∀x ∀y: eats(x, y) Λ ¬ killed(x) → food(y)

d. eats (Anil, Peanuts) Λ alive(Anil).

e. ∀x : eats(Anil, x) → eats(Harry, x)

f. ∀x: ¬ killed(x) → alive(x) } added predicates.

g. ∀x: alive(x) →¬ killed(x)

h. likes(John, Peanuts)

Step-2: Conversion of FOL into CNF

In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.

- **Eliminate all implication (→) and rewrite**
  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
  6. ∀x¬ [¬ killed(x)] V alive(x)
  7. ∀x ¬ alive(x) V ¬ killed(x)
  8. likes(John, Peanuts).
- **Move negation (¬)inwards and rewrite**

1. ∀x ¬ food(x) V likes(John, x)
   2. food(Apple) Λ food(vegetables)

3. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
4. eats (Anil, Peanuts) Λ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
6. ∀x ¬killed(x)] V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(John, Peanuts).

- **Rename variables or standardize variables**

1. ∀x ¬ food(x) V likes(John, x)
2. food(Apple) Λ food(vegetables)
3. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
4. eats (Anil, Peanuts) Λ alive(Anil)
5. ∀w¬ eats(Anil, w) V eats(Harry, w)
6. ∀g ¬killed(g)] V alive(g)
7. ∀k ¬ alive(k) V ¬ killed(k)
8. likes(John, Peanuts).

- **Eliminate existential instantiation quantifier by elimination.**
  In this step, we will eliminate existential quantifier ∃, and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.
- **Drop Universal quantifiers.**
  In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.

1. ¬ food(x) V likes(John, x)
2. food(Apple)
3. food(vegetables)
4. ¬ eats(y, z) V killed(y) V food(z)
5. eats (Anil, Peanuts)
6. alive(Anil)
7. ¬ eats(Anil, w) V eats(Harry, w)
8. killed(g) V alive(g)
9. ¬ alive(k) V ¬ killed(k)
10. likes(John, Peanuts).

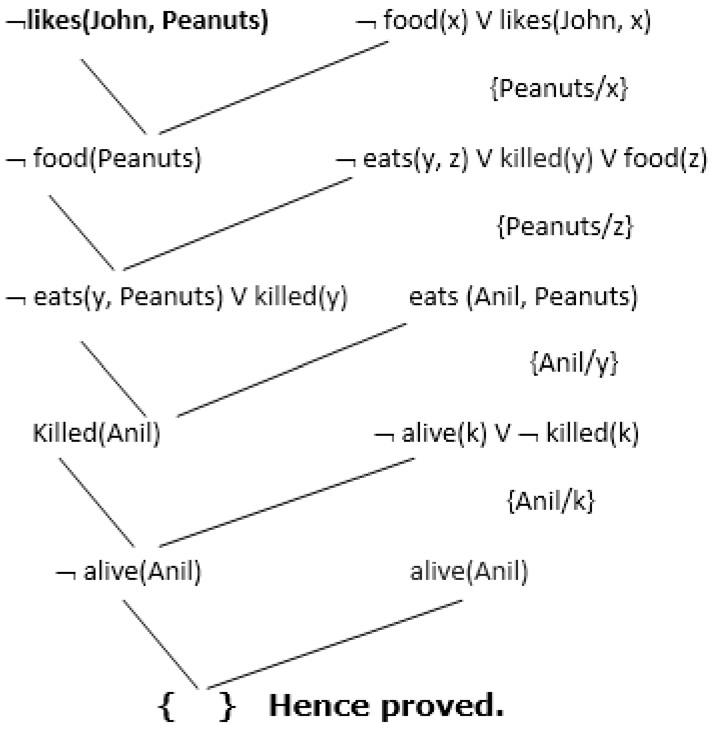
- **Distribute conjunction ∧ over disjunction ¬.**
  This step will not make any change in this problem.

Step-3: Negate the statement to be proved

In this statement, we will apply negation to the conclusion statements, which will be written as ¬likes(John, Peanuts)

Step-4: Draw Resolution graph:

Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:



Hence the negation of the conclusion has been proved as a complete contradiction with the given set of statements.

Explanation of Resolution graph:

- In the first step of resolution graph, **¬likes(John, Peanuts)** , and **likes(John, x)** get resolved(canceled) by substitution of **{Peanuts/x}**, and we are left with **¬ food(Peanuts)**

- In the second step of the resolution graph, ¬ **food(Peanuts)** , and **food(z)** get resolved (canceled) by substitution of **{ Peanuts/z}**, and we are left with ¬ **eats(y, Peanuts) V killed(y)** .
- In the third step of the resolution graph, ¬ **eats(y, Peanuts)** and **eats (Anil, Peanuts)** get resolved by substitution **{Anil/y}**, and we are left with **Killed(Anil)** .
- In the fourth step of the resolution graph, **Killed(Anil)** and ¬ **killed(k)** get resolve by substitution **{Anil/k}**, and we are left with ¬ **alive(Anil)** .
- In the last step of the resolution graph ¬ **alive(Anil)** and **alive(Anil)** get resolved.

Validity and satisfiability:

In mathematical logic, including, in particular, first-order logic and propositional calculus, **satisfiability** and **validity** are elementary concepts of semantics. A formula is *satisfiable* if there exists an interpretation (model) that makes the formula true.[1] A formula is *valid* if all interpretations make the formula true. The opposites of these concepts are **unsatisfiability** and **invalidity**, that is, a formula is *unsatisfiable* if none of the interpretations make the formula true, and *invalid* if some such interpretation makes the formula false. These four concepts are related to each other in a manner exactly analogous to Aristotle's square of opposition.

The four concepts can be raised to apply to whole theories: a theory is satisfiable (valid) if one (all) of the interpretations make(s) each of the axioms of the theory true, and a theory is unsatisfiable (invalid) if all (one) of the interpretations make(s) one of the axioms of the theory false.

A propositional logic formula φ is called satisfiable if there is some assignment to its variables that makes it evaluate to true. ● p ∧ q is satisfiable. ● p ∧ ¬p is unsatisfiable. ● p → (q ∧ ¬q) is satisfiable. ● An assignment of true and false to the variables of φ that makes it evaluate to true is called a satisfying assignment.

Predicate Logic deals with predicates, which are propositions containing variables.

Predicate Logic – Definition

A predicate is an expression of one or more variables defined on some specific domain. A predicate with variables can be made a proposition by either assigning a value to the variable or by quantifying the variable.

The following are some examples of predicates −

- Let E(x, y) denote "x = y"
- Let X(a, b, c) denote "a + b + c = 0"
- Let M(x, y) denote "x is married to y"

Well Formed Formula

Well Formed Formula (wff) is a predicate holding any of the following −

- All propositional constants and propositional variables are wffs
- If x is a variable and Y is a wff, ∀xY∀xY and ∃xY∃xY are also wff
- Truth value and false values are wffs
- Each atomic formula is a wff
- All connectives connecting wffs are wffs

Quantifiers

The variable of predicates is quantified by quantifiers. There are two types of quantifier in predicate logic − Universal Quantifier and Existential Quantifier.

Universal Quantifier

Universal quantifier states that the statements within its scope are true for every value of the specific variable. It is denoted by the symbol ∀∀.
∀xP(x)∀xP(x) is read as for every value of x, P(x) is true.
Example − "Man is mortal" can be transformed into the propositional form ∀xP(x)∀xP(x) where P(x) is the predicate which denotes x is mortal and the universe of discourse is all men.

Existential Quantifier

Existential quantifier states that the statements within its scope are true for some values of the specific variable. It is denoted by the symbol ∃∃.
∃xP(x)∃xP(x) is read as for some values of x, P(x) is true.
Example − "Some people are dishonest" can be transformed into the propositional form ∃xP(x)∃xP(x) where P(x) is the predicate which denotes x is dishonest and the universe of discourse is some people.

Nested Quantifiers

If we use a quantifier that appears within the scope of another quantifier, it is called nested quantifier.

Example

- ∀ a∃bP(x,y)∀ a∃bP(x,y) where P(a,b)P(a,b) denotes a+b=0a+b=0
- ∀ a∀b∀cP(a,b,c)∀ a∀b∀cP(a,b,c) where P(a,b)P(a,b) denotes a+(b+c)=(a +b)+c

Syntax Of Predicate Logic:

Predicate logic is very expressive, but we need to clarify several important items.

First give a precise definition of what a formula in predicate logic is. Same as with programming languages: we have to pin down the syntax exactly.

Then associate a clear definition of truth (usually called validity) with these formulae. Somewhat like the semantics of a programming language.

To define validity, we have to define structures, domains over which a formula in predicate logic can be interpreted.

Every function symbol and relation symbol has a fixed number of arguments, its arity. Terms are defined inductively by Every constant and variable is a term. If f is an n-ary function symbol, and t1;::: ; t n are terms, then f (t1;::: ; t n) is also a term. An atomic formula is an expression of the form R(t1;::: ; t n) where R is an n-ary relation symbol, and t1;::: ; t n are terms. Lastly, formulae are defined inductively by Every atomic formula is a is a formula.