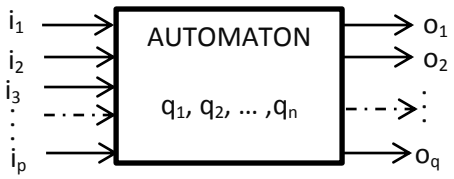
Definition of an *Automaton*:-An Automaton is defined as a system that preforms certain functions without human intervention. it accepts raw material and energy as input and converts them into the final product under the guidance of control signals. or an automata is defined as a system where energy, materials, and information are transformed, transmitted and used for performing some functions without direct involvement of man. Ex: Any automatic machine like printing machine, washing machine etc.



Model of a automaton

Characteristics of automaton:

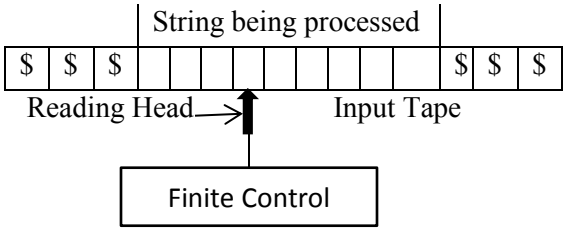
1. *Input*: $i_1, i_2,\ldots,i_p$ are the input of the model each of which can take a finite number of fixed values from an input I.
2. *Output*: $o_1, o_2,\ldots,o_q$ are the outputs of the model each of which can give the finite number of fixed values to an output O.
3. *States*: At any instant of time the automaton can be in one of the states $q_1, q_2, \ldots, q_n$.
4. *States Relation:* The next state of an automaton at any instant of time is determine by the present state and present input.
5. *Output relation*: The output is related to either state only or to both the input and the state.

Note: An automaton in which the output depends only on the input is called an automaton without a memory. Ex:- logic gate. An automaton in which the output depends on the state and input is called an automaton with a finite memory. Ex:- flip-flops, shift register, Mealy machine. An automaton in which the output depends only on the states of the machine is called Moore Machine.

Description of a Finite Automata (*Finite State Machine*): A Finite automaton can be represented by five-tuple structure $M(Q,\Sigma, \delta, q_0, F)$, where

1. *Q is a finite non empty set of states.*
2. *Σ is a finite non empty set of inputs called the input alphabets.*
3. *is a function which maps $Q \times \Sigma$ into Q and is called transmission function (next state function) (present state × input alphabet→ next state).*
4. *$q_0 \in Q$ is the initial state.*
5. *$F \subseteq Q$ is the set of final states (may be more than 1).*

Note: $Q\times\Sigma^*$ into Q means Present state × String of input symbols(including Λ) → Next State.



Block diagram of a finite automaton

I) *Input tape*: The input tape is divided onto squares, each square containing a single symbol from the input alphabet Σ. The end squares of the tape contain the end marker $. The absence of the end markers indicates that the tape is of infinite long. Input string is processed from left to right.

II) *Reading Head*: The head examines only one square at a time and can move one square either to the left or to the right, we restrict the movement of reading head only to the right side.

III) *Finite Control*: The input to the finite control will usually be the 'Symbol' under the reading head and the 'present state' of the machine. Outputs will be A). A motion of R-head along the tape to the next square. B). The next state of the finite state machine given by $\delta(q,a)$.

Transition system: A Transition system is a finite directed labeled graph in which each vertex(node) represents a state and the directed edge indicates the transmission of a state and the edges are labeled with input/output.

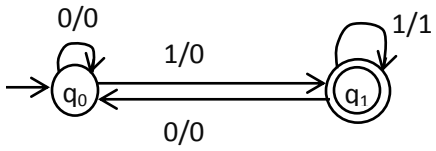


Fig. A Transition System.

In Figure, the initial state is represented by a circle with an arrow pointing toward it, the final state by two concentric circles and the other states are represented by a circle. The edges are labeled by input/output (eq. 1/0 or 1/1).

For example, if the system is in the state q_0 & the input is 1 is applied, the system moves to state q_1 as there is a directed edge from q_0 to q_1 with label1/0. It output 0.

Definition of Transition System: A transition System consist of 5-tuple $(Q,\Sigma, \delta, Q_0, F)$.

I. Here $Q,\sum$ and F are finite non empty set of states, the input alphabets, and set of final states respectively, as in the case of FA.

II. $Q_0 \subseteq Q$ and Q_0 is the non-empty set of initial state.

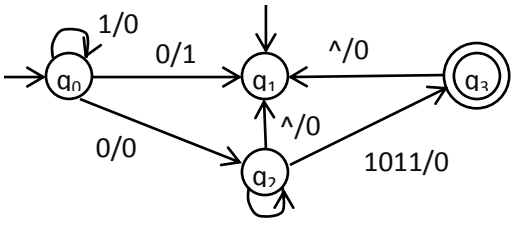
III. δ is finite $Q\times\Sigma \rightarrow Q$.

In other words, if (q_1, ω, q_2) is in δ. It means that the graph starts at the vertex q_1, goes along a set of edges and reaches the vertex q_2.

A Transition system accepts a string w in Σ^* if

i). There exists a path which originates from some initial state, goes along the arrows and terminates at some final state.

ii). The path value obtained by concatenation of all- edge-labels of the path is equal to w.



Determine the initial and final states.

q_0 & q_1 q_3

101011 will be accepted.

111010 will be rejected.

Properties of Transition Functions:

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

Property 2:- for all strings w and input symbol a

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

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

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

Ex: prove that for any input transition function δ and for any two input string x and y.

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

Proof: By method pf mathematical induction

1. **Basis**: on |y| i.e. length of y when |y|=1, $y=a\in\Sigma$.
   L.H.S= $\delta(q,xa)$
   $= \delta(\delta(q,x),a)$ (by using prop. 2)
   =R.H.S.
2. **Induction Hypothesis**: Assume the result for all string x and string y with |y|=n.
3. Let y be a string of length n+1.
   Write $y=y_1a$, where $|y_1| = n$.
   L.H.S. = $\delta(q,xy_1a) = \delta(q,x_1a)$ where $x_1=xy_1$
   $= \delta(\delta(q,x_1)a)$ (by using prop. 2).
   $= \delta(\delta(q,xy_1)a)$
   $= \delta(\delta(\delta(q,x), y_1)a)$ by step 2.result
   $= \delta(\delta(q,x)y_1a)$
   $= \delta(\delta(q,x)y) = R.H.S.$
   By the principle of mathematical induction, this is true for all strings.

Ex.:- prove that if $\delta(q,x) = \delta(q,y)$, then $\delta(q,xz) = \delta(q,yz)$ for all strings z in Σ^+.

Sol:- $\delta(q,xz) = \delta(\delta(q,x)z)$ by previous results.

$= \delta(\delta(q,y)z)$ (given)

=(q,yz) (reverse the previous result)

Regular Languages: a regular language over an alphabet Σ is one that can be obtained from these basic languages using the operations Union, Concatenation and Kleene* (Kleene* operation arises from the concatenation to produce infinite languages).

A regular language can be described by explicit formula { } by leaving out the set of { } or replacing them with ( ) and replacing ∪ by +; the result is called a regular expression.

	Language	Corresponding Regular Expression
1.	{^}	^
2.	{0}	0
3.	{001} or ({0},{0},{1})	001
4.	{0,1} or ({0}∪{1})	0+1
5.	{0,10} or ({0} ∪ {10})	0+10
6.	{1,^}{001}	(1+^)001
7.	$\{110\}^*\{10\}$	$(110)^*(0+1)$
8.	$\{1\}^*\{10\}$	$(1)^*10$ or 1^*0 or 1^+0
9.	$\{10,111,11010\}^*$	$(10+111+11010)^*$
10.	$\{0,10\}^*\{\{11\}^*U\{001,^\}\})$	$(0+10)^*((11)^*+001+^)$

Definition: Regular Languages and Regular Expressions over Σ: The set R of regular languages over Σ and the corresponding regular expressions are defined as follows:

1. Φ is an element of R, then the corresponding RE is Φ.
2. {^} is an element of R, then corresponding RE is ^.
3. For each a∈Σ is an element of R, and the corresponding RE is a.
4. If L_1 and L_2 are any element of R and r_1 and r_2 are corresponding RE.
   a. $L_1 \cup L_2$ are any element of R and the corresponding RE is (r_1+r_2).
   b. L_1L_2 is an elements of any element of R and the corresponding RE is (r_1r_2).
   c. L_1^* is an elements of R and the corresponding RE is (r_1^*).

Note: Only those languages that can be obtained by using statements 1 to 4 are regular languages over Σ.

Acceptability of a string by a finite Automation:

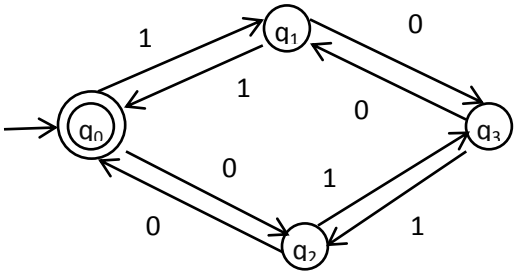
Definition: A string x is accepted by a finite automaton M=(Q, Σ, δ, q_0, F) if $\delta(q_0, x)=q$ for some q∈ F.

This is basically the acceptability of a string by the final state,

Example: The FSM is given below

Table:

State	Input symbol 0	1
→ q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2



Here Q = { q_0, q_1, q_2, q_3 }

Σ = {0, 1 }

F = { q_0 }

Input String(x) = 110101

$\delta(q_0, 110101) \Longrightarrow \delta(q_1, 10101) \Longrightarrow$

$\delta(q_0, 0101) \Longrightarrow \delta(q_2, 101) \Longrightarrow \delta(q_3, 01) \Longrightarrow$

$\delta(q_1, 1) \Longrightarrow \delta(q_0, \wedge) \Longrightarrow q_0$

Hence $q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_0 \xrightarrow{0} q_2 \xrightarrow{1} q_3 \xrightarrow{0} q_1 \xrightarrow{1} q_0$

This ↓ indicates that the current input symbol is being processed by machine.

Non-Deterministic FSM:

The concept of non-determinism plays a central role in both the theory of languages and the theory of computation, and it is useful to understand this notion fully in a very simple context initially. The finite automaton model allows zero, one, or more transitions from a state on the same input symbol. This model is called a *nondeterministic finite automaton(NFA).*

For one input there is one or more output states.

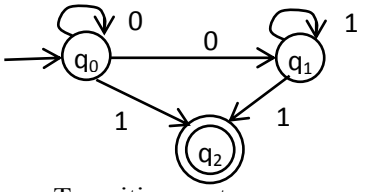


Figure: Transition system representing NDA

If an automaton is in a state $\{q_0\}$ and the input symbol is 0, what will be the next state?, from figre it is clear that next state will be either $\{q_0\}$ or $\{q_1\}$. Some moves of the machine cannot be determined uniquely by the input symbol and the present state. Such machines are called NDA.

Definition: A Non-Deterministic Finite Automata (NDFA) is a 5 tuple $(Q, \Sigma, \delta, q_0, F)$, where

I) Q is a finite non-empty set of states.

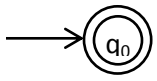
II) Σ is a finite non-empty set of inputs.

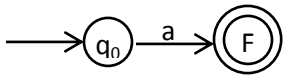
III) δ is the transition function mapping from $Q \times \sum$ into 2^Q which is the power set of Q, the set of all subset of Q.

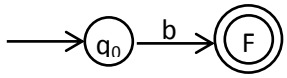
IV) $q_0 \in Q$ is the initial state; and

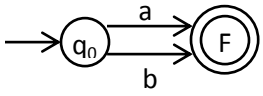
V) $F \subseteq Q$ is the set of final states.

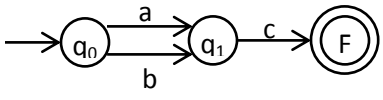
Some rules to generate FA

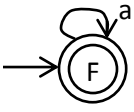
$\in, \wedge$ 

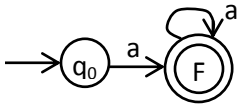
a 

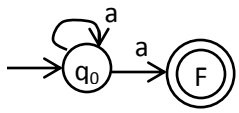
b 

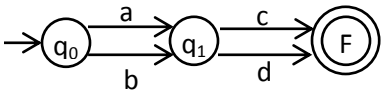
a + b 

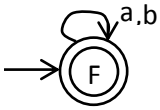
(a+b)c 

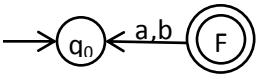
a^* 

aa^* 

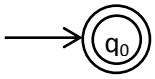
a^*a(NFA) 

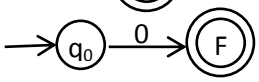
(a+b)(c+d) 

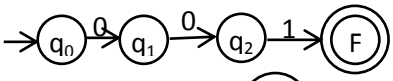
$(a + b)^*$ 

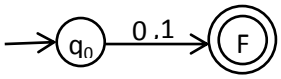
Φ 

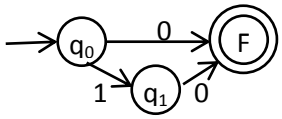
Some regular Expression and their corresponding FA

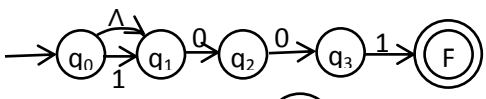
$\in, \wedge$ 

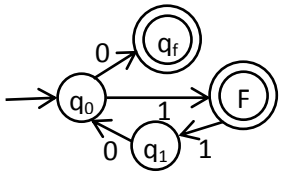
0 

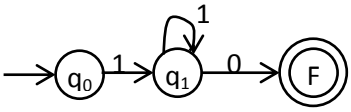
001 

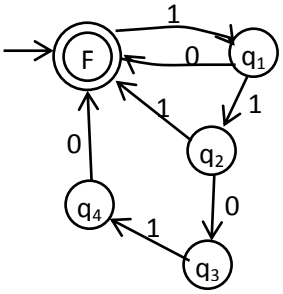
0 + 1 

0 + 10 

(1 + ∧) 001 

$(110)^*$ (0+1) 

1^+0 

(10+111+11010) 

Difference between DFA and NDFA is only in δ. for DFA outcome is a state, i.e. an element of Q; for NDFA, the outcome is a subset of Q.

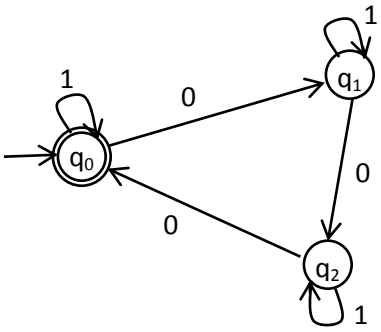
Facts of designing Procedure of FA

we can describe some facts or observation of FA

a. In the designing of the FA, first of all we have to analyze the set of strings i.e. language which is accepted by the FA.
b. Make sure that every state is check for the output state for every input symbol of Σ.
c. Make sure that no state must have two different output states for a single input symbol.
d. Make sure that there is one initial state and at least one final state in transition diagram of FA.

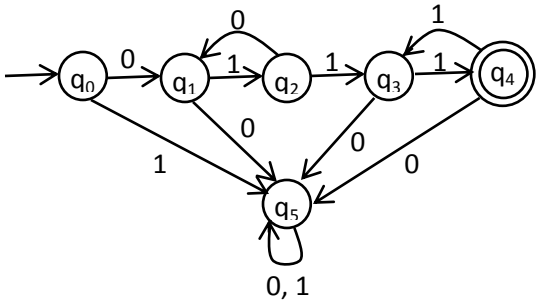
Example: Construct a FA that accepts set strings where the number of 0's in every string is multiple of three over alphabet Σ = {0, 1}.

Solution: Multiple of three means number of 0's on the string may be 0, 3, 6, 9, 12, . . .



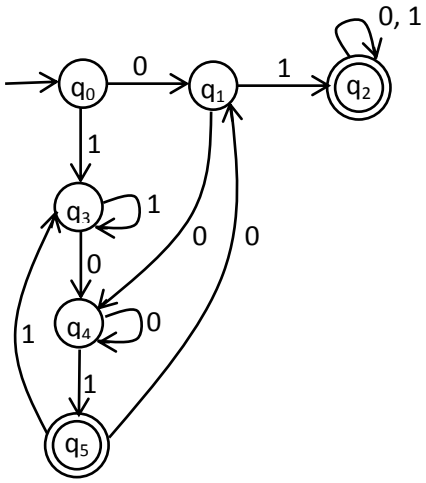
Example: Design FA for the Language
$L = \{(01)^i 1^{2j} \mid i \geq 1, j \geq 1\}$.

Solution: By analysis Language L, it is clear that FA will accepts strings start with any number of 01(not empty) and end with even number of 1's.



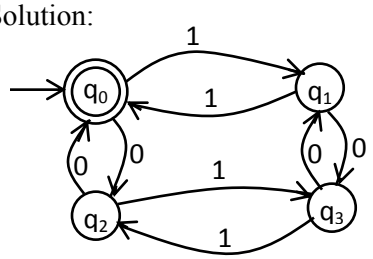
Example: Design a FA over alphabet Σ = {0, 1}, which accepts the set of strings either start with 01 and or end with 01.

Solution: By the analysis of problem, it is clear that FA will accepts the strings such as 01, 01111, 010000, 000101, 0001101,….

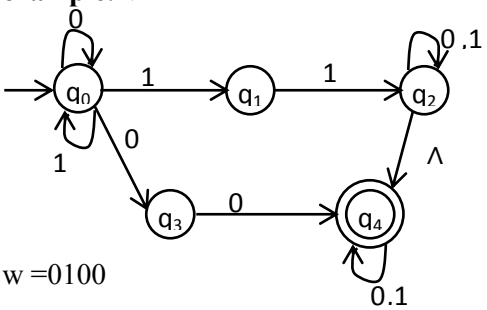


Example: Design a FA which accepts the language L= {w/ w has both an even number of 0's and an even number of 1's over alphabets Σ = {0, 1}}.

Solution:



example:NFA

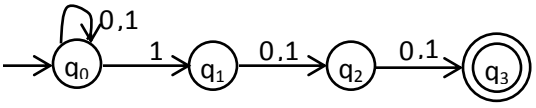


w =0100

$\delta(q_0, 0100) = \{ q_0, q_3, q_4 \}$

since q_4 is an accepting state, the input string will be accepted by NDFA.

Example: NFA



$\delta^*(q_0, 11) = \bigcup_{r \in (q_0,1)} \delta(r, 1)$

$= \bigcup_{r \in (q_0,q_1)} \delta(r, 1)$

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

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

$\delta^*(q_0, 01) = \bigcup_{r \in (q_0,0)} \delta(r, 1)$

$= \bigcup_{r \in \{q_0\}} \delta(r, 1)$

$= \delta(q_0, 1)$

$= \{q_0, q_1\}$

$\delta^*(q_0, 111) = \bigcup_{r \in (q_0,11)} \delta(r, 1)$

$= \bigcup_{r \in \{q_0, q_1, q_2\}} \delta(r, 1)$

$= \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)$

$= \{ q_0, q_1 \} \cup \{ q_2 \} \cup \{ q_3 \}$

$= \{ q_0, q_1, q_2, q_3 \}$

The Equivalence of DFA and NDFA:

or **Prove that for every NFA, there is an corresponding FA which simulate the behavior of NFA. for every string L.**

Solution: Before describing the equivalence of NFA and DFA, let us take a look at the term *equivalence*. The term equivalence means 'equal in some respect'. For example, a BA degree is equivalent to an B.E. degree, as both are bachelor degrees, and for appearing in the civil services examination, either of them is equally applicable. However, both are not equal, as a person with a BA degree cannot be appointed as engineer and a person with a BA degree cannot be appointed as a history teacher.

I. A DFA can simulate the behavior of NDFA by increasing the number of states.

II. Any NDFA is a more general machine without being more powerful.

let $M = (Q, \Sigma, q_0, \delta, F)$ is an NFA.

$M_1 = (Q_1, \Sigma, q_1, \delta_1, F_1)$ is a DFA.

I) $Q_1 = 2^Q$

II) $q_1 = \{ q_0 \}$

III) $\Sigma = \Sigma$

IV) $F_1 = \{ q \in Q_1 \text{ and } q \cap F \neq \emptyset \}$

V) $\delta_1(q, a) = \bigcup_{r \in q} \delta(r, a)$

$\delta_1([q_1, q_2, \dots, q_i], a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \dots \cup \delta(q_i, a)$

equivalently

$\delta_1([q_1, q_2, \dots, q_i], a) = [p_1, p_2, \dots, p_j]$

if and only if

$\delta_1(\{q_1, q_2, \dots, q_i\}, a) = \{p_1, p_2, \dots, p_j\}$

$\delta_1(q_1, x) = [q_1, q_2, \dots, q_i]$

if and only if

$\delta(q_0, x) = \{q_1, q_2, \dots, q_i\}$

Prove by using Mathematical Induction

1. Basis: $|x| = 0$
   $\delta(q_0, \wedge) = \{ q_0 \}$ and
   $\delta_1(q_1, \wedge) = q_1 = \{q_0\}$
   so the equation is true.
2. Assume equation is true for all string y with $|y| = m$
   $\delta_1(q_1, y) = \delta(q_0, y)$
3. let x be a string of length m+1.
   $x = ya$

$\delta_1(q_1, ya) = \delta_1(\delta_1(q_1, y), a)$

$= \delta_1(\delta(q_0, y), a)$

$= \bigcup_{r \in (q_0,y)} \delta(r, a)$

$= \delta(q_0, ya)$

Examples: Construct a deterministic automaton equivalent to $M = ([q_0, q_1], \{0,1\}, \delta, q_0, \{q_0\})$ where δ is

state/ Σ	0	1
→q_0	q_0	q_1
q_1	q_1	q_0q_1

Solution:

i. The states are subset of $\{q_0, q_1\}$ i.e. $\{\Phi\}$, $\{q_0\}$, $\{q_1\}$, $\{q_0, q_1\}$.

ii. $[q_0]$ is initial state.

iii. $[q_0]$ and $[q_0, q_1]$ are final states, both states contain q_0.

iv. Transition table for DFA

state/ Σ	0	1
Φ	Φ	Φ
→[q_0]	[q_0]	[q_1]
[q_1]	[q_1]	[q_0, q_1]
[q_0, q_1]	[q_0, q_1]	[q_0, q_1]

When M has n-states, the corresponding FA has 2^n states. However, we need not to construct δ for all these 2^n states. But only for those states are reachable from $\{q_0\}$, we halt on when no more new states appear under the input.

Example: Find a deterministic acceptor equivalent to M = ($[q_0, q_1, q_2]$, {a, b}, δ, q_0, $\{q_2\}$) where δ is

state/ Σ	0	1
→q_0	q_0, q_1	q_2
q_1	q_0	q_1
(q_2)	—	q_0, q_1

Solution:

M = { 2^Q, {a,b}, δ, $[q_0]$, F' }

F' = { $[q_2]$, $[q_0, q_2]$, $[q_1. q_2]$, $[q_0, q_1, q_2]$ }

state/ Σ	0	1
→[q_0]	[q_0, q_1]	[q_2]
[q_2]	—	[q_0, q_1]
[q_0, q_1]	[q_0, q_1]	[q_1, q_2]
[q_1, q_2]	[q_0]	[q_0, q_1]

Example: Find a deterministic acceptor equivalent to

M = ($\{q_0, q_1, q_2, q_3\}$, {a,b}, δ, q_0, $\{q_3\}$) where δ is

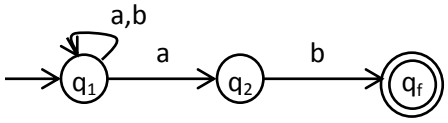
state/ Σ	0	1
→q_0	q_0, q_1	q_0
q_1	q_2	q_1
q_2	q_3	q_3
(q_3)	—	q_2

Sol: $M_1 = (2^Q, \{a,b\}, \delta, [q_0], F)$

F = $[q_3]$, $[q_0. q_3]$, $[q_1, q_3]$, $[q_2, q_3]$, $[q_0, q_1, q_3]$, $[q_0, q_2, q_3]$, $[q_1, q_2, q_3]$ and $[q_0, q_1, q_2, q_3]$

state/ Σ	0	1
→[q_0]	[q_0, q_1]	[q_0]
[q_0,q_1]	[q_0,q_1,q_2]	[q_0, q_1]
[q_0,q_1,q_2]	[q_0, q_1, q_2, q_3]	[q_0,q_1,q_3]
[q_0,q_1,q_3]	[q_0,q_1,q_2]	[q_0,q_1,q_2]
[q_0, q_1, q_2, q_3]	[q_0, q_1, q_2, q_3]	[q_0, q_1, q_2, q_3]

Example: Convert he NFA given in Figure to its equivalent DFA.



Transition table for the NFA in figure

Current State	Input Symbol	
	a	b
→q_1	q_1, q_2	q_1
q_2	—	q_f
(q_f)	—	—

Transition Table for DFA corresponding to DFA

Current State	Input Symbol	
	a	b
→[q_1]	[q_1, q_2]	[q_1]
[q_1, q_2]	[q_1, q_2]	[q_1, q_f]
([q_1, q_f])	[q_1, q_2]	[q_1]

Example: Convert the NFA given in Table to Corresponding DFA.

Transition Table for an NFA

Current State	Input Symbol	
	0	1
→q_1	q_2	q_f
q_2	—	q_3
q_3	q_4	q_3
q_4	q_3, q_f	—
(q_f)	—	q_1

Transition Table for DFA corresponding to DFA

Current State	Input Symbol	
	0	1
→[q_1]	[q_2]	[q_f]
[q_2]	ϕ	[q_3]
[q_3]	[q_4]	[q_3]

$[q_4]$	$[q_3, q_f]$	ϕ
$[q_f]$	ϕ	$[q_1]$
$[q_3, q_f]$	$[q_4]$	$[q_1, q_3]$
$[q_1, q_3]$	$[q_2, q_4]$	$[q_3, q_f]$
$[q_2, q_4]$	$[q_3, q_f]$	$[q_3]$

Example: Convert the NFA fiven in table to its corresponding DFA.

Transition Table for an NFA

Current State	Input Symbol	
	0	1
$\rightarrow q_0$	q_1	q_0, q_2
q_1	q_2	q_0
q_2	q_0	—

Transition Table for DFA corresponding to DFA

Current State	Input Symbol	
	0	1
$\rightarrow [q_0]$	$[q_1]$	$[q_0, q_2]$
$[q_1]$	$[q_2]$	$[q_0]$
$[q_2]$	$[q_0]$	ϕ
$[q_0, q_2]$	$[q_0, q_1]$	$[q_0, q_2]$
$[q_0, q_1]$	$[q_1, q_2]$	$[q_0, q_2]$
$[q_1, q_2]$	$[q_0, q_2]$	$[q_0]$
ϕ	ϕ	ϕ

NFA with ∧-moves :

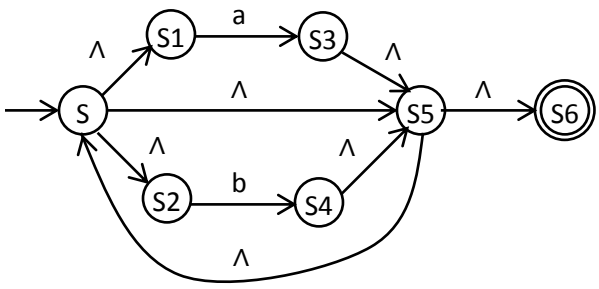


Fig:- NFA with ∧- moves

If there is a NFA M with ∧- moves, then there exists an equivalent DFA M_1 which has equal string recognizing power or if some NFA M with ∧-moves accepts an input string w, then there exists an equivalent DFA M_1 which also accepts w.

Method for constructing equivalent DFA from given NFA with ∧-moves.

1. <u>Using ∈- closure Method</u>

we use two operations ∈ - closure and move().

1. ∈ - closure(q): Set of states which are reachable from state q on ∈ - input including state q. It is equivalent to one state of equivalent DFA.
2. move(q,a) = set of reachable states on input a from state q.

∈ - closure (move (q,a)): Next state from state q on input a (Note: ∈ - closure(∅) = ∅)

Initial state of equivalent DFA is ∈ - closure(), is the initial state of given NFA and final states are those sets, which have atleast one final state of given NFA.

Example: Consider the NFA with ∧ - moves shown in figure.

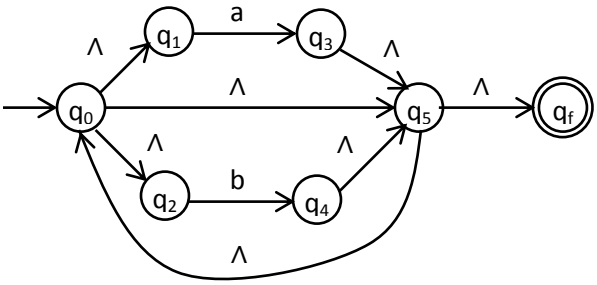


Fig: NFA with ∧ - moves.

Find the equivalent DFA.

Sol: M = (Q, Σ, δ, S, F)

S = ∈ - closure (initial state of NFA)

= ∈ - closure (q_0) (state reachable from q_0 on input ∈ - including q_0)

= { q_0, q_1, q_2 }

δ (S , a) = ∈ - closure (move ({ q_0, q_1, q_2 }, a))

= ∈ - closure (q_3)

= { q_3, q_5, q_f} (next state)

let A = { q_3, q_5, q_f}.

where

δ (S , b) = ∈ - closure (move ({ q_0, q_1, q_2 }, b))

= ∈ - closure (q_4)

= {q_4, q_5, q_f }

let B = {q_4, q_5, q_f }

Here we have two states A & B. so we have to define possible transitions from these states and we continue the process until no next state remain to be considered.

δ (A , a) = $\in$ - closure (move ({ q_3, q_5, q_f}, a))
= $\in$ - closure (move (ϕ)) = ϕ.

δ (A , b) = $\in$ - closure (move ({ q_3, q_5, q_f}, b))
= $\in$ - closure (move (ϕ)) = ϕ.

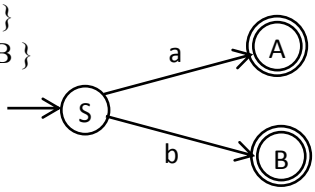
δ (B , a) = $\in$ - closure (move ({q_4, q_5, q_f }, a))
= $\in$ - closure (move (ϕ)) = ϕ.

δ (B , b) = $\in$ - closure (move ({q_4, q_5, q_f }, b))
= $\in$ - closure (move (ϕ)) = ϕ.

Table for DFA

state/Σ	a	b
$\rightarrow$S	A	B
A	ϕ	ϕ
B	ϕ	ϕ

Q = { S, A, B }
Σ = { a, b}
F = { A, B }



Example: Consider the following NFA with $\wedge$ - moves. Construct an equivalent DFA.

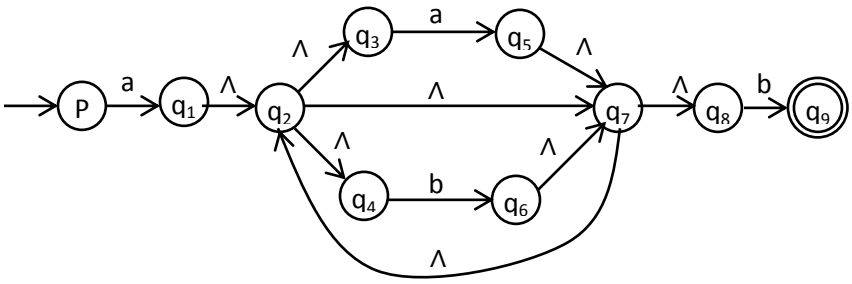


Figure: NFA having $\wedge$ - moves

let M = (Q, Σ, δ, S, F) is an equivalent DFA
S = $\in$ - closure (initial state of NFA)
= $\in$ - closure (P) (state reachable from P on input $\in$ - including P)
= { P } (next state)

δ (S , a) = $\in$ - closure (move ({ P }, a))
= $\in$ - closure (q_1)
= { q_1, q_2, q_3, q_4, q_8} (next state)

let A = { q_1, q_2, q_3, q_4, q_8},where

δ (S , b) = $\in$ - closure (move ({P}, b))
= $\in$ - closure (ϕ) = ϕ.

δ (A , a) = $\in$ - closure (move ({ q_1, q_2, q_3, q_4, q_8}, a)) = $\in$ - closure (q_5)
= {q_2, q_3, q_4, q_5, q_7, q_8} (next state)
let B ={q_2, q_3, q_4, q_5, q_7, q_8}

δ (A , b) = $\in$ - closure (move ({ q_1, q_2, q_3, q_4, q_8}, b)) = $\in$ - closure (q_6, q_f)
= {q_2, q_3, q_4, q_6, q_7, q_8, q_f} (next state)
let C = {q_2, q_3, q_4, q_6, q_7, q_8, q_f}

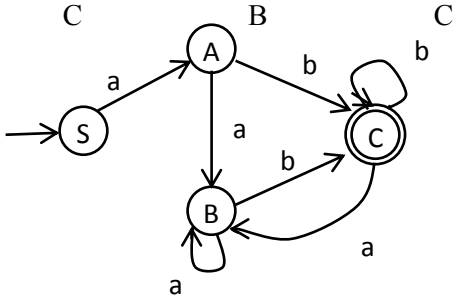
δ (B , a) = $\in$ - closure (move ({q_2, q_3, q_4, q_5, q_7, q_8}, a)) = $\in$ - closure(q_5)
={q_2, q_3, q_4, q_5, q_7, q_8} =B

δ (B , b) = $\in$ - closure (move ({q_2, q_3, q_4, q_5, q_7, q_8}, b)) = $\in$ - closure (q_6, q_f)
= {q_2, q_3, q_4, q_6, q_7, q_8, q_f} = C.

δ (C , a) = $\in$ - closure (move ({q_2, q_3, q_4, q_6, q_7, q_8, q_f}, a)) = $\in$ - closure(q_5)
={q_2, q_3, q_4, q_5, q_7, q_8}=B

δ (C , b) = $\in$ - closure (move ({q_2, q_3, q_4, q_6, q_7, q_8, q_f}, b)) = $\in$ - closure (q_6, q_f)
= {q_2, q_3, q_4, q_6, q_7, q_8, q_f} = C.

state/Σ	a	b
$\rightarrow$S	A	$\emptyset$
A	B	C
B	B	C
C	B	C

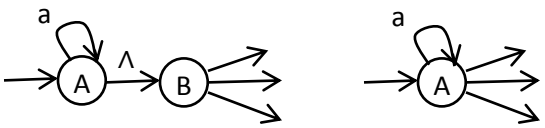


2. Removal of Null Moves

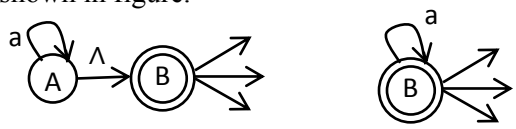
The basic Strategies for the removal of null string are as follows:

Let A and B be the two states connected by a null string $\wedge$.

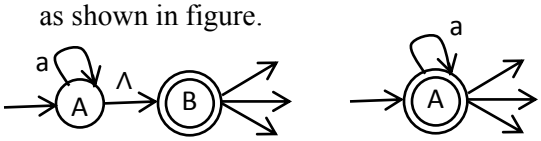
Strategy 1: If A is an initial state, then move A to overlap B with all its connected structures as shown in figure.



Strategy 2: If B is a final state, then move B to overlap A with all its connected structures as shown in figure.



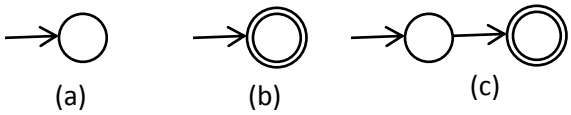
Strategy 3: If A is the initial state and B is the final state, then move A to overlap B with all its connected structures and make A the final state as shown in figure.



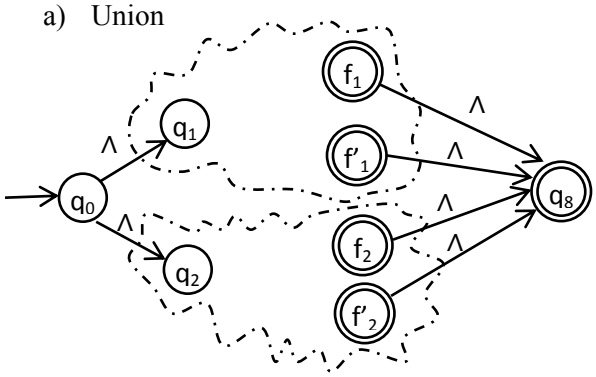
Strategy 4: If both A and B are normal states (neither final nor initial), either can be moved to overlap the other with all its connected structures.

Note: If there is confusion in removing the null moves, remove them using ∈-closure as discussed earlier.

Kleene's Theorem:

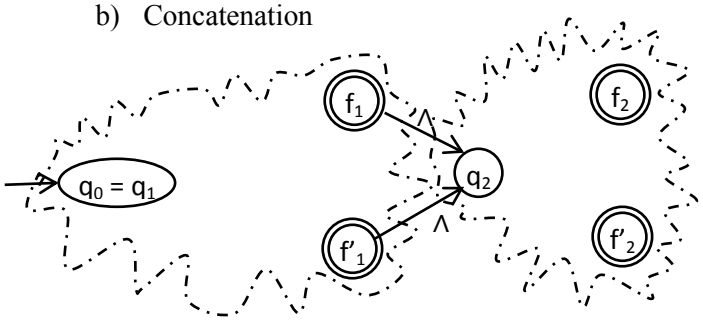


a) Union



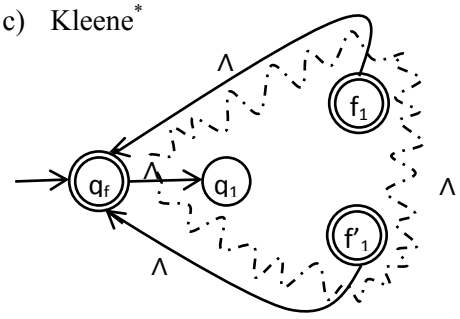
Union(P + Q)

b) Concatenation



Concatenation(PQ)

c) Kleene*



Kleenee* P*

Equivalent of FA to a Regular Set: The method going to give for constructing a finite automaton equivalent to a given regular expression is called subset method which involves two steps.

Step 1: Construct a transition graph equivalent to the regular expression using Λ- moves. This is done by kleene's theorem.

Step 2: Construct the transition table for the transition graph obtained in step 1. Using the method conversion of NFA to DFA & construct the equivalent DFA.

Example:- Construct a finite automaton for the regular language represented by the regular expression $(a + b)^*cd^*e$.

Solution: The following steps details the creation of the finite automaton for the given expression:

Step 1: The regular expression $(a + b)^*$ means the repetition of the symbol a and b any number of times (including zero) and in any order. Thus, the automaton for this part of the regular expression is given by loop that repeats for both a and b at any state as shown in figure i.

Step 2: The finite automaton corresponding to the regular expression cd^* consists of an arc with the c. this arc comes from the previous state of the finite automaton from where cd^* follows sequentially. This arc ends on a state, say A, on which there is a loop to indicate the nuber of repetitions (including zero) of d. The part of the finite automaton corresponding to the regular expression cd^* is shown in figure 2.

Step 3: The finite automaton corresponding to the regular expression e, consists of an arc with label e coming from a previous state and ending into some state, say F. since e is the last part of the total regular expression $(a + b)^*cd^*e$, F has to be a final state. The part of the finite automaton corresponding to the regular expression e is shown in figure 3.

Step 4: The complete finite automaton for the regular expression $(a + b)^*cd^*e$ can be obtained by combining the finite automata in fig 1 -3 serially, as shown in figure 4.

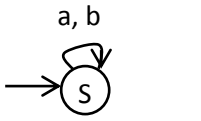


Figure 1

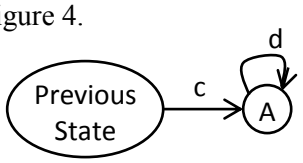


Figure 2

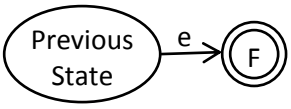


Figure 3

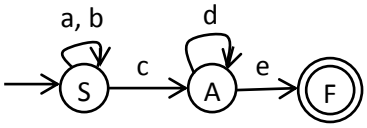
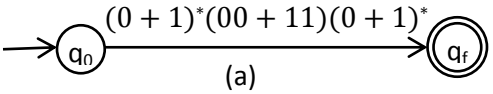


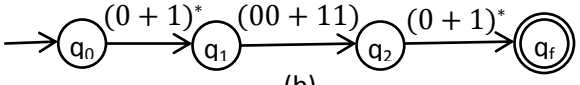
Figure 4

Example: Construct the FA equivalent to the RE $(0 + 1)^*(00 + 11)(0 + 1)^*$

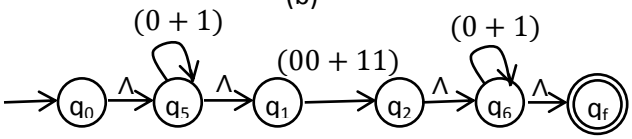
Solution: (Construction of transition graph): First we construct the transition graph with ∧-moves then eliminate ∧-moves.

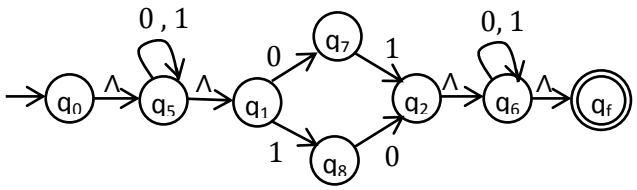


(a)

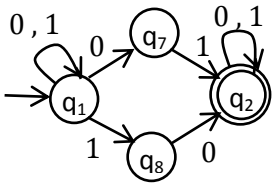


(b)





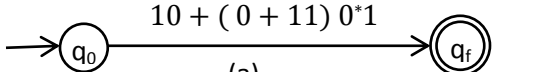
(d)



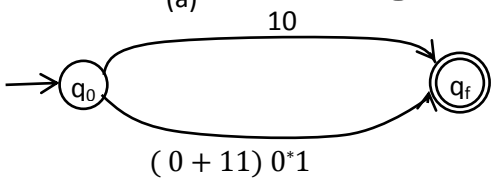
(e)

Example: Construct the FA equivalent to the regular expression $10 + (0 + 11) 0^*1$

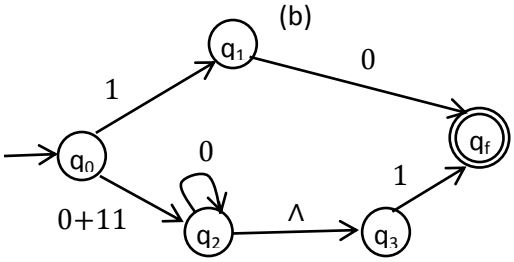
Solution: (Construction of transition graph): First we construct the transition graph with ∧-moves then eliminate ∧-moves.



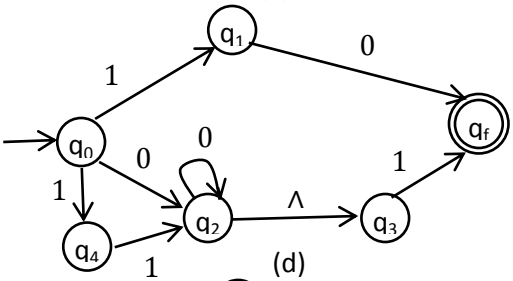
(a)



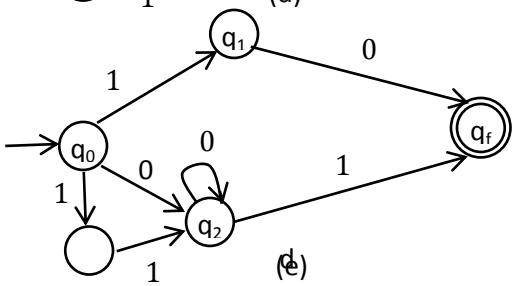
(b)



(c)



(d)



(e)

Example: Design a grammar generating the language L = { wcw^R| w ∈ (a, b)$^+$}.

Solution: (a, b)$^+$ indicates that w contain at least one symbol. The example strings are in the language L are aca, abcba, bbcbb, aacaa, abbcbba, . . . Thus productions P are

S ⟶ aSa | bSb | aXa| bXb

X ⟶ c

Example: Design a grammar generating the language L = {a^nb^n | n ≥ 1 }.

Solution: Every string contains equal number of a's and b's. Every string w in L contains a substring of a's followed by a substring of equal number of b's. The example strings in the language L are ab, aabb, aaaabbbb, aaabbb, ... Thus productions P are

S ⟶ aSb | ab

Example: Design a grammar generating the language L = { $a^nc^mb^n$ | m, n ≥ 1 }.

Solution: every string w belonging to the language begins with a substring containing a symbols. this follows a substring of c symbols. the substring of c's follows the substring of b symbols and the number of b's is the same as that of a's. the example strings in the language are acb, accb, aacccbbb, aaaccbbb, ... Thus, we have the set of productions P as follows:

S ⟶ aSb | aXb

X⟶ cX | c.

Example: Design a grammar generating the language L = { a^nb^m |m, n ≥ 1 }.

Solution: every string w belonging to the language begins with a substring containing a symbols. This follows substring of b symbols and the number of b's are independent of a's. The example strings in the language L are ab, aab, abb, aabbbb, abbb, abbb, etc. Thus the productions P are:

S ⟶ AB

A ⟶ aA | a

B ⟶ bB | b

Example: Design a grammar generating the language L = { $a^nb^nc^n$ | n ≥ 1 }.

Solution: Every string w belonging to the language begins with a substring containing a symbols. This follows a substring of equal number of b symbols, which further follows the substring of same number of c symbols. The example strings in the language L are abc, aaabbbccc, aabbcc, aaaabbbbcccc, . . . etc. Thus the productions P are:

S ⟶ aSXc | abc

cX ⟶ Xc

bX ⟶ bb

it can be seen that every string in the language L has the length 3n.

Example: Design a grammar generating the language L = { ww | w ∈ (a, b)$^+$ }

Solution: we see that each string in the language L is of the length 2n (n ≥ 1). The first substring of length n is the same as that of the subsequent substring of length n. The example strings in the language L are aa, abab, bbbb, baba, bbabba, babbab, ... thus productions P are :

S ⟶ XYZ

XY ⟶ aXA | bXB

AZ ⟶ YaZ

BZ ⟶ YbZ

Aa ⟶ aA

Ab ⟶ bA

Ba ⟶ aB

Bb ⟶ bB

aY ⟶ Ya

bY ⟶ Yb

XY ⟶ ∧

Z ⟶ ∧

Example: Design a grammar for a language of all palindromes over {a, b}.

Solution: The basis for our grammar is as follows:

Note: Final state is also known as Accepting state.

An alphabet a or b is a palindrome. If a string x is palindrome then the strings axa and bxb are also palindromes. thus the grammar can be designed as follows:

S ⟶ aSa | bSb | a | b | aa | bb.

Example: Design a grammar for a language L over {a, b} such that each string in L contains equal number of a's and b's.

Solution: The string can start with either a or b. Thus, we have the production

S ⟶ aX | bY

Now the design of X should be such that it inserts one b more than a in the output string. Similarly, design of Y should be such that it inserts one a more than b in the output string, Thus, we have

X ⟶ b | aXX | bS

Y ⟶ a | bYY | aS

Questions: **Find the regular expression corresponding to following:**

1. a, b ∈ Σ starting from (abb)
   - $abb\ (a + b)^*$
2. a, b ∈ Σ ending with (baa)
   - $(a + b)^*\ baa$
3. a, b ∈ Σ ending with (baa) and starting with (abb).
   - $abb\ (a + b)^*\ baa$
4. a, b ∈ Σ containing exactly 2a's
   - $b^*ab^*ab^*$
5. a, b ∈ Σ containing atmost 2a's
   - $b^* + b^*ab^* + b^*ab^*ab^*$
6. a, b ∈ Σ containing atleast 2a's
   - $b^*a\ (a + b)^*\ a\ (a + b)^*$
7. a, b ∈ Σ containing abb as substring
   - $(a + b)^*\ abb\ (a + b)^*$
8. String of length 2.
   - $(a + b)\ (a + b)$ or $(a + b)^2$
9. String of length 6.
   - $(a+b)(a+b)(a+b)(a+b)(a+b)(a+b)$
10. String of length 2 or less.
    - $(\wedge + a + b)\ (\wedge + a + b)$
11. String of even length
    - $(aa + ab + ba + bb)^*$
12. String with odd number of 1's
    - $0^*\ (10^*10^*)^*\ 10^*$
13. String of length of 6 or less.
    - $(\wedge + 0 + 1)^6$
14. Strings ending with 1 and not containing 00
    - $(1 + 01)^+$
15. Language of C identifiers.
    - $(l + \_)\ (l + d + \_)^*$
16. Real Literals in Pascal
    - $sd^+(pd^+ + pd^+Esd^+ + Esd^+)$
17. Strings Containing exactly 2 0's
    - $1^*01^*01^*$
18. String containing atleast 2 0's.
    - $1^*0\ (1 + 0)^*\ 0\ (1 + 0)^*$
19. Strings that do not end with 01.
    - $\wedge + 1 + (0 + 1)^*0 + (0 + 1)^*11$
20. Strings that begin or end with 00 or 11.
    - $(00 + 11)\ (0 + 1)^* + (0 + 1)^*\ (00 + 11)$
21. Strings not containing the substring 00
    - $(\wedge + 0)\ (1 + 10)^*$ or $(1 + 10)^*\ (\wedge + 0)$
22. Strings in which the number of 0's is even
    - $1^*\ (01^*01^*)^*$
23. Strings containing no more than one occurrence of the string 00.
    - $(1 + 01)^*\ (\wedge + 0 + 00)\ (1 + 10)^*$
24. Strings in which every 0 is followed immediately by 11.
    - $1^*\ (011^+)^*$
25. { 0, 1, 2}
    - $0 + 1 + 2$
26. $\{\ 1^{2n+1} \mid n > 0\ \}$
    - $(11)^+1$
27. $\{\ w \in \{\ a, b\ \}^* \mid w$ has only one a $\}$
    - b^*ab^*
28. Set of all strings over { 0, 1} which has atmost 2 0's.
    - $1^+ + 1^*01^* + 1^*01^*01^*$
29. $\{\ a^2, a^5, a^8, \ldots\ \}$
    - $(aaa)^*aa$
30. $\{\ a^n \mid n$ is divisible by 2 or 3 or n = 5 $\}$
    - $(aa)^* + (aaa)^* + aaaaa$
31. Strings beginning and ending with a.

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

32. Strings having atmost one pair 0's or atmost one pair of 1's.
    - $(1 + 01)^* + (1 + 01)^* 00 (1 + 10)^* + (0 + 10)^* + (0 + 10)^* 11(0 + 01)^*$
33. String in which number of occurrences of a is divisible by 3.
    - $(b^* a b^* a b^* a b^*)^*$
34. String with 3 consecutive
    - $(a + b)^* bbb (a + b)^*$
35. Strings beginning with 00
    - $00 (0 + 1)^*$
36. String ending with 00 and beginning with 1.
    - $1 (0 + 1)^* 00$

Difference between DFA and NDFA

Sr.	DFA	NDFA
1.	Deterministic Finite Automata	Non-Deterministic Finite Automata
2.	Every transition is unique and deterministic in nature.	Multiple transition for an input on a state are possible, which means moves are non-deterministic in nature.
3.	Null-transitions are not allowed	Null-transitions are allowed.
4.	Requires less memory as transition diagram needs less number of states.	Requires more memory as transition diagrams needs more number of states.
5.	Range of transition is $\delta: Q \times \Sigma \longrightarrow Q$.	Range of transition is $\delta: Q \times \Sigma \longrightarrow 2^Q$
6.	All DFA's are NDFA's but vice versa is not true.	All DFA are NDFA.

Applications of Finite Automaton.

1. Lexical Analyzers: Lexical analysis is a part of compilation and used to recognize the validity of input programs. Whether the input program is grammatically constructed or not.
2. Text Editor: smart and speedy text editors can be constructed using FA.
3. Spell checkers: spell checkers can be designed using FA. In our computer system, we have a dictionary, FA can recognize the correctness of a word and can give the appropriate suggestion to the users also.
4. Sequential Circuits: FA can also be used to design sequential circuits.
5. Image Processing.
6. Natural Language Processing.
7. Internet Browsers: To scan large body of information.
8. Communication: Designing of protocols for exchange of information.
9. Pattern matching.
10. Forensic Science
11. Cellular machines uses cellular Automata

Limitations of Finite Automaton:

FA is most restricted model of automatic machines. It is an abstract model of a computer system. a FA has following limitations:

1. The memory is limited.
2. Memory is read-only memory.
3. Memory is sequentially accessed.
4. FA has only string recognizing power.
5. Head movement is restricted in one direction, either from left to right or from right to left.
6. Limited Computability mean it can act like a scanner FA are not computing devices but these act only as scanner.
7. Periodicity: FA cannot remember large amount of information or strings of large size.
8. Impossibility of Multiplications: Full length of multiplier and multiplicand cannot be stored.
9. Binary Response: Response of FA is binary either accepted or rejected after a long sequence of moves.

Moore and Mealy Machines: Finite automata with output: The finite automata which we were considered in the earlier have binary output i.e. either they accept the string or they do not accept the string. This acceptability was decided on the basis of the reachability of final state by the initial state. Now, we remove this restriction and consider the model where the outputs can be chosen from some other alphabets.

The value of the output function Z(t) in the most general case is a function of the present state q(t) and present input x(t), i.e.

$Z(t) = \lambda(q(t), x(t))$

Where λ is called the output function. This generalized model is usually called the mealy machine.

if the output function Z(t) depends only on the present state and is independent of the current input, the output function may be written as

$Z(T) = \lambda(q(t))$

this restricted model is called the Moore machine.

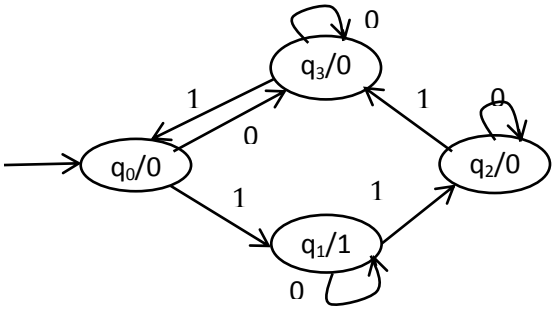
Definition: A Moore machine has 6-tuple

$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

I. Q is a finite set of states.
II. Σ is the input alphabet,
III. Δ is the output alphabet,
IV. δ is the transition function $\Sigma \times Q$ into Q.
V. λ is the output function mapping Q into Δ.
VI. q_0 is initial state.

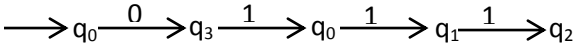
Example: Moore Machine

Present state	Next State a = 0	Next State a = 1	Output (λ)
$\longrightarrow q_0$	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q_2	q_3	0
q_3	q_3	q_0	0



Transition graph

input string ⟶ 0111



output ⟶ 00010

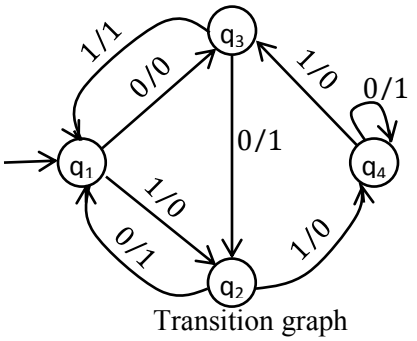
Definition: A Mealy machine has 6-tuple

$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

I. Q is a finite set of states.
II. Δ is the output alphabet,
III. Σ is the input alphabet,
IV. δ is the transition function $\Sigma \times Q$ into Q.
V. λ is the output function mapping $\Sigma \times Q$ into Δ.
VI. q_0 is initial state.

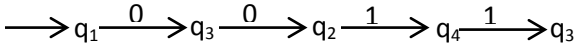
Example: Mealy Machine

Present State	Next State a = 0		Next State a = 1	
	state	output	state	output
$\longrightarrow q_1$	q_3	0	q_2	0
q_2	q_1	1	q_4	0
q_3	q_2	1	q_1	1
q_4	q_4	1	q_3	0



Transition graph

Input string : 0011



Output String = 0100

Note: for a Moore machine, if the input string is of length n, the output string is of length n+1. The first output is $\lambda(q_0)$ for all output strings. In the case of a Mealy machine if the input string is of length n, the output string is also same length.

Convert the Following Moore Machine to Mealy Machine.

Present State	Next State a = 0	Next State a = 1	Output (λ)
$\longrightarrow q_0$	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q_2	q_3	0
q_3	q_3	q_0	0

Present State	a = 0 Next	a = 0 O/p	a = 1 Next	a = 1 O/p
$\longrightarrow q_0$	q_3	0	q_1	1
q_1	q_1	1	q_2	0
q_2	q_2	0	$q_{3=}$	0
q_3	q_3	0	q_0	0

Convert the Following Moore Machine to Mealy Machine.

Present State	Next State a = 0	Next State a = 1	Output (λ)
$\longrightarrow q_1$	q_1	q_2	0
q_2	q_1	q_3	0
q_3	q_1	q_3	1

Present State	a = 0 Next	a = 0 O/p	a = 1 Next	a = 1 O/p
$\longrightarrow q_1$	q_1	0	q_2	0
q_2	q_2	0	q_3	1
q_3	q_3	0	q_0	1

Convert the Following Mealy Machine to Moore Machine.

Present State	a = 0 Next	a = 0 O/p	a = 1 Next	a = 1 O/p
$\longrightarrow q_1$	q_3	0	q_2	0
q_2	q_1	1	q_4	0
q_3	q_2	1	q_1	1
q_4	q_4	1	q_3	0

Present State	a = 0 Next	a = 0 O/p	a = 1 Next	a = 1 O/p
$\longrightarrow q_1$	q_3	0	q_{20}	0
q_{20}	q_1	1	q_{40}	0
q_{21}	q_1	1	q_{41}	1
q_3	q_{21}	1	q_1	1
q_{40}	q_{41}	1	q_3	0
q_{41}	q_{41}	1	q_3	0

Present State	Next State a = 0	Next State a = 1	Output (λ)
$\longrightarrow q_1$	q_3	q_{20}	1
q_{20}	q_1	q_{40}	0
q_{21}	q_1	q_{41}	1
q_3	q_{21}	q_1	0
q_{40}	q_{41}	q_3	0
q_{41}	q_{41}	q_3	1

Convert the Following Mealy Machine to Moore Machine.

Present State	a = 0 Next	a = 0 O/p	a = 1 Next	a = 1 O/p
$\longrightarrow q_1$	q_2	z_1	q_3	z_1
q_2	q_2	z_2	q_3	z_1
q_3	q_2	z_1	q_3	z_2

Present State	a = 0 Next	a = 0 O/p	a = 1 Next	a = 1 O/p
$\longrightarrow q_1$	q_{21}	z_1	q_{31}	z_1
q_{21}	q_{22}	z_2	q_{31}	z_1
q_{22}	q_{22}	z_2	q_{31}	z_1
q_3	q_{21}	z_1	q_{32}	z_2
q_3	q_{21}	z_1	q_{32}	z_2

Present State	Next State a = 0	Next State a = 1	Output (λ)
$\longrightarrow q_1$	q_{21}	q_{31}	Λ
q_{21}	q_{22}	q_{31}	z_1
q_{22}	q_{22}	q_{31}	z_2
q_3	q_{21}	q_{32}	z_1
q_3	q_{21}	q_{32}	z_2

Question: Design a Moore machine which counts the occurrence of substring aab in input string.

Solution: Let the Moore machine be

$M_0 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$

$Q = \{ q_0, q_1, q_2, q_3 \}$

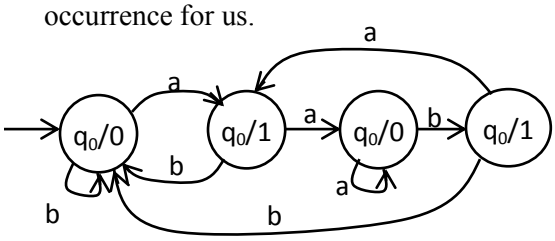
$\Sigma = \{ a, b\}$

We will design this machine in such a way that machine prints out the character 0 except for q_3, which prints a 1. To get q_3, we must come from state q_2 and have 1st read b. To get state q_2, we must read atleast two a's in a row, having started in any state.

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

$\lambda'(q_0) = 0,\ \lambda'(q_1) = 0,\ \lambda'(q_2) = 0,\ \lambda'(q_3) = 1$

The following machine will counts aab occurrence for us.



Question: Construct a mealy machine which calculates the residue mod-4 for each binary string treated as binary integer.

Solution: When we divide any number by 4 then remainder can be 0, 1, 2, &3. so clearly mealy machine will have four states.

Let the Mealy machine be

$M_0 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$

$Q = \{ q_1, q_2, q_3, q_4 \}$

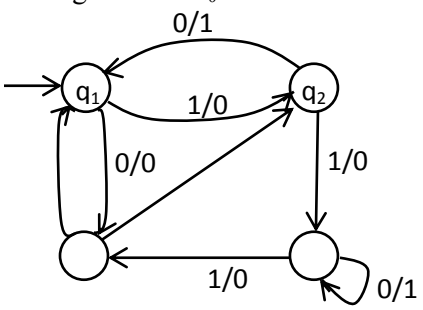
$\Sigma = \{ a, b\}$

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

q_1 is the initial state & δ is transition function.

$\lambda'(q_1, 0) = 0$	$\lambda' (q_1, 1) = 0$
$\lambda'(q_2, 0) = 1$	$\lambda' (q_2, 1) = 0$
$\lambda'(q_3, 0) = 1$	$\lambda' (q_3, 1) = 1$
$\lambda'(q_4, 0) = 1$	$\lambda' (q_4, 1) = 0$

Transition Diagram for M_0 is as follows:



By the analyzing mealy machine transition diagram, we can easily notice that it is also finite automata without any final state and output is there for input-symbol on corresponding transition. Output for every input symbol is represented as level on each transition corresponding output. For e.q. in between q_1 and q_2 is labeled by 0/1, 0 is input symbol and 1 is the output symbol.

Question: Design a mealy machine, which prints 1's complement of input bit string over alphabet $\Sigma = \{ 0, 1\}$.

Solution: If input string is 101110 then 1's complements of 101110 will be 010001, so we have to design a mealy machine which will print output 010001 for the input string 10110.

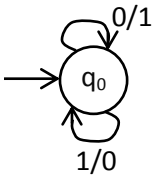
$M_0 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$

$Q = \{ q_0 \}$

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

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

q_0 is the initial state.



Question: Construct a mealy machine which calculate residue mod-4 for each binary string treated as binary integer.

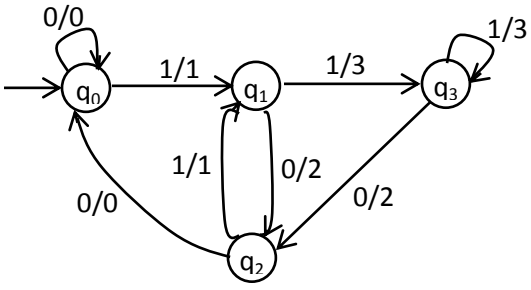
Solution:

$M_0 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$

$Q = \{ q_0, q_1, q_2, q_3 \}$

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

$\Delta = \{ 0, 1, 2, 3\}$



Arden's Theorem: let P and Q be two regular expressions over Σ. If P does not contain ∧, then the following equation in R, is

$$R = Q + RP \text{ ---(i)}$$

has a unique solution QP^*---(ii)

Proof: let us First check if QP^* is a solution to (i). To check this, let us substitute R by QP^* on both sides of (i)

$R = Q + RP = Q + QP^*P = Q(\wedge + P^*P) = QP^*$

Hence eqn(i) is satisfied when $R = QP^*$. This means $R = QP^*$ is a solution of eqn(i).

Let us check if it is the only solution. To do this, replace R by Q + RP on the Right Hand Side. of eqn(ii), we get

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

$= Q + QP + RP^2$

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

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

. . .

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

$= Q(\wedge + P + P^2 + \ldots + P^i) + RP^{i+1}$

We now show that any solution of equation (i) is equivalent to QP^*.

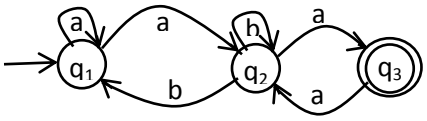
Let w be the string of length I in the set R. then w belongs to the set $Q(\wedge + P + P^2 + \ldots + P^i) + RP^{i+1}$. As P does not contain ∧, RP^{i+1} has no string less than i+1. This means that w belongs to the set RP^{i+1}. This means that w belongs to the set $Q(\wedge + P + P^2 + \ldots + P^i)$ and hence to QP^*.

Rules for using Arden's Theorem:

1. The transition graph does n't have ∧- moves.
2. It has only one initial state.
3. V_i the regular expression (of final state) represents the set of strings accepted by the system (V_i is final state).
4. $q = \Sigma\, q_i$ (incoming edge) alphabets + ∧ (only for initial state).

Conversion of FA to Regular Expression.

Example: The transition system is given in figure.



Prove that the strings recognized are

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

we can directly apply the method since the graph does not contain any ∧-move and there is only one initial state.

$q_1 = q_1a + q_2b + \wedge$

$q_2 = q_1a + q_2b + q_3a$

$q_3 = q_2a$

reduce the number of unknowns by repeated substitution.

$q_2 = q_1a + q_2b + q_3a$

$q_2 = q_1a + q_2b + q_2aa$

$q_2 = q_1a + q_2(b + aa)$

$R = Q + R\,P \longleftrightarrow QP^*$

$q_2 = q_1a(b + aa)^*$

Now substitute q_2 in q_1

$q_1 = q_1a + q_2b + \wedge$

$q_1 = q_1a + q_1a(b + aa)^*b + \wedge$

$q_1 = q_1(a + a(b + aa)^* + \wedge$

$R = R\,P + Q \longleftrightarrow QP^*$

$q_1 = \wedge.(a + a(b + aa)^*)^*$

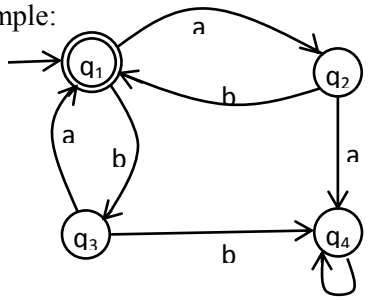
$= (a + a(b + aa)^*)^*$

$q_2 = (a + a(b + aa)^*)^* a(b + aa)^*$

$q_3 = (a + a(b + aa)^*)^* a(b + aa)^* a$

Since q_3 is a final state, the set of strings recognized by the graph.

Example:



We can directly apply the method since the graph does not contain any ∧-move and there is only one initial state.

$q_1 = q_2b + q_3a + \wedge$

$q_2 = q_1a$

$q_3 = q_1b$

$q_4 = q_2a + q_3b + q_4a + q_4b$

$q_1 = q_2b + q_3b + \wedge$

$q_1 = q_1ab + q_1ba + \wedge$

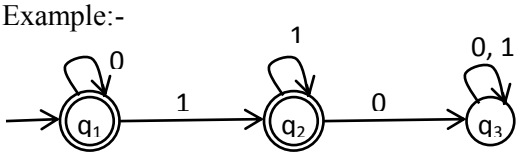
$q_1 = q_1 (ab + ba) + \wedge$

$R = R\ P + Q \longleftrightarrow QP^*$

$q_1 = \wedge .(ab + ba)^*$

$q_1 = (ab + ba)^*$

Example:-



We can directly apply the method since the graph does not contain any ∧-move and there is only one initial state.

$q_1 = q_10 + \wedge$

$q_2 = q_11 + q_21$

$q_3 = q_20 + q_3(0 + 1)$

$q_1 = q_10 + \wedge$

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

$q_1 = \wedge.(0)^* = 0^*$

$q_2 = q_11 + q_21$

$q_2 = 0^*1 + q_21$

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

$q_2 = 0^*11^*$

As the final states are q_1 and q_2, we need not solve q_3.

$q_1 + q_2$

$0^* + 0^*11^*$

$0^* (\wedge + 11^*) = 0^*1^*$

Pumping Leema for regular Languages/Regular Sets:

Pumping Lemma: let M = (Q, Σ, δ, q_0, F) be a finite automaton with n states. let L be the regular set accepted by M. let x ∈ L and |x|≥ n, then there exists μνω such that x = μνω, ν ≠ ∧ and $\mu\nu^m\omega \in L$ for each m ≥ 0.

Proof: suppose that the set Q has n elements. for any string x in L with length atleast n. of we write $x = a_1a_2\ldots a_n$;

then the sequence of n+1 states.

$q_0 = \delta^* (q_0 , \wedge)$

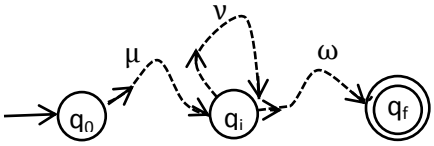
$q_1 = \delta^* (q_0 , a_1)$

$q_2 = \delta^* (q_0 , a_1a_2)$

$q_3 = \delta^* (q_0 , a_1a_2a_3)$

. . .

$q_n = \delta^* (q_0 , a_1a_2. . . a_n)$



suppose $q_i = q_{i+p}$

$0 \le i \le i + p \le n$

then

$\delta^* (q_0 , a_1a_2. . . a_i) = q_i$

$\delta^* (q_i , a_{i+1}a_{i+2}. . . a_{i+p}) = q_{i+p} = q_i$

$\delta^* (q_i , a_{i+p+1}a_{i+p+2}. . . a_n) = q_f \in F$

To simplify the notation,

let $\mu = a_1a_2. . . a_i$

$\nu = a_{i+1}a_{i+2}. . . a_{i+p}$

$\omega = a_{i+p+1}a_{i+p+2}. . . a_n$

$$\begin{bmatrix} \text{if } i = 0 \ \ \mu \text{ will be } \wedge \\ \text{if } i + p = 0 \ \ \omega \text{ will be } \wedge \end{bmatrix}$$

since $\delta^* (q_i , \nu) = q_i$, have $\delta^* (q_i , \nu^m) = q_i$ for every m ≥ 0 and it follows that

$\delta^* (q_0 , \mu\nu^m\omega) = q_f$ for every m ≥ 0 since p > 0 and i + p ≤ n.

Note: The decomposition is valid only for strings of length greater than or equal to the number of

states. for such a string x = μνω as many times as we like and get strings of the form $\mu\nu^m\omega$ which are longer than μνω and are in L. by considering the path from q_0 to q_i and then the path from q_i to q_f (without going through the loop), we get a path ending in a final state with path value μω (this corresponds to the case when m = 0).

Application of Pumping Lemma:

This problem can be used to prove that certain sets are not regular. we now give the steps needed for proving that a given set is not regular.

Step 1: Assume that L is regular. Let n be the number of states in the corresponding FA.

Step 2: Choose a string x such that |x| ≥ n, use pumping lemma to write x = μνω with |μν|≤ n and |ν| > 0.

Step 3: Find a suitable integer m such that $\mu\nu^m\omega$ ∈ L. this contradicts our assumption Hence L is not regular.

Example: Show that the set L = { a^{i^2} | I ≥ 1 } is not regular.

Sol: L = {∧, a^1, a^4, a^9, a^{16}, . . . }

= { ∧, a, aaaa, aaaaaaaaa, . . . }

Step 1: Suppose L is regular. let n be the number of states in finite automaton accepting L.

Step 2: let x = a^{n^2} . then |x| = n^2 > n. by pumping lemma, we can write x= μνω with |μν|≤ n and |ν| > 0.

Step 3: Consider $\mu\nu^2\omega$.

$|\mu\nu^2\omega| = |\mu| + 2|\nu| + |\omega| > |\mu| + |\nu| + |\omega|$ as |ν| > 0.

this means

$n^2 = |\mu\nu\omega| = |\mu| + |\nu| + |\omega| < |\mu\nu^2\omega|$

as |μν|≤ n we have |ν|≤ n. therefore

$|\mu\nu^2\omega| = |\mu| + 2|\nu| + |\omega| \le n^2 + n$.

i.e.

$n^2 < |\mu\nu^2\omega| \le n^2 + n < n^2 + n + n + 1$

$n^2 < |\mu\nu^2\omega| < (n + 1)^2$

Hence, $|\mu\nu^2\omega|$ strictly lies between n^2 and $(n + 1)^2$, but is not equal to any one of them. thus, $|\mu\nu^2\omega|$ is not a perfect square and so $\mu\nu^2\omega \notin L$. but by pumping lemma, $\mu\nu^2\omega \in L$. this is contradiction. thus L is not regular.

Example: Show that L = { a^Q | Q is a prime} is not a regular.

Solution: Step 1: We suppose L is a regular. let n be the number of states in the finite automaton accepting L.

Step 2: Let Q be a prime number greater than n. let x = a^Q. by pumping lemma, x be written as x = μνω, with |μν|≤ n and |ν| > 0. μ , ν, ω are simply strings of a's. so ν = a^p for some p ≥ 1 (and ≤ n).

Step 3: let m = Q+1, the $|\mu\nu^m\omega|$

$= |\mu\nu\omega| + |\nu^{m-1}| = Q + (m-1)p = Q + QP = Q(1+p)$.

by pumping lemma, $\mu\nu^m\omega \in L$. but $|\mu\nu^m\omega| = Q + Qp = Q(1+p)$ and Q(1+p) is not prime. so $\mu\nu^m\omega \notin L$. this is contradiction. thus L is not regular.

Example: Show that L = { 0^i1^i | i≥1} is not regular.

Solution: Step 1: Suppose L is regular. Let n be the number of states in the finite automaton accepting L.

Step 2: let x = 0^n1^n. then |x| = 2n > n. by pumping lemma, we write x = μνω with |μν|≤ n and |ν| > 0.

Step 3: we want to find m, s the $\mu\nu^m\omega \in L$ for getting a contradiction. the string ν can be in any of the following:

case 1: ν has only 0's i.e. ν = 0^k for some k≥1.

case 2: ν has only 1's i.e. ν = 1^l for some l≥1.

case 3: ν has both 0's and 1's i.e. ν = 0^k1^l for some k, l≥1.

in case 1, we can take m=0, as μνω = 0^n1^n, μω= $0^{n-k}1^n$. as k≥1, n-k≠n so μω ∉ L.

In case 2, we can take m =0, as μνω = 0^n1^n, μω= 0^n1^{n-l}. as l≥1, n-l≠n so μν∉ L.

In case 3, we can take m =2, as $\mu\nu\omega = 0^{n-k}0^k1^l1^{n-l}$. $\mu\nu^2\omega = 0^{n-k}0^k1^l0^k1^l1^{n-l}$, as $\mu\nu^2\omega$ is not of the form 0^n1^n, $\mu\nu^2\omega \notin$ L. thus in all the cases we get a contradiction. Thus L is not regular.

Example: Show that L = {zz | z∈{a, b}*} is not regular.

Sol: Step 1: suppose L is regular, let n be the number of states in the automaton M accepting L.

Step 2: let us consider zz = a^nba^nb in L |zz| = 2(n+1) > n. we can apply pumping lemma to write zz = μνω with |μν|≤ n and |ν| > 0.

Step 3: We want to find m so that $\mu\nu^m\omega \in$ L for getting a contradiction. the string ν can be in only one of the following form:

case 1: ν has no b's i.e. y =a^k for some k≥1.

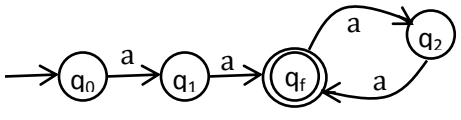
case 2: ν has only one b.

we may note that ν can not have two b's if so |ν| ≥ n+2. but |ν| ≤ |μν| ≤ n.

in case 1, we can take m =0. then $\mu\nu^0\omega = \mu\omega$ is of the for $a^{n-k}ba^nb$, here n-k ≠ n, we can not write μω in the form so $\mu\omega \notin$ L.

In case 2, we can take m = 0. then $\mu\nu^0\omega = \mu\omega$ has only one b (as one b is removed from μνω, b being in ν). so $\mu\omega \notin$ L as any element in L should have an even number of a's and an even number of b's. thus in both case, we get a contradiction. therefore, L is not regular.

Example: Is L = {a^{2n} | n ≥ 1} regular?
Solution: We can write a^{2n} as $aa(a^2)^m$, where m ≥ 0 } is simply {a^2 }* so L is represented by the regular expression $aa(P)^*$, where P represents {a^2}. the corresponding FA is shown in figure:



Pumping lemma fails to solve this problem because this is a regular grammar.

Closure properties of Regular sets:

The term 'closure property' relates to a mathematical system (structure) containing a set and operations performed on the elements of the set. For example, let I^+ be the set of positive integers and addition (+) and subtraction (-) be the two operations performed on the elements of this set. When the addition operation is performed on the elements of this set the result obtained is some positive integer and belongs I^+. When the subtraction operation is performed, the result obtained may or may not belongs to I^+. Thus the execution of the addition operation does not violate the boundary of the set I^+ whereas subtraction may do so. We say that the set I^+ is closed over the addition operation but not over subtraction. Similarly, a set of all 3 × 3matrices is closed over the operation of addition, subtraction and transpose, but not over the operation of the determinant.

Every regular set can be expressed by its corresponding regular expression. In addition, for every regular expression, there exists a corresponding regular set. If an operation performed on regular set(s) leads to another set, which is also regular, then the regular set(s) are closed over that operation.

The regular sets are closed over the operations of union, concatenation, transpose, complement, intersection, difference and Kleene's star.

1. Regular sets are closed over the union operation.
   Proof: let S_1 and S_2 be two regular sets associated with regular expressions R_1 and R_2 respectively. Let there be a set $S_3 = (S_1 \cup S_2)$. then every element of S_3 can be expressed either by expression R_1, R_2 or both. Thus, the set S_3 can be generated by the regular expression $(R_1 + R_2)$. The association of S_3 with regular sets has generated a regular set, regular sets are closed over the Union operation.
2. Regular sets are closed over the concatenation operation.

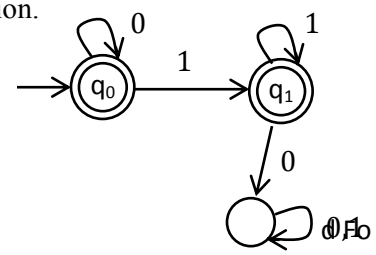
Proof: Let S_1 and S_2 be the two regular sets associated with the regular expressions R_1 and R_2 respectively. Let there be a set S_3 created by the concatenation of the elements of S_1 and S_2 in order. Now each string belonging to S_3 is such that it can be decomposed into two parts, with the first belonging to S_1 (corresponding to R_1) and the second to S_2 (corresponding to R_2). Thus, every element of S_3 can be generated by the regular expression R_1R_2. The association of S_3 with a regular expression (R_1R_2) indicates that S_3 is a regular set. Since concatenation of two regular sets has generated a regular set, regular sets are closed over concatenation operation.

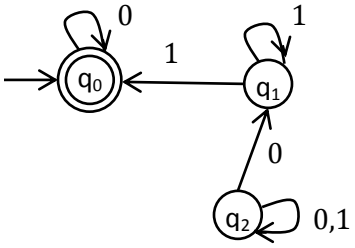
3. Regular sets are closed over the transpose (string reversal) operation.

   Proof: Let S_1 be a regular set associated with the regular expression R_1. let M_1 be the finite automaton corresponding to R_1. Now, for every string ($w \in S_1$), there is a path in M_1 from the initial state to final state. Now, the following operations are performed on M_1 to create a new finite automaton, say M_2.

   a. Reverse the direction of every arc/arm.
   b. Change the initial state (origionally in M_1) to the final state.
   c. Create a new start state S and connect it to all the acceptor states of M_1 using $\wedge$-transitions.
   d. Remove all $\wedge$-transitions.

   Let S_2 be a set of all strings accepted by M_2. Now every string ($x \in S_2$) is reverse (transpose) of some string ($w \in S_1$) thus S_2 is a set of transposed strings of S_1. Since S_2 is associated with a finite automaton M_2, it is associated with a regular expression, Hence S_2 is a regular set has created another regular set, regular sets are loosed over the transpose operation.





If final state is reachable from initial state then L^T is also regular.

4. Regular sets are closed over the complement operation

   Proof: let L be a regular set(language) over the character set Σ; then its complement $\tilde{L}$ is a set of all string over Σ^* that are not in L, that is $\tilde{L} = \Sigma^* - L$.

   Let M be the finite automaton for the language L. now, covert all the final states ofM to non-final states and vice versa to generate a new finite automaton, say M'. Now, if for a string ω, M stops in final state the M' stops in the non-final state and vice-

   versa. Hence M' is the finite automaton for $\tilde{}$

   Since there is a finite automaton for $\tilde{}$, $\tilde{}$ is a regular set. Hence, regular sets are closed over the complement operation.

5. Regular sets are closed over intersection operation

   Proof: let S_1 and S_2 be two regular sets. Therefore, their complements $\tilde{S}_1$ and $\tilde{S}_2$ are also regular sets. Union of $\tilde{S}_1$ and $\tilde{S}_2$, that is, ($\tilde{S}_1 \cup \tilde{S}_2$) is also a regular sets.

   Complement of ($\tilde{S}_1 \cup \tilde{S}_2$), that is, ($\overline{\tilde{S}_1 \cup \tilde{S}_2}$) is also a regular set.

   By De Morgan's Law, $S_1 \cap S_2 = (\overline{\tilde{S}_1 \cup \tilde{S}_2})$. Hence, ($S_1 \cap S_2$) is also a regular set. Therefore, regular sets are closed over the intersection operation.

6. Regular sets are closed over the difference operation.

Proof: let S_1 and S_2 be te two regular sets. since S_2 is a regular set, $\tilde{S}_2$ is also a regular set, $(S_1 \cap \tilde{S}_2)$ is also a regular set.

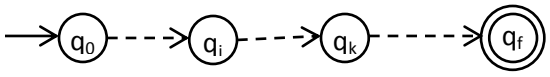
$S_1 \cap \tilde{S}_2 = S_1 - S_2$ is also a regular set.

Therefore, regular sets are closed over the intersection operation.

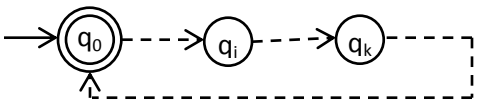
7. Regular sets are closed over the Kleene's star.

   proof: The closure over Kleene's star(∗) is also known as Kleene's closure. closure of a regular set over Kleene's star means that if R is a regular expression, then R^* is also a regular expression.

   Let R be a regular expression and M be its corresponding finite automaton, as shown in figure,



Now wrap around the finite automaton M and superimpose the final state q_f with the initial state q_0 to make the initial state as the final state and to create a new finite automaton, say M' as shown in figure.



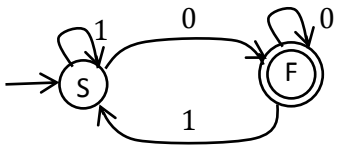
Now if a string ω is accepted by the finite automaton M, then M' accepts all the strings that involve the repetition of ω. such as ∧, ω, ωω, ωωω and so on. Thus, M' is a finite automaton for R^*. since there exists a finite automaton for R^*, it is a regular set corresponding to R^*. Hence, regular sets are closed over Kleene's closure.

Myhill-Nerode Theorem [Equivalent classes and Regular Languages]: Consider a language L with Σ = {0, 1} such that a string ω belong to L if and only if it ends with 0. we can categorized, all the strings into two categories (classes), those ending with 0(say class C1) and those ending with 1(say class C2). All the strings in the class C1 belongs to the language set L and those in the class C2 do not belongs to L.

The language L can be implemented on the finite state machine (FSM) such that each state in the FSM corresponds to a particular class.

FA contains minimum two states.

1. where the strings belonging to the class C1 reach the final state and
2. Other where the strings belonging to the class C2 reach the non-final state. Thus, the language L can be implemented using a FA.



Since L1 is regular.

Similarly consider a language L_2 over {0, 1} such that all strings belonging to L_2 wnd with 00.

all the string classified into 5-classes.

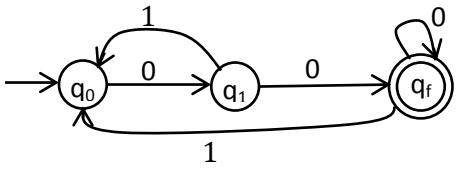
Class C1: string length less than 2 I.e. ∧, 0, 1.

Class C2: those ending with 00.

Class C3: those ending with 01.

Class C4: those ending with 10.

Class C5: those ending with 11.



since L_2 is regular

"If the number of equivalent classes over a language L is finite then the language L is regular". This statement is refered to as Myhill-Nerode theorem.

$L_3 = \{0^n1^n | n>0\}$ to recognize this machine should be able to remember how many 0's it has seen and if the no. of 1's is exactly the same as that of number of 0's. However, the FSM is devoid of any kind of memory the 0 count and match it

with the 1 count. the no. of equivalent classes generated by the language L_3 over $\{0,1\}^*$ is infinite, with each class corresponding to number of 0's on any string. Since the number equivalent classes are not finite, according to Myhill-Nerode theorem, the language L_3 is not regular.

Consider a language L_4 = {w|w is palindrome over {a,b}}.

Now, the recursive definition of a palindrome over {a, b} is as follows:

∧ - is a palindrome

a – is a palindrome

b – is a palindrome

if a string x is a palindrome then axa, bxb are palindromes.

To recognize L_4, a machine should be able to mark the middle of the string and match the symbols on the both sides of the middle. FSM has no memory, so counting is n't possible. another possible mechanism is to create a class corresponding to the string before the middle of the input string and to match this class with the class of the reverse string after the middle. if they are same then the string is a palindrome. Otherwise, it is not. However the number of possible classes for the string before the middle is infinite; hence the language L_4 can not be implemented on a finite automaton and hence, is not regular.

L_5 = {ww | w is a string over {0.1} is not regular} because the language L_5 belongs to the same group as the language L_4.

L_6 over Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; a string w over Σ^* belongs to L_6 if the number formed by w is divisible by 9.

For example: 9, 18, 27, . . . are belongs to L_6 1, 2, 3, . . . rest does not belongs to L_6.

Here string can be divided into 9 equivalent classes, with each class corresponding to a remainder 0 to 8. if remainder 0, then the string belongs to L_6, otherwise it does not since the number of equivalent classes. for L_6 is finite(9), the language L_6 can be implemented on the FSM and therefore L_6 is regular.

Current state	Input Symbol 0	1	2	3	4	5	6	7	8	9
→(q_0)	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_0
q_1	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_0	q_1
q_2	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_0	q_1	q_2
q_3	q_3	q_4	q_5	q_6	q_7	q_8	q_0	q_1	q_2	q_3
q_4	q_4	q_5	q_6	q_7	q_8	q_0	q_1	q_2	q_3	q_4
q_5	q_5	q_6	q_7	q_8	q_0	q_1	q_2	q_3	q_4	q_5
q_6	q_6	q_7	q_8	q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_7	q_7	q_8	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
q_8	q_8	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8

Minimization of Finite Automata:

Definition: Two state q_1 and q_2 are equivalent (denoted by $q_1 \equiv q_2$). if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states or both of them are non final state for all $x \in \Sigma^*$.

As it is difficult to construct $\delta(q_1, x)$ and $\delta(q_2, x)$ for all $x \in \Sigma^*$ because there are an infinite number of strings in Σ^*.

Definition: Two states q_1 and q_2 are k-equivalent($k \geq 0$) if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states or both non final states for all strings x of length k or less.

Example:

Current State	Input State a	b
→q_0	q_1(N)	q_4(N)
q_1	q_5(N)	q_f(F)
q_2	q_f(F)	q_5(N)
q_3	q_6(N)	q_4(N)
q_4	q_f(F)	q_5(N)
q_5	q_5(N)	q_3(N)
q_6	q_5(N)	q_f(F)
(q_f)	q_0(N)	q_f(F)

a) 0-level equivalence: The only string with length zero i.e. ∧, if ∧ is applied on any state q_i remains on the same state q_i.
if ∧ is applied in state q_f we get final state.

if ∧ is applied in state any other state, we get non final state.

The state of state Q can be partitioned into two subsets: $\{q_f\}$ and $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$

This partition is denoted by

$\pi_0 = \{ \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{q_f\}\}$

b) 1- Level equivalence: The strings with length one are a and b, equivalence at level 1 can exists if and only if, there exists equivalence at level 0. Hence, q_f cannot be equivalent to any other state at level 1.

1) The state q_0, q_3, & q_5 have same behavior. These state lead to a non-final state for both the strings a and b.
2) The state q_2 and q_4 have the same behavior a- F, b-NF.

Hence the set partition for 1-level equivalence is as follows:

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

Current state	Input String			
	aa	ab	ba	bb
q_0	q_5(N)	q_f(F)	q_f(F)	q_5(N)
q_3	q_5(N)	q_f(F)	q_f(F)	q_5(N)
q_5	q_5(N)	q_3(N)	q_6(N)	q_4(N)
q_1	q_5(N)	q_3(N)	q_0(N)	q_f(F)
q_6	q_5(N)	q_3(N)	q_0(N)	q_f(F)
q_2	q_0(N)	q_f(F)	q_5(N)	q_3(N)
q_4	q_0(N)	q_f(F)	q_5(N)	q_3(N)

c) 2-Level equivalence: The string length two is aa, ab, ba & bb. Equivalence at level 2 can exist. If there is equivalence at level 1.

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

$\pi_2 = \{ \{q_0, q_3\},\{q_5\}, \{q_1,q_6\}, \{q_2\ q_4\}, \{q_f\}\}$

d) 3-level equivalence: the strings with length three are aaa, aab, aba, abb, baa, bab, bba, and bbb.

$\pi_3 = \{ \{q_0, q_3\},\{q_5\}, \{q_1,q_6\}, \{q_2\ q_4\}, \{q_f\}\}$

π_3 is same as π_2. Hence, no further division is possible.

Current state	Input Symbol							
	aaa	aab	aba	abb	baa	bab	bba	bbb
q_0	q_5(N)	q_3(N)	q_0(N)	q_f(N)	q_0(N)	q_f(N)	q_5(N)	q_3(N)
q_3	q_5(N)	q_3(N)	q_0(N)	q_f(N)	q_0(N)	q_f(N)	q_5(N)	q_3(N)
q_1	q_5(N)	q_3(N)	q_0(N)	q_4(N)	q_1(N)	q_4(N)	q_0(N)	q_f(N)
q_6	q_5(N)	q_3(N)	q_3(N)	q_4(N)	q_1(N)	q_4(N)	q_0(N)	q_f(N)
q_2	q_1(N)	q_4(N)	q_0(N)	q_f(N)	q_5(N)	q_3(N)	q_6(N)	q_4(N)
q_4	q_1(N)	q_4(N)	q_0(N)	q_f(N)	q_5(N)	q_3(N)	q_6(N)	q_4(N)

Algorithm (minimization of Finite Automaton):

Step 1: (construction of π_0): By definition of 0-level equivalence, $\pi_0 = \{Q_1^0, Q_2^0\}$ where Q_1^0 is the set of all final states and $Q_2^0 = Q - Q_1^0$.

Step 2: (Construction of π_{k+1} and π_k): let Q_i^k be any subset in π_k. if q_1 and q_2 are in Q_i^k, they are (k+1)-equivalent provided $\delta(q_1, a)$ and $\delta(q_2, a)$ are in the same equivalent class in π_k for every $a \in \Sigma$.

if so, q_1 and q_2 are (k+1)- equivalent. in this way, Q_i^k is further divided into (k+1)-equivalent classes. repeat this for every Q_i^k in π_k to get all the elements of π_{k+1}.

Step 3: Construct π_n for n = 1, 2, … until $\pi_n = \pi_{n+1}$.

Example:

Current State	Input Symbol	
	a	b
$\rightarrow q_0$	q_1(N)	q_0(N)
q_1	q_0(N)	q_2(N)
q_2	q_3(F)	q_1(N)
q_3	q_3(F)	q_0(N)
q_4	q_3(F)	q_5(N)
q_5	q_6(N)	q_4(N)
q_6	q_5(N)	q_6(N)
q_7	q_6(N)	q_3(F)

By Step 1:

$Q_1^0 = \{q_3\}$, $Q_2^0 = Q - Q_1^0$

$\pi_0 = \{\{q_3\}, \{ q_0, q_1, q_2, q_4, q_5, q_6, q_7 \}\}$. As $\{q_3 \}$ can not be partitioned further.

$Q_1^1 = \{q_3 \}, Q_2^1 = \{q_2, q_4 \}, Q_3^1 = \{q_7 \}, Q_4^1 = \{q_0, q_1, q_5, q_6 \}$

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

$Q_1^2 = \{q_3\}, Q_3^2 = \{q_7 \}, Q_2^2 = \{q_0, q_6 \}, Q_4^2 = \{q_1, q_5\}, Q_5^2 = \{q_2, q_4 \}$.

Thus

$\pi_2 = \{\{q_3 \}, \{q_0, q_6\}, \{q_7 \}, \{q_1, q_5\}, \{q_2, q_4 \}\}$

$Q_1^3 = \{q_3\}, Q_3^3 = \{q_7 \}, Q_2^3 = \{q_0, q_6 \}, Q_4^3 = \{q_1, q_5\}, Q_5^3 = \{q_2, q_4 \}$.

$\pi_3 = \{\{q_3 \}, \{q_0, q_6\}, \{q_7 \}, \{q_1, q_5\}, \{q_2, q_4 \}\}$

$\pi_3 = \pi_2 (\pi_{n+1} = \pi_n)$

Hence at higher level, we will get the same equivalence.

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

$q_0' = [q_0, q_6]$ $F' = [q_3]$.

Example 2:

Current State	Input Symbol	
	0	1
$\longrightarrow q_0$	q_1(N)	q_5(N)
q_1	q_6(N)	q_2(F)
(q_2)	q_0(N)	q_2(F)
q_3	q_2(F)	q_6(N)
q_4	q_7(N)	q_5(N)
q_5	q_2(F)	q_6(N)
q_6	q_6(N)	q_4(N)
q_7	q_6(N)	q_2(F)

By Step 1:

$Q_1^0 = F = \{q_2\}, Q_2^0 = Q - Q_1^0 = \{ q_0, q_1, q_3, q_4, q_5, q_6, q_7 \}$

$\pi_0 = \{\{q_2\}, \{ q_0, q_1, q_3, q_4, q_5, q_6, q_7 \}\}$. As $\{q_2 \}$ can not be partitioned further.

$Q_1^1 = \{q_2 \}, Q_2^1 = \{q_3, q_5 \}, Q_3^1 = \{q_1, q_7 \}, Q_4^1 = \{q_0, q_4, q_6 \}$

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

$Q_1^2 = \{q_2\}, Q_3^2 = \{q_1, q_7 \}, Q_2^2 = \{q_6 \}, Q_4^2 = \{q_3, q_5\}, Q_5^2 = \{q_0, q_4 \}$.

$\pi_2 = \{\{q_2\}, \{q_1, q_7 \}, \{q_6 \}, \{q_3, q_5\}, \{q_0, q_4 \}\}$.

$Q_1^3 = \{q_2\}, Q_3^3 = \{q_1, q_7 \}, Q_2^3 = \{q_6 \}, Q_4^3 = \{q_3, q_5\}, Q_5^3 = \{q_0, q_4 \}$.

$\pi_3 = \{\{q_2\}, \{q_1, q_7 \}, \{q_6 \}, \{q_3, q_5\}, \{q_0, q_4 \}\}$.

$\pi_3 = \pi_2 (\pi_{n+1} = \pi_n)$

Hence at higher level, we will get the same equivalence.

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

$q_0' = [q_0, q_4]$, $F' = [q_2]$.

Equivalence of Two Finite Automata: Two finite automata over Σ are equivalent if they accept the same set of strings over Σ. when two finite automata are not equivalent; there is some string ω over Σ satisfying the following: One automaton reaches a final state on application of ω, whereas the other automaton reaches a non-final state.

Comparison Method: Let M and M' be two finite automata over Σ. we construct a comparison table consisting of n+1 columns, where n is the number of input symbols. the first column consists of pair of vertices of the form (q, q') where q∈M and q'∈M'.

Case 1: If we reach a pair of (q, q') such that q is final state of M and q' is a non-final of M' or vice-versa, we terminate the construction and conclude that M and M' are not equivalent.

Case 2: Here the construction is terminated when no new element appears in the second and subsequent columns which are not in the first column (i.e. when all the elements in the second and subsequent column appear in the first column). In this case we conclude that M and M' are equivalent.

Example:

Consider the following two DFAs M and M' over {0, 1} given in figure. determine M and M' are equivalent.

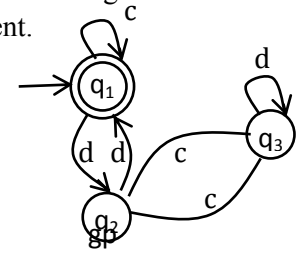


Figure: Automaton M

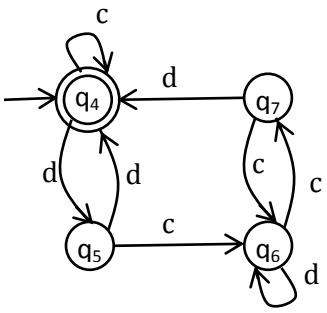


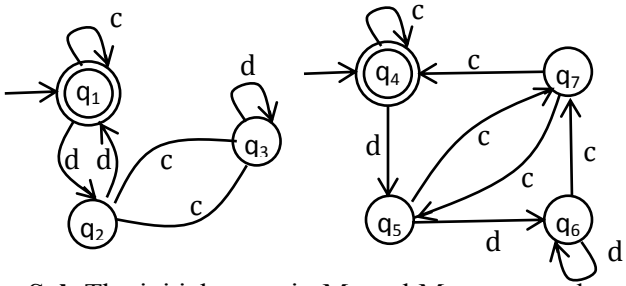
Figure: Automaton M'

Sol: The initial states in M and M' are q_1 and q_4 respectively. Hence the first element of the first column in the comparison table must be (q_1, q_4). the first element in the second column in (q_1, q_4) since both q_1 and q_4 are c-reachable from the respective initial states.

Current State (q, q')	Input Symbol c (q_c, q_c')	d (q_d, q_d')
→(q_1, q_4)	(q_1, q_4)	(q_2, q_5)
(q_2, q_5)	(q_3, q_6)	(q_1, q_4)
(q_3, q_6)	(q_2, q_7)	(q_3, q_6)
(q_2, q_7)	(q_3, q_6)	(q_1, q_4)

As we do not get a pair of (q, q'), where q is a final state and q' is a non-final state(or vice-versa) at every row, we proceed all the elements. Therefore M and M' are equivalent.

Example: show that the automat M_1 and M_2 defined by figure are not equivalent.



Sol: The initial states in M_1 and M_2 are q_1 and q_4 respectively. Hence the first column in comparison table is (q_1, q_4). q_2 and q_5 are d-reachable from q_1 and q_4. we see fro the comparison table that q_1 and q_6 are d-reachable from q_2 and q_5 respectively. As q_1 is a final state in M_1 and q_6 is a non-final state in M_2, we see that M_1 and M_2 are not equivalent: we can also not that q_1 id dd-reachable from q_1 and hence dd is accepted by M_1. dd is not acceptable by M_2 as only q_6 is dd-reachable from q_4, but q_6 is non-final.

Current state	Input Symbol c	d
(q,q')	(q_c,q_c')	(q_d,q_d')
(q_1,q_4)	(q_1,q_4)	(q_2,q_5)
(q_2,q_5)	(q_3,q_7)	(q_1,q_6)

Grammar: Before giving the definition of grammar, we shall study, two types of sentences in English, with a view to formalizing the construction of theses sentences. The sentences we consider are those with a noun and a verb, or those with a noun-verb and adverb (such as 'Ram ate quickly' or 'Sham ran'). the sentences 'Ram are quickly' has the words 'Ram', 'ate', 'quickly' written in that order. If we replace 'Ram' by 'sham', 'tom', etc. i.e. be any verb in the past tense, and 'quickly' by 'slowly' i.e. by an adverb, we get other grammatically correct sentences so the structure of 'Ram ate quickly' can be given as <noun><verb><adverb>.

for <noun>, we can substitute 'Ram', 'Sham', 'Tom' etc.

for <verb>, we can substitute 'ate', 'ran', 'walked' etc.

for <adverb>, we can substitute 'quickly', 'slowly' etc.

we have to note that <noun> <verb> <adverb> is not a sentence but only the description of a particular type of sentence. if we replace <noun> <verb> and <adverb> by suitable words, we get actual grammatically correct sentences.

Let S be a variable denoting a sentence. Now we can form the following rules to generate two types of sentences.

Productions:

S ⟶ <noun><verb><adverb> | <noun><verb>

<noun> ⟶ Ram|Sham|Tom|Geeta

<verb>⟶ ran|ate|walked

<adverb> ⟶ slowly|quickly

let us denote the collection of rules given above by P.

If our vocabulary is thus restricted to 'Ram', 'Sham', 'Geeta', 'Tom', 'ran', 'ate', 'walked', 'quickly' and 'slowly', and our sentences are of the form <noun><verb><adverb> and <noun><verb>, we can describe the grammar by a 4-tuple.

(V_N, T, P, S), where

V_N = {<noun>, <verb>, <adverb>}

T= {Ram, Sham, Tom, Geeta, ate, ran, walked, quickly, slowly}

P is the collection of rules described above (the rules are called productions).

S is the special symbol denoting a sequence. The sentences are obtained by i) starting with S ii) replacing words using the productions and iii) terminating when a string of terminals is obtained.

Definition: A phrase – structure grammar (or simply a grammar) is (V_N, T, P, S), where

i) V_N is a finite non empty set whose elements are called variables (non-terminals).
ii) T is a finite non-empty set whose elements are called terminals.
iii) $V_N \cap T \neq \Phi$.
iv) S is a special variable (i.e. an element of V_N) called start symbol.
v) P is a finite set whose elements are $\alpha \longrightarrow \beta$, where α and β are strings on $V_N \cup T$. α has at least one symbol from V_N. The elements of P are called productions or productions rules.

Note:

1. Reverse Substitution is not permitted.
2. No inversion operation is permitted, for example:, if S ⟶ AB is production, it is not necessary that AB ⟶ S is a production.

Remarks: The derivation of a string is complete when the working string can not be modified. If the final string does not contain any variable, it is a sentence in the language. if the final string contains a variable, it is a sentential form and in this case the production generator gets 'stuck'.

Definition: The language generated by a grammar G (denoted by L(G)) is defined as $\{\omega \in T^* | S \overset{*}{\underset{G}{\Rightarrow}} \omega\}$. The elements of L(G) are called sentences L(G) is the set of all terminal strings derived from the start symbol S.

Example:

If G = ({S}, {0,1}, { S ⟶ 0S1, S ⟶ ∧}, S), find L(G).

Sol: As S ⟶ ∧ is a production. $S \underset{G}{\Rightarrow} \wedge$. So ∧ is in L(G).

also, for all n≥ 1.

$S \underset{G}{\Rightarrow} 0S1 \underset{G}{\Rightarrow} 0^2S1^2 \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} 0^nS1^n \underset{G}{\Rightarrow} 0^n1^n$

therefore,

$0^n1^n \in L(G)$ for n≥ 1.

$L(G) = \{0^n1^n | n \geq 1\}$

Context-Free Grammars: A context-free grammar (CFG) is denoted by G = {V,T,P,S}, where V and T are finite sets of variables and terminals respectively. we assume V and T are disjoint. P is a finite set of productions, each production is of the form A⟶α. where A is a variable and α is a string of symbols from $(V \cup T)^*$ finally S is a special variable called the start symbol.

The right-hand side of a CFG is not restricted and it may be null or a combination of variables and terminals. The possible length of right-hand sentential from ranges from 0 to ∞ i.e. $0 \leq |\alpha| \leq \infty$.

As we know that a CFG has no context neither left nor right. this is why, it is known as context-free. Many programming languages have recursive structure that can be defined by CFGs.

Example: Consider a grammar G = (V, T, P, S) having productions. S ⟶ aSa | bSb | ∧. check the productions and find the languages generated.

Sol: let

P_1: S ⟶ aSa (RHS is terminal variable terminal)

P_2: S ⟶ bSb (RHS is terminal variable terminal)

P_3: S ⟶ ∧ (RHS is null string).

Since all productions are of the form A ⟶α, where $\alpha \in (V \cup T)^*$. hence G is a CFG.

Language Generated:

S ⟹	aSa	or	bSb
S ⟹	a^nSa^n	or	b^nSb^n (using n step derivation)
S ⟹	$a^nb^mSb^ma^n$	or	$b^na^mSa^mb^n$ (using m step derivation)
S ⟹	$a^nb^mb^ma^n$	or	$b^na^ma^mb^n$ (using S⟶∧

So $L(G) = \{\omega\omega^R : \omega \in (a+b)^*\}$

Leftmost and Rightmost Derivation:

Leftmost derivation: if G = (V, T, P, S) is a CFG and ω ∈ L(G) then a derivation $S \underset{L}{\overset{*}{\Rightarrow}} \omega$ is called leftmost derivation iff all steps involved in derivation have leftmost variable replacement only.

Rightmost derivation: if G = (V, T, P, S) is a CFG and ω ∈ L(G) then a derivation $S \underset{R}{\overset{*}{\Rightarrow}} \omega$ is called rightmost derivation iff all steps involved in derivation have rightmost variable replacement only.

Example: Consider the grammar S ⟶S + S | S ∗ S | a | b. find the leftmost and rightmost derivations for string ω = a ∗ a + b.

Sol: let

P_1: S ⟶S + S (RHS is variable terminal variable)

P_2: S ⟶S ∗ S (RHS is variable terminal variable)

P_3: S ⟶ a (RHS is terminal)

P_4: S ⟶ b (RHS is terminal)

Hence the given grammar is CFG.

Leftmost derivation: for ω = a ∗ a + b

$S \underset{L}{\Rightarrow} S * S$ (using S ⟶S ∗ S)

$\underset{L}{\Rightarrow} a * S$ (using S ⟶ a)

$\underset{L}{\Rightarrow} a * S + S$ (using S ⟶S + S)

$\underset{L}{\Rightarrow} a * a + S$ (using S ⟶ a)

$\underset{L}{\Rightarrow} a * a + b$ (using S ⟶ b)

The last symbol from the left is b, so using S ⟶ b).

(Note: Leftmost variable is highlighted in each step).

Rightmost Derivation:- for ω = a ∗ a + b.

$S \underset{R}{\Rightarrow} S * S$ (using S ⟶S ∗ S)

$S \underset{R}{\Rightarrow} S * S + S$ (using S ⟶S + S)

(since, in the above sentential form second symbol from the right is ∗ so, we can not use S ⟶ a|b. therefore, we use S ⟶S + S).

$\underset{R}{\Rightarrow} S * S + b$ (using S ⟶ b)

$\underset{R}{\Rightarrow} S * a + b$ (using S ⟶ a)

$\underset{R}{\Rightarrow} a * a + b$ (using S ⟶ a)

(Note: Rightmost variable is highlighted in each step).

Chomsky classification of Grammars| Languages (Chomsky Hierarchy): Noam chomsky has classified all grammars in four categories (type 0 to type 3) based on the right hand side forms of the productions.

Type 3: This is most restricted type. productions of type A ⟶α or A ⟶αB|Bα. where A, B ∈ V_N, and α ∈ T are known as type 3 or regular grammar productions. in production of type 3. S ⟶ ∧ is also allowed, if ∧ is in generated language.

for example: production S ⟶ aS, S ⟶ a are type 3 production.

Left-linear production: A ⟶ Ba

Right-linear production: A ⟶ aB

a left linear or right linear grammar is called regular grammar. The language generated by a regular grammar is known as regular language.

Type 2: Type 2 grammar is called context free grammars. A production of the form $\alpha \longrightarrow \beta$, where $\alpha, \beta \in (V_N, T)^*$ is known as type 2 production and the language generated by the type of grammars is called context-free language (CFL).

For example: $S \longrightarrow S + S$, $S \longrightarrow S * S$, $S \longrightarrow id$ are type 2 productions.

Type 1: Type 1 are context sensitive grammars (CSGs). if all productions of a grammar are of type 1 then grammar is known as type 1 grammar.

In CSG, there is left context or right context or both. For example, production $\alpha A \beta \longrightarrow \alpha a \beta$. In this α is left context or β is right context of A and A is the variable which is replaced by a. the production of $S \longrightarrow \wedge$ is allowed in type 1. If $\wedge$ is in L(G), but S should not appear on the right hand side of any production. for example, productions $S \longrightarrow AB$, $S \longrightarrow \wedge$, $A \longrightarrow c$ are type 1 productions, but the production of type $A \longrightarrow Sc$ is not allowed.

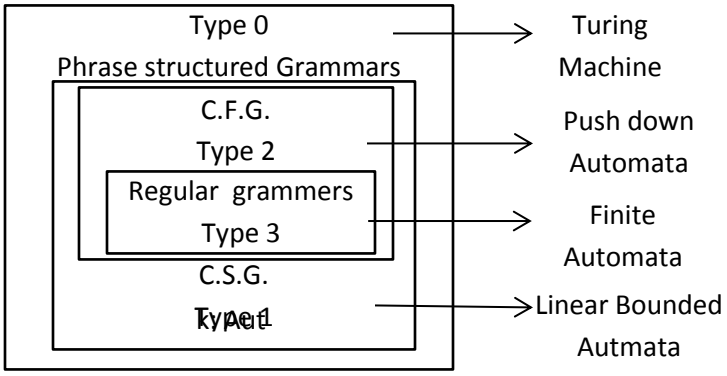
(Note: we assume that $\wedge$ is the context on left and right).

Type 0: These types of grammar are also known as phrase-structure grammars, and RHS of these are free from any restriction. All grammars are type 0 grammars.

For example: productions of type $AS \longrightarrow aS$, $SB \longrightarrow Sb$, $S \longrightarrow \wedge$ are type 0 productions.

Classifying the Type of grammars:

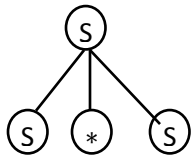
1. Type 0 grammars include Type 1, Type 2, Type 3 grammars.
2. Type 1 grammars include Type 2 and Type 3 grammars.
3. Type 2 grammar includes Type 3 grammars.



Derivation Trees: A natural way of exhibiting the structure of a derivation is to draw a derivation tree or parse tree. At the root of the tree is the variable with which the derivation begins. Interior (intermediated) nodes are variables and leaf nodes are terminals.

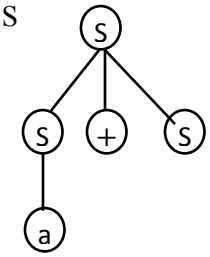
For example: CFG $S \longrightarrow S + S \mid S * S \mid a \mid b$ and construct the derivation trees for all productions. for the production

i) $S \longrightarrow S * S$



ii) $S \longrightarrow S + S$

iii) $S \longrightarrow a$

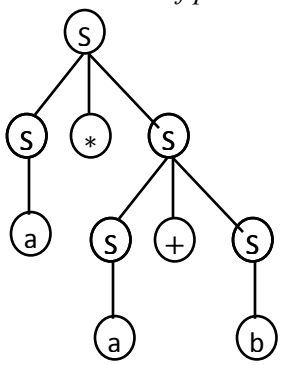


iv) $S \longrightarrow b$

If $\omega \in L(G)$ then it is represented by a tree called derivation tree or parse tree satisfying the following

1. The root has label S (the starting symbol).
2. The all interior (internal) vertices (nodes) are labeled with variables.
3. The leaves or terminal nodes are labeled with $\wedge$ or terminal symbols.
4. If $A \longrightarrow \alpha_1 \alpha_2 \alpha_3 \ldots \alpha_n$ is a production in G, then A becomes the parent of nodes labeled $\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n$ and
5. The collection of leaves from left to right produces string ω.

Leftmost derivation's tree of previous example.



Ambiguity: A grammar G is ambiguous if there exists some string $\omega \in L(G)$ for which there are two or more distinct derivation trees or there are two of more distinct leftmost derivations.

i.e. रोको मत जाने दो

Ambiguity is a negative property of a grammar and it is usually (but not always) possible to find an equivalent unambiguous grammar. An 'inherently ambiguous language' is a language for which no unambiguous grammar exists.

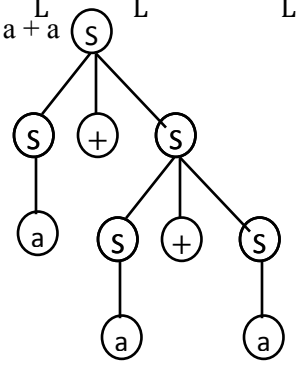
Example of Ambiguity:

$S \longrightarrow S + S \mid a$

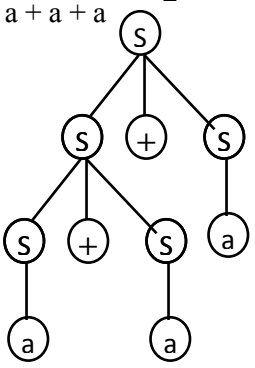
$\omega = a + a + a$

Left-most derivation

a) $S \underset{L}{\Rightarrow} S + S \underset{L}{\Rightarrow} a + S \underset{L}{\Rightarrow} a + S + S \underset{L}{\Rightarrow} a + a + S \underset{L}{\Rightarrow} a + a + a$



b) $S \underset{L}{\Rightarrow} S + S \underset{L}{\Rightarrow} S + S + S \underset{L}{\Rightarrow} a + S + S \underset{L}{\Rightarrow} a + a + S \underset{L}{\Rightarrow} a + a + a$



Elimination of Left Recursion: let the variable A has left recursive productions as follows:

$A \longrightarrow A\alpha_1|A\alpha_2|A\alpha_3|\ldots|A\alpha_n|\beta_1|\ \beta_2|\ \beta_3|\ldots|\ \beta_n$

Where $\beta_1, \beta_2, \ldots, \beta_n$ do not begin with A, then we replace A-productions by

$A \longrightarrow \beta_1\ A'|\ \beta_2\ A'|\ \beta_3\ A'|\ldots|\ \beta_n\ A'$

Where $A' \longrightarrow \alpha_1 A'|\ \alpha_2 A'|\ \alpha_3 A'|\ldots|\ \alpha_n A'|\wedge$.

Example: CFG $E \longrightarrow E + T \mid T$

$T \longrightarrow T * F \mid F$

$F \longrightarrow id.$

Where E is the starting symbol and id is the terminal. Remove the left recursion (if any).

Sol: Recursive production are

$E \longrightarrow E + T$, $T \longrightarrow T * F$

Eliminating Left Recursion:

$E \longrightarrow E + T$ is replaced by $E \longrightarrow TE'$

Where $E' \longrightarrow +TE' \mid \wedge$

$T \longrightarrow T * F$ is replaced by $T \longrightarrow FT'$

Where $T' \longrightarrow * FT' \mid \wedge$

So, productions of G are:

$E' \longrightarrow +TE'$

$E' \longrightarrow +TE' \mid \wedge$

$T \longrightarrow FT'$

$T' \longrightarrow * FT' \mid \wedge$

$F \longrightarrow id.$

Left Factoring: Two or more productions of a variable A of the grammar G are said to have left factoring if A – productions are of the form

$A \longrightarrow \alpha\beta_1|\ \alpha\beta_2|\ \alpha\beta_3|\ldots|\ \alpha\beta_n$, where $\beta_i \in (V_N \cup T)^*$ and does not start with α. All these A-productions have common left factor α.

Elimination of Left Factoring: let the variable A has (left factoring) productions as follows:

$A \longrightarrow \alpha\beta_1|\ \alpha\beta_2|\ \alpha\beta_3|\ldots|\ \alpha\beta_n|\gamma_1|\ \gamma_2|\ \gamma_3|\ldots|\ \gamma_n$, where $\beta_1, \beta_2, \beta_3, \ldots, \beta_n$ and $\gamma_1, \gamma_2, \gamma_3, \ldots, \gamma_n$, do not contain α as a prefix, then we replace A-production by,

$A \longrightarrow \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \gamma_3 \mid \ldots \mid \gamma_n$

Where $A' \longrightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$

Example: Consider the grammar $S \longrightarrow aSa \mid aa$ and remove the left factoring (if any).

Sol: $S \longrightarrow aSa$ and $S \longrightarrow aa$,

we have $\alpha = a$ as a left factoring, we get the productions:

$S \longrightarrow aS'$

$S' \longrightarrow Sa \mid a$

Removal of Ambiguity: for removing ambiguity

i) Remove the Left Recursion.
ii) Remove the Left Factoring.

(This is heuristic approach for removal of ambiguity).

Example:

$S \longrightarrow S + S \mid S * S \mid a \mid b$

Sol: Step 1:

Find left recursive statement

$S \longrightarrow S + S$

$S \longrightarrow S * S$

After removing left recursion

$S \longrightarrow aS' \mid bS'$

$S' \longrightarrow +SS' \mid *SS' \mid \wedge$

Now check for ambiguous string $\omega = a * a + a$.

$S \Longrightarrow aS'$

$S \Longrightarrow a *S\ S'$

$S \Longrightarrow a * a\ S'\ S'$

$S \Longrightarrow a * a + a\ S'\ S'\ S'$

$S \Longrightarrow a * a + a \wedge S'\ S'$

$S \Longrightarrow a * a + a \wedge S'$

$S \Longrightarrow a * a + a \wedge \equiv a * a + a$

So, we conclude that removal of left recursion (and left factoring also) helps in removal on ambiguity of the ambiguous grammar.

Simplification of CFG :

i) Removal of Null productions:

Definition: A production $A \longrightarrow \wedge$ is called a 'null production' and variable B having production $B \overset{*}{\Longrightarrow} \wedge$ is called null variable.

Removing Null productions: Consider the Null production $X \longrightarrow \wedge$ & for every non-null production

$Y \longrightarrow \alpha X \beta$, where $\alpha, \beta \in (V_N \cup T)^*$.

a) Add enough productions of the form $Y \longrightarrow \alpha\beta$ such that $\alpha\beta \neq \wedge$.
b) Delete the production $X \longrightarrow \wedge$.

Example: $S \longrightarrow aSa \mid bSb \mid \wedge$.

Sol: $S \longrightarrow aSa \mid bSb \mid aa \mid bb$

Example: $S \longrightarrow aS \mid aA \mid \wedge$, $A \longrightarrow \wedge$

Solution: $S \longrightarrow aS \mid aA \mid a \mid a$

$S \longrightarrow aS \mid a$

Example: $S \longrightarrow a \mid Xb \mid aYa$,

$X \longrightarrow Y \mid \wedge$

$Y \longrightarrow b \mid X$

Solution: $S \longrightarrow a \mid Xb \mid aYa \mid b \mid aa$

$X \longrightarrow Y$

$Y \longrightarrow b \mid X$

Removal of Unit Production: A production of the type $A \longrightarrow B$, where A, B are variables is called unit production. These productions create indirection and this result into requirement of extra steps in deriving the terminal strings.

For example: Consider the grammar $S \longrightarrow aA \mid a$

$A \longrightarrow B$

$B \longrightarrow C$

$C \longrightarrow aS$

Solution: we find $S \Longrightarrow aA \Longrightarrow bS \Longrightarrow aC \Longrightarrow aaS$. It means we are getting as after two intermediate steps, $A \Longrightarrow B \Longrightarrow C \Longrightarrow aS$. So, we have the production $S \longrightarrow aaS$, instead of $A \longrightarrow B$, $B \longrightarrow C$, $C \longrightarrow aS$, which create indirection.

Steps for removing unit production.

Step 1: Collect all the unit productions

Step 2: if $A \Longrightarrow A_1 \Longrightarrow A_2 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow \omega$, where is string of terminals, then replace the RHS of A by ω and remove the productions

$A_1 \longrightarrow A_2 \longrightarrow \ldots \longrightarrow A_n \longrightarrow \omega$

Example: Consider the grammar S ⟶ aS | bS | A | B, A ⟶ a, B ⟶ b remove the unit productions (if any).

Solution: we have two unit productions: S ⟶ A and S ⟶ B and from variable A and B we get terminals a and b respectively. So, replacing the occurrences of A and B variables in S productions, we get simplified productions:

S ⟶ aS | bS | a | b.

Removal of useless Symbol: The symbols that can be used in any production due to their unavailability in the productions or inability in deriving the terminal(s) are known as useless symbols.

Example: Consider the grammar S ⟶ aS | bS a | b, A ⟶ a, B ⟶ b, in this symbols (variables) A and B cannot be used because we can reach the variables A and B from the starting symbol S. so A and B are useless symbols and these are removed from the grammar.

Now, the simplified grammar is

S ⟶ aS | bS | a | b.

Application of Grammars:

1. Specifying syntax of programming languages.
2. Representation symtactic structure in natural languages,
3. Model of computation.

Order in which defects should be Removed:

1. Eliminate null productions
2. Eliminate Unit productions
3. Eliminate non-reachable Non-terminals

Reduced Form:

Example: Find the reduced grammar equivalent to the grammar G whose productions are S ⟶AB | CA, B ⟶ BC | AB, A ⟶ a, C ⟶ aB|b.

Solution:

Step 1: W_1 = { A, C } as A ⟶ a and C ⟶ b be productions with a terminal string on RHS.

W_2 = { A, C } ∪ { A_1 | A_1 ⟶ α with α ∈ (∑ ∪ { A, C })* }

= { A, C } ∪ { S } as we have S ⟶ CA.

W_3 = { A, C, S } ∪ { A_1 | A_1 ⟶ α with α ∈ (∑ ∪ { A, C })* }

= { A, C, S } ∪ ϕ = W_2

V_N' = W_2 = { S, A, C }

P' = { A_1 ⟶ α|A_1, α ∈ (V_N' ∪ ∑)* }

P' = { S ⟶ CA, A ⟶a, C ⟶b }

Thus,

G_1 = ({S, A, C} , { a, b},

{ S ⟶ CA, A ⟶a, C ⟶b }, S)

Step 2: W_1 = { S }

W_2 = { S } ∪ { A, C } [because S ⟶ CA]

= { S, A, C }

W_3 = { S, A, C } ∪ { a, b} = { S, A, C, a, b} [because A ⟶ a, C ⟶ b]

As W_3 = V_N' ∪ T, P'' = P'

Therefore G' = ({S, A, C} , { a, b}, { S ⟶ CA, A ⟶a, C ⟶b }, S) is reduced grammar.

Example:

S ⟶ aAa, A ⟶ Sb | bCC | DaA, C ⟶ abb | DD, E ⟶ aC, D ⟶aDa

Solution: Step 1: W_1 = {c} as C ⟶ abb be production with terminals string on RHS.

W_2 = {C} ∪ {E, A } as E ⟶ aC and A ⟶ bCC

= { C, E, A }

W_3 = { C, E, A } ∪ { S } as S ⟶ aAa

= { C, E, A, S}

W_4 = { C, E, A, S} ∪ ϕ = W_3 ∪ ϕ = W_3

Hence

V_N' = W_3 = {S, A, C, E}

P' = {S ⟶ aAa, A ⟶ Sb | bCC, C ⟶ abb, E ⟶ aC }

G_1 = (V_N', {a, b}, P', S)

Step 2: We start with W_1 = { S }

As we have S ⟶ aAa

W_2 = {S} ∪ {A, a}

As A ⟶ Sb | bCC

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

As we have $C \longrightarrow abb$

$W_4 = W_3 \cup \{a, b\} = W_3$

Hence

$P'' = \{ A \longrightarrow \alpha \mid A_1 \in W_3 \}$

$= \{ S \longrightarrow aAa, A \longrightarrow Sb \mid bCC, C \longrightarrow abb\}$

Therefore

G' = ({ S, A, C} , {a, b}, P'', S) is reduced grammar.

Normal Forms: In Context free productions, we have no restriction on RHS. If, we restrict the RHS in different manner and this results into different normal forms.

"If G is a CFG and productions of G satisfy certain restrictions, then G is said to in a normal form".

Two popular and important normal forms are:

1. Chomsky Normal Form (CNF) and
2. Griebach Normal Form (GNF).

Chomsky Normal Form: A Context-free grammar is in Chomsky Normal Form (CNF) if every production is of one of the two types.

$A \longrightarrow BC$ (<variable> ⟶ <Variable> < Variable>)

$A \longrightarrow a$ (< variable > ⟶ <Terminal>)

Where A, B and C are variables and a is a terminal symbol.

Methods for converting a CFG to CNF:

Step 1: Eliminate NULL productions

Step 2: Eliminate UNIT productions.

Step 3: Change the RHS to single terminal or string of two variables.

Example:

$S \longrightarrow aSa \mid bSb \mid a \mid b$, convert it into CNF.

Solution: Let $P_1 : S \longrightarrow aSa$ (Not in CNF)

$P_2 : S \longrightarrow bSb$ (Not in CNF)

$P_3 : S \longrightarrow a$ (In CNF)

$P_4 : S \longrightarrow b$ (In CNF)

There is no NULL production and Unit production.

Consider P_1 : Replace the terminal a by a new variable A, we get

$S \longrightarrow ASA$,

Now, replace SA by a new variable X_1 and so,

$S \longrightarrow AX_1$, where

$X_1 \longrightarrow SA$ (In CNF)

$A \longrightarrow a$ (In CNF)

Consider P_2: Replace the terminal b by a new variable B, we get

$S \longrightarrow BSB$, now, replace SB by a new variable Y_2 and so, $S \longrightarrow BY_2$, where

$Y_2 \longrightarrow SB$ (In CNF)

$B \longrightarrow b$ (In CNF)

Thus grammar G is in CNF has following productions

$S \longrightarrow AX_1 \mid BX_2 \mid a \mid b$

$A \longrightarrow a, \quad B \longrightarrow b$

$X_1 \longrightarrow SA, \quad Y_2 \longrightarrow SB.$

Example: Let G be the grammar with productions

$S \longrightarrow AACD$

$A \longrightarrow aAb \mid \wedge$

$C \longrightarrow aC \mid a$

$D \longrightarrow aDa \mid bDb \mid \wedge.$

Solution: Step 1: Remove the Null-productions: The nullable Variables are A and D.

$S \longrightarrow AACD \mid ACD \mid AAC \mid CD \mid AC \mid C$

$A \longrightarrow aAb \mid ab$

$C \longrightarrow aC \mid a$

$D \longrightarrow aDa \mid bDb \mid aa \mid bb$

Step 2: Remove Unit – productions

$S \longrightarrow aC \mid a$ and delete $S \longrightarrow C$

Step 3: $S \longrightarrow AACD \mid ACD \mid AAC \mid CD \mid AC \mid aC \mid a$ $A \longrightarrow aAb \mid ab$

$C \longrightarrow aC \mid a$

$D \longrightarrow aDa \mid bDb \mid aa \mid bb$

Replace the terminal b by a new variable X_b, and the terminal a by X_a, we get

$S \longrightarrow AACD \mid ACD \mid AAC \mid CD \mid AC \mid X_aC \mid X_a$

$A \longrightarrow X_aAX_b \mid X_a X_b$

$C \longrightarrow X_aC \mid X_a$

$D \longrightarrow X_aDX_a \mid X_bDX_b \mid X_aX_a \mid X_bX_b$

$X_a \longrightarrow a, X_b \longrightarrow b$

Final Step:

$S \longrightarrow AT_1 \quad T_1 \longrightarrow AT_2 \; T_2 \longrightarrow CD$

$S \longrightarrow AT_2$

$S \longrightarrow AU_1 \quad U_1 \longrightarrow AC$

$A \longrightarrow X_aW_1 \quad W_1 \longrightarrow AX_b$

$A \longrightarrow X_aX_b$

$C \longrightarrow X_aC \mid a$

$D \longrightarrow X_aY_1 \quad Y_1 \longrightarrow DX_a$

$D \longrightarrow X_bZ_1 \quad Z_1 \longrightarrow DX_b$

$D \longrightarrow X_aX_b \mid X_aX_b$

$X_a \longrightarrow a$

$X_b \longrightarrow b$

Griebach Normal Form: Every Context-Free Language L without ∧ can be generated by a grammar for which every production is of the form $A \longrightarrow a\alpha$, where A is a variable, a is a terminal and α is a string of variable (possibly empty or ($\alpha \in V_N^*$)).

A grammar in GNF is the natural generalization of a regular grammar (right linear).

Method for converting a CFG into GNF :

1. Eliminate the NULL productions
2. Change the intermediate terminal into variables.
3. Remove all the variables of G as $A_1, A_2, A_3, \ldots, A_n$.
4. Repeat Step 5 and Step 6 for I = 1, 2, . . . , n
5. If $A_i \longrightarrow a\,\alpha_1\,\alpha_2 \ldots \alpha_n$, where $a \in T$, and α_j is a variable or a terminal symbol.
   Repeat for j = 1, 2,. . . , m
   If α_j is a terminal then replace it by a variable A_{n+j} consider the next A_i- production and go to step 5.
6. If $A_i \longrightarrow \alpha_1\,\alpha_2 \ldots \alpha_n$, where α_1 is a variable, then preform the following:
   If α_1 is same as A_i, then remove the left-recursion and go to step 5.
   Else replace α_1 by all RHS of α_1-productions one by one. Consider the remaining A_i-productions, which are not in GNF and go to step 5.
7. Exit.

Advantages of GNF:

1. Avoids Left Recursion
2. Always has terminal in leftmost position in RHS of each production.
3. Helps select production correctly.
4. Guarantees derivation length no longer than string length.

Example: G = ({A_1, A_2, A_3}, {a, b}, P, A_1}, where P consists of the following production rules.

$A_1 \longrightarrow A_2A_3$

$A_2 \longrightarrow A_3A_1 \mid b$

$A_3 \longrightarrow A_1A_2 \mid a$

Convert it into GNF.

Solution: (Renaming is not required)

Consider the A_1-production

$A_1 \longrightarrow A_2A_3$ (Not in GNF)

Replacing A_2 by its RHS, we get

$A_1 \longrightarrow bA_3$ (In GNF)

$A_1 \longrightarrow A_3A_1A_3$ (Not in GNF)

Now, consider $A_1 \longrightarrow A_3A_1A_3$ and replacing A_3 by its RHS, we get

$A_1 \longrightarrow aA_1A_3$ (In GNF)

$A_1 \longrightarrow A_1A_2A_1A_3$ (Not in GNF)

So, A_1-productions are $A_1 \longrightarrow bA_3 \mid aA_1A3 \mid A_1A_2A_1A_3$.

Now, Consider $A_1 \longrightarrow A_1A_2A_1A_3$ and removing left recursion, we get

$A_1 \longrightarrow bA_3A_4 \mid bA_3$ (In GNF)

$A_1 \longrightarrow aA_1A_3A_4 \mid aA_1A_3$ (In GNF) where

$A_4 \longrightarrow A_2A_1A_3A_4 \mid A_2A_1A_3$

(A_4 is a new variable and it is a production that is not in GNF)

So, now all A_1-productions are in GNF. Consider A_2-productions:

$A_2 \longrightarrow A_3A_1$ (Not in GNF)

$A_2 \longrightarrow b$ (In GNF)

Now, consider $A_2 \longrightarrow A_3A_1$ and replacing A_3 by its RHS, we get

$A_2 \longrightarrow aA_1$ (In GNF)

$A_2 \longrightarrow A_1A_2A_1$ (Not in GNF)

Now, consider $A_2 \longrightarrow A_1A_2A_1$ and replacing A_1 by its RHS, we get

$A_2 \longrightarrow bA_3A_4A_2A_1$ (In GNF)

$A_2 \longrightarrow bA_3A_2A_1$ (In GNF)

$A_2 \longrightarrow aA_1A_3A_4A_2A_1$ (In GNF)

$A_2 \longrightarrow aA_1A_3A_2A_1$ (In GNF)

So, all A_2-productions are in GNF.

Consider A_3-productions:

$A_3 \longrightarrow a$ (In GNF)

$A_3 \longrightarrow A_1A_2$ (Not in GNF)

Now, consider $A_3 \longrightarrow A_1A_2$ and replacing A_1 by its RHS, we get

$A_3 \longrightarrow bA_3A_4A_2$ (In GNF)

$A_3 \longrightarrow bA_3A_2$ (In GNF)

$A_3 \longrightarrow aA_1A_3A_4A_2$ (In GNF)

$A_3 \longrightarrow aA_1A_3A_2$ (In GNF)

Consider A_4-productions:

$A_4 \longrightarrow A_2A_1A_3A_4 \mid A_2A_1A_3$ (Not in GNF)

Replacing A_2 by its RHS, we get

$A_4 \longrightarrow bA_1A_3A_4 \mid bA_1A_3$

$A_4 \longrightarrow aA_1A_3A_4 \mid aA_1A_3$

$A_4 \longrightarrow bA_3A_4A_2A_1A_1A_3A_4 \mid bA_3A_4A_2A_1A_1A_3$

$A_4 \longrightarrow bA_3A_2A_1A_1A_3A_4 \mid bA_3A_2A_1A_1A_3$

$A_4 \longrightarrow aA_1A_3A_4A_2A_1A_1A_3A_4 \mid aA_1A_3A_4A_2A_1A_1A_3$

$A_4 \longrightarrow aA_1A_3A_2A_1A_1A_3A_4 \mid aA_1A_3A_2A_1A_1A_3$

Now, all A_4-productions are in GNF. All productions are in GNF.

Example: $E \longrightarrow E + T \mid T$,

$T \longrightarrow T * F \mid F$,

$F \longrightarrow (E) \mid a$,

Solution:

Eliminate the Unit Productions

$T \longrightarrow T * F \mid (E) \mid a$,

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

The equivalent grammars without unit productions is, $G_1 = (V, T, P_1, S)$, where P_1 consists of

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

ii) $T \longrightarrow T * F \mid (E) \mid a$,

iii) $F \longrightarrow (E) \mid a$,

Step 2: Reduce the Terminals into Variables

We introduce new variables A, B, C corresponding to +, *,) respectively. The modified productions are

i) $E \longrightarrow EAT \mid TBF \mid (EC \mid a$,

ii) $T \longrightarrow TBF \mid (EC \mid a$,

iii) $F \longrightarrow (E) \mid a$,

iv) $A \longrightarrow +$,

v) $B \longrightarrow *$,

vi) $C \longrightarrow)$

Step 3: The variables A, B, C, F, T and E are renamed as A_1, A_2, A_3, A_4, A_5, A_6 respectively. The productions becomes

$A_1 \longrightarrow +$ (In GNF)

$A_2 \longrightarrow *$ (In GNF)

$A_3 \longrightarrow)$ (In GNF)

$A_4 \longrightarrow (A_6A_3$ (In GNF)

$A_4 \longrightarrow a$ (In GNF)

$A_5 \longrightarrow A_5A_2A_4$ (Not in GNF)

$A_5 \longrightarrow (A_6 A_3$ (In GNF)

$A_5 \longrightarrow a$ (In GNF)

$A_6 \longrightarrow A_6A_1A_5$ (Not in GNF)

$A_6 \longrightarrow A_5A_2A_4$ (Not in GNF)

$A_6 \longrightarrow (A_6A_3$ (In GNF)

$A_6 \longrightarrow a$ (In GNF)

Production A_1, A_2, A_3 and A_4 are in GNF.

Consider production A_5-production

$A_5 \longrightarrow A_5A_2A_4$ (Not in GNF)

and renaming left recursion, we get

$A_5 \longrightarrow (A_6 A_3 A_7 | (A_6 A_3$ (In GNF)

$A_5 \longrightarrow aA_7 | a$

$A_7 \longrightarrow A_2A_4A_7 | A_2A_4$

(A_7 is a new variable and it is production that is Not in GNF).

So, now all A_5- productions are in GNF.

Consider A_6-productions

$A_6 \longrightarrow A_6A_1A_5$ (Not in GNF)

$A_6 \longrightarrow A_5A_2A_4$ (Not in GNF)

now, Consider $A_6 \longrightarrow A_5A_2A_4$ and replacing A_5 by its RHS, we get

$A_6 \longrightarrow aA_7A_2A_4 | aA_2A_4$ (In GNF)

$A_6 \longrightarrow (A_6A_3A_7A_2A_4 |(A_6A_3A_7A_2A_4$ (In GNF)

Now, Consider $A_6 \longrightarrow A_6A_1A_5$ and removing left recursion, we get

$A_6 \longrightarrow (A_6A_3A_8 |(A_6A_3$

$A_6 \longrightarrow aA_8 | a$

$A_6 \longrightarrow aA_7A_2A_4A_8 | aA_7A_2A_4 | aA_2A_4A_8 | aA_2A_4$

$A_6 \longrightarrow (A_6A_3A_7A_2A_4A_8 |(A_6A_3A_7A_2A_4$

$A_6 \longrightarrow (A_6A_3A_2A_4A_8 |(A_6A_3A_2A_4$

All above A_6 grammar are in GNF.

$A_8 \longrightarrow A_1A_5A_8 | A_1A_5$ (Not in GNF)

(A_8 is a new variable and it is production is not in GNF).

So, now all A_6-productions are in GNF.

Consider A_7-production

$A_7 \longrightarrow A_2A_4A_7 | A_2A_4$

and replacing A_2 by its RHS, we get

$A_7 \longrightarrow +A_4 | +A_4A_7$ (In GNF)

Consider A_8-production

$A_8 \longrightarrow A_1A_5A_8 | A_1A_5$

and replacing A_1 by its RHS, we get

$A_8 \longrightarrow *A_5A_8 | *A_5$ (In GNF)

Now, all the productions are in GNF. Hence grammar is in GNF.

Applications Of CFG:

1. Parser Design
2. Design of Natural Languages Construction.
3. Design of Mark-up Languages.

Defects in CFG:

- It may restrict the grammar from generating the string (No generation of terminal)
- It may involve multiple Names and definitions so confusion.
- Illegal operation (null production misused).
- Symbols not appearing any sentential form.
- In CFG, not necessary to use all symbols.

Push-Down Automata

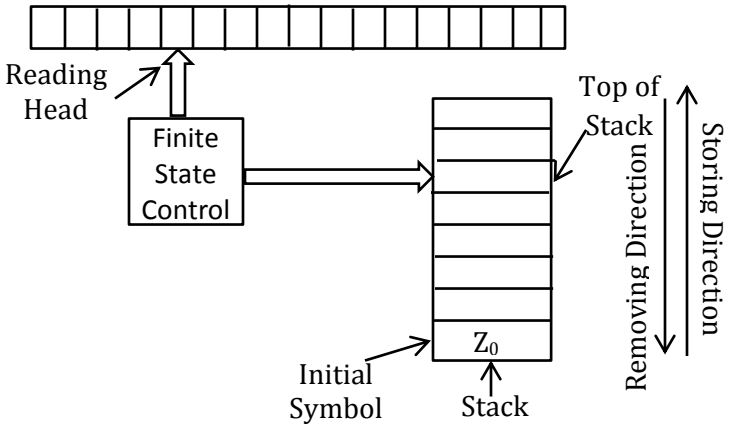
Push-Down Automata: A PDA is a tool to implement context-free languages (CFLs). The biggest drawback of finite automata is absence of memory. there is no mechanism to remember the count of Input symbol or to match.

let us consider L= $\{0^n1^n \mid n \geq 1\}$ .This is a context free language but not regular.

($S \longrightarrow 0S1 | 01$ } generates L.

A Finite automata cannot accept L, i.e. strings of the form 0^n1^n as it has to remember the number of 0's in a string and so it will require an infinite number of states. this difficulty can be avoided by adding an auxiliary memory in the form of stack. To implement the language L on the PDA, the 0's in the given string are added to stack. when a symbol 1 is encountered in the Input string, 0 is removed from the stack. thus the matching of number of 0's and number of 1's is done. An empty stack at the consumption of the string indicates the acceptance of the string by the PDA.

Model of Push-Down Automata: It consists of a finite tape, a reading head, which reads from the tape, a stack memory (Push-down store).



Definition (Mathematical Description of PDA) : A PDA Consists of & tuple (Q, Σ, Γ, δ, q_0, Z_0, F)

1. Q is a finite and non-empty set of states.
2. Σ is a finite and non-empty set of input alphabets.
3. Γ is a set of symbols that can be pushed into the stack.
4. δ is the transition function which maps from $Q \times (\Sigma \cup \{\wedge\}) \times \Gamma$ to $Q \times \Gamma^*$ (finite subset).
5. $q_0 \in Q$ is the initial (starting) state.
6. $Z_0 \in \Gamma$ is the initial (starting) stack symbol.
7. $F \subseteq Q$ is the set of final states.

Moves of Push-down Automata: The move of PD means that what are the options to proceed further after reading inputs in some state and writing some string on the stack. PDA is basically non-deterministic device having some finite number of choices of moves in each situation.

The move will be of two types:

DPDA

1. In first type, an Input symbol is read from the tape, and depending upon the topmost symbol on the stack and present state, PDA has number of choices to proceed next.
   Ex: $\delta (q, a, Z) \vdash \{(q, \beta)\}$
   $q \in Q$, $\beta \in \Gamma^*$.

NDPDA

2. In the second type of move, the input symbol is not read from the tape and topmost symbol of the stack is used. The topmost of the stack is modified without reading the input symbol. It is also known as ∧ - move.

Non-deterministic PDA : NPDA has finite number of choices for its inputs. A NPDA accepts an Input if a sequence of choices lead to some final state or cause PDA to empty its stack.

Deterministic PDA : pda is deterministic if:

1. for each q in Q and Z in Γ, whenever δ(q, ∧, Z) is non empty, then δ(q, a, Z) is empty for all a in ∑.
2. For no q in Q, and Z in Γ, and a in Σ∪ {∧} does δ(q, a, Z) contains more than one element.

Acceptance by PDA: A PDA has final state like $\delta (q_0, \omega, Z_0) \vdash^* (F, \wedge, \gamma)$, $\gamma \in \Gamma^*$

Design a PDA to accept strings with More a's than b's.

Solution:

$\delta (q_0, a, Z_0) \vdash (q_1, aZ_0)$
$\delta (q_0, b, Z_0) \vdash (q_1, bZ_0)$
$\delta (q_1, a, a) \vdash (q_1, aa)$
$\delta (q_1, a, b) \vdash (q_1, \wedge)$
$\delta (q_1, b, a) \vdash (q_1, \wedge)$
$\delta (q_1, b, b) \vdash (q_1, bb)$
$\delta (q_1, \wedge , a) \vdash (q_f, a)$

Design a PDA to accept the language $\{a^n c a^n \mid n \geq 1\}$

Solution:

$\delta (q_0, a, Z_0) \vdash (q_1, aZ_0)$
$\delta (q_1, a, a) \vdash (q_1, aa)$
$\delta (q_1, c, a) \vdash (q_2, a)$
$\delta (q_2, a, a) \vdash (q_2, \wedge)$
$\delta (q_2, \wedge, Z_0) \vdash (q_f, Z_0)$

Design a PDA to accept language $\{a^n b^n \mid n>0\}$

Solution:

$$\begin{aligned}
\delta(q_0, a, Z_0) &\vdash (q_1, aZ_0)\\
\delta(q_1, a, a) &\vdash (q_1, aa)\\
\delta(q_1, b, a) &\vdash (q_2, \wedge)\\
\delta(q_2, b, a) &\vdash (q_2, \wedge)\\
\delta(q_2, \wedge, Z_0) &\vdash (q_f, Z_0)
\end{aligned}$$

Note: LL parsers (left to right scan using Leftmost derivation) usually are top-down parsers, and LR parser (left to right scan; Rightmost derivation in reverse order) usually are bottom –up parsers.

Design a PDA to accept language $\{wcw^R \mid w \in (a+b)^*\}$

Solution:

$$\begin{aligned}
\delta(q_0, a, Z_0) &\vdash (q_1, aZ_0)\\
\delta(q_0, b, Z_0) &\vdash (q_1, bZ_0)\\
\delta(q_1, a, a) &\vdash (q_1, aa)\\
\delta(q_1, b, a) &\vdash (q_1, ba)\\
\delta(q_1, a, b) &\vdash (q_1, ab)\\
\delta(q_1, b, b) &\vdash (q_1, bb)\\
\delta(q_1, c, a) &\vdash (q_2, a)\\
\delta(q_1, c, b) &\vdash (q_2, b)\\
\delta(q_2, a, a) &\vdash (q_2, \wedge)\\
\delta(q_2, b, b) &\vdash (q_2, \wedge)\\
\delta(q_2, \wedge, Z_0) &\vdash (q_f, Z_0)
\end{aligned}$$

Design a PDA for language $L = \{a^n c b^{2n} \mid n \geq 1\}$

Solution:

$$\begin{aligned}
\delta(q_0, a, Z_0) &\vdash (q_1, aaZ_0)\\
\delta(q_1, a, a) &\vdash (q_1, aaa)\\
\delta(q_1, c, a) &\vdash (q_2, a)\\
\delta(q_2, b, a) &\vdash (q_2, \wedge)\\
\delta(q_2, \wedge, Z_0) &\vdash (q_f, Z_0)
\end{aligned}$$

Solution 2:

$$\begin{aligned}
\delta(q_0, a, Z_0) &\vdash (q_1, aZ_0)\\
\delta(q_1, a, a) &\vdash (q_1, aa)\\
\delta(q_1, c, a) &\vdash (q_2, a)\\
\delta(q_2, b, a) &\vdash (q_3, a)\\
\delta(q_3, b, a) &\vdash (q_2, \wedge)\\
\delta(q_2, \wedge, Z_0) &\vdash (q_f, Z_0)
\end{aligned}$$

Design a PDA for language $\{a^n b^m a^n \mid m, n \geq 1\}$.

Solution

$$\begin{aligned}
\delta(q_0, a, Z_0) &\vdash (q_1, aZ_0)\\
\delta(q_1, a, a) &\vdash (q_1, aa)\\
\delta(q_1, b, a) &\vdash (q_2, a)\\
\delta(q_2, b, a) &\vdash (q_2, a)\\
\delta(q_2, a, a) &\vdash (q_3, \wedge)\\
\delta(q_3, a, a) &\vdash (q_3, \wedge)\\
\delta(q_3, \wedge, Z_0) &\vdash (q_f, Z_0)
\end{aligned}$$

Design a PDA for language $L = \{a^n b^{n+2} \mid n \geq 1\}$

Solution:

$$\begin{aligned}
\delta(q_0, a, Z_0) &\vdash (q_1, aZ_0)\\
\delta(q_1, a, a) &\vdash (q_1, aa)\\
\delta(q_1, b, a) &\vdash (q_2, a)\\
\delta(q_2, b, a) &\vdash (q_3, a)\\
\delta(q_3, b, a) &\vdash (q_3, \wedge)\\
\delta(q_3, \wedge, Z_0) &\vdash (q_f, Z_0)
\end{aligned}$$

Parsing: The importance of a grammer is that it provides the platform to describe (generate or produce) a certain language.

1. **Syntax**: It means the grammatical construct of a program.
2. **Semantics**: It means, the meaning of a program. it is all about the operations that are to be carried out by some means.

We define parsing, Parsing is a technique to construct a parse (derivation) tree or to check whether there is some leftmost derivation or not for a given word (string) using a certain grammar.

if the syntax of a given string is correct then the answer is 'YES' otherwise the answer is 'NO'.

In general parsing is categorized into two categories.

1. Top-down parsing
2. Bottom-up parsing

Top-Down Parsing: We start with initial symbol and replace the leftmost variable in each step and finally get the string. This approach is known as top-down parsing. In the parse tree, start symbol is root, terminals are leaves (input symbols) and

other nodes are variables. we start from the root and replacing the intermediate nodes one by one from left to right reach the leaves. this is the reason why this approach is known as top-down approach. This approach is also known as recursive descent parsing.

$S \xrightarrow{*} \omega$

(From the starting symbol to generate sentence)

Note: If a leftmost derivation is constructed by looking k symbols ahead in the given string, then the grammar is known as LL(k) grammar. Due to non-determinism problem in the grammar , we have to use LL(k) approach in the parsing. we can reduce the non-determinism to some extent by converting a grammar in CNF and removing the left recursion and left factoring.

A grammar G having the property to look k ($k \geq 1$) symbols ahead in the given string to parse that given string is known as LL(k) grammar.

Example: let grammar G=({S, A, B}, {a, b}, P, S) where P_1 : $S \longrightarrow aA$

$S \longrightarrow bB$

$S \longrightarrow \wedge$.

$A \longrightarrow aS$ and

$B \longrightarrow bS$

Design the parsing Table.

Variable	Symbols		
	∧	a	b
S	P_3	P_1	P_2
A	ϵ	P_4	ϵ
B	ϵ	ϵ	P_5

How to implement a Parser: A parser is a program, which read input string, using certain context-free grammar and finds a leftmost derivation or construct a parse tree.

There are two forms of parsers:

1. Top-down parsers: Construct parse tree from root toward leaves.
2. Bottom-up parsers: Construct parse tree from leaves to root.

Note: Both top-down and bottom-up parsers scan the input from left to right.

Bottom-up parsing: The basic idea of a bottom-up parser is that we start with input string and replace the terminal(s) by respective variable such that these replacements lead to the starting symbol of the grammar So, in each step, we reduce the input string to reach the starting symbol. This approach is reverse of the top-down approach.

Bottom-up parser reads the input symbol from left and uses rightmost derivation in reverse order like predictive parsing, with most tables, have we use stack to push symbols. If the first few symbol (sentential form) at the top of the stack match the RHS of some variable, then we pop out these symbols from the stack and we push that variable (left-hand-side of production on the stack). This is called a reduction. another name for bottom-up parsers is shift-reduce(SR) parsers.

'Handles' are the sequences (terminals and variables or both) on the stack that match with some RHS in the given grammar productions and after replacement lead to the start symbol. there are three actions in the bottom-up parsing.

1. 'Shift' the current input(token) on the stack and read the next token.
2. 'Reduce' by some suitable LHS of production rule.
3. 'Acceptance' Final reduction, which leads to starting symbol.

Context Free Grammar is G = {V_N, T, P, S }, design the parsing table and parse the string babbba and check $\omega \in L(G)$?

P_1: $S \longrightarrow bAB$
P_2: $S \longrightarrow aAB$
P_3: $A \longrightarrow aA$
P_4: $B \longrightarrow bB$
P_5: $A \longrightarrow b$
P_6: $B \longrightarrow a$

Parsing Table

Variables	Symbols		
	∧	a	b
S	ϵ	P_2	P_1
A	ϵ	P_3	P_5
B	ϵ	P_6	P_4

Suppose: ω = babbba $\in$ L(G) ?

$S \xrightarrow{P_1} bAB \xrightarrow{P_3} baAB \xrightarrow{P_5} babB \xrightarrow{P_4} babbB \xrightarrow{P_4} babbbB \xrightarrow{P_6} babbba$.

Hence babbba $\in$ L(G). YES

Context-Free grammar is G = {V_N, T, P, S}, design the parsing table and parse the string ω= a + a ∗ a.

productions are:

P_1 : S ⟶ F + S

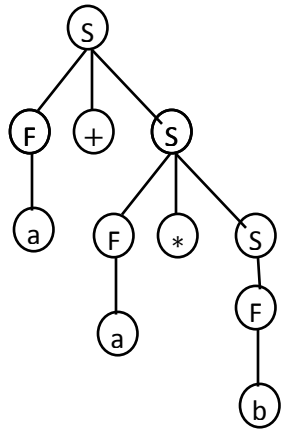
P_2 : S ⟶ F ∗ S

P_3: S ⟶ F

P_4 : F ⟶ a

Parsing Table

	Λ	a	+	∗	a+	+a	a∗	∗a	aa	++	+∗	∗+	∗∗
S	ϵ	P_3	ϵ	ϵ	P_1	ϵ	P_2	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
F	ϵ	P_4	ϵ	ϵ	P_4	ϵ	P_4	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ



$S \xrightarrow{P_1} F + S \xrightarrow{P_4} a + S \xrightarrow{P_2} a + F * S \xrightarrow{P_4} a + a * S \xrightarrow{P_3} a + a * F \xrightarrow{P_4} a + a * a.$

Hence the string a + a ∗ a ∈ L(G). YES

Rules to Construct Top-Down Parser:

Step 1: Remove left recursion, if any, from the production set P.

Step 2: Remove the left factoring, if any, from the production set P.

Step 3: Since the PDA takes one symbol at a time, reduce the given grammar into LL(1).

step 4: Now design the PDA as follow:

4.1 To put the starting symbol S in the pushdown store. make a transition function $\delta(q_0, \Lambda, Z_0) \vdash (q, SZ_0)$ (q_0 ⟶ initial state)

4.2 For every production X ⟶ α in P, make a transition function.

$\delta(q, \Lambda, X) \vdash (q, \alpha)$

4.3 For every terminal a∈Σ, make a transition function.

$\delta(q, a, a) \vdash (q, \Lambda)$

4.4 To indicate the end of the string, we put a marker $ at the end of the string. this marker is used to remove the initial symbol Z_0 from the push-down store to make the stack empty.

$\delta(q, \$, Z_0) \vdash (q_f, \$Z_0)$

Design Parsing table and Top-Down parser for given productions.

E ⟶ E + T

E ⟶ T

T ⟶ T ∗ F

T ⟶ F

F ⟶ (E)

F ⟶ x1

F ⟶ x2

Solution:

Remove Left Recursion and Left Factoring.

P_1: E' ⟶ +TE'

P_2: E ⟶ TE'

P_3: E' ⟶ Λ

P_4: T' ⟶ ∗FT'

P_5: T ⟶ FT'

P_6: T' ⟶ Λ

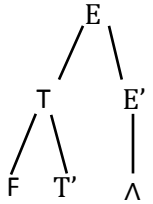
P_7: F ⟶ (E)

P_8: F ⟶ xN

P_9: N ⟶ 1

P_{10}: N ⟶ 2

we have LL(1) grammar.



Parsing Table:

V/T	∧	x	1	2	(	)	+	∗
E	ϵ	P_2	ϵ	ϵ	P_2	ϵ	ϵ	ϵ
E’	P_3	ϵ	ϵ	ϵ	ϵ	ϵ	P_1	ϵ
T	ϵ	P_5	ϵ	ϵ	P_5	ϵ	ϵ	ϵ
T’	P_6	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	P_4
F	ϵ	P_8	ϵ	ϵ	P_7	ϵ	ϵ	ϵ
N	ϵ	ϵ	P_9	P_{10}	ϵ	ϵ	ϵ	ϵ

Parser:

Rule	Input		output	Step
R_0	$\delta(q_0, \Lambda, Z_0)$	⊢	(q, EZ_0)	4.1
R_1	$\delta(q, \Lambda, E)$	⊢	(q, TE’)	4.2
R_2	$\delta(q, \Lambda, E')$	⊢	(q, +TE’)	4.2
R_3	$\delta(q, \Lambda, E')$	⊢	(q, ∧)	4.2
R_4	$\delta(q, \Lambda, T)$	⊢	(q, FT’)	4.2
R_5	$\delta(q, \Lambda, T')$	⊢	(q, ∗FT’)	4.2
R_6	$\delta(q, \Lambda, T')$	⊢	(q, ∧)	4.2
R_7	$\delta(q, \Lambda, F)$	⊢	(q, (E))	4.2
R_8	$\delta(q, \Lambda, F)$	⊢	(q, xN)	4.2
R_9	$\delta(q, \Lambda, N)$	⊢	(q, 1)	4.2
R_{10}	$\delta(q, \Lambda, N)$	⊢	(q, 2)	4.2
R_{11}	δ (q, 1, 1)	⊢	(q, ∧)	4.3
R_{12}	δ (q, 2, 2)	⊢	(q, ∧)	4.3
R_{13}	δ (q, (, ( )	⊢	(q, ∧)	4.3
R_{14}	δ (q,),))	⊢	(q, ∧)	4.3
R_{15}	δ (q, x, x)	⊢	(q, ∧)	4.3
R_{16}	δ (q, +, +)	⊢	(q, ∧)	4.3
R_{17}	δ (q, ∗, ∗)	⊢	(q, ∧)	4.3
R_{18}	$\delta(q, \$, Z_0)$	⊢	$(q_f, \$Z_0)$	4.4

Consider ω = x1 + (x2). parse on above PDA.

State	Unread Input	PushDown store	Rule used
q_0	x1 + (x2) $	Z_0	
q	x1 + (x2) $	EZ_0	R_0
q	x1 + (x2) $	$TE'Z_0$	R_1
q	x1 + (x2) $	$FT'E'Z_0$	R_4
q	x1 + (x2) $	$xNT'E'Z_0$	R_8
q	1 + (x2) $	$NT'E'Z_0$	R_{15}
q	1 + (x2) $	$1T'E'Z_0$	R_9
q	+ (x2) $	$T'E'Z_0$	R_{11}
q	+ (x2) $	$E'Z_0$	R_6
q	+ (x2) $	$+TE'Z_0$	R_2
q	(x2) $	$TE'Z_0$	R_{16}
q	(x2) $	$FT'E'Z_0$	R_4
q	(x2) $	$(E)T'E'Z_0$	R_7
q	x2) $	$E)T'E'Z_0$	R_{13}
q	x2) $	$TE')T'E'Z_0$	R_1
q	x2) $	$FT'E')T'E'Z_0$	R_4
q	x2) $	$xNT'E')T'E'Z_0$	R_8
q	2) $	$NT'E')T'E'Z_0$	R_{15}
q	) $	$2T'E')T'E'Z_0$	R_{10}
q	) $	$T'E')T'E'Z_0$	R_{12}
q	) $	$E')T'E'Z_0$	R_6
q	) $	$)T'E'Z_0$	R_3
q	$	$T'E'Z_0$	R_{14}
q	$	$E'Z_0$	R_6
q	$	Z_0	R_3
q_f		$\$Z_0$	R_{18}

Rules to Construct bottom-up parser:

Rule 1: $\delta(q, a, Z_0) \vdash (q, aZ_0)$ $a \in T$

Rule 2: $\delta(q, a, \Lambda) \vdash (q, a)$ $a \in T$

Rule 3: $\delta(q, \Lambda, a^R) \vdash (q, A)$ $a \in T$, a^R is reverse of a. for each $A \rightarrow \alpha$

Rule 4: $\delta(q, \Lambda, S) \vdash (q_f, \Lambda)$

Design bottom-up parser for given productions

Productions:

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow x1$

$F \rightarrow x2$

Solution:

Here E is the starting symbol and +, ∗, (,), x1 & x2 are terminals. a ∈ T.

Productions	Input		Output	Rule
P_1	$\delta(q, a, Z_0)$	⊢	(q, aZ_0)	1
P_2	$\delta(q, a, \Lambda)$	⊢	(q, a)	2

P_3	$\delta(q, \wedge, T+E)$	$\vdash$	(q, E)	3	$E \longrightarrow E + T$
P_4	$\delta(q, \wedge, T)$	$\vdash$	(q, E)	3	$E \longrightarrow T$
P_5	$\delta(q, \wedge, F*T)$	$\vdash$	(q, T)	3	$T \longrightarrow T * F$
P_6	$\delta(q, \wedge, F)$	$\vdash$	(q, T)	3	$T \longrightarrow F$
P_7	$\delta(q, \wedge,)E()$	$\vdash$	(q, F)	3	$F \longrightarrow (E)$
P_8	$\delta(q, \wedge, 1x)$	$\vdash$	(q, F)	3	$F \longrightarrow x1$
P_9	$\delta(q, \wedge, 2x)$	$\vdash$	(q, F)	3	$F \longrightarrow x2$
P_{10}	$\delta(q, \wedge, E)$	$\vdash$	$(q_f, \wedge)$	4	

Consider ω = x1 +(x2) parse on PDA.

State	Rest of Input	Stack	action	Production
q	x1 + (x2)	Z_0		
q	1 + (x2)	x Z_0	Shift	P_1
q	+ (x2)	1x Z_0	Shift	P_2
q	+(x2)	F Z_0	Reduce	P_8
q	+ (x2)	T Z_0	Reduce	P_6
q	+ (x2)	E Z_0	Reduce	P_4
q	(x2)	+E Z_0	Shift	P_2
q	x2)	(+E Z_0	Shift	P_2
q	2)	x(+E Z_0	Shift	P_2
q	)	2x(+E Z_0	Shift	P_2
q	)	F(+E Z_0	Reduce	P_9
q	)	T(+E Z_0	Reduce	P_6
q	)	E(+E Z_0	Reduce	P_4
q	∧	)E(+E Z_0	Shift	P_2
q	∧	F+E Z_0	Reduce	P_7
q	∧	T+EZ_0	Reduce	P_6
q	∧	EZ_0	Reduce	P_3
q_f	∧	Z_0	Reduce	P_{10}

Design a CFG for the language L = $\{0^n1^n \mid n \geq 0\}$

Solution: At n = 0, ∈ becomes a valid string in the language L. The corresponding production is S → ∈.

To incorporate the equal numbers of 0s and 1s, the production used is S → 0S1. Every transitive substitution of S inserts an equal number of 0s and 1s.

Hence, the production set for the CFG G corresponding to CFL L is

S → ∈ | 0S1 | 01

Design a CFG for the language L = $\{ 0^n1^{2n} \mid n \geq 0\}$

Solution: At n = 0, ∈ becomes a valid string in the language L. The corresponding production is S → ∈.

To incorporate the double numbers of 1s as 0s, the production used in S → 0S11. Every transitive substitution of S inserts double the number of 1s following 0s.

S → ∈ | 0S11 | 011

Design a CFG for the language L over { 0, 1 } to generate all possible strings of even length.

Solution: The number 0 is considered to be even. The string of length 0 is ∈ and the corresponding production is S → ∈. A String of length 2 over {0, 1} consists of one of the strings 00, 01, 10, or 11. The corresponding productions are S → 00 | 01 | 10 | 11.

An even length of string of length 2n (n > 1) contains one or more repetitions of the strings 00, 01, 10, or 11. The corresponding productions are S → 00 | 01 | 10 | 11.

Hence, the productions set for the CFG G corresponding to the CFL L is

S → ∈ | 00 | 01 | 10 | 11 | 00S | 01S | 10S | 11S

Design a CFG for the language L over {0, 1} to generate all the strings having alternate sequence of 0 and 1.

Solution: The minimum string length in L is 2. The corresponding strings are 01 or 10. If a string ω in L begins with 01, then it should follow a repetition of 01 and terminate with either 01 or 0, and if it begins with 10, then it should follow a repetition of 10 and terminate either with 10 or 1. The corresponding productions in CFG are

S → 01 | 10 | 01A | 10B

A → 01A | 0 | 01

B → 10B | 1 | 10

Design a CFG generating the set of odd-length strings in $\{a, b\}^*$ with middle symbol a.

Solution: $S \rightarrow aSa \mid aSb \mid bSa \mid bSb \mid a$.

Design a CFG generating the set of even-length strings in $\{a, b\}^*$ with the two middle symbols.

Solution: $S \rightarrow aSa \mid aSb \mid bSa \mid bSb \mid aa \mid bb$

Design a CFG generating the set of odd-length strings in $\{a, b\}^*$ whose first, middle, and last symbols are all the same.

Solution: $S \rightarrow aTa \mid bUb$

$T \rightarrow aTa \mid aTb \mid bTa \mid bTb \mid a$

$U \rightarrow aUa \mid aUb \mid bUa \mid bUb \mid b.$

Design the CFG equivalent to a Regular Expression $(011 + 1)^* (01)^*$.

Solution: $S \rightarrow AC$

$A \rightarrow \Lambda|BA$

$B \rightarrow 011 \mid 1$

$C \rightarrow \Lambda \mid DC$

$D \rightarrow 01$

Design the CFG equivalent to a Regular Expression $(100 + 1) (1)^* (011 + 1)^+$ Solution:

$S \rightarrow ABD$

$A \rightarrow 100 \mid 1$

$B \rightarrow \Lambda \mid 1B$

$C \rightarrow D \mid DC$

$D \rightarrow 011 \mid 1$

Design the CFG equivalent to language $L = \{ 0^i1^j0^k \mid j = I + k \}$

Solution: $L = 0^i 1^{i+k} 0^k$

$0^i 1^i 1^k 0^k$

$S \rightarrow AB$

$A \rightarrow 0A1 \mid \Lambda$

$B \rightarrow 1B0 \mid \Lambda$

Design the CFG equivalent to language $L = \{ 0^i1^j0^k \mid j > i + k \}$

Solution: $j = i + j + k$

$0^i 1^{i+j+k} 0^k$

$0^i 1^i 1^m 1^k 0^k$

$S \rightarrow BCD$

$B \rightarrow 0B1 \mid \Lambda$

$D \rightarrow 1D0 \mid \Lambda$

$C \rightarrow 1 \mid 1C$

Turing Machine

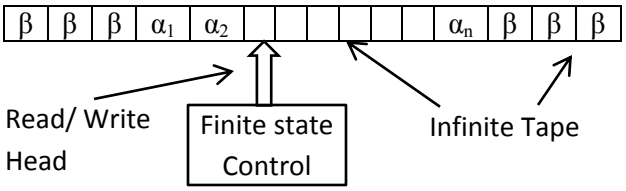
We need such machines that can perform recognizing as well as computation. Without computational capability a machine is not much useful as desired. We have computer system, which is capable of doing recognizing as well as computational. so, we are interested in designing of an automaton(machine) that can solve both objectives.

1. Recognizing
2. Computation

A Turing machine is a generalization of a PDA, which uses a ∞-tape instead of 'a tape and stack'. the potential length of the tape is assumed to be infinite and divided into cells. One cell can hold one input symbol. The head of TM is capable to read as well as write on the tape and move left or right or remain stationary. Turing machines accept the languages are known as type 0 languages or the recursively enumerable (RE) languages.

Note: One thesis called Church's thesis support the fact that a Turing machine can simulate the behavior of a general-purpose computer system. The church-turing thesis states that any algorithm procedure that can be carried out by human begins/computer can be carried out by turing machine. It has been universally accepted by computer scientist that the turing machine provides an ideal theoretical model of a computer.

Model of a Turing Machine: The turing machine can be thought of as finite control connected to a R/W head. It has one tape which is divided into a number of cells. The block diagram of the basic model for Turing is



Each cell can store only one symbol. The input to and output from the the finite state automaton are effected by the R/W head which can examine one cell at a time. In one move, the machine examines the present symbol under the R/W head on the tape and the present state of an automaton to determine

i. a new symbol to be written on the tape in the cell under the R/W head.
ii. a motion of the R/W head along the tape: either the head moves one cell left(L), or one cell right(R).
iii. The next state of the automaton, and
iv. Whether to half or not.

Mathematical description of a Turing Machine:

A Turing Machine M is described by 7-tuple namely $(Q, \Sigma, \Gamma, \delta, q_0, \beta, F)$, where

i. Q is a finite non empty set of states.
ii. Σ is a non empty set of symbols,
iii. Γ is a finite non-empty set of symbols including β. $\beta \cup \Sigma = \Gamma$,
iv. δ is a transition function which maps from $Q \times \Gamma$ to $Q \times \Gamma \times (L|R|S)$
i.e. $(q, x) \vdash (q', y, D)$ D is direction of movement of R/W head: D= L or R or S according as the movement is to the left or right or stationary.

Note: i. The acceptability of a string is decided by the reachability from the initial state to some final state.
ii. δ may not be defined for some elements of $Q \times \Gamma$.

v. $q_0 \in Q$ is the initial state.
vi. $\beta \in \Gamma$ represent the blank on the tape,
vii. F is set of final states.

Depending upon the number of moves in transition, a TM be deterministic or non-deterministic. If TM has almost one move in a transition then it is called deterministic TM (DTM), if one or more than one move is in a transition table it is called non-deterministic TM (NTM). A standard TM reads one cell of the tape and changes its state (optional) and moves left or right one cell.

The halt state(h) means the stopping point of processing by a TM when TM processes some input then TM may fall into one of these given below:

a. Halting State(h)
b. Non-Halting State(non-final)
c. Loop (finite or infinite)
d. Crash and
e. Hang State (Undefined Transition)

When TM reads inputs on the tape then it keeps write on the tape and when it halts or stops the processing then whatever written on the tape is the output of the TM. So, we have no need to have final states as such. When TM does not stop its processing and it is on for infinite time this situation is known as infinite loop and when head damages the tape then it is known as crash. for some input, if TM does not decide what to do, then we will say that TM is in hang state. A TM can stop its processing in halting state or in non-halting states.

Halting Problem: A turing machine accepts a language means that the TM will stop eventually in a halting state for each word that is in the language. Halting state does not mean the accepting state but it is additional state similar to accepting state. However, if the word is not in the language then Turing machine may stop in halting or non-halting state or loop forever or crash. In this case, we can never be sure whether the given word is accepted or rejected i.e. the Turing machine does not decide the word problem (known as halting problem of TM).

Note: All languages (Regular Language, CFL, CSL, Type 0 (pharse structured) language) are accepted by TM. A language that is accepted by a TM is known as recursive enumerable(R.E.). being enumerable means, a TM precisely lists all the strings of the language. Some type-0 languages are not decidable by TMs known as recursive set (language). For a recursive language a TM always halts for its member strings and rejects other.

Computation with Turing Machine: we represent integer numbers on the tape in forms of 0. suppose i is a positive integer, the 0^i represents this integer numbers.

For example:

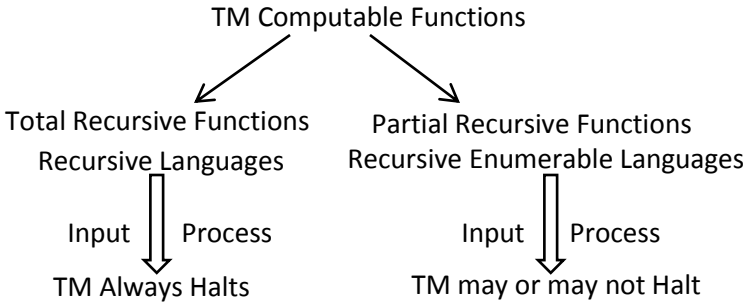
1. 5 is represented by 0^5 as shown in figure.

β	β	β	0	0	0	0	0	β	β	β

2. integers 2 and 4 are represented by $0^2 1 0^4$ as shown in figure.

β	β	β	0	0	1	0	0	0	0	β	β	β

Note: If f_1 is defined for all the arguments a_1, a_2, . . . ,a_n then f_1 is known as total recursive function. "The total recursive function is analogous to recursive langages". A function $f(a_1, a_2, \ldots, a_n)$ computed by a TM is known as partial recursive function analogous to the recursive enumerable set (RE languages).



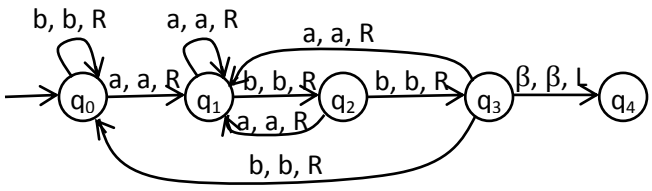
UNIVERSAL TURING MACHINE (UTM): A Universal Turing machine is a specific Turing machine that can simulate the behavior of any TM. Alan Turing describes such a machine. Its existence means that there is a turing machine known as UTM, which is capable of running any algorithm. The key idea is to think of the description of a Turing machine itself as data. we think a UTM as a reprogrammable TM.
A universal Turing Machine takes as input the description of a turing machine along with the initial tape contents, and simulates the behavior of any individual TM. Building such an UTM appears to be a daunting task.

Problem: Design a TM for L = { w: w ∈ $(a + b)^*$ } ending in substring abb.

Solution:

State	a	b	β
$\rightarrow q_0$	q_1, a, R	q_0, b, R	___
q_1	q_1, a, R	q_2, b, R	___
q_2	q_1, a, R	q_3, b, R	___
q_3	q_1, a, R	q_0, b, R	q_f, β, L
$*q_f$	___	___	___

Transition Graph:



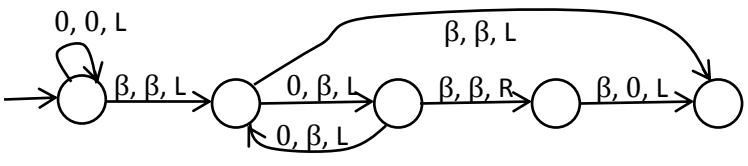
TM for L = $(a + b)^*$abb

Problem: Design a TM for finding n mod 2.

Solution:

State	0	β
$\rightarrow q_0$	q_0, 0 , R	q_1, β, L
q_1	q_2, β, L	q_f, β, L
q_2	q_1, β, L	q_3, β, R
q_3	__	q_f, 0, L
$*q_f$	__	__

Transition Graph For n mod 2



The Encoding of a simple TM (The Encoding function e): First we associate to each tape symbol (including β), to each state (including accepting and rejecting) and to each of the three directions, a string of 0;s. let

$S(\beta) = 0$

$S(a_i) = 0^{i+1}$

Result of String

$S(A) = 0$ Acceptance

$S(R) = 00$ Rejection

States

$S(q_i) = 0^{i+2}$ State on the TM

Direction

$S(S) = 0$ Stationary Direction on TM

$S(L) = 00$ Left Direction on TM

$S(R) = 000$ Right Direction on TM

Problem: Design a TM over $\Sigma = \{1\}$ to accept the language $L = \{ 1^m \mid m \text{ is odd} \}$.

Solution:

State ↓	1	β
$\rightarrow q_0$	q_1, 1, R	__
$*q_1$	q_0, 1, R	__

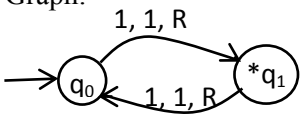
The asterisk (*) before q_1 indicates that q_1 is a final state.

let consider the string 1111

$\beta\beta q_0 1111\beta\beta \rightarrow \beta\beta 1 q_1 111\beta\beta \rightarrow \beta\beta 11 q_0 11\beta\beta \rightarrow \beta\beta 111 q_1 1\beta\beta \rightarrow \beta\beta 1111 q_0 \beta\beta$

Since the TM halts on q_0, the string 1111 is not accepted.

Transition Graph:



Problem: Design TM over $\Sigma = \{0, 1\}$ to accept the language $L = \{0^m 1^m \mid m > 0 \}$.

Solution: The example strings in the language L are 01, 0011, 000111, 00001111, . . .

let the string be 000111

The string recognition process of the TM is:

1. Read the first symbol 0 of the string and convert this 0 to blank(β).
2. The R/W head keeps moving towards the right till the input string is crossed and the first blank symbol β after the string is reached.
3. From this blank, turn one cell left and reach the last symbol of the input string. If it is 1, convert it into a blank.
4. Now, after converting the last 1 symbol to β, the R/W head keeps moving towards the left till a blank symbol β is reached.
5. As the first blank is found move the head toward right and repeat step 1 to step 4 till all 0's are cancelled by the corresponding 1's.
6. If the cancellation process is completed successfully, then the string is accepted, otherwise there are residual un-cancelled 0s and 1s, then the string is not accepted.

Current State ↓	Input Symbol 0	1	β
$\rightarrow q_0$	(q_1, β, R)	__	__
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_2, β, L)
q_2	__	(q_3, β, L)	
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_4, β, R)
q_4	(q_1, β, R)	__	(q_f, β, R)
$*q_f$	__	__	__

Design a Turing Machine over $\Sigma = \{ 0 \}$ to accept the language $L = \{0^m \mid m \text{ is even}\}$.

Solution: The problem is similar to the previous one. The only change is that now q_0 is a final state and q_1 is non-final state. The transition table shown in table provides the complete set of transition function.

Current state ↓	Input Symbol 0	β
$\rightarrow *q_0$	$(q_1, 0, R)$	__
q_1	$(q_0, 0, R)$	__

Design a Turing machine over $\Sigma = \{0\}$ to accept the language $L = \{ 0^m \mid m \text{ is multiple of } 3\}$.

Solution: The example strings in the language L are ϵ, 000, 000000, 000000000, … Normally, the R/W head points to the first 0 of the input string ω in the state q_0. If the length of the string ω, $|\omega| = 0$, then it is a null string and the R/W

points to an arbitrary blank on the Turing tape in the state q_0. In this case, the Turing machine halts. Since zero is a multiple of 3, the Turing machine has to halt in the final state. Therefore, q_0 is declared as a final state.

Current state ↓	Input Symbol 0	β
$\rightarrow{}^*q_0$	$(q_1, 0, R)$	—
q_1	$(q_2, 0, R)$	—
q_2	$(q_0, 0, R)$	—

Design a Turing machine over $\Sigma = \{0, 1\}$ to accept the language $L = \{ 0^m 1^{2m} \mid m > 0\}$.

Solution: The example strings in the language L are 011, 001111, 000111111,… The r/w head points to the first symbol of the input string in the string in the state q_0.

The recognition process of the string on the Turing machine is as follows:

1. The r/w head reads the first symbol 0 of the input string and converts this 0 into a blank (β).
2. The r/w head keeps moving towards the right till the input string is crossed and the first blank after the string is reached.
3. From this blank, the r/w head turns one cell to the left and reaches the last symbol of the input string. If the last two symbols in the string are 11, then the match occurs.
   a. If the match occurs, then the r/w head converts both 1 symbols to blanks one by one. The r/w head traverses back to the beginning of the remaining string and perform the matching and cancellation of one 0 by 11. This process is repeated till all 0s are matched by the corresponding 11 pair. If this occurs, then it indicates the acceptance of the string.
   b. If the match does not occur at any stage, the process stops in a non-final state and indicates the rejection of the string.

The transition table is:

Current State ↓	Input Symbol 0	1	β
$\rightarrow q_0$	$(q_1, β, R)$	—	—
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	$(q_2, β, L)$
q_2	—	$(q_3, β, L)$	—
q_3	—	$(q_4, β, L)$	—
q_4	—	$(q_5, 1, L)$	$(q_6, β, R)$
q_5	$(q_5, 0, L)$	$(q_5, 1, L)$	$(q_0, β, R)$
q_6	—	—	$(q_f, β, R)$
$*q_f$	—	—	—

Design a Turing machine to compute m – n where m and n are positive integers and m > n.

Current State ↓	Input Symbol 0	1	β
$\rightarrow q_0$	$(q_1, β, R)$	—	—
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	—
q_2	$(q_2, 0, R)$	—	$(q_3, β, L)$
q_3	$(q_4, β, L)$	$(q_f, 0, L)$	—
q_4	$(q_4, 0, L)$	$(q_5, 1, L)$	—
q_5	$(q_5, 0, L)$	—	$(q_6, β, R)$
q_6	$(q_1, β, R)$	—	—
$*q_f$	—	—	—

Design a Turing machine to compute m × n where m and n are positive integers.

Curren State↓	Input State 0	1	x	y	β
$\rightarrow q_0$	(q_1,x, R)	$(q_9,1,L)$	—	—	—
q_1	$(q_1,0,R)$	$(q_2,1,R)$	—	—	—
q_2	(q_3,y,R)	$(q_7,1,L)$	—	—	—
q_3	$(q_3,0,R)$	$(q_4,1,R)$	—	—	—
q_4	$(q_4,0,R)$	—	—	—	$(q_5,0,L)$
q_5	$(q_5,0,L)$	$(q_6,1,L)$	—	—	—
q_6	$(q_6,0,L)$	—	—	(q_2,y,R)	—
q_7	—	$(q_8,1,L)$	—	$(q_7,0,L)$	—
q_8	$(q_8,0,L)$	—	(q_0,x,R)	—	—
q_9	—	—	$(q_9,0,L)$	—	$(q_f. β, R)$
$*q_f$	—	—	—	—	—

Design a TM over Σ = {a, b} to accept the language L = {ww^R | w ∈ $(a, b)^*$ }

Solution:

Current State ↓	Input Symbol a	b	β
$\longrightarrow q_0$	(q_1, β, R)	(q_4, β, R)	(q_f, β, R)
q_1	(q_1, a, R)	(q_1, b, R)	(q_2, β, L)
q_2	(q_3, β, L)		
q_3	(q_3, a, L)	(q_3, b, L)	(q_0, β, R)
q_4	(q_4, a, R)	(q_4, b, R)	(q_5, β, R)
q_5	—	(q_3, β, L)	—
$*q_f$	—	—	—

Design a TM to compute m+n where m and n are positive integers.

Sol: Initially two integers m and n are separated by '1' symbol on the Turing tape.

β	β	β	0	0	0	1	0	0	0	0	0	β	β	β

The numbers to be added 3 and 5.

After the addition operation, the Turing tape contains the result of addition.

β	β	β	0	0	0	0	0	0	0	0	β	β	β

Current State ↓	Input Symbol 0	1	β
$\longrightarrow q_0$	$(q_0, 0, R)$	$(q_1, 0, R)$	—
q_1	$(q_1, 0, R)$	—	(q_2, β, R)
q_2	(q_f, β, L)	—	—
$*q_f$	—	—	—

Post Correspondence Problem (PCP): Post's correspondence problem is a combinatorial problem formulated by Emil Post in 1946. This problem has many applications in the field theory of formal languages.

Definition: A correspondence system P is a finite set of ordered pairs of no empty strings over some alphabet.

Let Σ be an alphabet, then P is finite subset of $\Sigma^+ \times \Sigma^*$. A match or solution P is any string w∈ Σ such that pairs $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n) \in P$ and $w = u_1 u_2 u_3 \ldots u_n = v_1 v_2 v_3 \ldots v_n$ for some n > 0. The selected pairs $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ are not necessary distinct.

Let string $u_1 u_2 u_3 \ldots u_n$ are in U and string $v_1 v_2 v_3 \ldots v_n$ are in V, then

U = { $u_1 u_2 u_3 \ldots u_m$ } and V = { $v_1 v_2 v_3 \ldots v_m$} for some m>0.

PCP is to determine whether there is any match or not for a given correspondence system.

The PCP is to determine whether or not there exist I, where $1 \le i \le m$ such that

$u_{i1} \ldots u_{in} = v_{i1} \ldots v_{in}$

Example: Does the PCP with two lists u = (b, bab^3, ba) and v = (b^3, ba, a) have a solution.

bab^3	b	b	ba	=	ba	b^3	b^3	a
u_2	u_1	u_1	u_3		v_2	v_1	v_1	v_3

Example: Does the PCP given below have a solution?

	1	2	3	4	5
u_i	10	01	0	100	1
v_i	101	100	10	0	010

Solution:

10	1	01	0	100	100	0	100
101	010	100	10	0	0	10	0
1	5	2	3	4	4	3	4

Note: If the first substring used in PCP is always x_i and y_i, then PCP is known as the modified Post Correspondence Problem.

Unrestricted Grammars: An Unrestricted or phrase-structure grammar is a 4-tuple G= (V_N, T, P, S) where V_N and T are disjoint sets of variables and terminals respectively, S is an element of V called the start symbol; and P is a set of productions of the form

$$\alpha \longrightarrow \beta$$

where α, β ∈ $(V_N \cup T)^*$ and α contains at least one variable.

Context-Sensitive Grammar and Languages:

A Context-Sensitive grammar (CSG) is an unrestricted grammar in which every production has the from $\alpha \longrightarrow \beta$

$$\alpha \longrightarrow \beta \quad \text{with } |\beta| \geq |\alpha|$$

A context-Sensitive language (CSL) is a language that can generated by such a grammar.

A slightly different characterization of these language makes it easier to understand the phrase context sensitive. A language is context-sensitive if and only if it can be generated by a grammar in which every production has the form

$$\alpha A \beta \longrightarrow \alpha X \beta$$

Where α, β, and X are the strings of variables and/or terminals, with X not null, and A is a variable such a production may allow A to be replaced by X. depending on the context.

The production of type $S \longrightarrow \wedge$ is allowed in type 1 if $\wedge$ is in L(G), but S should not appear on right hand side of any production.

For example, Productions $S \longrightarrow AB$, $S \longrightarrow \wedge$, $A \longrightarrow c$ are type 1 productions, but the production of type $A \longrightarrow Sc$ is not allowed. Atmost every language can be thought as CSL.

Note: If left or right context is missing then we assume that $\wedge$ is the context.

Relation between Languages of Classes (Languages And Their Relation): let l_0, l_{csl}, l_{cfl}, l_{rl} denote the family of type 0 languages, context sensitive languages, context free languages and regular languages respectively.

Property 1: From the definition, it follows that $l_{rl} \subseteq l_0$, $l_{cfl} \subseteq l_0$.

Property 2: $l_{cfl} \subseteq l_{csl}$. The inclusion relation is not immediate as we allow $A \longrightarrow \wedge$ in context free grammar even when $A \neq S$, but not in context-sensitive grammars (we allow only $S \longrightarrow \wedge$ in context sensitive grammars). a context free grammar G with productions of the $A \longrightarrow \wedge$ is equivalent to a context-free grammar G_1 which has no productions of the form $A \longrightarrow \wedge$ (except $S \longrightarrow \wedge$). Also, when G_1 has $S \longrightarrow \wedge$, S does not appear on the right-hand side of any production. So G_1 is context sensitive. this proves $l_{cfl} \subseteq l_{csl}$.

Property 3: $l_{rl} \subseteq l_{cfl} \subseteq l_{csl} \subseteq l_0$. this follows from properties 1 and properties 2.

Property 4: $l_{rl} \subset_{\neq} l_{clf} \subset_{\neq} l_{csl} \subset_{\neq} l_0$.

Computability (Basic Concept):

we start with the definition of partial and total functions.

$\longrightarrow$ A partial function f from X to Y is a rule which assigns to every element of X atmost one element of Y.

$\longrightarrow$ A total function from X to Y is a rule which assigns to every element of X a unique element of Y. for example:, If R denotes the set of all real numbers, the rule f from R to itself given by $f(r) = +\sqrt{r}$ is a partial function since f(r) is not defined as a real number when r is negative. But $g(r) = 2r$ is a total function from R to itself.

Remarks: A partial or total function from X^k to X is also called a function of k variables and denoted by $f(x_1, x_2, \ldots, x_n)$. for example, $f(x_1, x_2) = 2x_1 + x_2$ is a function of two variables. f(1,2) = 4, 1 and 2 are called arguments and 4 is called a value. $g(w_1, w_2) = w_1w_2$ is a function of two variables ($w_1, w_2 \in \Sigma^*$); g(ab,aa) = abaa, ab and aa are called arguments and abaa is a value.

Primitive Recursive Functions:

A function is primitive recursive if

1. It is an initial function
2. It is obtained from recursion or composition of initial functions.

Most Popular total functions are primitive recursive functions; however some total functions are not. for example, the ackerman function is total but not primitive recursive.

Initial functions: the initial funvctions over N are given below:

S(4) = 5, Z(7) = 0

$U^3_2\{2, 4, 7\} = 4$, $U^3_1\{2, 4, 7\} = 2$, $U^3_3\{2, 4, 7\} = 7$

Zero function Z defined by $Z(x) = 0$.

Successor function S defined by $S(x) = x+1$.

Projection function $U^N_i\{x_1, x_2, \ldots, x_N) = x_i$

The initial functions over Σ are given below.

nil(abab) = ∧

cons a(abab) = aabab

cons b(abab) = babab

nil(x) = ∧

cons a(x) = ax

cons b(x) = bx

Note: we note that cons a(x) and cons b(x) simply denote the concatenation of the 'constant' string a and x, and concatenation of the 'constat' string b and x.

definition: If $f_1, f_2, \ldots, f_n$ are partial functions of n variables and g is a partial function of k variables then the composition of g with $f_1, f_2, \ldots, f_k$ is a partial function of n variables defined by $g(f_1(x_1, x_2, \ldots, x_n), f_2(x_1, x_2, \ldots, x_n), \ldots, f_k(x_1, x_2, \ldots, x_n))$.

Example: let $f_1(x, y) = x + y$,

$f_2(x, y) = 2x$,

$f_3(x, y) = xy$ and

$g(x, y, z) = x + y + z$ be a function over N.

then $g(f_1(x, y), f_2(x, y), f_3(x, y)) = g(x + y, 2x, xy) = x + y + 2x + xy$.

Example:

let $f_1(x_1, x_2) = x_1x_2$,

$f_2(x_1, x_2) = ∧$

$f_3(x_1, x_2) = x_1$ and

$g(x_1, x_2, x_3) = x_2x_3$ be the function over Σ.

Then $g(f_1(x_1, x_2), f_2(x_1, x_2), f_3(x_1, x_2))$

$= g(x_1x_2, ∧, x_1) = ∧x_1 = x_1$

Example:

Define n! by recursion

Solution:

$f(0) = 1$ and $f(n+1) = h(n, f(n))$

where $h(x,y) = S(x) * y$.

Definition: A total function over N is called primitive recursive.

1. If it is any one of the three initial functions, or
2. If it can be obtained by applying composition and recursion a finite number of times to the set of initial functions.

Example: Show that the function $f_1(x, y) = x + y$ is primitive recursive OR addition function of two integers are primitive recursive.

Solution: f_1 is a function of two variables. If we want f_1 to be defined by recursion, f_1 for x and y (in equation).

$f_1(m, 0) = m$

$f_1(m, n+1) = f(m, n) + 1 = S(f_1(m, n))$

for example. consider $f_1(5, 2)$

$f_1(5, 2) = S(f_1(5, 1))$

$= f_1(5, 1,) + 1$

$= S(f_1(5, 0)) + 1$

$= f_1(5, 0) + 1 + 1$

$= 5 + 1 + 1$

$= 7$.

Hence f_1 is a primitive recursive function.

Example: The function $f_2(x, y)$ $x * y$ is primitive recursive.

Solution: As multiplication of Two natural number is simply repeated addition, f_2 has to be primitive recursive. we prove this as follow

$f_2(x, 0) = 0$

$f_2(x, y+1) = x * (y + 1)$

$= f_2(x, y) + x$

$= f_1(f_2(x, y), x)$

Example: Show that the concatenation of two string over {a, b} is primitive recursive.

solution:

we have defined concatenation as

$f(\wedge, w_2) = w_2$

$f(a_1, a_2, \ldots, a_n, w_2) = a_1 f_1(a_2, \ldots, a_n, w_2)$ where $w_1, w_2 \in \Sigma^*$.

we defined f using initial functions a follows:

$f(\wedge, w_1) = \text{cons } \wedge(w_1) = w_1$ and

$f(a_1, a_2, \ldots, a_n, w_2) = \text{cons } a_1 (f(a_2, \ldots, a_n, w_2))$

for example, consider f(ab, cd),

f(ab, cd) = cons a f(b, cd)

= af(b, cd)

= acons bf(∧, cd)

= ab f(∧, cd)

= abcd (since $f(a,w_1) = w_1$)

Hence concatenation is primitive recursive function.

Application of FA:

1. NFA and DFA are used in passes parser construction of Compliers.
2. Pattern Recognition
3. Programming language processing -in scanning phase of compilation for recognizing tokens.
4. Text pattern matching
5. Signal processing
6. Design Controllers for systems (Washing Machines, Industrial processes).

Detail (Halting Problem): The halting problem is a decision problem which is informally stated as follows:

"Given a description of an algorithm and a description of its initial arguments, determine whether the algorithm, when executed with these arguments, ever halts. The alternative is that a given algorithm runs forever without halting.

Alan Turing proved in 1936 that there is no general method or algorithm which can solve the halting problem for all possible inputs. An infinite or finite in length depending on the input and behavior of the algorithm usually depends on the input size. Algorithm may consist of various number of loops, nested or in sequence. The HP asks the question:

Given a program and an input to the program, determine if the program will eventually stop when it is given that input?

One thing we can do here to find the solution of HP. let the program run with the given input and if the program stops and we conclude that problem is solved. But, if the program does not stop on a reasonable amount of time, we can not conclude that it would not stop. The question is: "How long we can wait… ?". The waiting time may be long enough to exhaust whole life. So, we can not take it as easier as it seems to be we want specific answer either "YES" or "NO", and hence some algorithm to decide answer.

The importance of the halting problem lies in the fact that it is the first problem which was proved undecidable.

CHURCH'S Thesis: A FA is a device having finite states and operates on a string of symbols with the fixed and finite set of rules. A PDA and TM are more powerful machines than FA. A TM captures behaviors of FA and PDA as well as general-purpose computer system. We take TM to be equivalent to an algorithm. Nothing will be considered as algorithm if it can not be implemented as a TM.

The principle that TMs are formal versions of algorithms and no computational procedure will be considered as an algorithm unless it can be implemented as a TM is known as church's thesis.

According to the church's thesis, behavior of a general purpose computer can be simulated by some TM. It is a thesis, not a theorem because it is not a mathematical result. As we have discussed that a TM can perform all the

computations (recognization and computation) which are carried out by a general-purpose digital computer.

Allan Turing also suggested in his Turing's thesis "Any computation that can be performed by mechanical means can also be performed by some TM". Again, we have to define terms "mechanical means". If we consider this, then the model of TM comes true.

Some arguments for accepting the church's thesis are as follows:

1. No one has been able to suggest a counter example to disprove it till now.
2. Some TMs can also perform anything that can be performed by digital computer.

There are several alternate models of computation like random access machine (RAM), Post machine(PM) etc., but none is more powerful than TM.