

Theory of Computation



Theory of automata is a theoretical branch of computer science and mathematical. It is the study of abstract machines and the computation problems that can be solved using these machines. The abstract machine is called the automata. An automaton with a finite number of states is called a Finite automaton.

In this tutorial, we are going to learn how to construct deterministic finite automata, non-deterministic finite automata, Regular expression, context-free grammar, context-free language, Push down automata, Turning machines, etc.

Prerequisite

Before learning Automata, you should have a basic understanding of string, language, alphabets, symbols.

Audience

Our Automata Tutorial is designed to help beginners and professionals.

Problems

We assure that you will not find any problem in this Automata Tutorial.
But if there is any mistake, please post the problem in contact form.

Theory of Automata

Theory of automata is a theoretical branch of computer science and mathematical. It is the study of abstract machines and the computation

problems that can be solved using these machines. The abstract machine is called the automata. The main motivation behind developing the automata theory was to develop methods to describe and analyse the dynamic behaviour of discrete systems.

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

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

There are the basic terminologies that are important and frequently used in automata:

Symbols:

Symbols are an entity or individual objects, which can be any letter, alphabet or any picture.

Example:

1, a, b, #

Alphabets:

Alphabets are a finite set of symbols. It is denoted by Σ .

Examples:

1. $\Sigma = \{a, b\}$
- 2.
3. $\Sigma = \{A, B, C, D\}$
- 4.
5. $\Sigma = \{0, 1, 2\}$
- 6.
7. $\Sigma = \{0, 1, \dots, 5\}$
- 8.
9. $\Sigma = \{\#, \beta, \Delta\}$

String:

It is a finite collection of symbols from the alphabet. The string is denoted by w .

Example 1:

If $\Sigma = \{a, b\}$, various string that can be generated from Σ are {ab, aa, aaa, bb, bbb, ba, aba.....}.

- A string with zero occurrences of symbols is known as an empty string. It is represented by ε .
- The number of symbols in a string w is called the length of a string. It is denoted by $|w|$.

Example 2:

1. $w = 010$
- 2.
3. Number of String $|w| = 3$

Language:

A language is a collection of appropriate string. A language which is formed over Σ can be **Finite** or **Infinite**.

Example: 1

$L_1 = \{\text{Set of string of length 2}\}$

= {aa, bb, ba, bb} **Finite Language**

Example: 2

$L_2 = \{\text{Set of all strings starts with 'a'}\}$

= {a, aa, aaa, abb, abbb, ababb} **Infinite Language**

Finite Automata

- Finite automata are used to recognize patterns.
- It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata can either move to the next state or stay in the same state.

- Finite automata have two states, **Accept state or Reject state**. When the input string is processed successfully, and the automata reached its final state, then it will accept.

Formal Definition of FA

A finite automaton is a collection of 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

1. Q : finite set of states
2. Σ : finite set of the input symbol
3. q_0 : initial state
4. F : **final** state
5. δ : Transition function

Finite Automata Model:

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

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

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

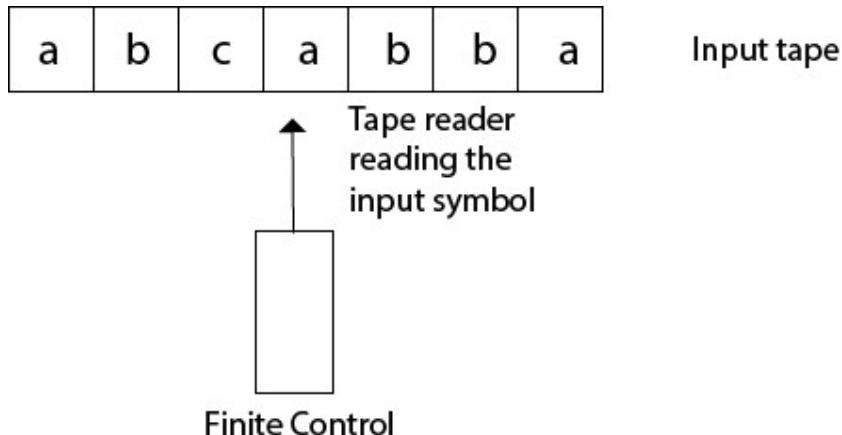


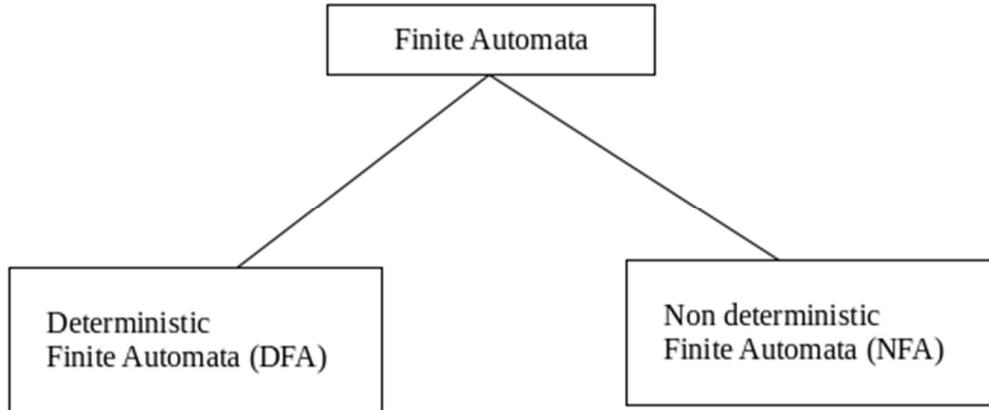
Fig :- Finite automata model

Types of Automata:

There are two types of finite automata:

1. **DFA(deterministic finite automata)**

2. NFA(non-deterministic finite automata)



1. DFA

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

2. NFA

NFA stands for non-deterministic finite automata. It is used to transmit any number of states for a particular input. It can accept the null move.

Some important points about DFA and NFA:

1. Every DFA is NFA, but NFA is not DFA.
2. There can be multiple final states in both NFA and DFA.
3. DFA is used in Lexical Analysis in Compiler.
4. NFA is more of a theoretical concept.

Transition Diagram

A transition diagram or state transition diagram is a directed graph which can be constructed as follows:

- o There is a node for each state in Q , which is represented by the circle.
- o There is a directed edge from node q to node p labeled a if $\delta(q, a) = p$.
- o In the start state, there is an arrow with no source.
- o Accepting states or final states are indicating by a double circle.

Some Notations that are used in the transition diagram:

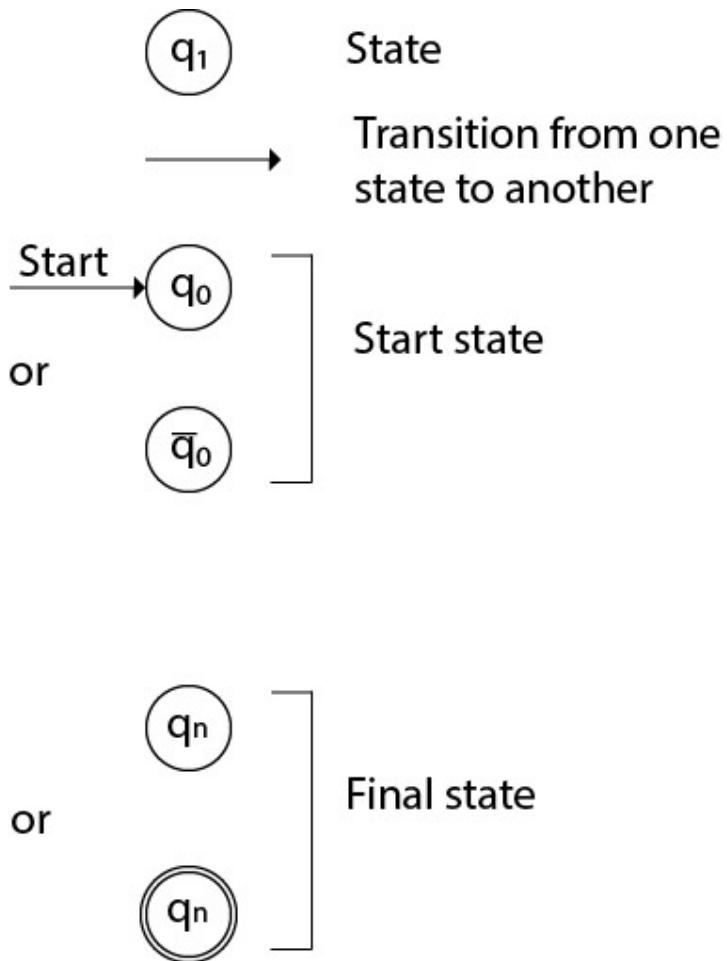


Fig:- Notations

There is a description of how a DFA operates:

1. In DFA, the input to the automata can be any string. Now, put a pointer to the start state q and read the input string w from left to right and move the pointer according to the transition function, δ . We can read one symbol at a time. If the next symbol of string w is a and the pointer is on state p , move the pointer to $\delta(p, a)$. When the end of the input string w is encountered, then the pointer is on some state F .
2. The string w is said to be accepted by the DFA if $r \in F$ that means the input string w is processed successfully and the automata reached its final state. The string is said to be rejected by DFA if $r \notin F$.

Example 1:

DFA with $\Sigma = \{0, 1\}$ accepts all strings starting with 1.

Solution:

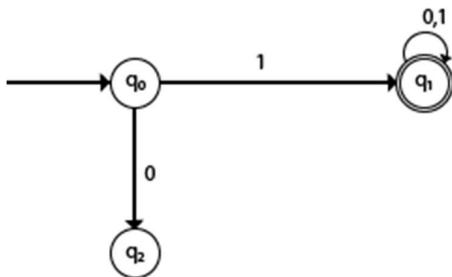


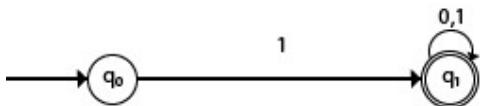
Fig: Transition diagram

The finite automata can be represented using a transition graph. In the above diagram, the machine initially is in start state q_0 then on receiving input 1 the machine changes its state to q_1 . From q_0 on receiving 0, the machine changes its state to q_2 , which is the dead state. From q_1 on receiving input 0, 1 the machine changes its state to q_1 , which is the final state. The possible input strings that can be generated are 10, 11, 110, 101, 111....., that means all string starts with 1.

Example 2:

NFA with $\Sigma = \{0, 1\}$ accepts all strings starting with 1.

Solution:



The NFA can be represented using a transition graph. In the above diagram, the machine initially is in start state q_0 then on receiving input 1 the machine changes its state to q_1 . From q_1 on receiving input 0, 1 the machine changes its state to q_1 . The possible input string that can be generated is 10, 11, 110, 101, 111....., that means all string starts with 1.

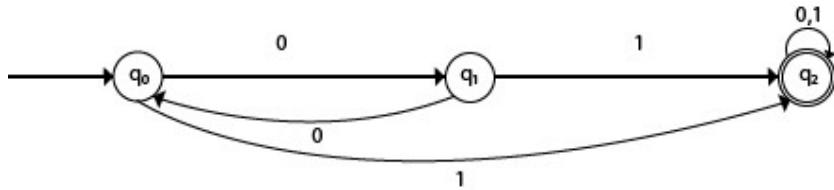
Transition Table

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

A transition table is represented by the following things:

- Columns correspond to input symbols.
- Rows correspond to states.
- Entries correspond to the next state.
- The start state is denoted by an arrow with no source.
- The accept state is denoted by a star.

Example 1:



Solution:

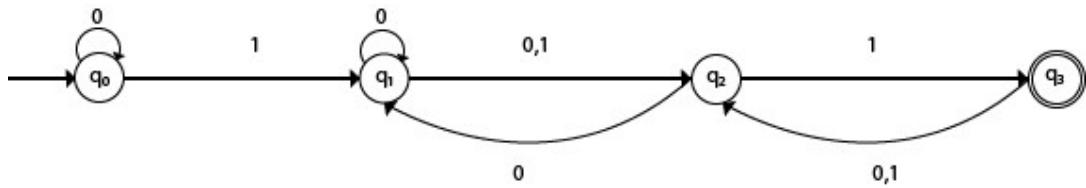
Transition table of given DFA is as follows:

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

Explanation:

- In the above table, the first column indicates all the current states. Under column 0 and 1, the next states are shown.
- The first row of the transition table can be read as, when the current state is q0, on input 0 the next state will be q1 and on input 1 the next state will be q2.
- In the second row, when the current state is q1, on input 0, the next state will be q0, and on 1 input the next state will be q2.
- In the third row, when the current state is q2 on input 0, the next state will be q2, and on 1 input the next state will be q2.
- The arrow marked to q0 indicates that it is a start state and circle marked to q2 indicates that it is a final state.

Example 2:



Solution:

Transition table of given NFA is as follows:

Present State	Next state for Input 0	Next State of Input 1
→q0	q0	q1
q1	q1, q2	q2
q2	q1	q3
*q3	q2	q2

Explanation:

- The first row of the transition table can be read as, when the current state is q0, on input 0 the next state will be q0 and on input 1 the next state will be q1.
- In the second row, when the current state is q1, on input 0 the next state will be either q1 or q2, and on 1 input the next state will be q2.
- In the third row, when the current state is q2 on input 0, the next state will be q1, and on 1 input the next state will be q3.
- In the fourth row, when the current state is q3 on input 0, the next state will be q2, and on 1 input the next state will be q2.

DFA (Deterministic finite automata)

- DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.

- In DFA, there is only one path for specific input from the current state to the next state.
- DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
- DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

In the following diagram, we can see that from state q_0 for input a , there is only one path which is going to q_1 . Similarly, from q_0 , there is only one path for input b going to q_2 .

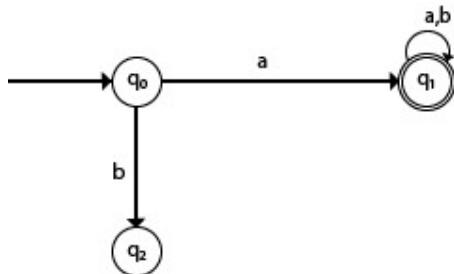


Fig:- DFA

Formal Definition of DFA

A DFA is a collection of 5-tuples same as we described in the definition of FA.

1. Q : finite set of states
2. Σ : finite set of the input symbol
3. q_0 : initial state
4. F : final state
5. δ : Transition function

Transition function can be defined as:

1. $\delta: Q \times \Sigma \rightarrow Q$

Graphical Representation of DFA

A DFA can be represented by digraphs called state diagram. In which:

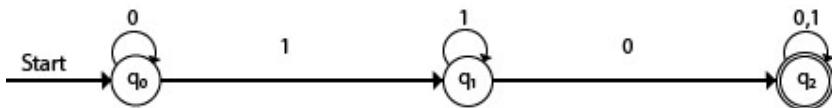
1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by a double circle.

Example 1:

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $q_0 = \{q_0\}$
4. $F = \{q_2\}$

Solution:

Transition Diagram:



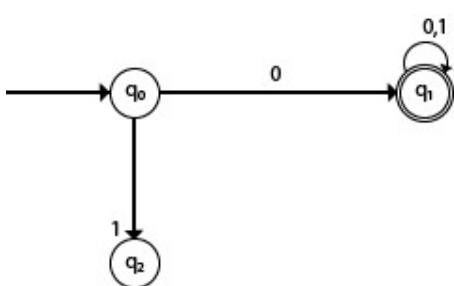
Transition Table:

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_0	q_1
q_1	q_2	q_1
$*q_2$	q_2	q_2

Example 2:

DFA with $\Sigma = \{0, 1\}$ accepts all starting with 0.

Solution:



Explanation:

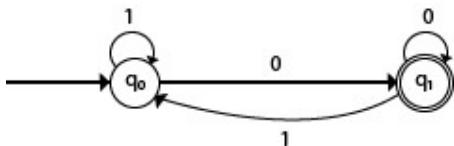
- o In the above diagram, we can see that on given 0 as input to DFA in state q_0 the DFA changes state to q_1 and always go to final state q_1 on starting input 0. It

can accept 00, 01, 000, 001....etc. It can't accept any string which starts with 1, because it will never go to final state on a string starting with 1.

Example 3:

DFA with $\Sigma = \{0, 1\}$ accepts all ending with 0.

Solution:



Explanation:

In the above diagram, we can see that on given 0 as input to DFA in state q_0 , the DFA changes state to q_1 . It can accept any string which ends with 0 like 00, 10, 110, 100....etc. It can't accept any string which ends with 1, because it will never go to the final state q_1 on 1 input, so the string ending with 1, will not be accepted or will be rejected.

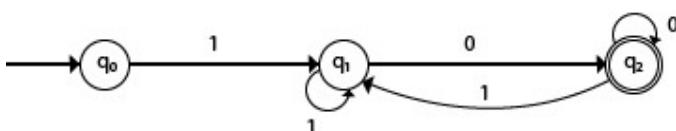
Examples of DFA

Example 1:

Design a FA with $\Sigma = \{0, 1\}$ accepts those string which starts with 1 and ends with 0.

Solution:

The FA will have a start state q_0 from which only the edge with input 1 will go to the next state.



In state q_1 , if we read 1, we will be in state q_1 , but if we read 0 at state q_1 , we will reach to state q_2 which is the final state. In state q_2 , if we read either 0 or 1, we will go to q_2 state or q_1 state respectively. Note that if the input ends with 0, it will be in the final state.

Example 2:

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

Solution:

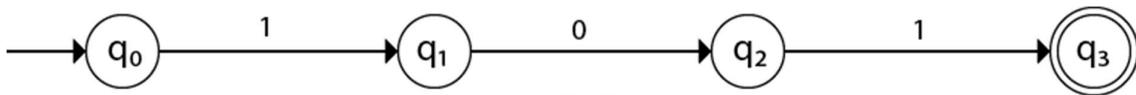


Fig: FA

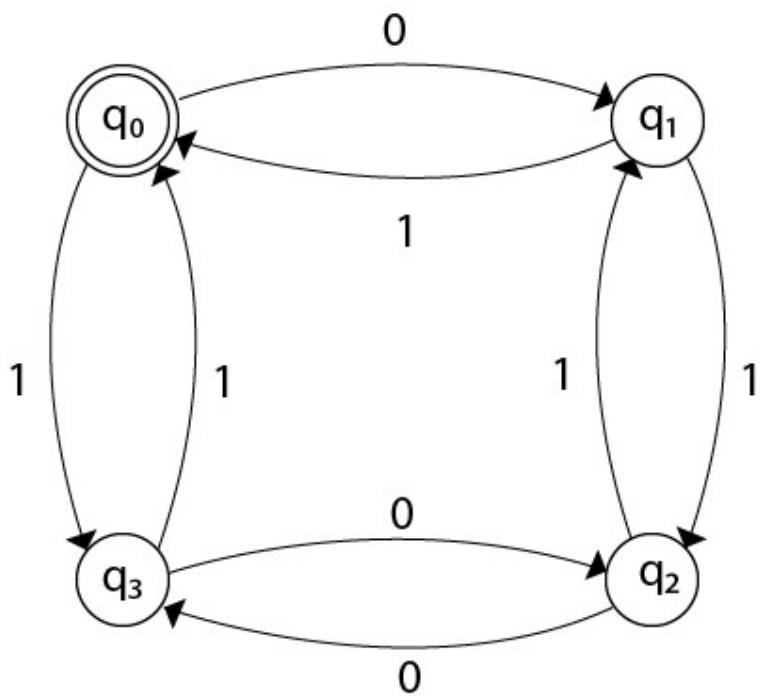
In the given solution, we can see that only input 101 will be accepted. Hence, for input 101, there is no other path shown for other input.

Example 3:

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

Solution:

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



Here q_0 is a start state and the final state also. Note carefully that a symmetry of 0's and 1's is maintained. We can associate meanings to each state as:

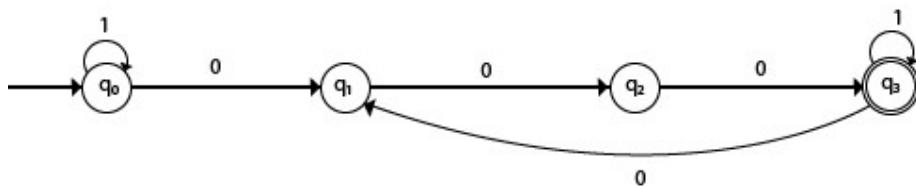
- q0: state of even number of 0's and even number of 1's.
- q1: state of odd number of 0's and even number of 1's.
- q2: state of odd number of 0's and odd number of 1's.
- q3: state of even number of 0's and odd number of 1's.

Example 4:

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

Solution:

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



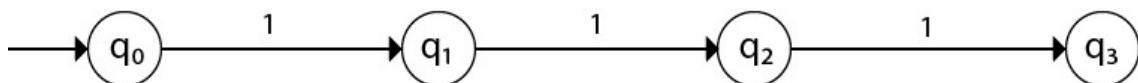
Note that the sequence of triple zeros is maintained to reach the final state.

Example 5:

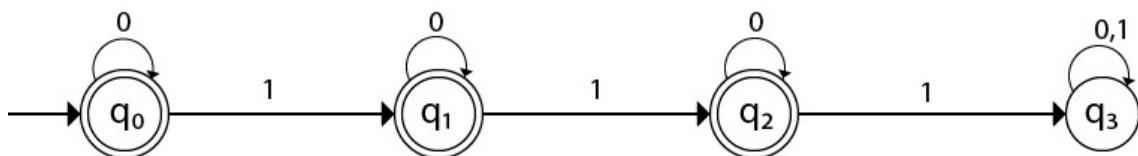
Design a DFA $L(M) = \{w \mid w \in \{0, 1\}^*\}$ and W is a string that does not contain consecutive 1's.

Solution:

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



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



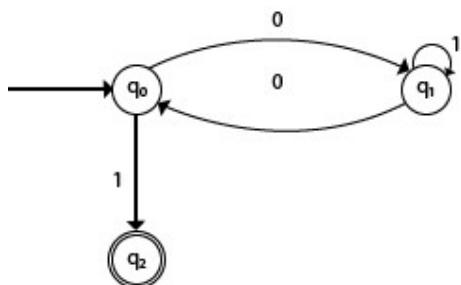
The stages q0, q1, q2 are the final states. The DFA will generate the strings that do not contain consecutive 1's like 10, 110, 101,..... etc.

Example 6:

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

Solution:

The DFA can be shown by a transition diagram as:



NFA (Non-Deterministic finite automata)

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

In the following image, we can see that from state q_0 for input a , there are two next states q_1 and q_2 , similarly, from q_0 for input b , the next states are q_0 and q_1 . Thus it is not fixed or determined that with a particular input where to go next. Hence this FA is called non-deterministic finite automata.

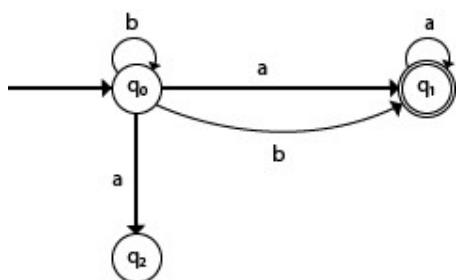


Fig:- NDFA

Formal definition of NFA:

NFA also has five states same as DFA, but with different transition function, as shown follows:

$$\delta: Q \times \Sigma \rightarrow 2^Q$$

where,

1. **Q: finite set of states**
2. **Σ : finite set of the input symbol**
3. **q_0 : initial state**
4. **F: final state**
5. **δ : Transition function**

Graphical Representation of an NFA

An NFA can be represented by digraphs called state diagram. In which:

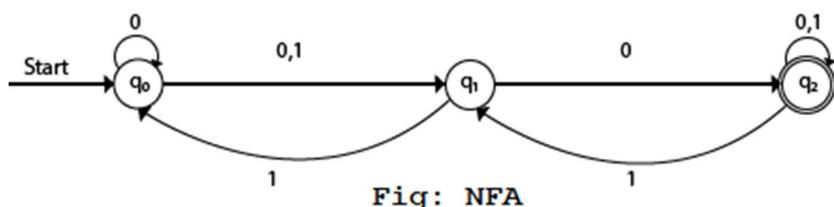
1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by the double circle.

Example 1:

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $q_0 = \{q_0\}$
4. $F = \{q_2\}$

Solution:

Transition diagram:



Transition Table:

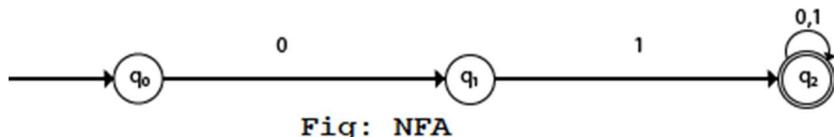
Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_0, q_1	q_1
q_1	q_2	q_0
$*q_2$	q_2	q_1, q_2

In the above diagram, we can see that when the current state is q_0 , on input 0, the next state will be q_0 or q_1 , and on 1 input the next state will be q_1 . When the current state is q_1 , on input 0 the next state will be q_2 and on 1 input, the next state will be q_0 . When the current state is q_2 , on 0 input the next state is q_2 , and on 1 input the next state will be q_1 or q_2 .

Example 2:

NFA with $\Sigma = \{0, 1\}$ accepts all strings with 01.

Solution:



Transition Table:

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_1	ϵ
q_1	ϵ	q_2
$*q_2$	q_2	q_2

Example 3:

NFA with $\Sigma = \{0, 1\}$ and accept all string of length atleast 2.

Solution:

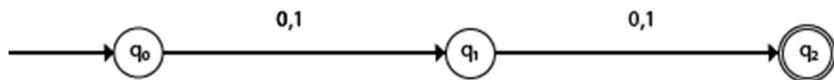


Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_1	q_1
q_1	q_2	q_2
$*q_2$	ϵ	ϵ

Examples of NFA

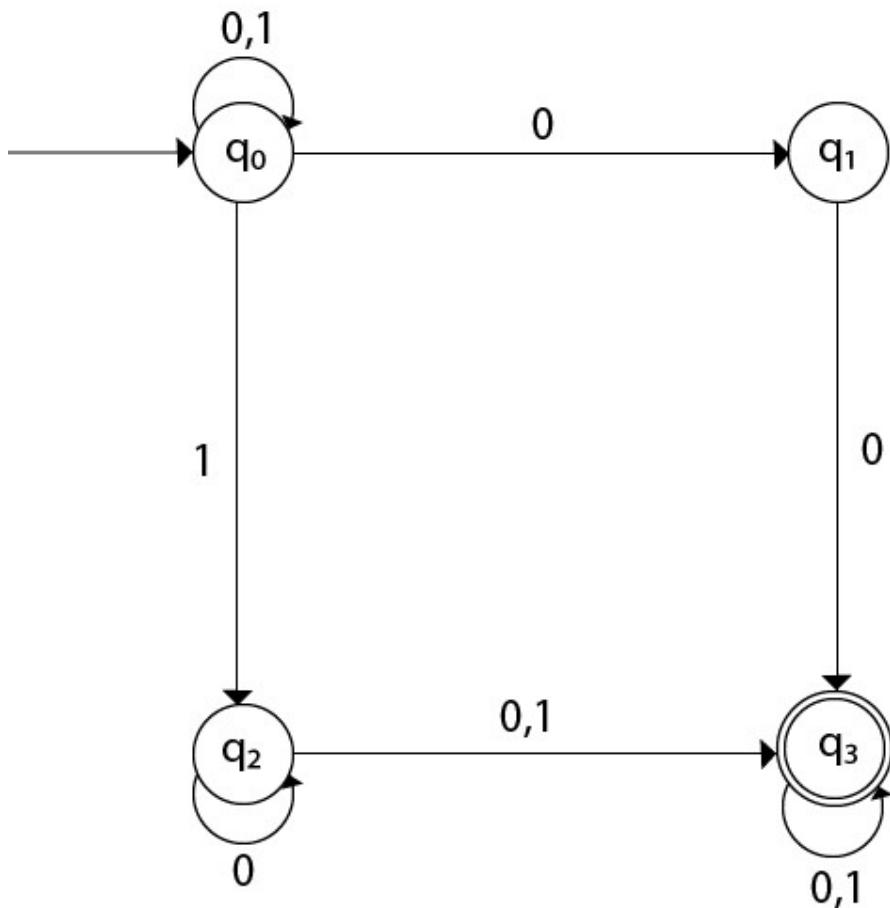
Example 1:

Design a NFA for the transition table as given below:

Present State	0	1
$\rightarrow q_0$	q_0, q_1	q_0, q_2
q_1	q_3	ϵ
q_2	q_2, q_3	q_3
$\rightarrow q_3$	q_3	q_3

Solution:

The transition diagram can be drawn by using the mapping function as given in the table.



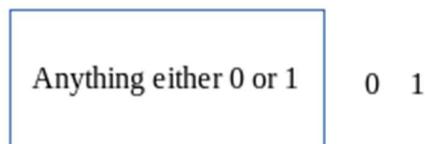
Here,

1. $\delta(q_0, 0) = \{q_0, q_1\}$
2. $\delta(q_0, 1) = \{q_0, q_2\}$
3. Then, $\delta(q_1, 0) = \{q_3\}$
4. Then, $\delta(q_2, 0) = \{q_2, q_3\}$
5. $\delta(q_2, 1) = \{q_3\}$
6. Then, $\delta(q_3, 0) = \{q_3\}$
7. $\delta(q_3, 1) = \{q_3\}$

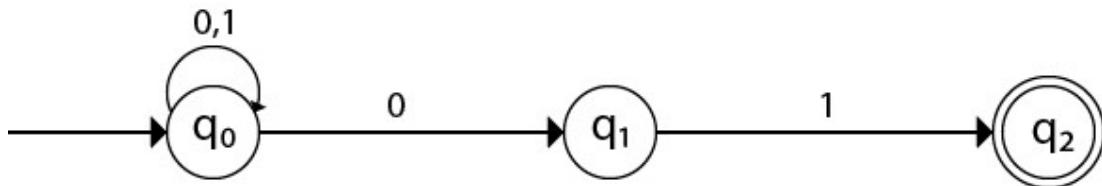
Example 2:

Design an NFA with $\Sigma = \{0, 1\}$ accepts all string ending with 01.

Solution:



Hence, NFA would be:

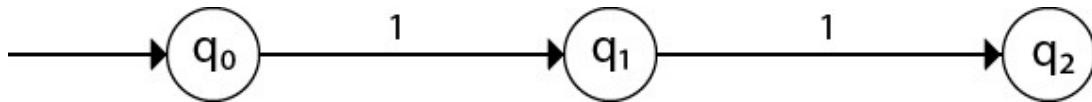


Example 3:

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

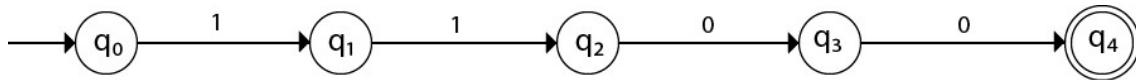
Solution:

The FA with double 1 is as follows:



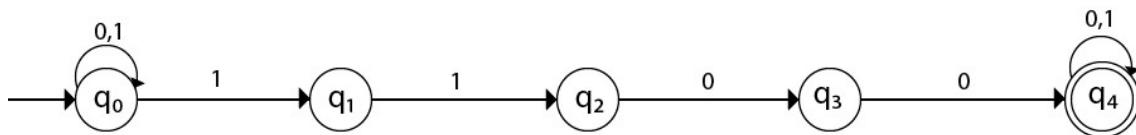
It should be immediately followed by double 0.

Then,



Now before double 1, there can be any string of 0 and 1. Similarly, after double 0, there can be any string of 0 and 1.

Hence the NFA becomes:



Now considering the string 01100011

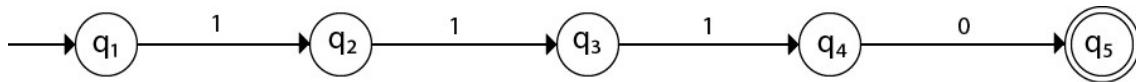
1. $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4 \rightarrow q_4 \rightarrow q_4 \rightarrow q_4$

Example 4:

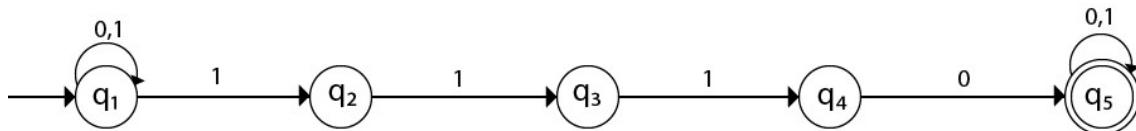
Design an NFA in which all the string contain a substring 1110.

Solution:

The language consists of all the string containing substring 1110. The partial transition diagram can be:



Now as 1010 could be the substring. Hence we will add the inputs 0's and 1's so that the substring 1010 of the language can be maintained. Hence the NFA becomes:



Transition table for the above transition diagram can be given below:

Present State	0	1
$\rightarrow q_1$	q_1	q_1, q_2
q_2		q_3
q_3		q_4
q_4	q_5	
$*q_5$	q_5	q_5

Consider a string 111010,

1. $\delta(q_1, \text{111010}) = \delta(q_1, \text{1100})$
2. $= \delta(q_1, \text{100})$
3. $= \delta(q_2, \text{00})$

Got stuck! As there is no path from q_2 for input symbol 0. We can process string 111010 in another way.

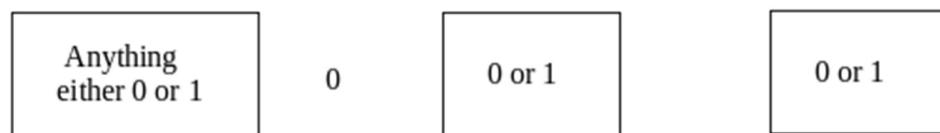
1. $\delta(q_1, \text{111010}) = \delta(q_2, \text{1100})$
2. $= \delta(q_3, \text{100})$
3. $= \delta(q_4, \text{00})$
4. $= \delta(q_5, \text{0})$
5. $= \delta(q_5, \varepsilon)$

As state q_5 is the accept state. We get the complete scanned, and we reached to the final state.

Example 5:

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

Solution:



Thus we get the third symbol from the right end as '0' always. The NFA can be:



The above image is an NFA because in state q_0 with input 0, we can either go to state q_0 or q_1 .

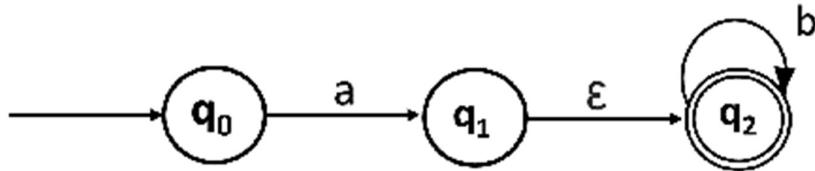
Eliminating ϵ Transitions

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

1. Find out all the ϵ transitions from each state from Q . That will be called as ϵ -closure{ q_1 } where $q_i \in Q$.
2. Then δ' transitions can be obtained. The δ' transitions mean a ϵ -closure on δ moves.
3. Repeat Step-2 for each input symbol and each state of given NFA.
4. Using the resultant states, the transition table for equivalent NFA without ϵ can be built.

Example:

Convert the following NFA with ϵ to NFA without ϵ .



Solutions: We will first obtain ϵ -closures of q_0 , q_1 and q_2 as follows:

1. $\epsilon\text{-closure}(q_0) = \{q_0\}$
2. $\epsilon\text{-closure}(q_1) = \{q_1, q_2\}$
3. $\epsilon\text{-closure}(q_2) = \{q_2\}$

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

1. $\delta'(q_0, a) = \epsilon\text{-closure}(\delta(\delta^\wedge(q_0, \epsilon), a))$
2. $= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0), a))$
3. $= \epsilon\text{-closure}(\delta(q_0, a))$
4. $= \epsilon\text{-closure}(q_1)$
5. $= \{q_1, q_2\}$
- 6.
7. $\delta'(q_0, b) = \epsilon\text{-closure}(\delta(\delta^\wedge(q_0, \epsilon), b))$
8. $= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0), b))$
9. $= \epsilon\text{-closure}(\delta(q_0, b))$
10. $= \Phi$

Now the δ' transition on q_1 is obtained as:

1. $\delta'(q_1, a) = \epsilon\text{-closure}(\delta(\delta^\wedge(q_1, \epsilon), a))$
2. $= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1), a))$
3. $= \epsilon\text{-closure}(\delta(q_1, q_2), a)$
4. $= \epsilon\text{-closure}(\delta(q_1, a) \cup \delta(q_2, a))$
5. $= \epsilon\text{-closure}(\Phi \cup \Phi)$
6. $= \Phi$
- 7.
8. $\delta'(q_1, b) = \epsilon\text{-closure}(\delta(\delta^\wedge(q_1, \epsilon), b))$
9. $= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1), b))$
10. $= \epsilon\text{-closure}(\delta(q_1, q_2), b)$
11. $= \epsilon\text{-closure}(\delta(q_1, b) \cup \delta(q_2, b))$
12. $= \epsilon\text{-closure}(\Phi \cup q_2)$

$$13. \quad = \{q2\}$$

The δ' transition on q_2 is obtained as:

1. $\delta'(q_2, a) = \epsilon\text{-closure}(\delta(\delta^*(q_2, \epsilon), a))$
2. $= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_2), a))$
3. $= \epsilon\text{-closure}(\delta(q_2, a))$
4. $= \epsilon\text{-closure}(\Phi)$
5. $= \Phi$
- 6.
7. $\delta'(q_2, b) = \epsilon\text{-closure}(\delta(\delta^*(q_2, \epsilon), b))$
8. $= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_2), b))$
9. $= \epsilon\text{-closure}(\delta(q_2, b))$
10. $= \epsilon\text{-closure}(q_2)$
11. $= \{q_2\}$

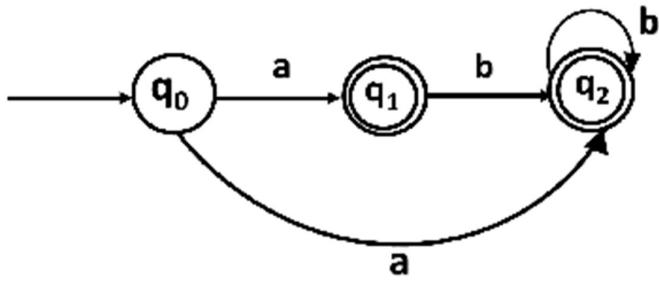
Now we will summarize all the computed δ' transitions:

1. $\delta'(q_0, a) = \{q_0, q_1\}$
2. $\delta'(q_0, b) = \Phi$
3. $\delta'(q_1, a) = \Phi$
4. $\delta'(q_1, b) = \{q_2\}$
5. $\delta'(q_2, a) = \Phi$
6. $\delta'(q_2, b) = \{q_2\}$

The transition table can be:

States	a	b
$\rightarrow q_0$	$\{q_1, q_2\}$	Φ
$*q_1$	Φ	$\{q_2\}$
$*q_2$	Φ	$\{q_2\}$

State q_1 and q_2 become the final state as ϵ -closure of q_1 and q_2 contain the final state q_2 . The NFA can be shown by the following transition diagram:



Conversion from NFA to DFA

In this section, we will discuss the method of converting NFA to its equivalent DFA. In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol.

Let, $M = (Q, \Sigma, \delta, q_0, F)$ is an NFA which accepts the language $L(M)$. There should be equivalent DFA denoted by $M' = (Q', \Sigma', q_0', \delta', F')$ such that $L(M) = L(M')$.

Steps for converting NFA to DFA:

Step 1: Initially $Q' = \emptyset$

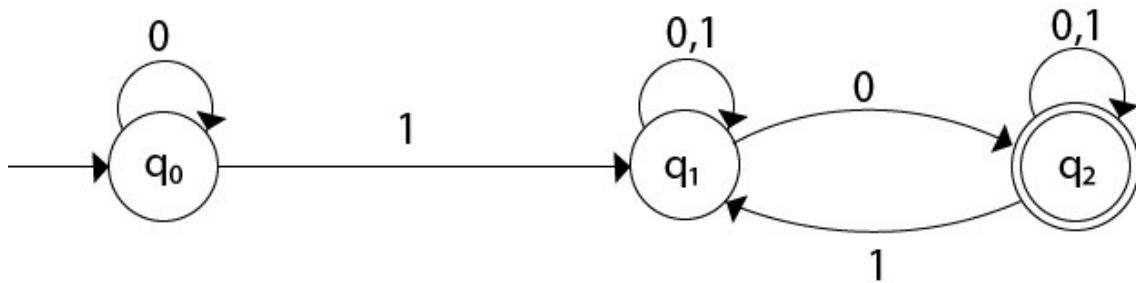
Step 2: Add q_0 of NFA to Q' . Then find the transitions from this start state.

Step 3: In Q' , find the possible set of states for each input symbol. If this set of states is not in Q' , then add it to Q' .

Step 4: In DFA, the final state will be all the states which contain F (final states of NFA)

Example 1:

Convert the given NFA to DFA.



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

State	0	1
$\rightarrow q_0$	q_0	q_1
q_1	$\{q_1, q_2\}$	q_1
$*q_2$	q_2	$\{q_1, q_2\}$

Now we will obtain δ' transition for state q_0 .

1. $\delta'([q_0], 0) = [q_0]$
2. $\delta'([q_0], 1) = [q_1]$

The δ' transition for state q_1 is obtained as:

1. $\delta'([q_1], 0) = [q_1, q_2]$ (new state generated)
2. $\delta'([q_1], 1) = [q_1]$

The δ' transition for state q_2 is obtained as:

1. $\delta'([q_2], 0) = [q_2]$
2. $\delta'([q_2], 1) = [q_1, q_2]$

Now we will obtain δ' transition on $[q_1, q_2]$.

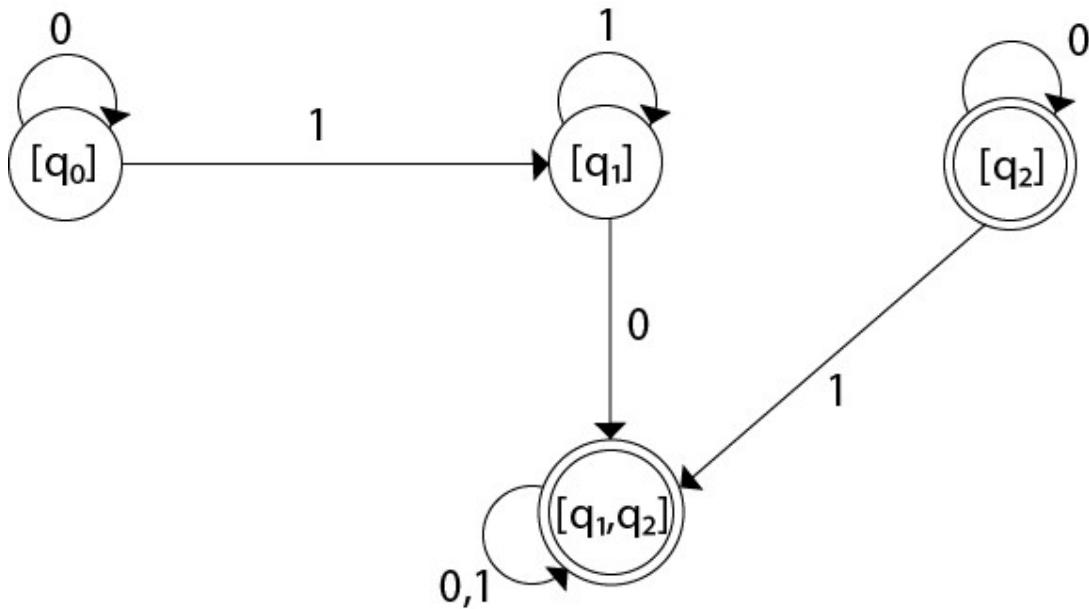
1. $\delta'([q_1, q_2], 0) = \delta(q_1, 0) \cup \delta(q_2, 0)$
2. $= \{q_1, q_2\} \cup \{q_2\}$
3. $= [q_1, q_2]$
4. $\delta'([q_1, q_2], 1) = \delta(q_1, 1) \cup \delta(q_2, 1)$
5. $= \{q_1\} \cup \{q_1, q_2\}$
6. $= \{q_1, q_2\}$

$$7. \quad = [q_1, q_2]$$

The state $[q_1, q_2]$ is the final state as well because it contains a final state q_2 . The transition table for the constructed DFA will be:

State	0	1
$\rightarrow[q_0]$	$[q_0]$	$[q_1]$
$[q_1]$	$[q_1, q_2]$	$[q_1]$
$*[q_2]$	$[q_2]$	$[q_1, q_2]$
$*[q_1, q_2]$	$[q_1, q_2]$	$[q_1, q_2]$

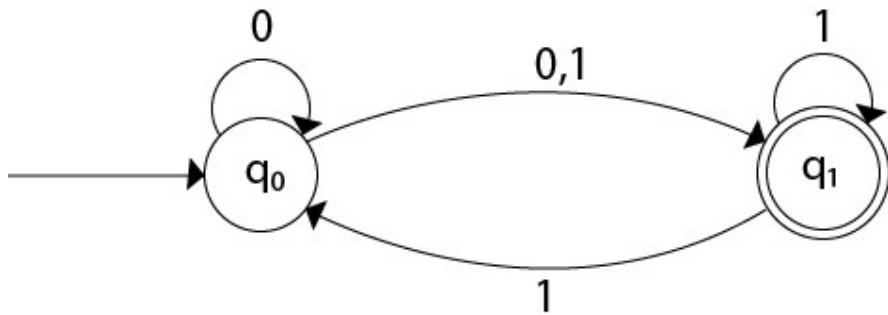
The Transition diagram will be:



The state q_2 can be eliminated because q_2 is an unreachable state.

Example 2:

Convert the given NFA to DFA.



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

State	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_1\}$
$*q_1$	\emptyset	$\{q_0, q_1\}$

Now we will obtain δ' transition for state q_0 .

1. $\delta'([q_0], 0) = \{q_0, q_1\}$
2. $= [q_0, q_1]$ (new state generated)
3. $\delta'([q_0], 1) = \{q_1\} = [q_1]$

The δ' transition for state q_1 is obtained as:

1. $\delta'([q_1], 0) = \emptyset$
2. $\delta'([q_1], 1) = [q_0, q_1]$

Now we will obtain δ' transition on $[q_0, q_1]$.

1. $\delta'([q_0, q_1], 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$
2. $= \{q_0, q_1\} \cup \emptyset$
3. $= \{q_0, q_1\}$
4. $= [q_0, q_1]$

Similarly,

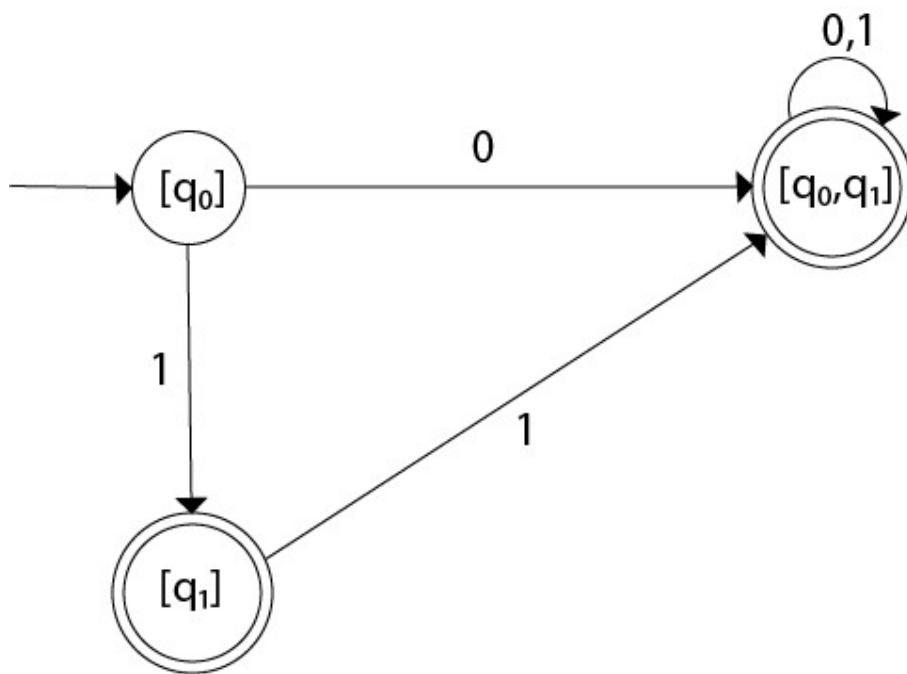
1. $\delta'([q_0, q_1], 1) = \delta(q_0, 1) \cup \delta(q_1, 1)$
2. $= \{q_1\} \cup \{q_0, q_1\}$
3. $= \{q_0, q_1\}$
4. $= [q_0, q_1]$

As in the given NFA, q_1 is a final state, then in DFA wherever, q_1 exists that state becomes a final state. Hence in the DFA, final states are $[q_1]$ and $[q_0, q_1]$. Therefore set of final states $F = \{[q_1], [q_0, q_1]\}$.

The transition table for the constructed DFA will be:

State	0	1
$\rightarrow[q_0]$	$[q_0, q_1]$	$[q_1]$
$*[q_1]$	\emptyset	$[q_0, q_1]$
$*[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$

The Transition diagram will be:

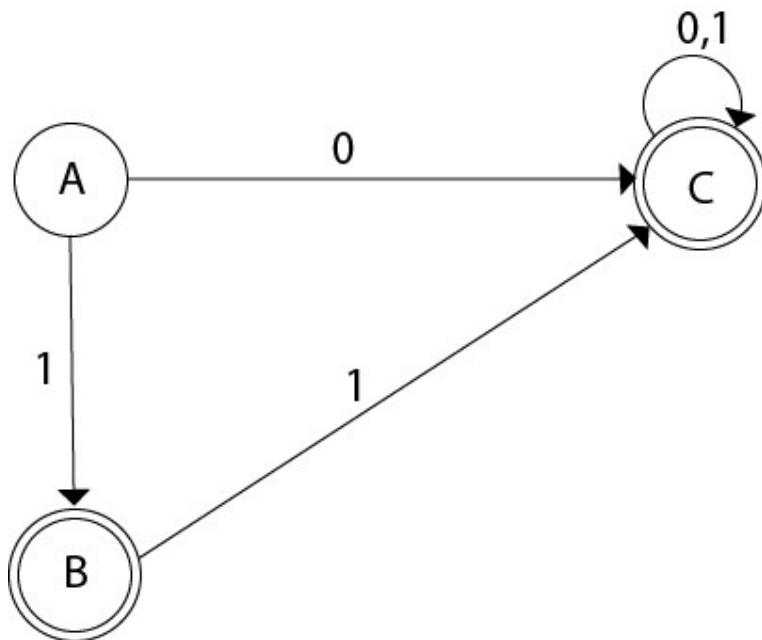


Even we can change the name of the states of DFA.

Suppose

1. $A = [q_0]$
2. $B = [q_1]$
3. $C = [q_0, q_1]$

With these new names the DFA will be as follows:



Conversion from NFA with ϵ to DFA

Non-deterministic finite automata(NFA) is a finite automata where for some cases when a specific input is given to the current state, the machine goes to multiple states or more than 1 states. It can contain ϵ move. It can be represented as $M = \{ Q, \Sigma, \delta, q_0, F \}$.

Where

1. **Q : finite set of states**
2. **Σ : finite set of the input symbol**
3. **q_0 : initial state**
4. **F : final state**
5. **δ : Transition function**

NFA with ϵ move: If any FA contains ϵ transaction or move, the finite automata is called NFA with ϵ move.

ϵ -closure: ϵ -closure for a given state A means a set of states which can be reached from the state A with only ϵ (null) move including the state A itself.

Steps for converting NFA with ϵ to DFA:

Step 1: We will take the ϵ -closure for the starting state of NFA as a starting state of DFA.

Step 2: Find the states for each input symbol that can be traversed from the present. That means the union of transition value and their closures for each state of NFA present in the current state of DFA.

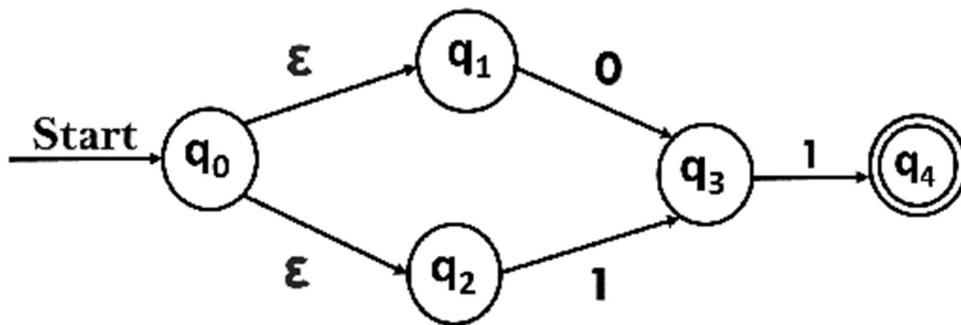
Step 3: If we found a new state, take it as current state and repeat step 2.

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

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

Example 1:

Convert the NFA with ϵ into its equivalent DFA.



Solution:

Let us obtain ϵ -closure of each state.

1. ϵ -closure $\{q_0\} = \{q_0, q_1, q_2\}$
2. ϵ -closure $\{q_1\} = \{q_1\}$
3. ϵ -closure $\{q_2\} = \{q_2\}$
4. ϵ -closure $\{q_3\} = \{q_3\}$
5. ϵ -closure $\{q_4\} = \{q_4\}$

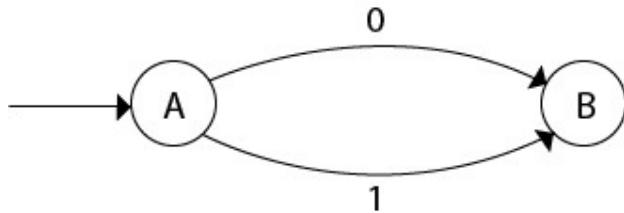
Now, let ϵ -closure $\{q_0\} = \{q_0, q_1, q_2\}$ be state A.

Hence

$$\begin{aligned}\delta'(A, 0) &= \epsilon\text{-closure } \{\delta(\{q_0, q_1, q_2\}, 0)\} \\ &= \epsilon\text{-closure } \{\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)\} \\ &= \epsilon\text{-closure } \{q_3\} \\ &= \{q_3\} \quad \text{call it as state B.}\end{aligned}$$

$$\begin{aligned}
 \delta'(A, 1) &= \varepsilon\text{-closure } \{\delta((q_0, q_1, q_2), 1)\} \\
 &= \varepsilon\text{-closure } \{\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)\} \\
 &= \varepsilon\text{-closure } \{q_3\} \\
 &= \{q_3\} = B.
 \end{aligned}$$

The partial DFA will be



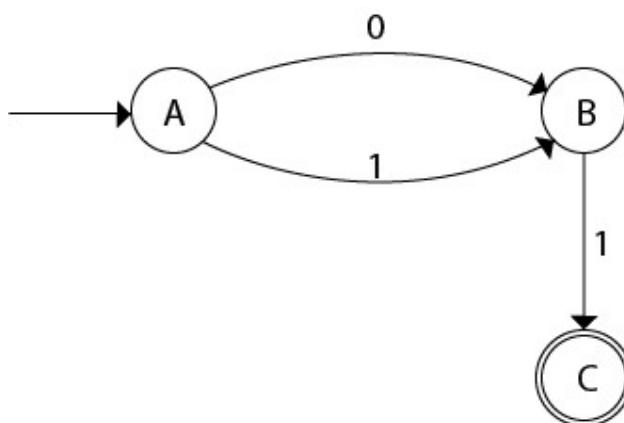
Now,

$$\begin{aligned}
 \delta'(B, 0) &= \varepsilon\text{-closure } \{\delta(q_3, 0)\} \\
 &= \emptyset \\
 \delta'(B, 1) &= \varepsilon\text{-closure } \{\delta(q_3, 1)\} \\
 &= \varepsilon\text{-closure } \{q_4\} \\
 &= \{q_4\} \quad \text{i.e. state C}
 \end{aligned}$$

For state C:

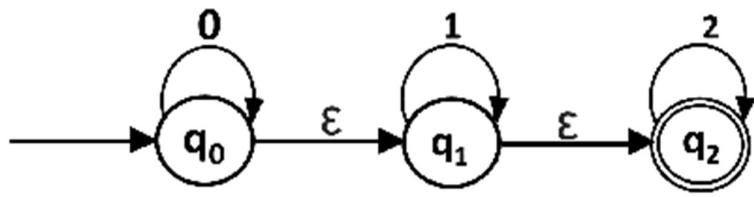
1. $\delta'(C, 0) = \varepsilon\text{-closure } \{\delta(q_4, 0)\}$
2. $= \emptyset$
3. $\delta'(C, 1) = \varepsilon\text{-closure } \{\delta(q_4, 1)\}$
4. $= \emptyset$

The DFA will be,



Example 2:

Convert the given NFA into its equivalent DFA.



Solution: Let us obtain the ϵ -closure of each state.

1. $\epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$
2. $\epsilon\text{-closure}(q_1) = \{q_1, q_2\}$
3. $\epsilon\text{-closure}(q_2) = \{q_2\}$

Now we will obtain δ' transition. Let $\epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$ call it as **state A**.

$$\begin{aligned}\delta'(A, 0) &= \epsilon\text{-closure}\{\delta((q_0, q_1, q_2), 0)\} \\ &= \epsilon\text{-closure}\{\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)\} \\ &= \epsilon\text{-closure}\{q_0\} \\ &= \{q_0, q_1, q_2\}\end{aligned}$$

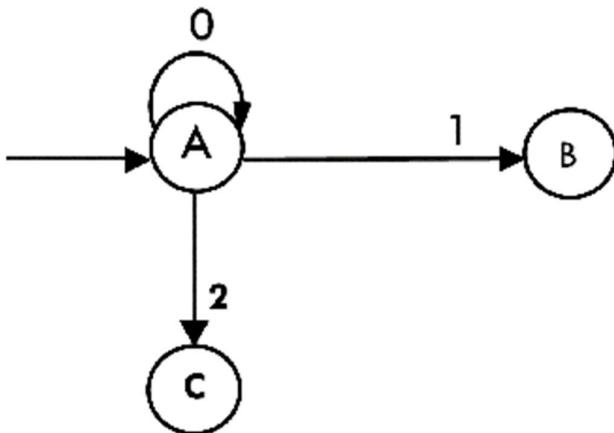
$$\begin{aligned}\delta'(A, 1) &= \epsilon\text{-closure}\{\delta((q_0, q_1, q_2), 1)\} \\ &= \epsilon\text{-closure}\{\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)\} \\ &= \epsilon\text{-closure}\{q_1\} \\ &= \{q_1, q_2\} \quad \text{call it as state B}\end{aligned}$$

$$\begin{aligned}\delta'(A, 2) &= \epsilon\text{-closure}\{\delta((q_0, q_1, q_2), 2)\} \\ &= \epsilon\text{-closure}\{\delta(q_0, 2) \cup \delta(q_1, 2) \cup \delta(q_2, 2)\} \\ &= \epsilon\text{-closure}\{q_2\} \\ &= \{q_2\} \quad \text{call it state C}\end{aligned}$$

Thus we have obtained

1. $\delta'(A, 0) = A$
2. $\delta'(A, 1) = B$
3. $\delta'(A, 2) = C$

The partial DFA will be:



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

Hence

$$\begin{aligned}
 \delta'(B, 0) &= \varepsilon\text{-closure}\{\delta(q_1, q_2), 0\} \\
 &= \varepsilon\text{-closure}\{\delta(q_1, 0) \cup \delta(q_2, 0)\} \\
 &= \varepsilon\text{-closure}\{\phi\} \\
 &= \phi
 \end{aligned}$$

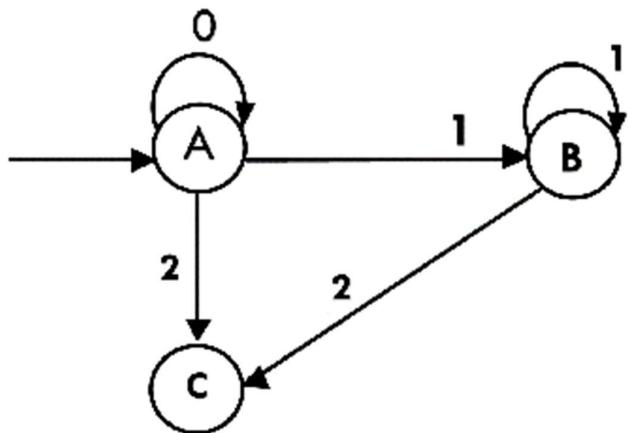
$$\begin{aligned}
 \delta'(B, 1) &= \varepsilon\text{-closure}\{\delta(q_1, q_2), 1\} \\
 &= \varepsilon\text{-closure}\{\delta(q_1, 1) \cup \delta(q_2, 1)\} \\
 &= \varepsilon\text{-closure}\{q_1\} \\
 &= \{q_1, q_2\} \quad \text{i.e. state B itself}
 \end{aligned}$$

$$\begin{aligned}
 \delta'(B, 2) &= \varepsilon\text{-closure}\{\delta(q_1, q_2), 2\} \\
 &= \varepsilon\text{-closure}\{\delta(q_1, 2) \cup \delta(q_2, 2)\} \\
 &= \varepsilon\text{-closure}\{q_2\} \\
 &= \{q_2\} \quad \text{i.e. state C itself}
 \end{aligned}$$

Thus we have obtained

1. $\delta'(B, 0) = \phi$
2. $\delta'(B, 1) = B$
3. $\delta'(B, 2) = C$

The partial transition diagram will be



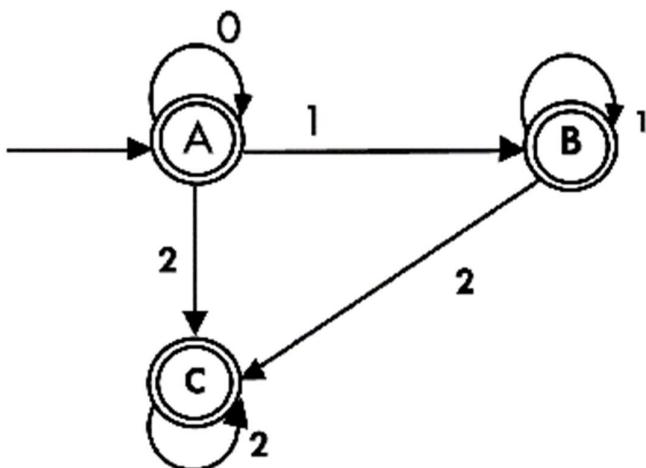
Now we will obtain transitions for C:

$$\begin{aligned}\delta'(C, 0) &= \varepsilon\text{-closure}\{\delta(q_2, 0)\} \\ &= \varepsilon\text{-closure}\{\phi\} \\ &= \phi\end{aligned}$$

$$\begin{aligned}\delta'(C, 1) &= \varepsilon\text{-closure}\{\delta(q_2, 1)\} \\ &= \varepsilon\text{-closure}\{\phi\} \\ &= \phi\end{aligned}$$

$$\begin{aligned}\delta'(C, 2) &= \varepsilon\text{-closure}\{\delta(q_2, 2)\} \\ &= \{q_2\}\end{aligned}$$

Hence the DFA is



As $A = \{q_0, q_1, q_2\}$ in which final state q_2 lies hence A is final state. $B = \{q_1, q_2\}$ in which the state q_2 lies hence B is also final state. $C = \{q_2\}$, the state q_2 lies hence C is also a final state.

Minimization of DFA

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

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

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

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

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

Step 4: Find similar rows from T1 such that:

1. $\delta(q, a) = p$
2. $\delta(r, a) = p$

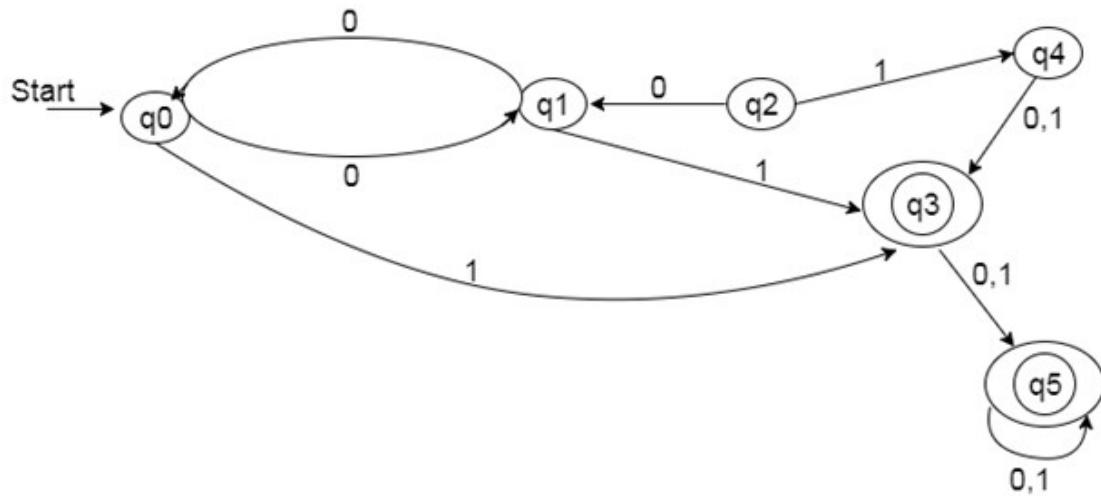
That means, find the two states which have the same value of a and b and remove one of them.

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

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

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

Example:



Solution:

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

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

State	0	1
$\rightarrow q_0$	q_1	q_3
q_1	q_0	q_3
$*q_3$	q_5	q_5
$*q_5$	q_5	q_5

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

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

State	0	1
q_0	q_1	q_3
q_1	q_0	q_3

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

State	0	1
q3	q5	q5
q5	q5	q5

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

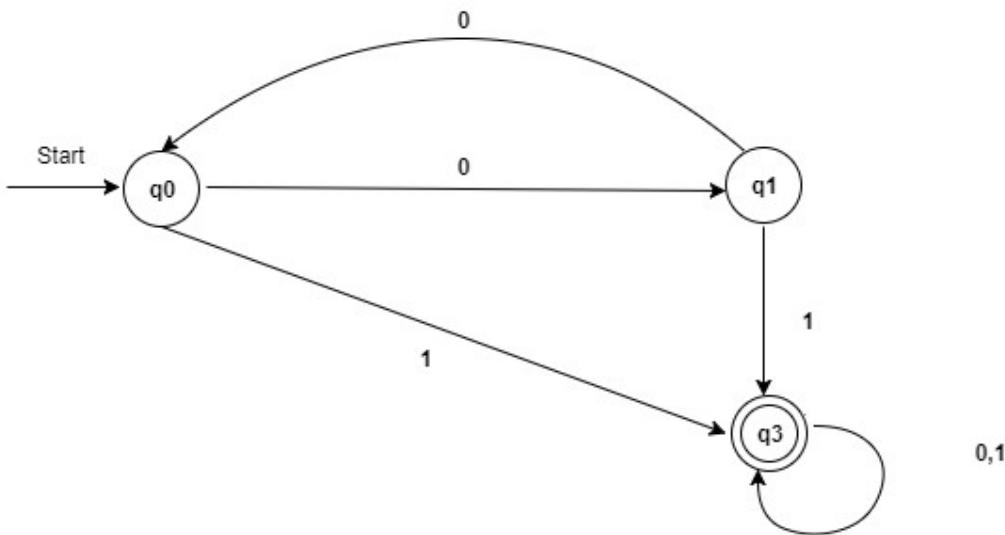
Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0	1
q3	q3	q3

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

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q3	q3

Now it is the transition table of minimized DFA.



Regular Expression

- The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
- The languages accepted by some regular expression are referred to as Regular languages.
- A regular expression can also be described as a sequence of pattern that defines a string.
- Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

For instance:

In a regular expression, x^* means zero or more occurrence of x . It can generate $\{e, x, xx, xxx, xxxx, \dots\}$

In a regular expression, x^+ means one or more occurrence of x . It can generate $\{x, xx, xxx, xxxx, \dots\}$

Operations on Regular Language

The various operations on regular language are:

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

1. $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$

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

1. $L \cap M = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$

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

1. $L^* = \text{Zero or more occurrence of language } L$.

Example 1:

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

Solution:

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

1. $R = a^*$

That is Kleen closure of a.

Example 2:

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

Solution:

The regular expression has to be built for the language

1. $L = \{a, aa, aaa, \dots\}$

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

$$R = a^+$$

Example 3:

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

Solution:

The regular expression will be:

$$1. \text{ r.e. } = (a + b)^*$$

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

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

Examples of Regular Expression

Example 1:

Write the regular expression for the language accepting all the strings which are starting with 1 and ending with 0, over $\Sigma = \{0, 1\}$.

Solution:

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

$$1. \text{ R } = 1(0+1)^* 0$$

Example 2:

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

Solution:

The regular expression will be:

$$1. \text{ R } = a b^* b$$

Example 3:

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

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

$$1. \text{ L } = \{a, aba, aab, aba, aaa, abab, \dots\}$$

The regular expression for the above language is:

$$1. \text{ R } = \{a + ab\}^*$$

Example 4:

Write the regular expression for the language accepting all the string in which any number of a's is followed by any number of b's is followed by any number of c's.

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

1. $R = a^* b^* c^*$

Example 5:

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

Solution:

The regular expression has to be built for the language:

1. $L = \{\epsilon, \textcolor{red}{00}, \textcolor{red}{0000}, \textcolor{red}{000000}, \dots\}$

The regular expression for the above language is:

1. $R = (\textcolor{red}{00})^*$

Example 6:

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

Solution:

The regular expression will be:

1. $R = [(\textcolor{red}{0} + \textcolor{red}{1})^* \textcolor{red}{0} (\textcolor{red}{0} + \textcolor{red}{1})^* \textcolor{red}{1} (\textcolor{red}{0} + \textcolor{red}{1})^*] + [(\textcolor{red}{0} + \textcolor{red}{1})^* \textcolor{red}{1} (\textcolor{red}{0} + \textcolor{red}{1})^* \textcolor{red}{0} (\textcolor{red}{0} + \textcolor{red}{1})^*]$

Example 7:

Describe the language denoted by following regular expression

1. r.e. = $(b^* (aaa)^* b^*)^*$

Solution:

The language can be predicted from the regular expression by finding the meaning of it. We will first split the regular expression as:

r.e. = (any combination of b's) (aaa)* (any combination of b's)

L = {The language consists of the string in which a's appear triples, there is no restriction on the number of b's}

Example 8:

Write the regular expression for the language L over $\Sigma = \{0, 1\}$ such that all the strings do not contain the substring 01.

Solution:

The Language is as follows:

1. $L = \{\epsilon, 0, 1, 00, 11, 10, 100, \dots\}$

The regular expression for the above language is as follows:

1. $R = (1^* 0^*)$

Example 9:

Write the regular expression for the language containing the string over $\{0, 1\}$ in which there are at least two occurrences of 1's between any two occurrences of 0's.

Solution: At least two 1's between two occurrences of 0's can be denoted by $(0111^* 0)^*$.

Similarly, if there is no occurrence of 0's, then any number of 1's are also allowed. Hence the r.e. for required language is:

1. $R = (1 + (0111^* 0))^*$

Example 10:

Write the regular expression for the language containing the string in which every 0 is immediately followed by 11.

Solution:

The regular expectation will be:

1. $R = (011 + 1)^*$

Conversion of RE to FA

To convert the RE to FA, we are going to use a method called the subset method. This method is used to obtain FA from the given regular expression. This method is given below:

Step 1: Design a transition diagram for given regular expression, using NFA with ϵ moves.

Step 2: Convert this NFA with ϵ to NFA without ϵ .

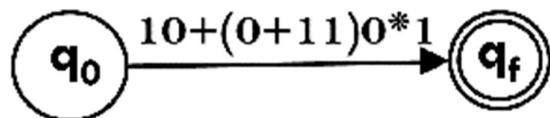
Step 3: Convert the obtained NFA to equivalent DFA.

Example 1:

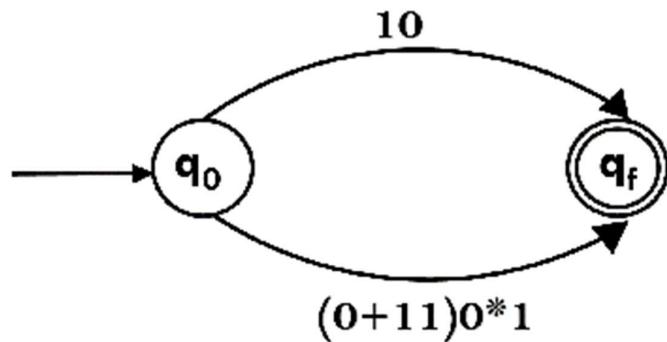
Design a FA from given regular expression $10 + (0 + 11)0^*1$.

Solution: First we will construct the transition diagram for a given regular expression.

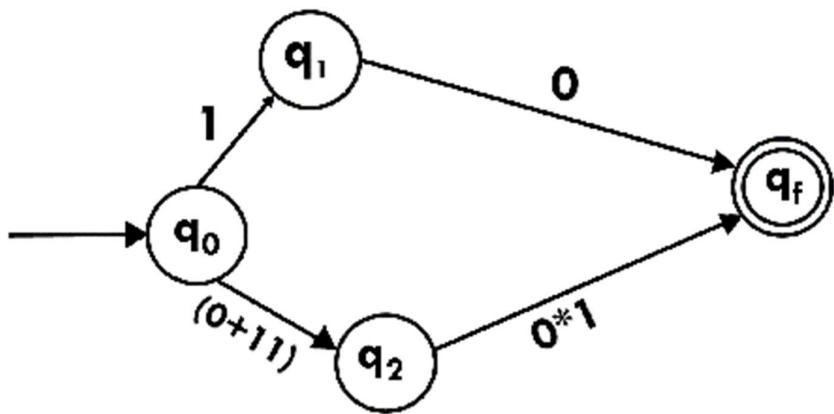
Step 1:



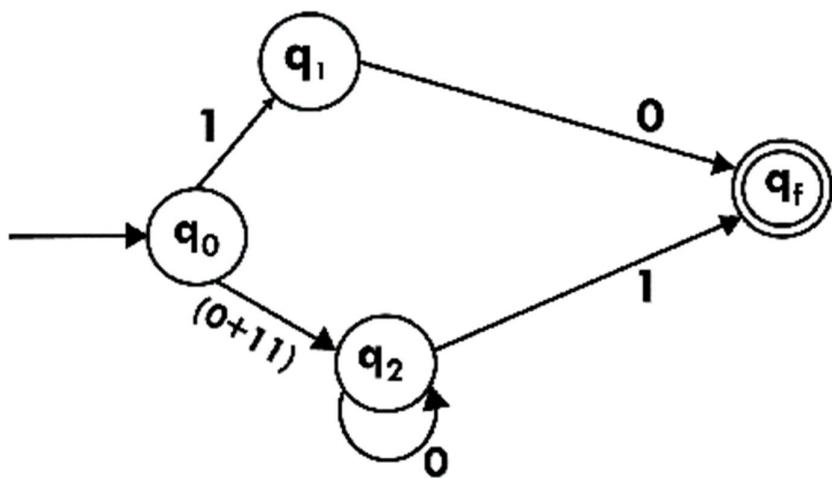
Step 2:



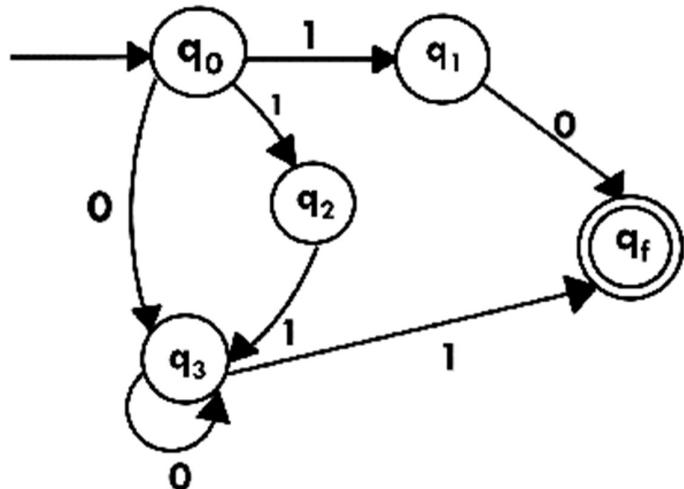
Step 3:



Step 4:



Step 5:



Now we have got NFA without ϵ . Now we will convert it into required DFA for that, we will first write a transition table for this NFA.

State	0	1
$\rightarrow q_0$	q_3	$\{q_1, q_2\}$
q_1	q_f	\emptyset
q_2	\emptyset	q_3
q_3	q_3	q_f
$*q_f$	\emptyset	\emptyset

The equivalent DFA will be:

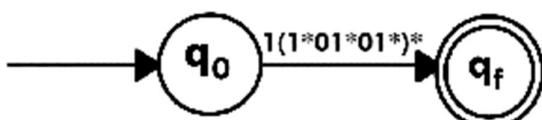
State	0	1
$\rightarrow [q_0]$	$[q_3]$	$[q_1, q_2]$
$[q_1]$	$[q_f]$	\emptyset
$[q_2]$	\emptyset	$[q_3]$
$[q_3]$	$[q_3]$	$[q_f]$
$[q_1, q_2]$	$[q_f]$	$[q_f]$
$*[q_f]$	\emptyset	\emptyset

Example 2:

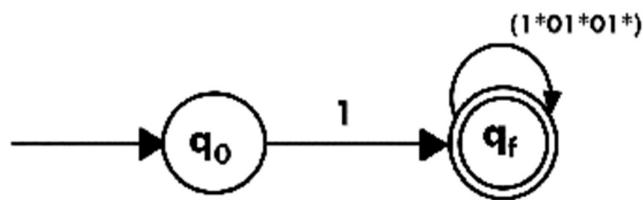
Design a NFA from given regular expression $1(1^* 01^* 01^*)^*$.

Solution: The NFA for the given regular expression is as follows:

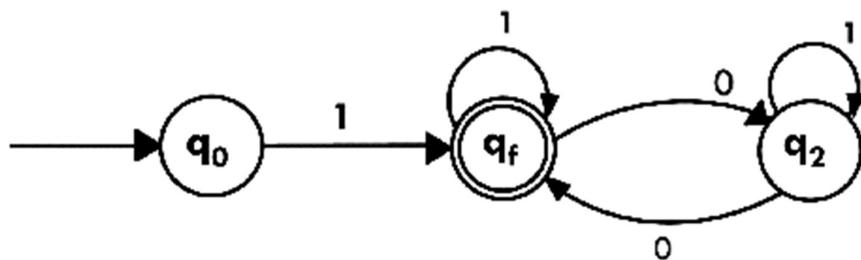
Step 1:



Step 2:



Step 3:



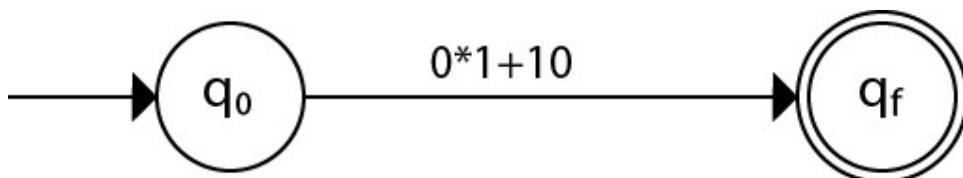
Example 3:

Construct the FA for regular expression $0^*1 + 10$.

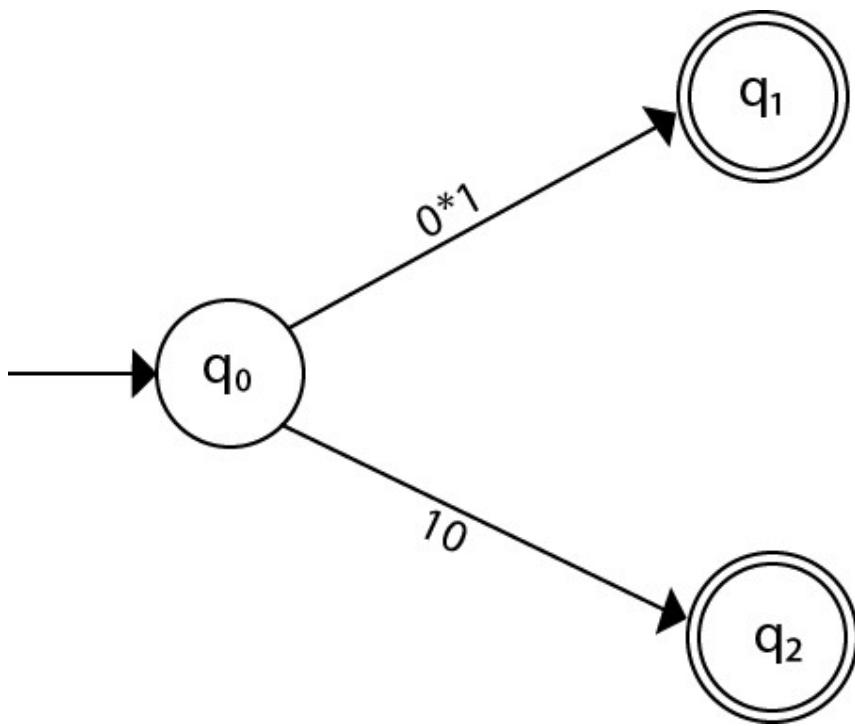
Solution:

We will first construct FA for $R = 0^*1 + 10$ as follows:

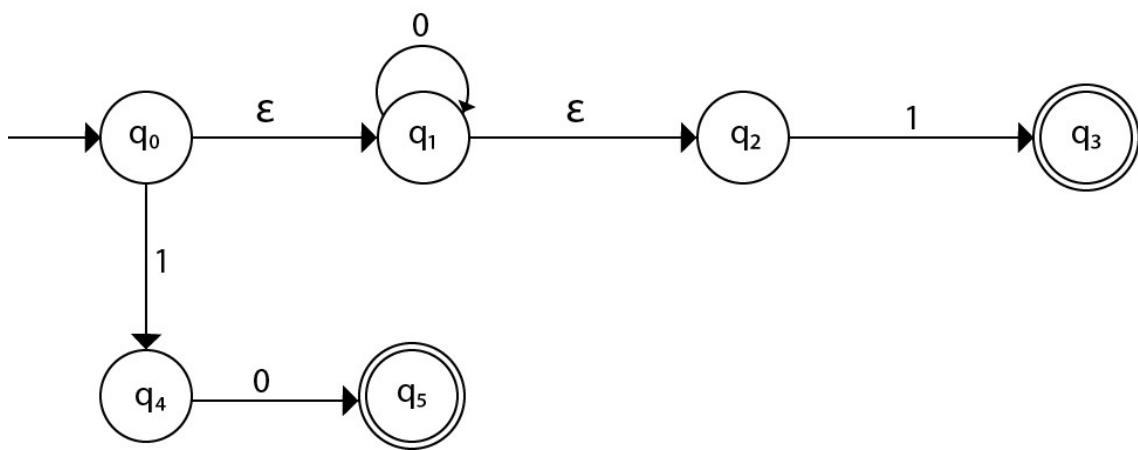
Step 1:



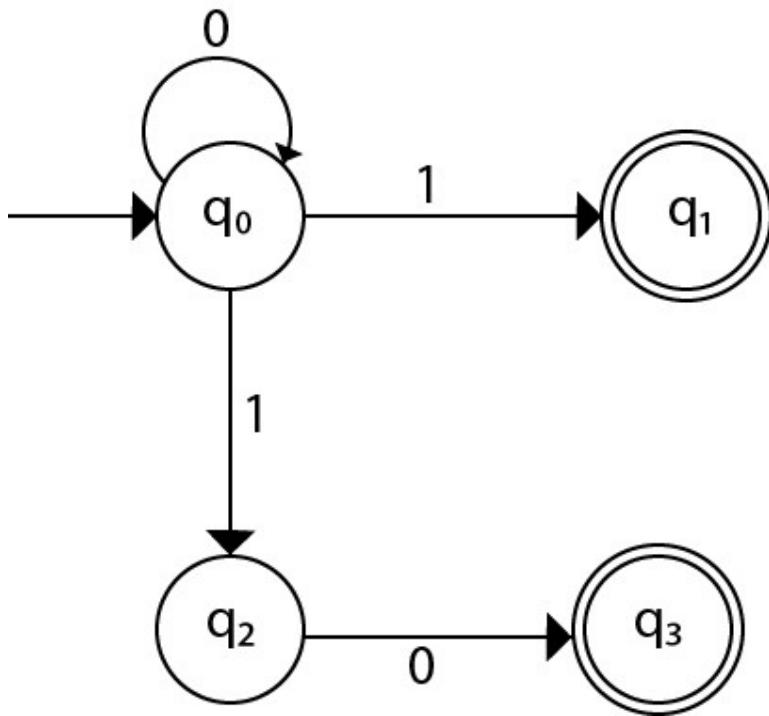
Step 2:



Step 3:



Step 4:



Arden's Theorem

The Arden's Theorem is useful for checking the equivalence of two regular expressions as well as in the conversion of DFA to a regular expression.

Let us see its use in the conversion of DFA to a regular expression.

Following algorithm is used to build the regular expression form given DFA.

1. Let q_1 be the initial state.
2. There are $q_2, q_3, q_4 \dots q_n$ number of states. The final state may be some q_j where $j \leq n$.
3. Let α_{ji} represents the transition from q_j to q_i .
4. Calculate q_i such that

$$q_i = \alpha_{ji} * q_j$$

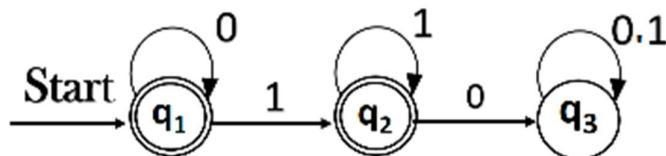
If q_j is a start state then we have:

$$q_i = \alpha_{ji} * q_j + \epsilon$$

5. Similarly, compute the final state which ultimately gives the regular expression ' r' .

Example:

Construct the regular expression for the given DFA



Solution:

Let us write down the equations

$$q_1 = q_1 0 + \epsilon$$

Since q_1 is the start state, so ϵ will be added, and the input 0 is coming to q_1 from q_1 hence we write
State = source state of input \times input coming to it

Similarly,

$$q_2 = q_1 1 + q_2 1$$

$$q_3 = q_2 0 + q_3 (0+1)$$

Since the final states are q_1 and q_2 , we are interested in solving q_1 and q_2 only. Let us see q_1 first

$$q_1 = q_1 0 + \epsilon$$

We can re-write it as

$$q_1 = \epsilon + q_1 0$$

Which is similar to $R = Q + RP$, and gets reduced to $R = QP^*$.

Assuming $R = q_1$, $Q = \epsilon$, $P = 0$

We get

$$q_1 = \epsilon \cdot (0)^*$$

$$q_1 = 0^* \quad (\epsilon \cdot R^* = R^*)$$

Substituting the value into q_2 , we will get

$$q_2 = 0^* 1 + q_2 1$$

$$q_2 = 0^* 1(1)^* \quad (R = Q + RP \rightarrow Q P^*)$$

The regular expression is given by

$$\begin{aligned} r &= q_1 + q_2 \\ &= 0^* + 0^* 1 \cdot 1^* \\ r &= 0^* + 0^* 1^* \quad (1 \cdot 1^* = 1^*) \end{aligned}$$

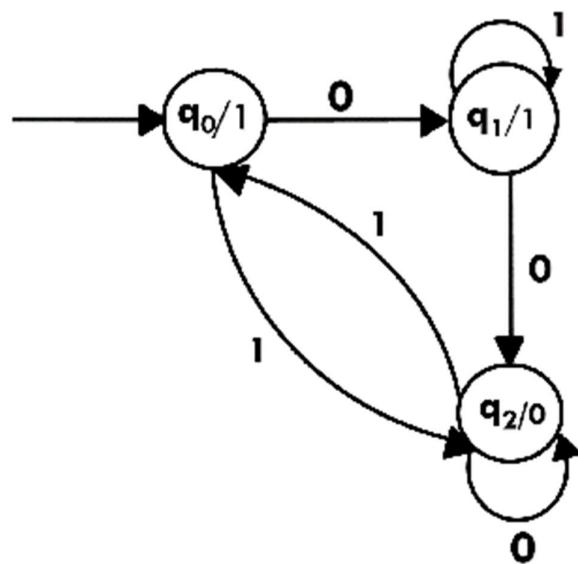
Moore Machine

Moore machine is a finite state machine in which the next state is decided by the current state and current input symbol. The output symbol at a given time depends only on the present state of the machine. Moore machine can be described by 6 tuples $(Q, q_0, \Sigma, O, \delta, \lambda)$ where,

1. **Q: finite set of states**
2. **q_0 : initial state of machine**
3. **Σ : finite set of input symbols**
4. **O: output alphabet**
5. **δ : transition function where $Q \times \Sigma \rightarrow Q$**
6. **λ : output function where $Q \rightarrow O$**

Example 1:

The state diagram for Moore Machine is



Transition table for Moore Machine is:

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

In the above Moore machine, the output is represented with each input state separated by /. The output length for a Moore machine is greater than input by 1.

Input: 010

Transition: $\delta(q_0, 0) \Rightarrow \delta(q_1, 1) \Rightarrow \delta(q_1, 0) \Rightarrow q_2$

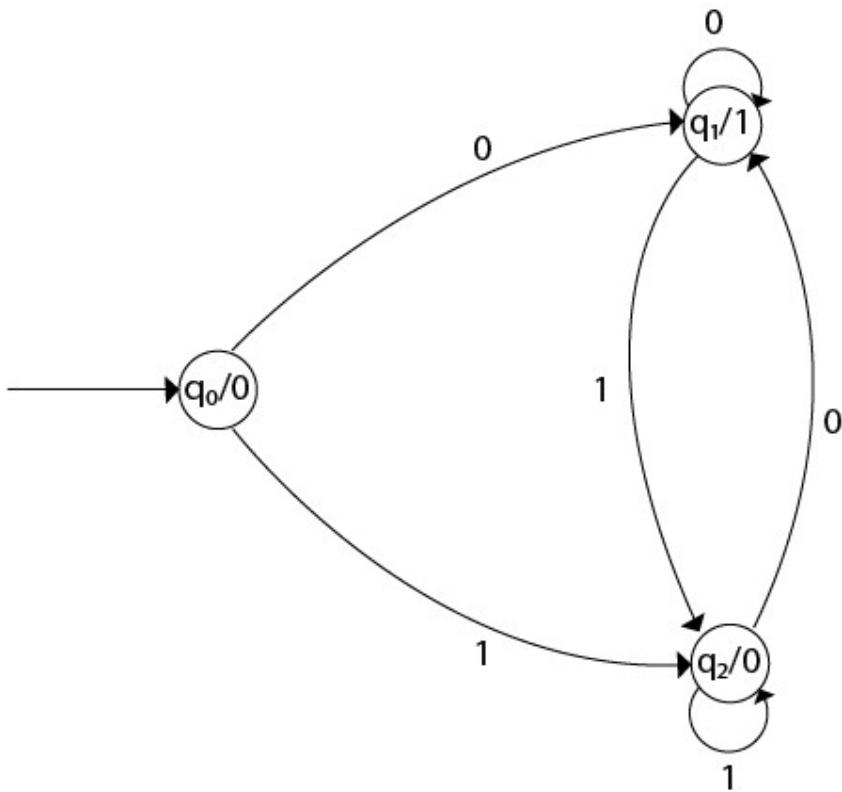
Output: 1110 (1 for q_0 , 1 for q_1 , again 1 for q_1 , 0 for q_2)

Example 2:

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

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

Hence the Moore machine will be,



For instance, take one binary number 1011 then

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

Thus we get 00100 as 1's complement of 1011, we can neglect the initial 0 and the output which we get is 0100 which is 1's complement of 1011. The transaction table is as follows:

Current State	Next State		Output
	0	1	
→ q_0	q_1	q_2	0
q_1	q_1	q_2	1
q_2	q_1	q_2	0

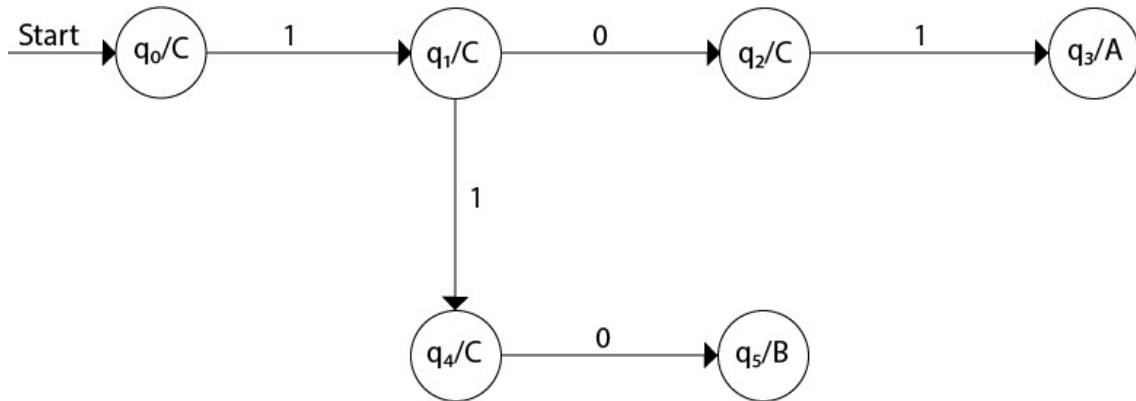
Thus Moore machine $M = (Q, q_0, \Sigma, O, \delta, \lambda)$; where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $O = \{0, 1\}$. the transition table shows the δ and λ functions.

Example 3:

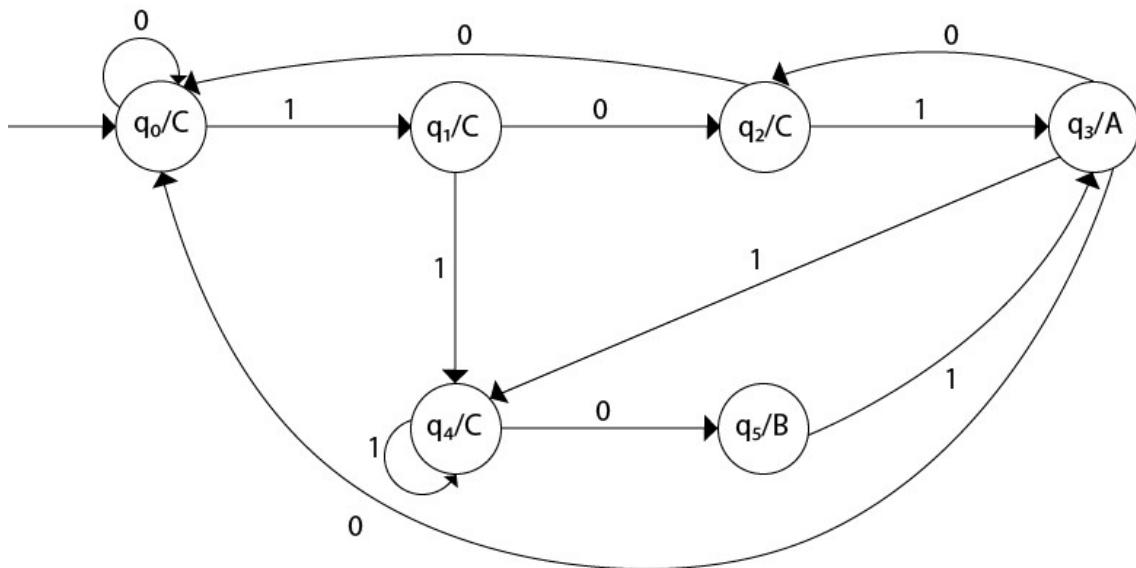
Design a Moore machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.

Solution: For designing such a machine, we will check two conditions, and those are 101 and 110. If we get 101, the output will be A, and if we recognize 110, the output will be B. For other strings, the output will be C.

The partial diagram will be:



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

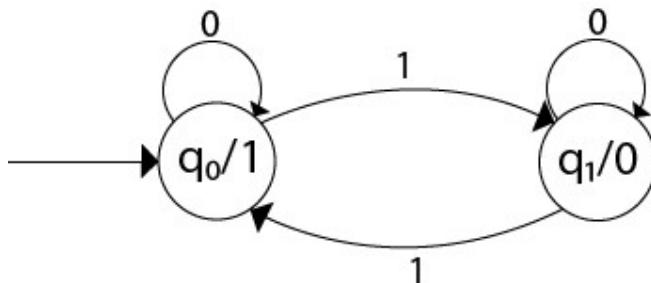


Example 4:

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

Solution:

The Moore machine will be:



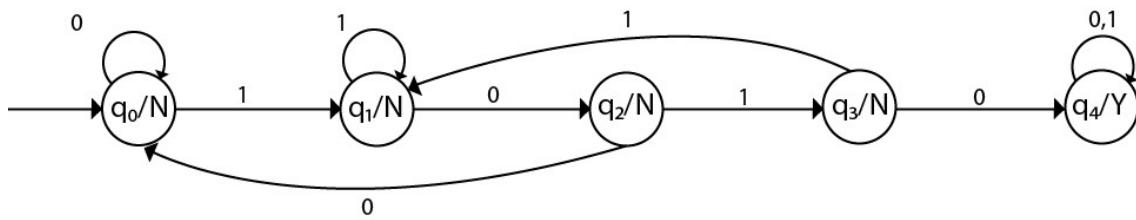
This is the required Moore machine. In this machine, state q_1 accepts an odd number of 1's and state q_0 accepts even number of 1's. There is no restriction on a number of zeros. Hence for 0 input, self-loop can be applied on both the states.

Example 5:

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

Solution:

The Moore machine will be:



Mealy Machine

A Mealy machine is a machine in which output symbol depends upon the present input symbol and present state of the machine. In the Mealy machine, the output is represented with each input symbol for each state separated by /. The Mealy machine can be described by 6 tuples $(Q, q_0, \Sigma, \Omega, \delta, \lambda')$ where

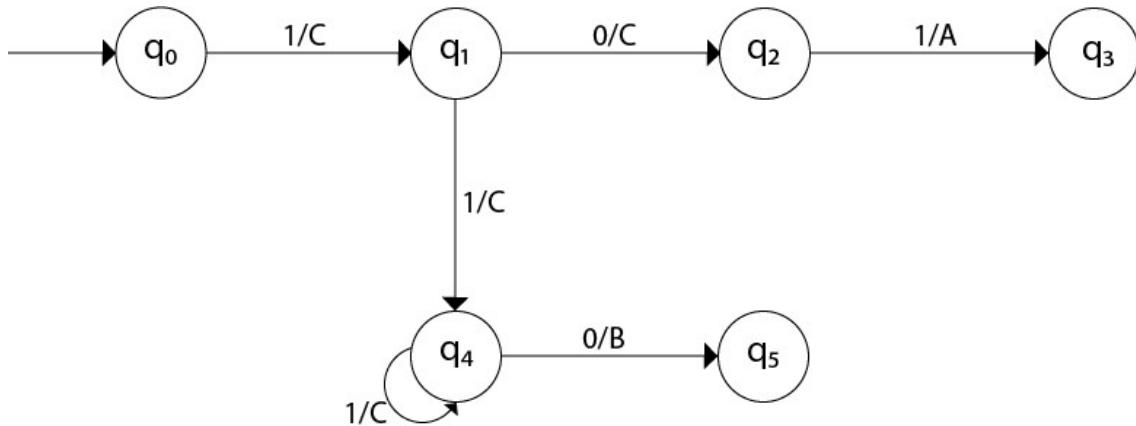
1. **Q : finite set of states**
2. **q_0 : initial state of machine**
3. **Σ : finite set of input alphabet**
4. **Ω : output alphabet**
5. **δ : transition function where $Q \times \Sigma \rightarrow Q$**
6. **λ' : output function where $Q \times \Sigma \rightarrow \Omega$**

Example 1:

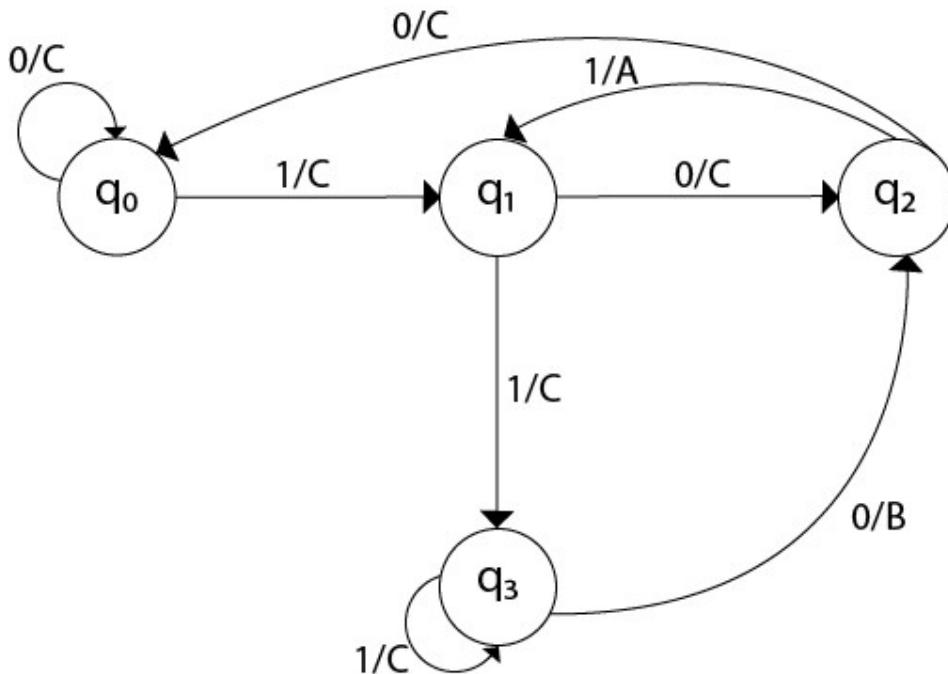
Design a Mealy machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.

Solution: For designing such a machine, we will check two conditions, and those are 101 and 110. If we get 101, the output will be A. If we recognize 110, the output will be B. For other strings the output will be C.

The partial diagram will be:



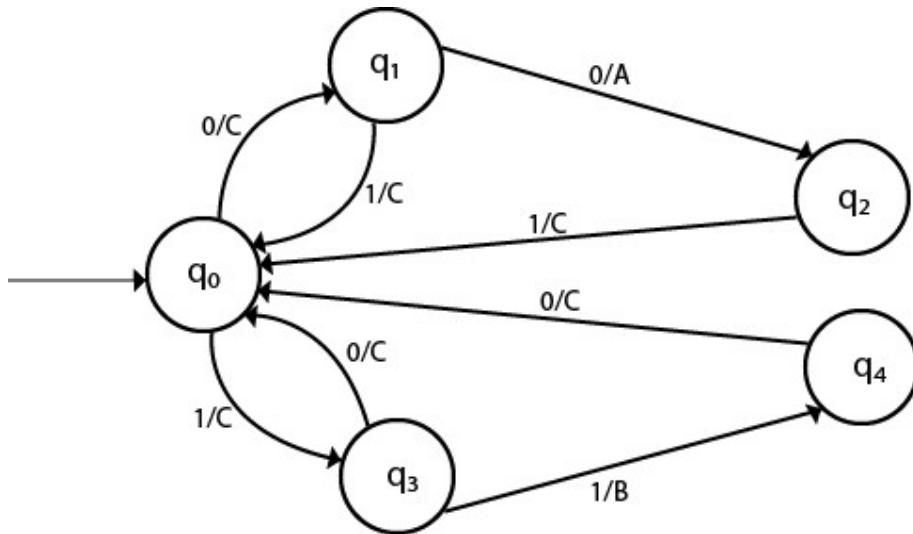
Now we will insert the possibilities of 0's and 1's for each state. Thus the Mealy machine becomes:



Example 2:

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

Solution: The mealy machine will be:



Conversion from Mealy machine to Moore Machine

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

The following steps are used for converting Mealy machine to the Moore machine:

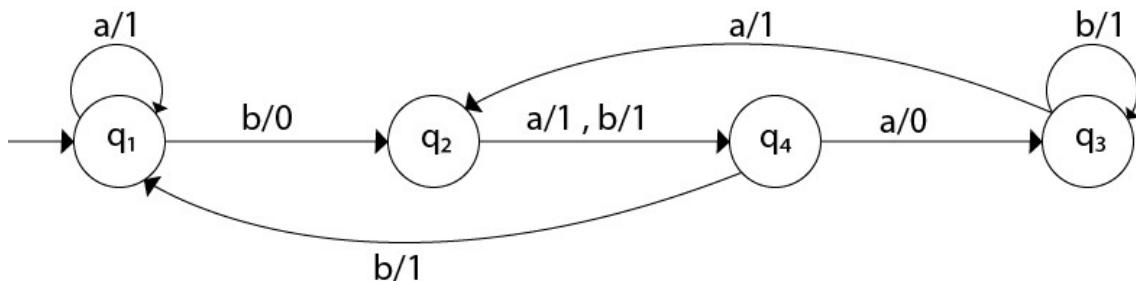
Step 1: For each state(Q_i), calculate the number of different outputs that are available in the transition table of the Mealy machine.

Step 2: Copy state Q_i , if all the outputs of Q_i are the same. Break q_i into n states as Q_{in} , if it has n distinct outputs where $n = 0, 1, 2, \dots$

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

Example 1:

Convert the following Mealy machine into equivalent Moore machine.



Solution:

Transition table for above Mealy machine is as follows:

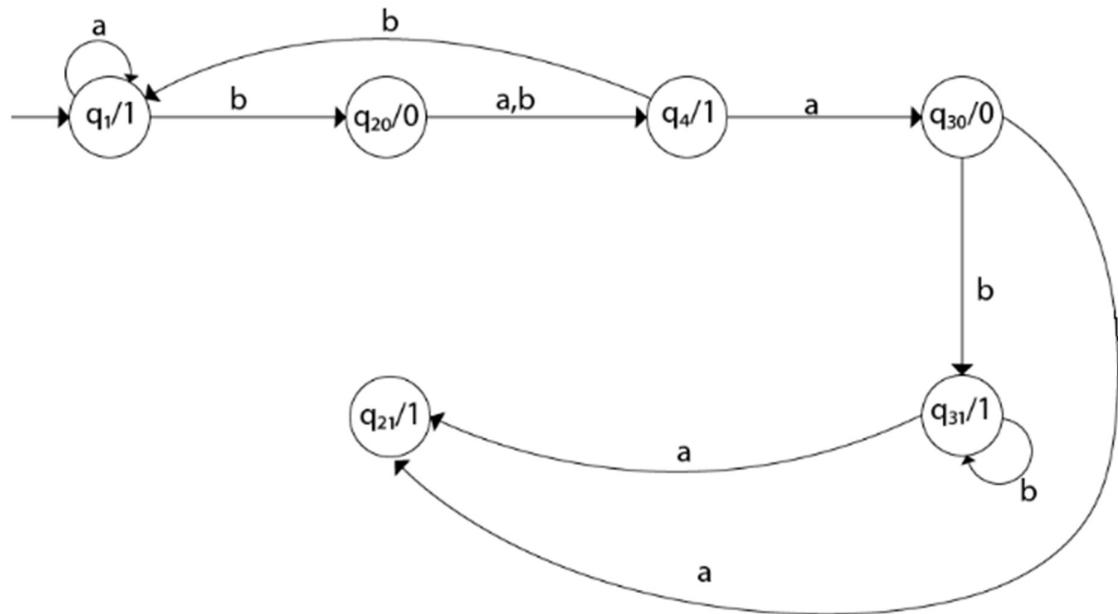
Present State	Next State			
	a		b	
	State	O/P	State	O/P
q_1	q_1	1	q_2	0
q_2	q_4	1	q_4	1
q_3	q_2	1	q_3	1
q_4	q_3	0	q_1	1

- For state q_1 , there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.
- For state q_2 , there is 2 incident edge with output 0 and 1. So, we will split this state into two states q_{20} (state with output 0) and q_{21} (with output 1).
- For state q_3 , there is 2 incident edge with output 0 and 1. So, we will split this state into two states q_{30} (state with output 0) and q_{31} (state with output 1).
- For state q_4 , there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.

Transition table for Moore machine will be:

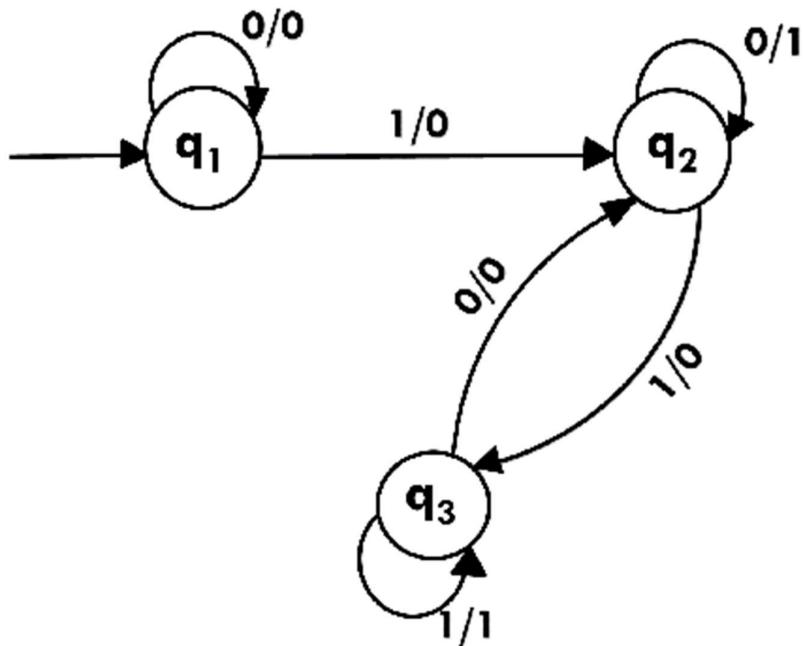
Present State	Next State		Output
	a=0	a=1	
q_1	q_1	q_2	1
q_{20}	q_4	q_4	0
q_{21}	\emptyset	\emptyset	1
q_{30}	q_{21}	q_{31}	0
q_{31}	q_{21}	q_{31}	1
q_4	q_3	q_4	1

Transition diagram for Moore machine will be:



Example 2:

Convert the following Mealy machine into equivalent Moore machine.



Solution:

Transition table for above Mealy machine is as follows:

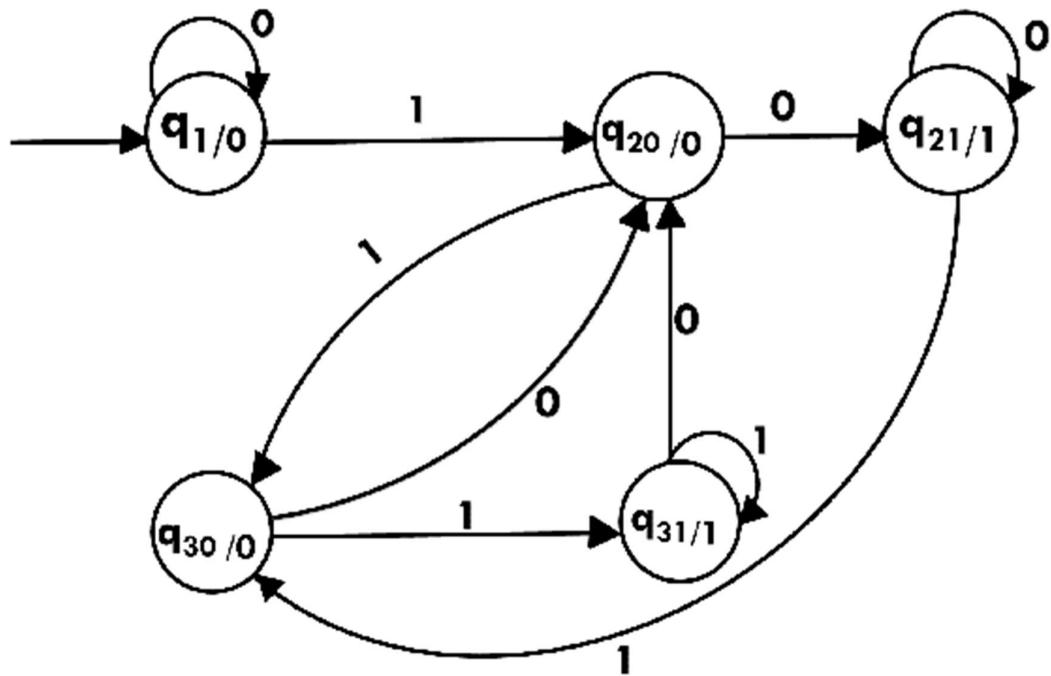
Present State	Next State 0		Next State 1	
	State	O/P	State	O/P
q_1	q_1	0	q_2	0
q_2	q_2	1	q_3	0
q_3	q_2	0	q_3	1

The state q_1 has only one output. The state q_2 and q_3 have both output 0 and 1. So we will create two states for these states. For q_2 , two states will be q_{20} (with output 0) and q_{21} (with output 1). Similarly, for q_3 two states will be q_{30} (with output 0) and q_{31} (with output 1).

Transition table for Moore machine will be:

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

Transition diagram for Moore machine will be:



Conversion from Moore machine to Mealy Machine

In the Moore machine, the output is associated with every state, and in the mealy machine, the output is given along the edge with input symbol. The equivalence of the Moore machine and Mealy machine means both the machines generate the same output string for same input string.

We cannot directly convert Moore machine to its equivalent Mealy machine because the length of the Moore machine is one longer than the Mealy machine for the given input. To convert Moore machine to Mealy machine, state output symbols are distributed into input symbol paths. We are going to use the following method to convert the Moore machine to Mealy machine.

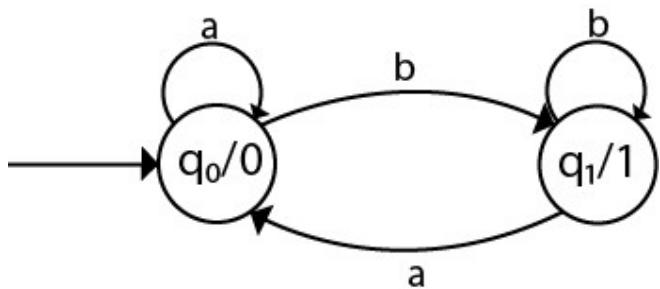
Method for conversion of Moore machine to Mealy machine

Let $M = (Q, \Sigma, \delta, \lambda, q_0)$ be a Moore machine. The equivalent Mealy machine can be represented by $M' = (Q, \Sigma, \delta, \lambda', q_0)$. The output function λ' can be obtained as:

1. $\lambda'(q, a) = \lambda(\delta(q, a))$

Example 1:

Convert the following Moore machine into its equivalent Mealy machine.



Solution:

The transition table of given Moore machine is as follows:

Q	a	b	Output(λ)
q0	q0	q1	0
q1	q0	q1	1

The equivalent Mealy machine can be obtained as follows:

1. $\lambda'(q0, a) = \lambda(\delta(q0, a))$
2. $= \lambda(q0)$
3. $= 0$
- 4.
5. $\lambda'(q0, b) = \lambda(\delta(q0, b))$
6. $= \lambda(q1)$
7. $= 1$

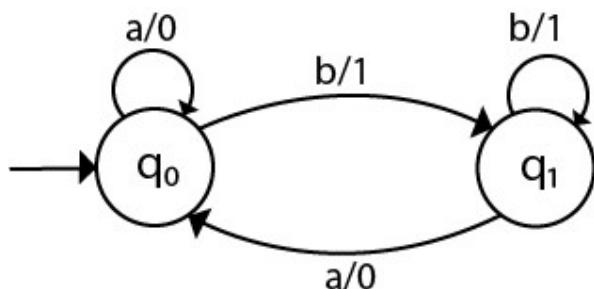
The λ for state q1 is as follows:

1. $\lambda'(q1, a) = \lambda(\delta(q1, a))$
2. $= \lambda(q0)$
3. $= 0$
- 4.
5. $\lambda'(q1, b) = \lambda(\delta(q1, b))$
6. $= \lambda(q1)$
7. $= 1$

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

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

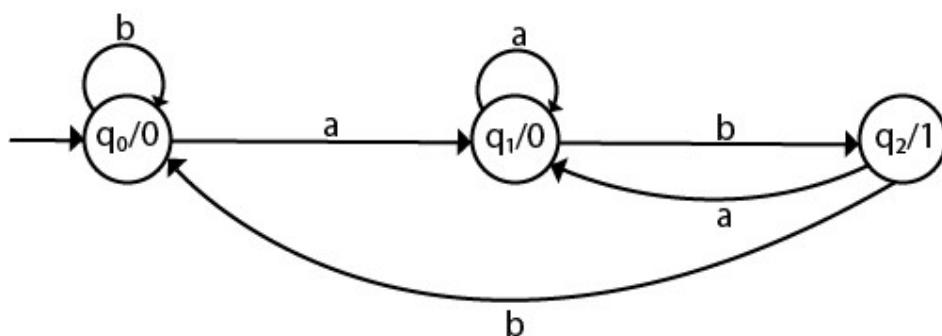
The equivalent Mealy machine will be,



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

Example 2:

Convert the given Moore machine into its equivalent Mealy machine.



Solution:

The transition table of given Moore machine is as follows:

Q	a	b	Output(λ)
$q_0/0$			

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

The equivalent Mealy machine can be obtained as follows:

$$1. \lambda'(q0, a) = \lambda(\delta(q0, a))$$

$$2. \quad = \lambda(q1)$$

$$3. \quad = \text{0}$$

4.

$$5. \lambda'(q0, b) = \lambda(\delta(q0, b))$$

$$6. \quad = \lambda(q0)$$

$$7. \quad = \text{0}$$

The λ for state q1 is as follows:

$$1. \lambda'(q1, a) = \lambda(\delta(q1, a))$$

$$2. \quad = \lambda(q1)$$

$$3. \quad = \text{0}$$

4.

$$5. \lambda'(q1, b) = \lambda(\delta(q1, b))$$

$$6. \quad = \lambda(q2)$$

$$7. \quad = \text{1}$$

The λ for state q2 is as follows:

$$1. \lambda'(q2, a) = \lambda(\delta(q2, a))$$

$$2. \quad = \lambda(q1)$$

$$3. \quad = \text{0}$$

4.

$$5. \lambda'(q2, b) = \lambda(\delta(q2, b))$$

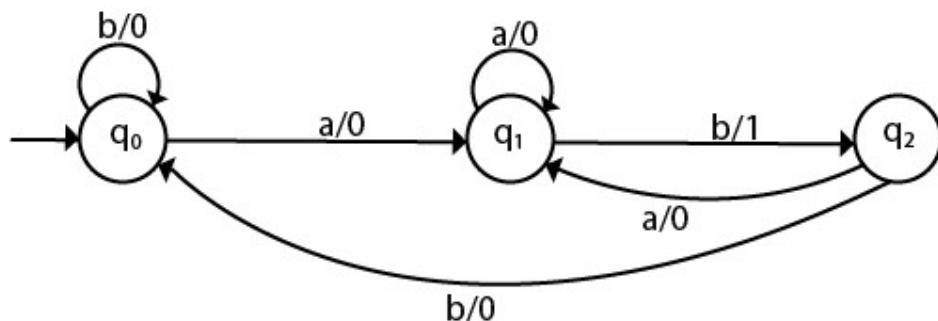
$$6. \quad = \lambda(q0)$$

$$7. \quad = \text{0}$$

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

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

The equivalent Mealy machine will be,



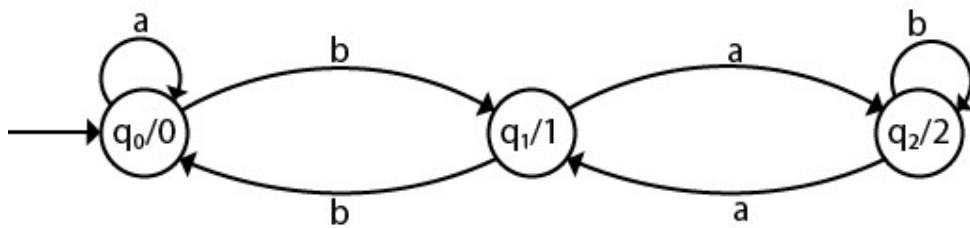
Example 3:

Convert the given Moore machine into its equivalent Mealy machine.

Q	a	b	Output(λ)
q_0	q_0	q_1	0
q_1	q_2	q_0	1
q_2	q_1	q_2	2

Solution:

The transaction diagram for the given problem can be drawn as:



The equivalent Mealy machine can be obtained as follows:

1. $\lambda'(q_0, a) = \lambda(\delta(q_0, a))$
2. $= \lambda(q_0)$
3. $= 0$
- 4.
5. $\lambda'(q_0, b) = \lambda(\delta(q_0, b))$
6. $= \lambda(q_1)$
7. $= 1$

The λ for state q_1 is as follows:

1. $\lambda'(q_1, a) = \lambda(\delta(q_1, a))$
2. $= \lambda(q_2)$
3. $= 2$
- 4.
5. $\lambda'(q_1, b) = \lambda(\delta(q_1, b))$
6. $= \lambda(q_0)$
7. $= 0$

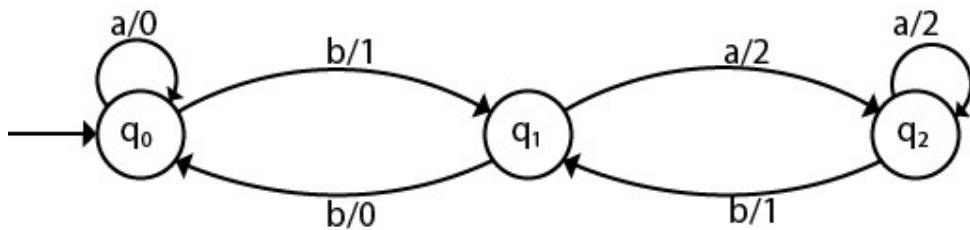
The λ for state q_2 is as follows:

1. $\lambda'(q_2, a) = \lambda(\delta(q_2, a))$
2. $= \lambda(q_1)$
3. $= 1$
- 4.
5. $\lambda'(q_2, b) = \lambda(\delta(q_2, b))$
6. $= \lambda(q_2)$
7. $= 2$

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

Σ	Input a		Input b	
	State	O/P	State	O/P
q_0	q_0	0	q_1	1
q_1	q_2	2	q_0	0
q_2	q_1	1	q_2	2

The equivalent Mealy machine will be,



Context-Free Grammar (CFG)

CFG stands for context-free grammar. It is a formal grammar which is used to generate all possible patterns of strings in a given formal language. Context-free grammar G can be defined by four tuples as:

1. $G = (V, T, P, S)$

Where,

G is the grammar, which consists of a set of production rule. It is used to generate the string of a language.

T is the final set of a terminal symbol. It is denoted by lower case letters.

V is the final set of a non-terminal symbol. It is denoted by capital letters.

P is a set of production rules, which is used for replacing non-terminals symbols(on the left side of the production) in a string with other terminal or non-terminal symbols(on the right side of the production).

S is the start symbol which is used to derive the string. We can derive the string by repeatedly replacing a non-terminal by the right-hand side of

the production until all non-terminal have been replaced by terminal symbols.

Example 1:

Construct the CFG for the language having any number of a's over the set $\Sigma = \{a\}$.

Solution:

As we know the regular expression for the above language is

1. r.e. = a^*

Production rule for the Regular expression is as follows:

1. $S \rightarrow aS$ rule 1
2. $S \rightarrow \epsilon$ rule 2

Now if we want to derive a string "aaaaaaaa", we can start with start symbols.

1. S
2. aS
3. aaS rule 1
4. $aaaS$ rule 1
5. $aaaaS$ rule 1
6. $aaaaaS$ rule 1
7. $aaaaaaS$ rule 1
8. $aaaaaaa\epsilon$ rule 2
9. $aaaaaaaa$

The r.e. = a^* can generate a set of string $\{\epsilon, a, aa, aaa, \dots\}$. We can have a null string because S is a start symbol and rule 2 gives $S \rightarrow \epsilon$.

Example 2:

Construct a CFG for the regular expression $(0+1)^*$

Solution:

The CFG can be given by,

1. Production rule (P):
2. $S \rightarrow 0S \mid 1S$

3. $S \rightarrow \epsilon$

The rules are in the combination of 0's and 1's with the start symbol. Since $(0+1)^*$ indicates $\{\epsilon, 0, 1, 01, 10, 00, 11, \dots\}$. In this set, ϵ is a string, so in the rule, we can set the rule $S \rightarrow \epsilon$.

Example 3:

Construct a CFG for a language $L = \{wcwR \mid w \in (a, b)^*\}$.

Solution:

The string that can be generated for a given language is {aacaa, bcb, abcba, bacab, abbcbbba,}

The grammar could be:

1. $S \rightarrow aSa$ rule 1
2. $S \rightarrow bSb$ rule 2
3. $S \rightarrow c$ rule 3

Now if we want to derive a string "abbcbbba", we can start with start symbols.

1. $S \rightarrow aSa$
2. $S \rightarrow abSba$ from rule 2
3. $S \rightarrow abbSbbba$ from rule 2
4. $S \rightarrow abbcbbba$ from rule 3

Thus any of this kind of string can be derived from the given production rules.

Example 4:

Construct a CFG for the language $L = a^n b^{2n}$ where $n \geq 1$.

Solution:

The string that can be generated for a given language is {abb, aabb, aaabbbbb,}.

The grammar could be:

1. $S \rightarrow aSbb \mid abb$

Now if we want to derive a string "aabbbb", we can start with start symbols.

1. $S \rightarrow aSbb$
2. $S \rightarrow aabbbb$

Derivation

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

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

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

1. Leftmost Derivation:

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

Example:

Production rules:

1. $E = E + E$
2. $E = E - E$
3. $E = a | b$

Input

1. $a - b + a$

The leftmost derivation is:

1. $E = E + E$
2. $E = E - E + E$
3. $E = a - E + E$
4. $E = a - b + E$

$$5. E = a - b + a$$

2. Rightmost Derivation:

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

Example

Production rules:

1. $E = E + E$
2. $E = E - E$
3. $E = a | b$

Input

1. $a - b + a$

The rightmost derivation is:

1. $E = E - E$
2. $E = E - E + E$
3. $E = E - E + a$
4. $E = E - b + a$
5. $E = a - b + a$

When we use the leftmost derivation or rightmost derivation, we may get the same string. This type of derivation does not affect on getting of a string.

Examples of Derivation:

Example 1:

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

1. $S \rightarrow AB | \epsilon$
2. $A \rightarrow aB$
3. $B \rightarrow Sb$

Solution:

Leftmost derivation:

S
AB
aB B
a Sb B
A ε bB
ab Sb
ab ε b
abb

Rightmost derivation:

S
AB
A Sb
A ε b
aB b
a Sb b
a ε bb
abb

Example 2:

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

1. $S \rightarrow aB \mid bA$
2. $S \rightarrow a \mid aS \mid bAA$
3. $S \rightarrow b \mid aS \mid aBB$

Solution:

Leftmost derivation:

1. S
2. $aB \quad S \rightarrow aB$

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

Rightmost derivation:

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

Example 3:

Derive the string "00101" for leftmost derivation and rightmost derivation using a CFG given by,

1. S → A1B
2. A → 0A | ε
3. B → 0B | 1B | ε

Solution:

Leftmost derivation:

1. S
2. A1B
3. 0A1B
4. 00A1B
5. 001B
6. 0010B
7. 00101B
8. 00101

Rightmost derivation:

1. S
2. A1B
3. A10B
4. A101B
5. A101
6. OA101
7. OOA101
8. **OO101**

Derivation Tree

Derivation tree is a graphical representation for the derivation of the given production rules for a given CFG. It is the simple way to show how the derivation can be done to obtain some string from a given set of production rules. The derivation tree is also called a parse tree.

Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

A parse tree contains the following properties:

1. The root node is always a node indicating start symbols.
2. The derivation is read from left to right.
3. The leaf node is always terminal nodes.
4. The interior nodes are always the non-terminal nodes.

Example 1:

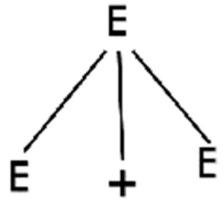
Production rules:

1. $E = E + E$
2. $E = E * E$
3. $E = a | b | c$

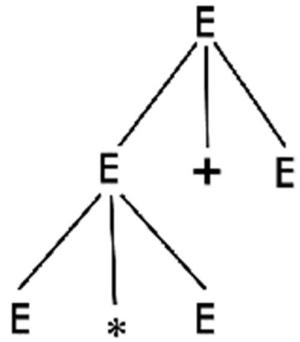
Input

1. $a * b + c$

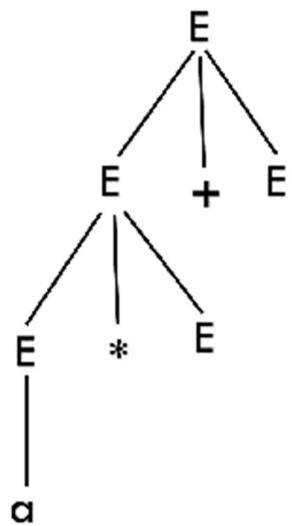
Step 1:



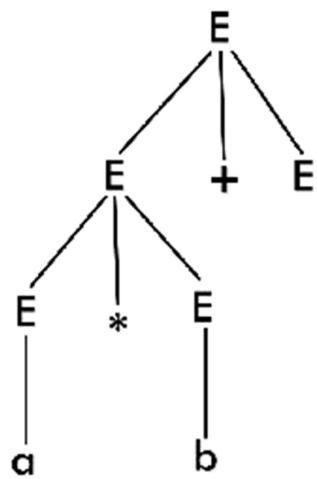
Step 2:



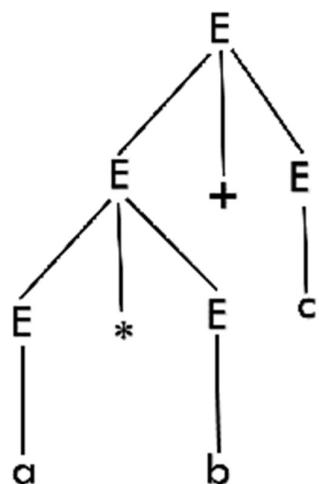
Step 2:



Step 4:



Step 5:



Note: We can draw a derivation tree step by step or directly in one step.

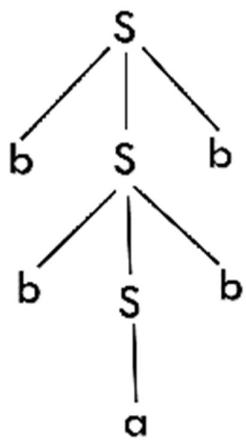
Example 2:

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

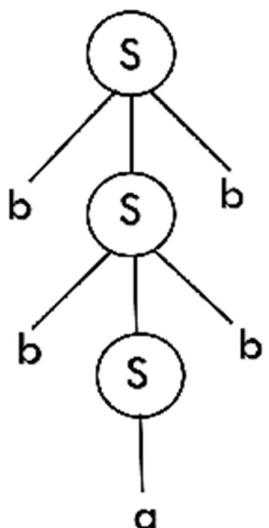
1. $S \rightarrow bSb \mid a \mid b$

Solution:

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



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



Example 3:

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

1. $S \rightarrow aB \mid bA$
2. $A \rightarrow a \mid aS \mid bAA$
3. $B \rightarrow b \mid bS \mid aBB$

Solution:

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

S

aB

a [aBB]

aa [bS] B

aab [bA] B

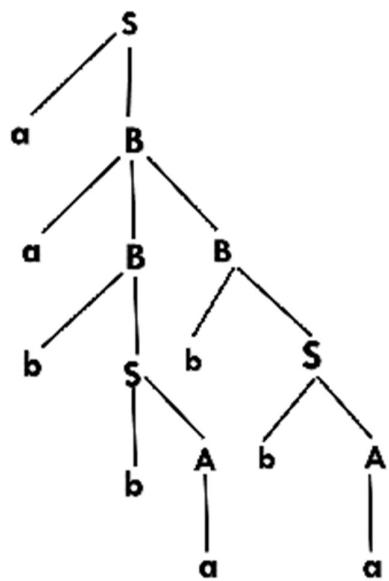
aabb [a] B

aabba [bS]

aabbab [bA]

aabbabb [a]

Now, the derivation tree is as follows:



Example 4:

Show the derivation tree for string "aabbabb" with the following grammar.

1. $S \rightarrow AB \mid \epsilon$
2. $A \rightarrow aB$
3. $B \rightarrow Sb$

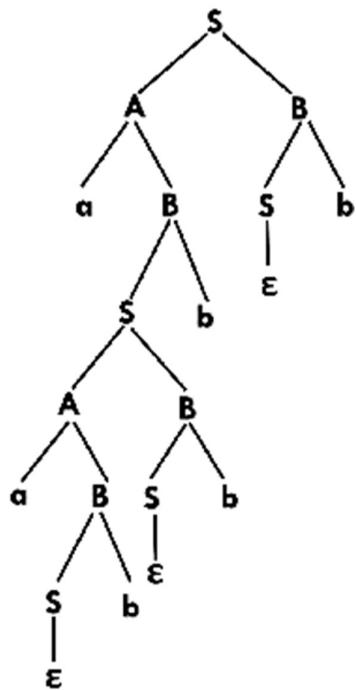
Solution:

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

S

AB
aB B
a Sb B
a AB bB
a aB BbB
aa Sb BbB
aa ε bBbB
aab Sb bB
aab ε bbB
aabb Sb
aabb ε b
aabbbb

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



Ambiguity in Grammar

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string. If the grammar is not ambiguous, then it is called unambiguous.

If the grammar has ambiguity, then it is not good for compiler construction. No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

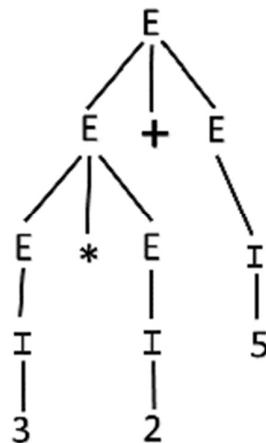
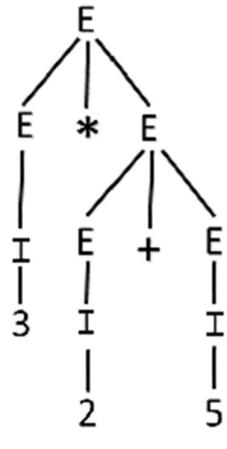
Example 1:

Let us consider a grammar G with the production rule

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow \epsilon | 0 | 1 | 2 | \dots | 9$

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:



Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.

Example 2:

Check whether the given grammar G is ambiguous or not.

1. $E \rightarrow E + E$

2. $E \rightarrow E - E$
3. $E \rightarrow id$

Solution:

From the above grammar String "id + id - id" can be derived in 2 ways:

First Leftmost derivation

1. $E \rightarrow E + E$
2. $\rightarrow id + E$
3. $\rightarrow id + E - E$
4. $\rightarrow id + id - E$
5. $\rightarrow id + id - id$

Second Leftmost derivation

1. $E \rightarrow E - E$
2. $\rightarrow E + E - E$
3. $\rightarrow id + E - E$
4. $\rightarrow id + id - E$
5. $\rightarrow id + id - id$

Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

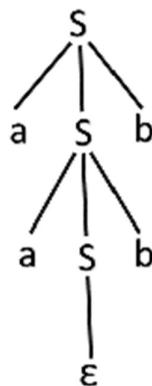
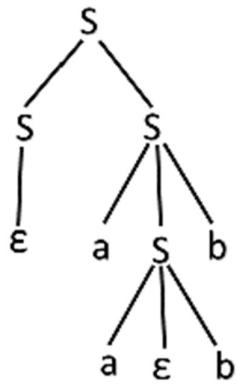
Example 3:

Check whether the given grammar G is ambiguous or not.

1. $S \rightarrow aSb \mid SS$
2. $S \rightarrow \epsilon$

Solution:

For the string "aabb" the above grammar can generate two parse trees



Since there are two parse trees for a single string "aabbb", the grammar G is ambiguous.

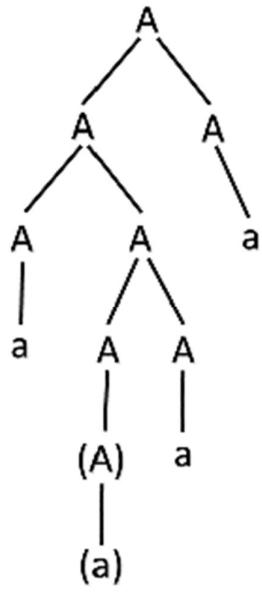
Example 4:

Check whether the given grammar G is ambiguous or not.

1. $A \rightarrow AA$
2. $A \rightarrow (A)$
3. $A \rightarrow a$

Solution:

For the string "a(a)aa" the above grammar can generate two parse trees:



Since there are two parse trees for a single string "a(a)aa", the grammar G is ambiguous.

Unambiguous Grammar

A grammar can be unambiguous if the grammar does not contain ambiguity that means if it does not contain more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.

To convert ambiguous grammar to unambiguous grammar, we will apply the following rules:

1. If the left associative operators (+, -, *, /) are used in the production rule, then apply left recursion in the production rule. Left recursion means that the leftmost symbol on the right side is the same as the non-terminal on the left side. For example,

$$1. \quad X \rightarrow Xa$$

2. If the right associative operator (^) is used in the production rule then apply right recursion in the production rule. Right recursion means that the rightmost symbol on the left side is the same as the non-terminal on the right side. For example,

$$1. \quad X \rightarrow aX$$

Example 1:

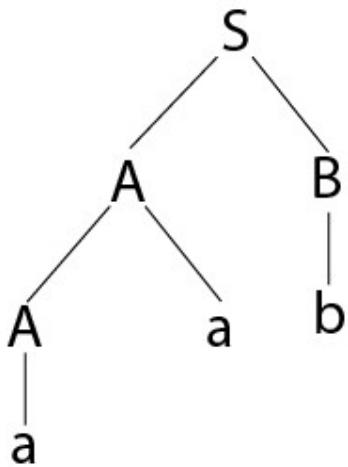
Consider a grammar G is given as follows:

1. $S \rightarrow AB \mid aAB$
2. $A \rightarrow a \mid Aa$
3. $B \rightarrow b$

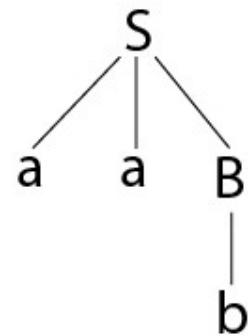
Determine whether the grammar G is ambiguous or not. If G is ambiguous, construct an unambiguous grammar equivalent to G.

Solution:

Let us derive the string "aab"



Parse tree 1



Parse tree 2

As there are two different parse tree for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

1. $S \rightarrow AB$
2. $A \rightarrow Aa \mid a$
3. $B \rightarrow b$

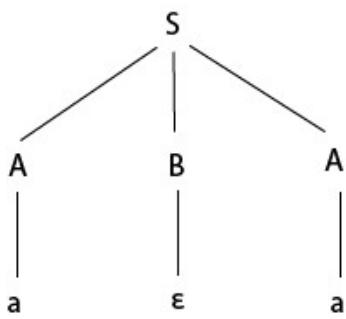
Example 2:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

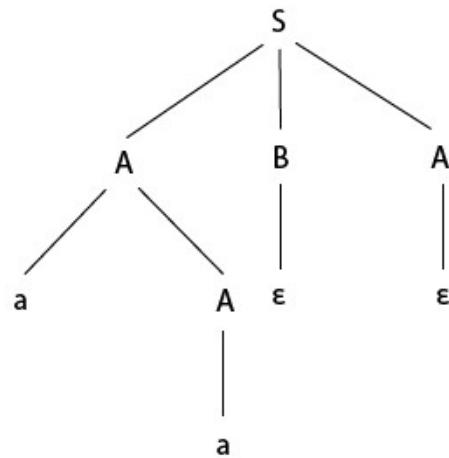
1. $S \rightarrow ABA$
2. $A \rightarrow aA \mid \epsilon$
3. $B \rightarrow bB \mid \epsilon$

Solution:

The given grammar is ambiguous because we can derive two different parse tree for string aa.



Parse tree 1



Parse tree 2

The unambiguous grammar is:

1. $S \rightarrow aXY \mid bYZ \mid \epsilon$
2. $Z \rightarrow aZ \mid a$
3. $X \rightarrow aXY \mid a \mid \epsilon$
4. $Y \rightarrow bYZ \mid b \mid \epsilon$

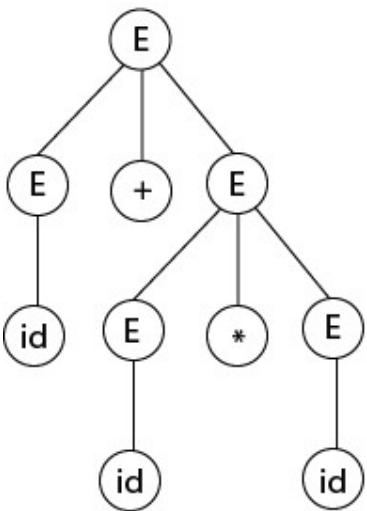
Example 3:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

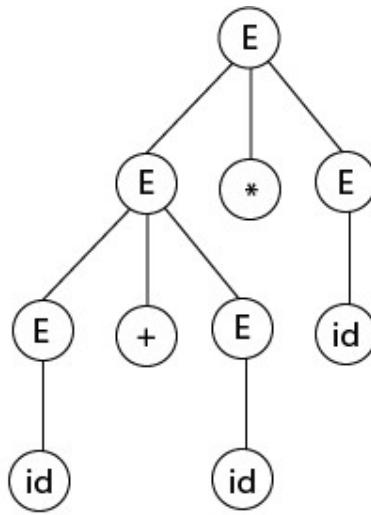
1. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow id$

Solution:

Let us derive the string "id + id * id"



Parse tree 1



Parse tree 2

As there are two different parse tree for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow id$

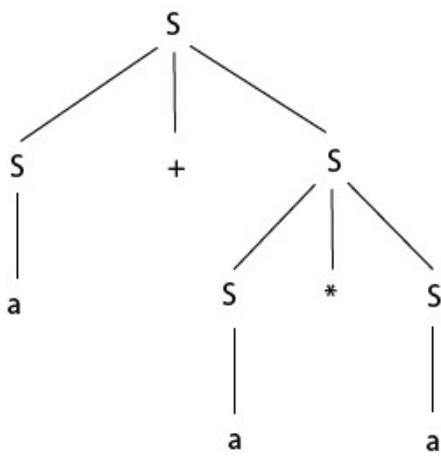
Example 4:

Check that the given grammar is ambiguous or not. Also, find an equivalent unambiguous grammar.

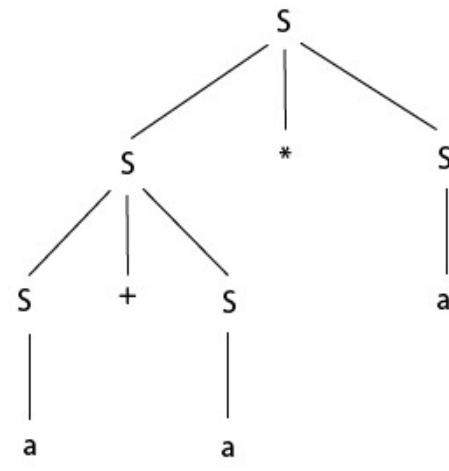
1. $S \rightarrow S + S$
2. $S \rightarrow S * S$
3. $S \rightarrow S^S$
4. $S \rightarrow a$

Solution:

The given grammar is ambiguous because the derivation of string aab can be represented by the following string:



Parse tree 1



Parse tree 2

Unambiguous grammar will be:

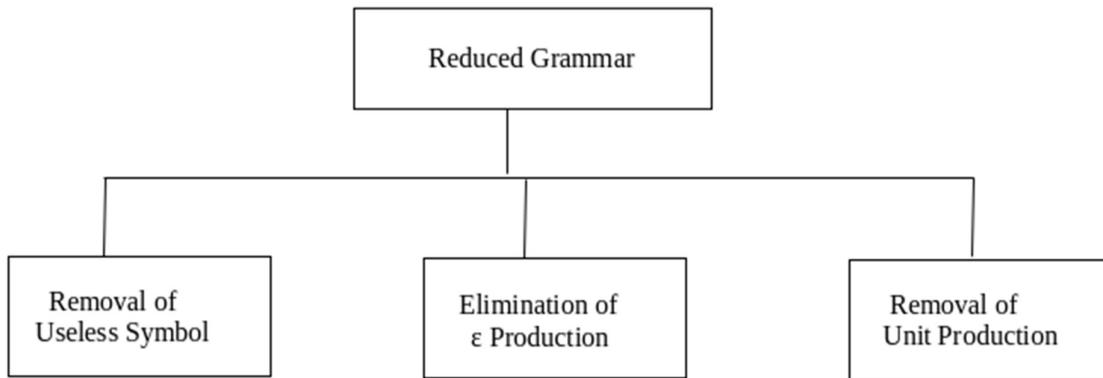
1. $S \rightarrow S + A \mid$
2. $A \rightarrow A^* B \mid B$
3. $B \rightarrow C^* B \mid C$
4. $C \rightarrow a$

Simplification of CFG

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

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

Let us study the reduction process in detail./p>



Removal of Useless Symbols

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

For Example:

1. $T \rightarrow aaB \mid abA \mid aaT$
2. $A \rightarrow aA$
3. $B \rightarrow ab \mid b$
4. $C \rightarrow ad$

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

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

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

Elimination of ϵ Production

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

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

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

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

Example:

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

1. $S \rightarrow XYX$
2. $X \rightarrow 0X \mid \epsilon$
3. $Y \rightarrow 1Y \mid \epsilon$

Solution:

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

Let us take

1. $S \rightarrow XYX$

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

1. $S \rightarrow YX$

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

1. $S \rightarrow XY$

If $Y = \epsilon$ then

1. $S \rightarrow XX$

If Y and X are ϵ then,

1. $S \rightarrow X$

If both X are replaced by ϵ

1. $S \rightarrow Y$

Now,

$$1. S \rightarrow XY | YX | XX | X | Y$$

Now let us consider

$$1. X \rightarrow 0X$$

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

$$1. X \rightarrow 0$$

$$2. X \rightarrow 0X | \epsilon$$

Similarly $Y \rightarrow 1Y | \epsilon$

Collectively we can rewrite the CFG with removed ϵ production as

$$1. S \rightarrow XY | YX | XX | X | Y$$

$$2. X \rightarrow 0X | 0$$

$$3. Y \rightarrow 1Y | 1$$

Removing Unit Productions

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

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

Step 2: Now delete $X \rightarrow Y$ from the grammar.

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

For example:

$$1. S \rightarrow 0A | 1B | C$$

$$2. A \rightarrow 0S | 00$$

$$3. B \rightarrow 1 | A$$

$$4. C \rightarrow 01$$

Solution:

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

$$1. S \rightarrow 0A | 1B | 01$$

Similarly, $B \rightarrow A$ is also a unit production so we can modify it as

$$1. B \rightarrow 1 | 0S | 00$$

Thus finally we can write CFG without unit production as

1. $S \rightarrow 0A | 1B | 01$
2. $A \rightarrow 0S | 00$
3. $B \rightarrow 1 | 0S | 00$
4. $C \rightarrow 01$

Chomsky's Normal Form (CNF)

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

- o Start symbol generating ϵ . For example, $A \rightarrow \epsilon$.
- o A non-terminal generating two non-terminals. For example, $S \rightarrow AB$.
- o A non-terminal generating a terminal. For example, $S \rightarrow a$.

For example:

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

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

Steps for converting CFG into CNF

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

$$1. S1 \rightarrow S$$

Where $S1$ is the new start symbol.

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

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

1. $S \rightarrow RA$
2. $R \rightarrow a$

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

1. $S \rightarrow RS$
2. $R \rightarrow AS$

Example:

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

1. $S \rightarrow a | aA | B$
2. $A \rightarrow aBB | \epsilon$
3. $B \rightarrow Aa | b$

Solution:

Step 1: We will create a new production $S_1 \rightarrow S$, as the start symbol S appears on the RHS. The grammar will be:

1. $S_1 \rightarrow S$
2. $S \rightarrow a | aA | B$
3. $A \rightarrow aBB | \epsilon$
4. $B \rightarrow Aa | b$

Step 2: As grammar G1 contains $A \rightarrow \epsilon$ null production, its removal from the grammar yields:

1. $S_1 \rightarrow S$
2. $S \rightarrow a | aA | B$
3. $A \rightarrow aBB$
4. $B \rightarrow Aa | b | a$

Now, as grammar G1 contains Unit production $S \rightarrow B$, its removal yield:

1. $S_1 \rightarrow S$
2. $S \rightarrow a | aA | Aa | b$
3. $A \rightarrow aBB$
4. $B \rightarrow Aa | b | a$

Also remove the unit production $S_1 \rightarrow S$, its removal from the grammar yields:

1. $S_0 \rightarrow a | aA | Aa | b$
2. $S \rightarrow a | aA | Aa | b$
3. $A \rightarrow aBB$
4. $B \rightarrow Aa | b | a$

Step 3: In the production rule $S_0 \rightarrow aA | Aa$, $S \rightarrow aA | Aa$, $A \rightarrow aBB$ and $B \rightarrow Aa$, terminal a exists on RHS with non-terminals. So we will replace terminal a with X :

1. $S_0 \rightarrow a | XA | AX | b$
2. $S \rightarrow a | XA | AX | b$
3. $A \rightarrow XBB$
4. $B \rightarrow AX | b | a$
5. $X \rightarrow a$

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

1. $S_0 \rightarrow a | XA | AX | b$
2. $S \rightarrow a | XA | AX | b$
3. $A \rightarrow RB$
4. $B \rightarrow AX | b | a$
5. $X \rightarrow a$
6. $R \rightarrow XB$

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

Greibach Normal Form (GNF)

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

- o A start symbol generating ϵ . For example, $S \rightarrow \epsilon$.
- o A non-terminal generating a terminal. For example, $A \rightarrow a$.
- o A non-terminal generating a terminal which is followed by any number of non-terminals. For example, $S \rightarrow aASB$.

For example:

- $G1 = \{S \rightarrow aAB \mid aB, A \rightarrow aA \mid a, B \rightarrow bB \mid b\}$
- $G2 = \{S \rightarrow aAB \mid aB, A \rightarrow aA \mid \epsilon, B \rightarrow bB \mid \epsilon\}$

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

Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

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

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

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

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

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

Example:

- $S \rightarrow XB \mid AA$
- $A \rightarrow a \mid SA$
- $B \rightarrow b$
- $X \rightarrow a$

Solution:

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

The production rule $A \rightarrow SA$ is not in GNF, so we substitute $S \rightarrow XB \mid AA$ in the production rule $A \rightarrow SA$ as:

- $S \rightarrow XB \mid AA$
- $A \rightarrow a \mid XBA \mid AAA$
- $B \rightarrow b$
- $X \rightarrow a$

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

1. $S \rightarrow aB \mid AA$
2. $A \rightarrow a \mid aBA \mid AAA$
3. $B \rightarrow b$
4. $X \rightarrow a$

Now we will remove left recursion ($A \rightarrow AAA$), we get:

1. $S \rightarrow aB \mid AA$
2. $A \rightarrow aC \mid aBAC$
3. $C \rightarrow AAC \mid \epsilon$
4. $B \rightarrow b$
5. $X \rightarrow a$

Now we will remove null production $C \rightarrow \epsilon$, we get:

1. $S \rightarrow aB \mid AA$
2. $A \rightarrow aC \mid aBAC \mid a \mid aBA$
3. $C \rightarrow AAC \mid AA$
4. $B \rightarrow b$
5. $X \rightarrow a$

The production rule $S \rightarrow AA$ is not in GNF, so we substitute $A \rightarrow aC \mid aBAC \mid a \mid aBA$ in production rule $S \rightarrow AA$ as:

1. $S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$
2. $A \rightarrow aC \mid aBAC \mid a \mid aBA$
3. $C \rightarrow AAC$
4. $C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$
5. $B \rightarrow b$
6. $X \rightarrow a$

The production rule $C \rightarrow AAC$ is not in GNF, so we substitute $A \rightarrow aC \mid aBAC \mid a \mid aBA$ in production rule $C \rightarrow AAC$ as:

1. $S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$
2. $A \rightarrow aC \mid aBAC \mid a \mid aBA$
3. $C \rightarrow aCAC \mid aBACAC \mid aAC \mid aBAAC$
4. $C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$
5. $B \rightarrow b$
6. $X \rightarrow a$

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

Pushdown Automata(PDA)

- Pushdown automata is a way to implement a CFG in the same way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.
- Pushdown automata is simply an NFA augmented with an "external stack memory". The addition of stack is used to provide a last-in-first-out memory management capability to Pushdown automata. Pushdown automata can store an unbounded amount of information on the stack. It can access a limited amount of information on the stack. A PDA can push an element onto the top of the stack and pop off an element from the top of the stack. To read an element into the stack, the top elements must be popped off and are lost.
- A PDA is more powerful than FA. Any language which can be acceptable by FA can also be acceptable by PDA. PDA also accepts a class of language which even cannot be accepted by FA. Thus PDA is much more superior to FA.

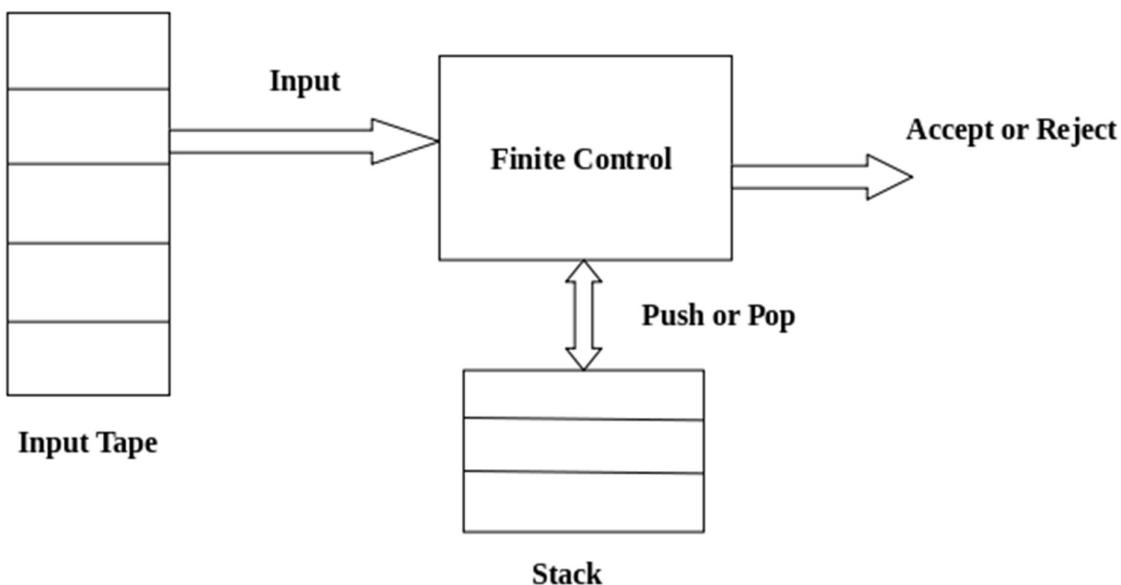


Fig: Pushdown Automata

PDA Components:

Input tape: The input tape is divided in many cells or symbols. The input head is read-only and may only move from left to right, one symbol at a time.

Finite control: The finite control has some pointer which points the current symbol which is to be read.

Stack: The stack is a structure in which we can push and remove the items from one end only. It has an infinite size. In PDA, the stack is used to store the items temporarily.

Formal definition of PDA:

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

Q: the finite set of states

Σ : the input set

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

q_0 : the initial state

Z : a start symbol which is in Γ .

F: a set of final states

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

Instantaneous Description (ID)

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

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

q describes the current state.

w describes the remaining input.

α describes the stack contents, top at the left.

Turnstile Notation:

\vdash sign describes the turnstile notation and represents one move.

\vdash^* sign describes a sequence of moves.

For example,

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

In the above example, while taking a transition from state p to q, the input symbol 'b' is consumed, and the top of the stack 'T' is represented by a new string α .

Example 1:

Design a PDA for accepting a language $\{a^n b^{2n} \mid n \geq 1\}$.

Solution: In this language, n number of a's should be followed by $2n$ number of b's. Hence, we will apply a very simple logic, and that is if we read single 'a', we will push two a's onto the stack. As soon as we read 'b' then for every single 'b' only one 'a' should get popped from the stack.

The ID can be constructed as follows:

1. $\delta(q_0, a, Z) = (q_0, aaZ)$
2. $\delta(q_0, a, a) = (q_0, aaa)$

Now when we read b, we will change the state from q_0 to q_1 and start popping corresponding 'a'. Hence,

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

Thus this process of popping 'b' will be repeated unless all the symbols are read. Note that popping action occurs in state q_1 only.

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

After reading all b's, all the corresponding a's should get popped. Hence when we read ϵ as input symbol then there should be nothing in the stack. Hence the move will be:

1. $\delta(q_1, \epsilon, Z) = (q_2, \epsilon)$

Where

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

We can summarize the ID as:

1. $\delta(q_0, a, Z) = (q_0, aaZ)$
2. $\delta(q_0, a, a) = (q_0, aaa)$
3. $\delta(q_0, b, a) = (q_1, \epsilon)$
4. $\delta(q_1, b, a) = (q_1, \epsilon)$
5. $\delta(q_1, \epsilon, Z) = (q_2, \epsilon)$

Now we will simulate this PDA for the input string "aaabbbbbbb".

1. $\delta(q_0, aaabbbbbbb, Z) \vdash \delta(q_0, aabbbbbbb, aaZ)$
2. $\vdash \delta(q_0, abbbbbbb, aaaaZ)$
3. $\vdash \delta(q_0, bbbbbbb, aaaaaaZ)$
4. $\vdash \delta(q_1, bbbbb, aaaaaZ)$
5. $\vdash \delta(q_1, bbbb, aaaaZ)$
6. $\vdash \delta(q_1, bbb, aaaZ)$
7. $\vdash \delta(q_1, bb, aaZ)$
8. $\vdash \delta(q_1, b, aZ)$
9. $\vdash \delta(q_1, \epsilon, Z)$
10. $\vdash \delta(q_2, \epsilon)$
11. **ACCEPT**

Example 2:

Design a PDA for accepting a language $\{0^n 1^m 0^n \mid m, n \geq 1\}$.

Solution: In this PDA, n number of 0's are followed by any number of 1's followed n number of 0's. Hence the logic for design of such PDA will be as follows:

Push all 0's onto the stack on encountering first 0's. Then if we read 1, just do nothing. Then read 0, and on each read of 0, pop one 0 from the stack.

For instance:

0011100 Δ	<table border="1"><tr><td>0</td></tr></table>	0	Pushing 0
0			
↑			
0011100 Δ	<table border="1"><tr><td>0</td></tr></table>	0	Pushing 0
0			
↑			
0011100 Δ	<table border="1"><tr><td>0</td></tr></table>	0	Skip 1
0			
↑			
0011100 Δ	<table border="1"><tr><td>0</td></tr></table>	0	Skip 1
0			
↑			
0011100 Δ	<table border="1"><tr><td>0</td></tr></table>	0	Skip 1
0			
↑			
0011100 Δ	<table border="1"><tr><td>0</td></tr></table>	0	Pop 0
0			
↑			
0011100 Δ	<table border="1"><tr><td>0</td></tr></table>	0	Pop 0
0			
↑			
0011100 Δ	<table border="1"><tr><td>0</td></tr></table>	0	Accept
0			
↑			

This scenario can be written in the ID form as:

1. $\delta(q_0, 0, Z) = \delta(q_0, 0Z)$
2. $\delta(q_0, 0, 0) = \delta(q_0, 00)$
3. $\delta(q_0, 1, 0) = \delta(q_1, 0)$
4. $\delta(q_0, 1, 0) = \delta(q_1, 0)$
5. $\delta(q_1, 0, 0) = \delta(q_1, \epsilon)$
6. $\delta(q_0, \epsilon, Z) = \delta(q_2, Z)$ (ACCEPT state)

Now we will simulate this PDA for the input string "0011100".

1. $\delta(q_0, 0011100, Z) \vdash \delta(q_0, 011100, 0Z)$
2. $\vdash \delta(q_0, 11100, 00Z)$
3. $\vdash \delta(q_0, 1100, 00Z)$
4. $\vdash \delta(q_1, 100, 00Z)$
5. $\vdash \delta(q_1, 00, 00Z)$
6. $\vdash \delta(q_1, 0, 0Z)$
7. $\vdash \delta(q_1, \epsilon, Z)$
8. $\vdash \delta(q_2, Z)$
9. ACCEPT

PDA Acceptance

A language can be accepted by Pushdown automata using two approaches:

1. Acceptance by Final State: The PDA is said to accept its input by the final state if it enters any final state in zero or more moves after reading the entire input.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ be a PDA. The language acceptable by the final state can be defined as:

1. $L(PDA) = \{w \mid (q_0, w, Z) \vdash^* (p, \epsilon, \epsilon), p \in F\}$

2. Acceptance by Empty Stack: On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ be a PDA. The language acceptable by empty stack can be defined as:

1. $N(PDA) = \{w \mid (q_0, w, Z) \vdash^* (p, \epsilon, \epsilon), p \in Q\}$

Equivalence of Acceptance by Final State and Empty Stack

- If $L = N(P1)$ for some PDA $P1$, then there is a PDA $P2$ such that $L = L(P2)$. That means the language accepted by empty stack PDA will also be accepted by final state PDA.
- If there is a language $L = L(P1)$ for some PDA $P1$ then there is a PDA $P2$ such that $L = N(P2)$. That means language accepted by final state PDA is also acceptable by empty stack PDA.

Example:

Construct a PDA that accepts the language L over $\{0, 1\}$ by empty stack which accepts all the string of 0's and 1's in which a number of 0's are twice of number of 1's.

Solution:

There are two parts for designing this PDA:

- If 1 comes before any 0's
- If 0 comes before any 1's.

We are going to design the first part i.e. 1 comes before 0's. The logic is that read single 1 and push two 1's onto the stack. Thereafter on reading two 0's, POP two 1's from the stack. The δ can be

1. $\delta(q0, 1, Z) = (q0, 11, Z)$ Here Z represents that stack is empty
2. $\delta(q0, 0, 1) = (q0, \epsilon)$

Now, consider the second part i.e. if 0 comes before 1's. The logic is that read first 0, push it onto the stack and change state from $q0$ to $q1$. [Note that state $q1$ indicates that first 0 is read and still second 0 has yet to read].

Being in $q1$, if 1 is encountered then POP 0. Being in $q1$, if 0 is read then simply read that second 0 and move ahead. The δ will be:

1. $\delta(q0, 0, Z) = (q1, 0Z)$
2. $\delta(q1, 0, 0) = (q1, 0)$
3. $\delta(q1, 0, Z) = (q0, \epsilon)$ (indicate that one 0 and one 1 is already read, so simply read the second 0)
4. $\delta(q1, 1, 0) = (q1, \epsilon)$

Now, summarize the complete PDA for given L is:

1. $\delta(q_0, 1, Z) = (q_0, 11Z)$
2. $\delta(q_0, 0, 1) = (q_1, \epsilon)$
3. $\delta(q_0, 0, Z) = (q_1, 0Z)$
4. $\delta(q_1, 0, 0) = (q_1, 0)$
5. $\delta(q_1, 0, Z) = (q_0, \epsilon)$
6. $\delta(q_0, \epsilon, Z) = (q_0, \epsilon)$ ACCEPT state

Non-deterministic Pushdown Automata

The non-deterministic pushdown automata is very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG which accepts deterministic PDA accepts non-deterministic PDAs as well. Similarly, there are some CFGs which can be accepted only by NPDA and not by DPDA. Thus NPDA is more powerful than DPDA.

Example:

Design PDA for Palindrome strips.

Solution:

Suppose the language consists of string $L = \{aba, aa, bb, bab, bbabb, aabaa, \dots\}$. The string can be odd palindrome or even palindrome. The logic for constructing PDA is that we will push a symbol onto the stack till half of the string then we will read each symbol and then perform the pop operation. We will compare to see whether the symbol which is popped is similar to the symbol which is read. Whether we reach to end of the input, we expect the stack to be empty.

This PDA is a non-deterministic PDA because finding the mid for the given string and reading the string from left and matching it with from right (reverse) direction leads to non-deterministic moves. Here is the ID.

1. $\delta(q1, a, Z) = (q1, aZ)$	
2. $\delta(q0, b, Z) = (q1, bZ)$	
3. $\delta(q0, a, a) = (q1, aa)$	
4. $\delta(q1, a, b) = (q1, ab)$	Pushing the symbols onto the stack
5. $\delta(q1, a, b) = (q1, ba)$	
6. $\delta(q1, b, b) = (q1, bb)$	
7. $\delta(q1, a, a) = (q2, \epsilon)$	
8. $\delta(q1, b, b) = (q2, \epsilon)$	
9. $\delta(q2, a, a) = (q2, \epsilon)$	
10. $\delta(q2, b, b) = (q2, \epsilon)$	Popping the symbols on reading the same kind of symbol
11. $\delta(q2, \epsilon, Z) = (q2, \epsilon)$	

Simulation of abaaba

1. $\delta(q1, abaaba, Z)$ **Apply rule 1**
2. $\vdash \delta(q1, baaba, aZ)$ **Apply rule 5**
3. $\vdash \delta(q1, aaba, baZ)$ **Apply rule 4**
4. $\vdash \delta(q1, aba, abaZ)$ **Apply rule 7**
5. $\vdash \delta(q2, ba, baZ)$ **Apply rule 8**
6. $\vdash \delta(q2, a, aZ)$ **Apply rule 7**
7. $\vdash \delta(q2, \epsilon, Z)$ **Apply rule 11**
8. $\vdash \delta(q2, \epsilon)$ **Accept**

CFG to PDA Conversion

The first symbol on R.H.S. production must be a terminal symbol. The following steps are used to obtain PDA from CFG is:

Step 1: Convert the given productions of CFG into GNF.

Step 2: The PDA will only have one state {q}.

Step 3: The initial symbol of CFG will be the initial symbol in the PDA.

Step 4: For non-terminal symbol, add the following rule:

$$1. \delta(q, \varepsilon, A) = (q, \alpha)$$

Where the production rule is $A \rightarrow \alpha$

Step 5: For each terminal symbols, add the following rule:

$$1. \delta(q, a, a) = (q, \varepsilon) \text{ for every terminal symbol}$$

Example 1:

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

$$1. S \rightarrow 0S1 | A$$

$$2. A \rightarrow 1AO | S | \varepsilon$$

Solution:

The CFG can be first simplified by eliminating unit productions:

$$1. S \rightarrow 0S1 | 1SO | \varepsilon$$

Now we will convert this CFG to GNF:

$$1. S \rightarrow 0SX | 1SY | \varepsilon$$

$$2. X \rightarrow 1$$

$$3. Y \rightarrow 0$$

The PDA can be:

$$R1: \delta(q, \varepsilon, S) = \{(q, 0SX) | (q, 1SY) | (q, \varepsilon)\}$$

$$R2: \delta(q, \varepsilon, X) = \{(q, 1)\}$$

$$R3: \delta(q, \varepsilon, Y) = \{(q, 0)\}$$

$$R4: \delta(q, 0, O) = \{(q, \varepsilon)\}$$

$$R5: \delta(q, 1, 1) = \{(q, \varepsilon)\}$$

Example 2:

Construct PDA for the given CFG, and test whether 0104 is acceptable by this PDA.

$$1. S \rightarrow 0BB$$

$$2. B \rightarrow 0S | 1S | 0$$

Solution:

The PDA can be given as:

$$1. A = \{(q), (0, 1), (S, B, 0, 1), \delta, q, S, ?\}$$

The production rule δ can be:

$$\text{R1: } \delta(q, \epsilon, S) = \{(q, 0BB)\}$$

$$\text{R2: } \delta(q, \epsilon, B) = \{(q, 0S) \mid (q, 1S) \mid (q, 0)\}$$

$$\text{R3: } \delta(q, 0, 0) = \{(q, \epsilon)\}$$

$$\text{R4: } \delta(q, 1, 1) = \{(q, \epsilon)\}$$

Testing 010^4 i.e. 010000 against PDA:

1. $\delta(q, 010000, S) \vdash \delta(q, 010000, 0BB)$
2. $\vdash \delta(q, 10000, BB) \quad \text{R1}$
3. $\vdash \delta(q, 10000, 1SB) \quad \text{R3}$
4. $\vdash \delta(q, 0000, SB) \quad \text{R2}$
5. $\vdash \delta(q, 0000, 0BBB) \quad \text{R1}$
6. $\vdash \delta(q, 000, BBB) \quad \text{R3}$
7. $\vdash \delta(q, 000, 0BB) \quad \text{R2}$
8. $\vdash \delta(q, 00, BB) \quad \text{R3}$
9. $\vdash \delta(q, 00, 0B) \quad \text{R2}$
10. $\vdash \delta(q, 0, B) \quad \text{R3}$
11. $\vdash \delta(q, 0, 0) \quad \text{R2}$
12. $\vdash \delta(q, \epsilon) \quad \text{R3}$
13. **ACCEPT**

Thus 010^4 is accepted by the PDA.

Example 3:

Draw a PDA for the CFG given below:

1. $S \rightarrow aSb$
2. $S \rightarrow a \mid b \mid \epsilon$

Solution:

The PDA can be given as:

1. $P = \{(q), (a, b), (S, a, b, z0), \delta, q, z0, q\}$

The mapping function δ will be:

$$\text{R1: } \delta(q, \epsilon, S) = \{(q, aSb)\}$$

$$\text{R2: } \delta(q, \epsilon, S) = \{(q, a) \mid (q, b) \mid (q, \epsilon)\}$$

$$\text{R3: } \delta(q, a, a) = \{(q, \epsilon)\}$$

$$\text{R4: } \delta(q, b, b) = \{(q, \epsilon)\}$$

$$\text{R5: } \delta(q, \epsilon, z0) = \{(q, \epsilon)\}$$

Simulation: Consider the string aaabb

1. $\delta(q, \epsilon aaabb, S) \vdash \delta(q, aaabb, aSb)$ R3
2. $\vdash \delta(q, \epsilon aabb, Sb)$ R1
3. $\vdash \delta(q, aabb, aSbb)$ R3
4. $\vdash \delta(q, \epsilon abb, Sbb)$ R2
5. $\vdash \delta(q, abb, abb)$ R3
6. $\vdash \delta(q, bb, bb)$ R4
7. $\vdash \delta(q, b, b)$ R4
8. $\vdash \delta(q, \epsilon, z0)$ R5
9. $\vdash \delta(q, \epsilon)$
10. ACCEPT

Turing Machine

Turing machine was invented in 1936 by **Alan Turing**. It is an accepting device which accepts Recursive Enumerable Language generated by type 0 grammar.

There are various features of the Turing machine:

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

Formal definition of Turing machine

A Turing machine can be defined as a collection of 7 components:

Q: the finite set of states
 Σ : the finite set of input symbols
T: the tape symbol
 q_0 : the initial state
F: a set of final states
B: a blank symbol used as an end marker for input
 δ : a transition or mapping function.

The mapping function shows the mapping from states of finite automata and input symbol on the tape to the next states, external symbols and the direction for moving the tape head. This is known as a triple or a program for turing machine.

$$1. (q_0, a) \rightarrow (q_1, A, R)$$

That means in q_0 state, if we read symbol 'a' then it will go to state q_1 , replaced a by X and move ahead right(R stands for right).

Example:

Construct TM for the language $L = \{0^n 1^n\}$ where $n \geq 1$.

Solution:

We have already solved this problem by PDA. In PDA, we have a stack to remember the previous symbol. The main advantage of the Turing machine is we have a tape head which can be moved forward or backward, and the input tape can be scanned.

The simple logic which we will apply is read out each '0' mark it by A and then move ahead along with the input tape and find out 1 convert it to B. Now, repeat this process for all a's and b's.

Now we will see how this turing machine work for 0011.

The simulation for 0011 can be shown as below:

0	0	1	1	Δ	-----
---	---	---	---	----------	-------

Now, we will see how this turing machine will works for 0011. Initially, state is q_0 and head points to 0 as:

0	0	1	1	Δ
↑				

The move will be $\delta(q_0, 0) = \delta(q_1, A, R)$ which means it will go to state q_1 , replaced 0 by A and head will move to the right as:

A	0	1	1	Δ
---	---	---	---	----------

↑

The move will be $\delta(q_1, 0) = \delta(q_1, 0, R)$ which means it will not change any symbol, remain in the same state and move to the right as:

A	0	1	1	Δ
---	---	---	---	----------

↑

The move will be $\delta(q_1, 1) = \delta(q_2, B, L)$ which means it will go to state q_2 , replaced 1 by B and head will move to left as:

A	0	B	1	Δ
---	---	---	---	----------

↑

Now move will be $\delta(q_2, 0) = \delta(q_2, 0, L)$ which means it will not change any symbol, remain in the same state and move to left as:

A	0	B	1	Δ
---	---	---	---	----------

↑

The move will be $\delta(q_2, A) = \delta(q_0, A, R)$, it means will go to state q_0 , replaced A by A and head will move to the right as:

A	0	B	1	Δ
---	---	---	---	----------

↑

The move will be $\delta(q_0, 0) = \delta(q_1, A, R)$ which means it will go to state q_1 , replaced 0 by A, and head will move to right as:

A	A	B ↑	1	Δ
---	---	--------	---	---

The move will be $\delta(q_1, B) = \delta(q_1, B, R)$ which means it will not change any symbol, remain in the same state and move to right as:

A	A	B	1 ↑	Δ
---	---	---	--------	---

The move will be $\delta(q_1, 1) = \delta(q_2, B, L)$ which means it will go to state q_2 , replaced 1 by B and head will move to left as:

A	A	B ↑	B	Δ
---	---	--------	---	---

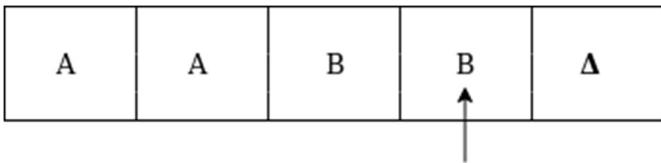
The move $\delta(q_2, B) = (q_2, B, L)$ which means it will not change any symbol, remain in the same state and move to left as:

A	A ↑	B	B	Δ
---	--------	---	---	---

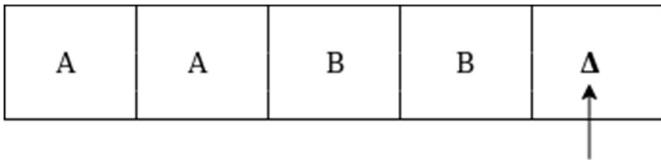
Now immediately before B is A that means all the 0's are marked by A. So we will move right to ensure that no 1 is present. The move will be $\delta(q_2, A) = (q_0, A, R)$ which means it will go to state q_0 , will not change any symbol, and move to right as:

A	A	B ↑	B	Δ
---	---	--------	---	---

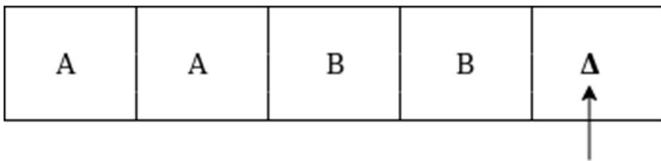
The move $\delta(q_0, B) = (q_3, B, R)$ which means it will go to state q_3 , will not change any symbol, and move to right as:



The move $\delta(q_3, B) = (q_3, B, R)$ which means it will not change any symbol, remain in the same state and move to right as:



The move $\delta(q_3, \Delta) = (q_4, \Delta, R)$ which means it will go to state q_4 which is the HALT state and HALT state is always an accept state for any TM.

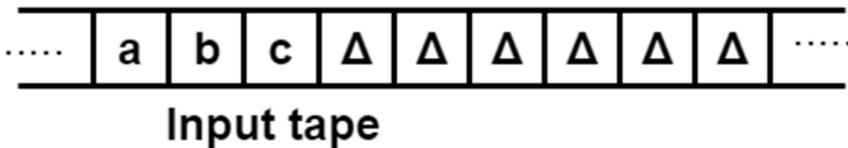


The same TM can be represented by Transition Diagram:

Basic Model of Turing machine

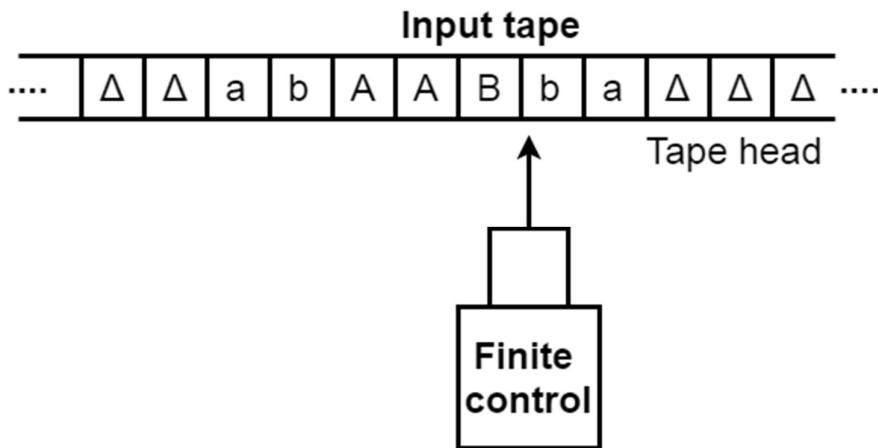
The turning machine can be modelled with the help of the following representation.

1. The input tape is having an infinite number of cells, each cell containing one input symbol and thus the input string can be placed on tape. The empty tape is filled by blank characters.



2. The finite control and the tape head which is responsible for reading the current input symbol. The tape head can move to left to right.
3. A finite set of states through which machine has to undergo.

4. Finite set of symbols called external symbols which are used in building the logic of turing machine.



Language accepted by Turing machine

The turing machine accepts all the language even though they are recursively enumerable. Recursive means repeating the same set of rules for any number of times and enumerable means a list of elements. The TM also accepts the computable functions, such as addition, multiplication, subtraction, division, power function, and many more.

Example:

Construct a turing machine which accepts the language of aba over $\Sigma = \{a, b\}$.

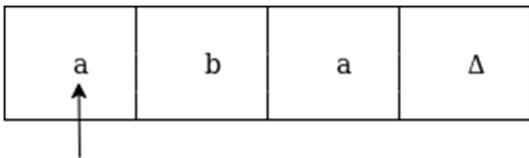
Solution:

We will assume that on input tape the string 'aba' is placed like this:

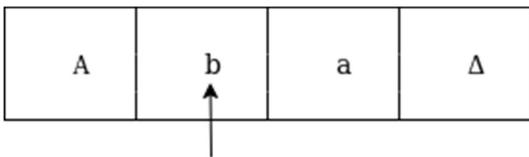
a	b	a	Δ	-----
---	---	---	---	-------

The tape head will read out the sequence up to the Δ characters. If the tape head is readout 'aba' string then TM will halt after reading Δ .

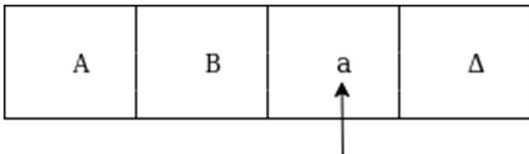
Now, we will see how this turing machine will work for aba. Initially, state is q_0 and head points to a as:



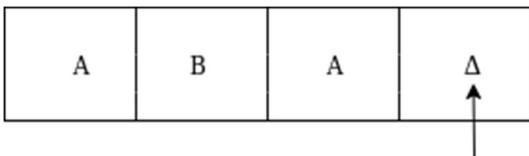
The move will be $\delta(q_0, a) = \delta(q_1, A, R)$ which means it will go to state q_1 , replaced a by A and head will move to right as:



The move will be $\delta(q_1, b) = \delta(q_2, B, R)$ which means it will go to state q_2 , replaced b by B and head will move to right as:



The move will be $\delta(q_2, a) = \delta(q_3, A, R)$ which means it will go to state q_3 , replaced a by A and head will move to right as:



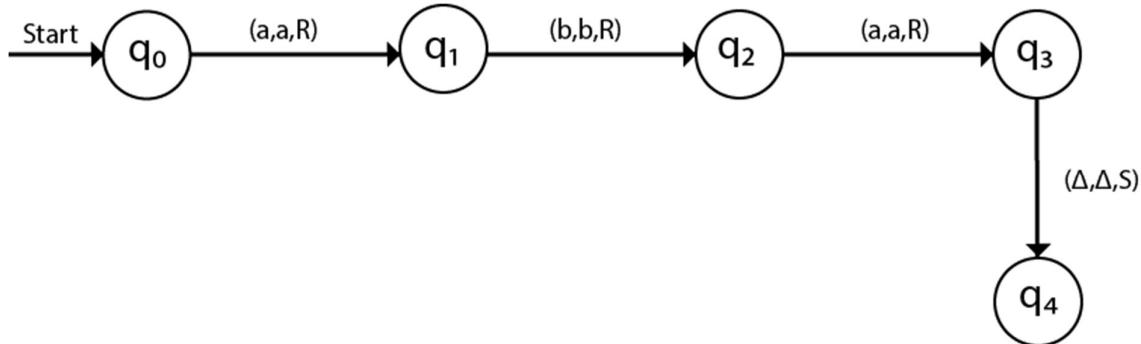
The move $\delta(q_3, \Delta) = (q_4, \Delta, S)$ which means it will go to state q_4 which is the HALT state and HALT state is always an accept state for any TM.

The same TM can be represented by Transition Table:

States	a	b	Δ
q_0	(q_1, A, R)	-	-
q_1	-	(q_2, B, R)	-
q_2	(q_3, A, R)	-	-

q3	-	-	(q4, Δ, S)
q4	-	-	-

The same TM can be represented by Transition Diagram:



Examples of TM

Example 1:

Construct a TM for the language $L = \{0^n 1^n 2^n \mid n \geq 1\}$

Solution:

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

The simulation for 001122 can be shown as below:

0	0	1	1	2	2	X	-----
---	---	---	---	---	---	---	-------

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

0	0	1	1	2	2	X
↑						

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

A	0	1	1	2	2	X
---	---	---	---	---	---	---

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

A	0	1	1	2	2	X
---	---	---	---	---	---	---

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

A	0	B	1	2	2	X
---	---	---	---	---	---	---

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

A	0	B	1	2	2	X
---	---	---	---	---	---	---

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

A	0	B	1	C	2	X
---	---	---	---	---	---	---

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

A	0	B	1	C	2	X
---	---	---	---	---	---	---

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

A	A	B	1	C	2	X
		↑				

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

A	A	B	1	C	2	X
		↑				

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

A	A	B	B	C	2	X
		↑				

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

A	A	B	B	C	2	X
		↑				

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

A	A	B	B	C	C	X
	↑					

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

A	A	B	B	C	C	X
		↑				

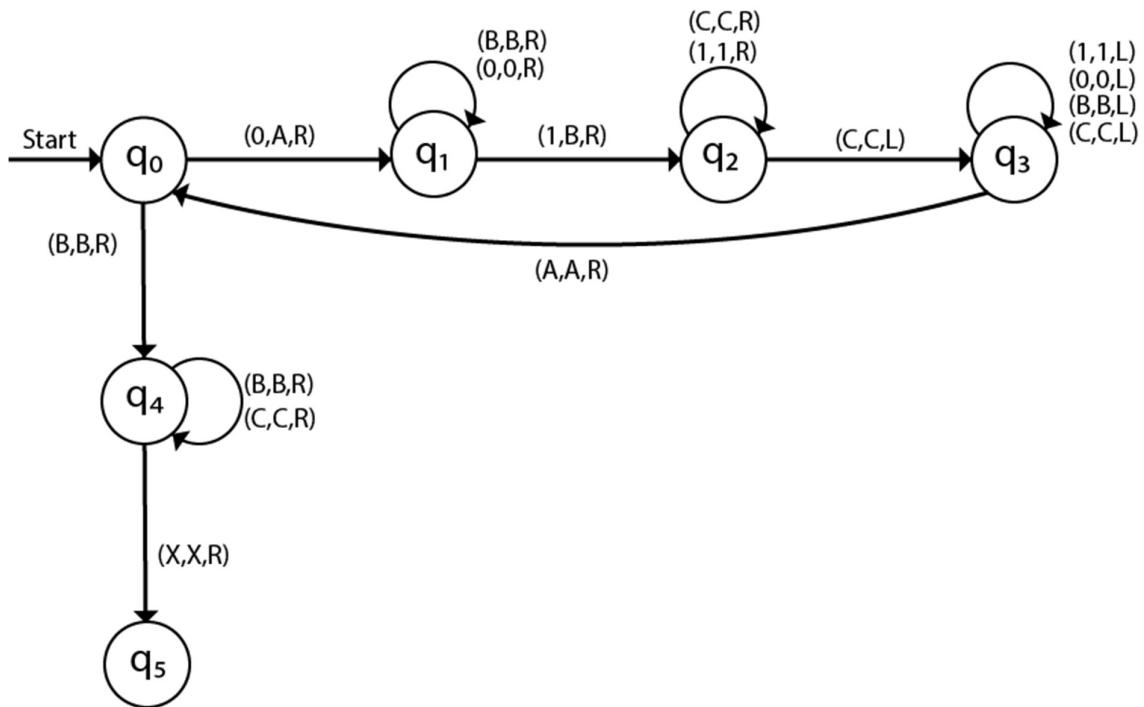
The move will be $(q_4, B) = \delta(q_4, B, R)$ and $(q_4, C) = \delta(q_4, C, R)$ which means it will not change any symbol, remain in the same state and move to right as:

A	A	B	B	C	C	X
		↑				

The move $\delta(q_4, X) = (q_5, X, R)$ which means it will go to state q_5 which is the HALT state and HALT state is always an accept state for any TM.

A	A	B	B	C	C	X
		↑				

The same TM can be represented by Transition Diagram:



Example 2:

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

Solution:

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

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

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

a	b	a	b	b	a	b	a	Δ	-----
---	---	---	---	---	---	---	---	----------	-------

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

a	b	a	b	b	a	b	a	Δ
\uparrow								

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

*	b	a	b	b	a	b	a	Δ
\uparrow								

We will move right up to Δ as:

*	b	a	b	b	a	b	a	Δ
\uparrow								

We will move left and check if it is a:

*	b	a	b	b	a	b	a	Δ
\uparrow								

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

*	b	a	b	b	a	b	Δ
---	---	---	---	---	---	---	----------

Now move to left up to * as:

*	b	a	b	b	a	b	Δ
---	---	---	---	---	---	---	----------

Move right and read it

*	b	a	b	b	a	b	Δ
---	---	---	---	---	---	---	----------

Now convert b by * and move right as:

*	*	a	b	b	a	b	Δ
---	---	---	---	---	---	---	----------

Move right up to Δ in search of b as:

*	*	a	b	b	a	b	Δ
---	---	---	---	---	---	---	----------

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

*	*	a	b	b	a	Δ
---	---	---	---	---	---	----------

Now move left until * as:

*	*	a	b	b	a	Δ
---	---	---	---	---	---	----------

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

*	*	*	b	b	a	Δ
---	---	---	---	---	---	---



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

*	*	*	b	b	a	Δ
---	---	---	---	---	---	---



It is 'a' so replace it by Δ as:

*	*	*	b	b	Δ
---	---	---	---	---	---



Now move left until *

*	*	*	b	b	Δ
---	---	---	---	---	---



Now move right as:

*	*	*	b	b	Δ
---	---	---	---	---	---



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

*	*	*	*	b	Δ
---	---	---	---	---	---



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

*	*	*	*	Δ
↑				

Move left till *

*	*	*	*	Δ
↑				

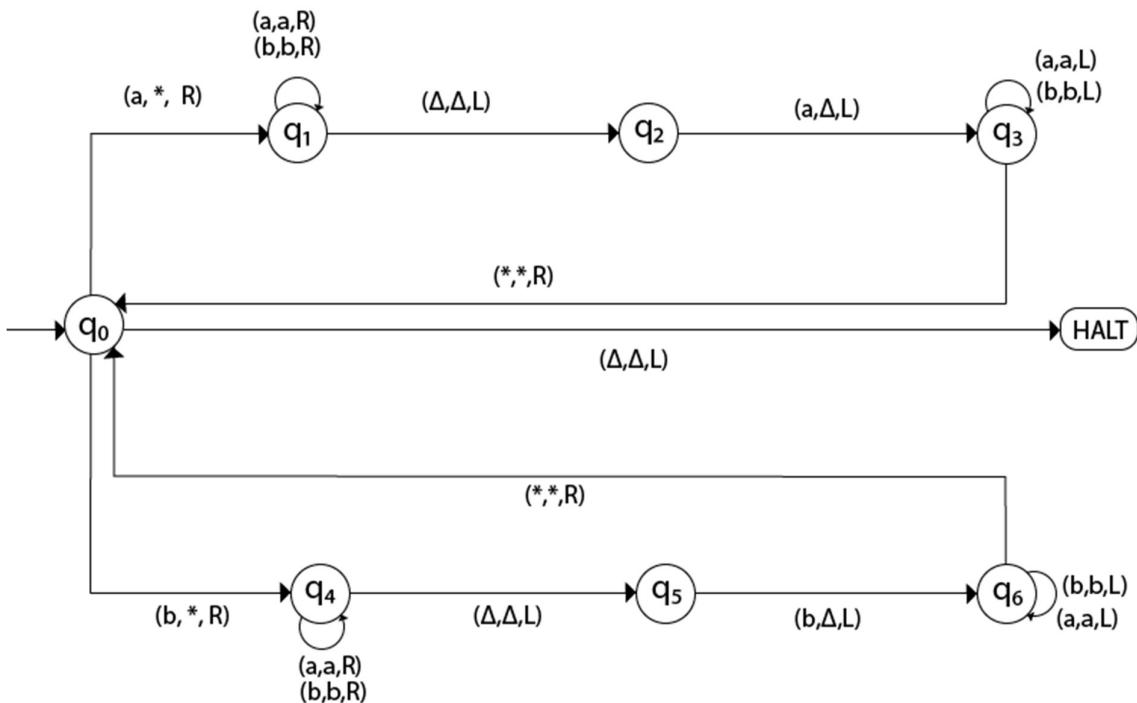
Move right and check whether it is Δ

*	*	*	*	Δ
↑				

Go to HALT state

*	*	*	*	Δ
↑				

The same TM can be represented by Transition Diagram:



Example 3:

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

Solution:

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

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

Suppose the string is 00100Δ. The simulation for 00100Δ can be shown as follows:

Now, we will see how this Turing machine will work for 00100Δ. Initially, state is q0 and head points to 0 as:

0	0	1	0	0	Δ
↑					

Now replace 0 by * and move right as:

*	0	1	0	0	Δ
↑	↑				

Move right up to Δ as:

*	0	1	0	0	Δ
				↑	

Move left and replace 0 by Δ and move left:

*	0	1	0	Δ
			↑	

Now move left up to * as:

*	0	1	0	Δ
---	---	---	---	----------

Move right, convert 0 by * and then move right as:

*	*	1	0	Δ
---	---	---	---	----------

Moved right up to Δ

*	*	1	0	Δ
---	---	---	---	----------

Move left and replace 0 by Δ as:

*	*	1	Δ	Δ
---	---	---	----------	----------

Move left till * as:

*	*	1	Δ	Δ
---	---	---	----------	----------

Move right and convert 1 to * as:

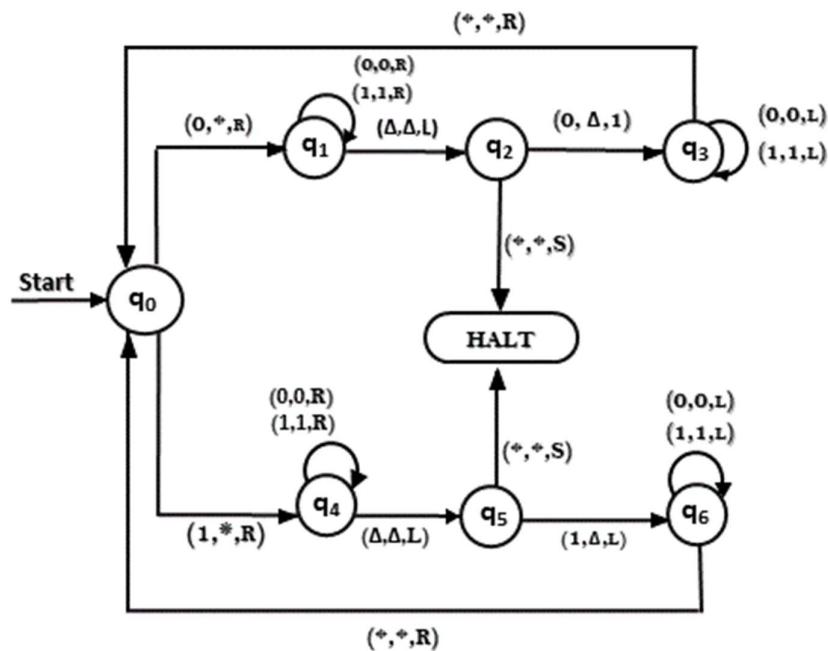
*	*	*	Δ	Δ
---	---	---	----------	----------

Move left

*	*	*	Δ	Δ
---	---	---	----------	----------

Since it is *, goto HALT state.

The same TM can be represented by Transition Diagram:



Example 4:

Construct TM for the addition function for the unary number system.

Solution:

The unary number is made up of only one character, i.e. The number 5 can be written in unary number system as 11111. In this TM, we are going to perform the addition of two unary numbers.

For example

$$\begin{array}{ccc} 2 & + & 3 \\ \text{i.e. } 11 & + & 111 = 11111 \end{array}$$

If you observe this process of addition, you will find the resemblance with string concatenation function.

In this case, we simply replace + by 1 and move ahead right for searching end of the string we will convert last 1 to Δ .

Input: 3+2

The simulation for 111+11 Δ can be shown as below:

1	1	1	+	1	1	Δ
---	---	---	---	---	---	---

Move right up to + sign as:

1	1	1	+	1	1	Δ
---	---	---	---	---	---	---

Convert + to 1 and move right as:

1	1	1	1	1	1	Δ
---	---	---	---	---	---	---

Now, move right

1	1	1	1	1	1	Δ
---	---	---	---	---	---	---

Again move right

1	1	1	1	1	1	Δ
---	---	---	---	---	---	---

Now Δ has encountered, so just move left as:

1	1	1	1	1	1	Δ
---	---	---	---	---	---	---

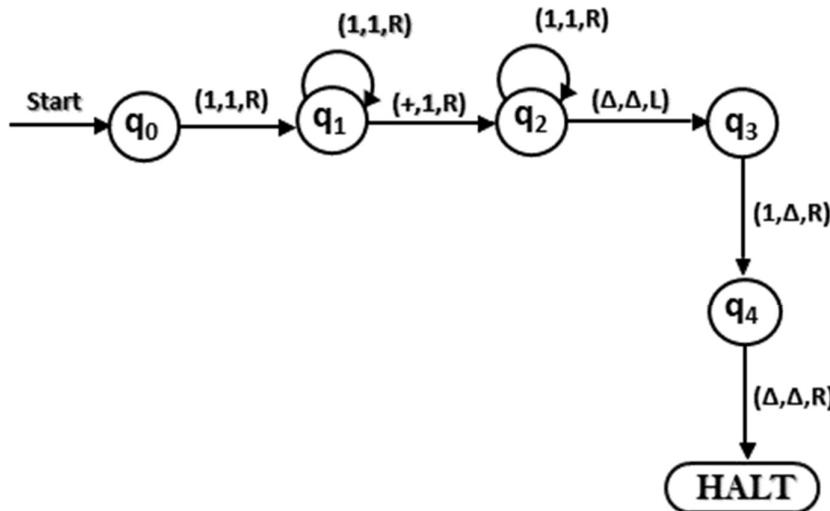
Convert 1 to Δ

1	1	1	1	1	Δ	Δ
---	---	---	---	---	---	---

Thus the tape now consists of the addition of two unary numbers.

The TM will look like as follows:

Here, we are implementing the function of $f(a + b) = c$. We assume a and b both are non zero elements.



Example 5:

Construct a TM for subtraction of two unary numbers $f(a-b) = c$ where a is always greater than b.

Solution: Here we have certain assumptions as the first number is greater than the second one. Let us assume that $a = 3$, $b = 2$, so the input tape will be:

1	1	1	-	1	1	Δ
---	---	---	---	---	---	----------

We will move right to - symbol as perform reduction of a number of 1's from the first number. Let us look at the simulation for understanding the logic:

1	1	1	-	1	1	Δ

Move right up to - as:

1	1	1	-	1	1	Δ
---	---	---	---	---	---	----------

Move right and convert 1 to * as:

1	1	1	-	*	1	Δ
---	---	---	---	---	---	----------

Now move left

1	1	1	-	*	1	Δ
---	---	---	---	---	---	----------

Again move left

1	1	1	-	*	1	Δ
---	---	---	---	---	---	----------

Convert 1 to * and move right-hand

1	1	*	-	*	1	Δ
---	---	---	---	---	---	----------

Now move right till 1

1	1	*	-	*	1	Δ
---	---	---	---	---	---	----------

Convert 1 to * and move left

1	1	*	-	*	*	Δ
---	---	---	---	---	---	----------

Convert 1 to * and move

1	1	*	-	*	*	Δ
1	*	*	-	*	*	Δ

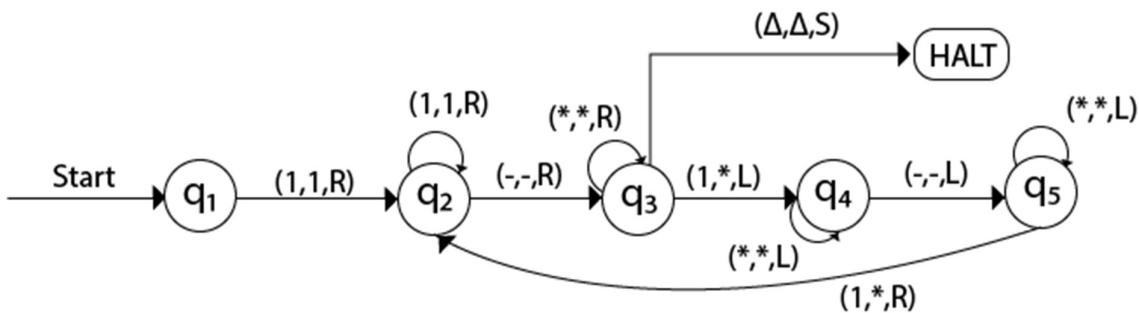
Move right till Δ as:

1	*	*	-	*	*	△
---	---	---	---	---	---	---

Now we are in the HALT state.

Thus we get 1 on the input tape as the answer for $f(3-2)$.

The Turing machine will look like this:



Introduction to Undecidability

In the theory of computation, we often come across such problems that are answered either 'yes' or 'no'. The class of problems which can be answered as 'yes' are called solvable or decidable. Otherwise, the class of problems is said to be unsolvable or undecidable.

Undecidability of Universal Languages:

The universal language L_u is a recursively enumerable language and we have to prove that it is undecidable (non-recursive).

Theorem: L_u is RE but not recursive.

Proof:

Consider that language L_u is recursively enumerable language. We will assume that L_u is recursive. Then the complement of L_u that is L'_{u} is also recursive. However, if we have a TM M to accept L'_{u} then we can construct a TM L_d . But L_d the diagonalization language is not RE. Thus our assumption that L_u is recursive is wrong (not RE means not recursive). Hence we can say that L_u is RE but not recursive. The construction of M for L_d is as shown in the following diagram:

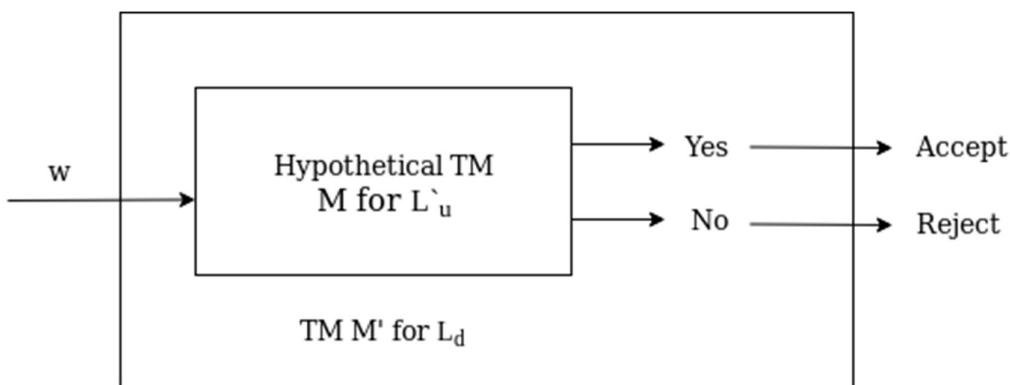


Fig: Construction of M' for L_d

Undecidable Problem about Turing Machine

In this section, we will discuss all the undecidable problems regarding turing machine. The reduction is used to prove whether given language is desirable or not. In this section, we will understand the concept of reduction first and then we will see an important theorem in this regard.

Reduction

Reduction is a technique in which if a problem P1 is reduced to a problem P2 then any solution of P2 solves P1. In general, if we have an algorithm to convert an instance of a problem P1 to an instance of a problem P2 that have the same answer then it is called as P1 reduced P2. Hence if P1 is not recursive then P2 is also not recursive. Similarly, if P1 is not recursively enumerable then P2 also is not recursively enumerable.

Theorem: if P1 is reduced to P2 then

1. If P1 is undecidable, then P2 is also undecidable.
2. If P1 is non-RE, then P2 is also non-RE.

Proof:

1. Consider an instance w of P1. Then construct an algorithm such that the algorithm takes instance w as input and converts it into another instance x of P2. Then apply that algorithm to check whether x is in P2. If the algorithm answer 'yes' then that means x is in P2, similarly we can also say that w is in P1. Since we have obtained P2 after reduction of P1. Similarly if algorithm answer 'no' then x is not in P2, that also means w is not in P1. This proves that if P1 is undecidable, then P1 is also undecidable.
2. We assume that P1 is non-RE but P2 is RE. Now construct an algorithm to reduce P1 to P2, but by this algorithm, P2 will be recognized. That means there will be a Turing machine that says 'yes' if the input is P2 but may or may not halt for the input which is not in P2. As we know that one can convert an instance of w in P1 to an instance x in P2. Then apply a TM to check whether x is in P2. If x is accepted that also means w is accepted. This procedure describes a TM whose language is P1 if w is in P1 then x is also in P2 and if w is not in P1 then x is also not in P2. This proves that if P1 is non-RE then P2 is also non-RE.

Empty and non empty languages:

There are two types of languages empty and non empty language. Let L_e denotes an empty language, and L_{ne} denotes non empty language. Let w be a binary string, and M_i be a TM. If $L(M_i) = \emptyset$ then M_i does not accept input then w is in L_e . Similarly, if $L(M_i)$ is not the empty language, then w is in L_{ne} . Thus we can say that

$$L_e = \{M \mid L(M) = \emptyset\}$$

$$L_{ne} = \{M \mid L(M) \neq \emptyset\}$$

Both L_e and L_{ne} are the complement of one another.

Post Correspondence Problem

In this section, we will discuss the undecidability of string and not of Turing machines. The undecidability of the string is determined with the help of Post's Correspondence Problem (PCP). Let us define the PCP.

"The Post's correspondence problem consists of two lists of string that are of equal length over the input. The two lists are $A = w_1, w_2, w_3, \dots, w_n$ and $B = x_1, x_2, x_3, \dots, x_n$ then there exists a non empty set of integers $i_1, i_2, i_3, \dots, i_n$ such that, $w_1, w_2, w_3, \dots, w_n = x_1, x_2, x_3, \dots, x_n$ "

To solve the post correspondence problem we try all the combinations of $i_1, i_2, i_3, \dots, i_n$ to find the $w_1 = x_1$ then we say that PCP has a solution.

Example 1:

Consider the correspondence system as given below

$A = (b, bab^3, ba)$ and $B = (b^3, ba, a)$. The input set is $\Sigma = \{0, 1\}$. Find the solution.

Solution:

A solution is 2, 1, 1, 3. That means $w_2w_1w_1w_3 = x_2x_1x_1x_3$

The constructed string from both lists is bab^3b^3a .



Example 2:

Does PCP with two lists $x = (b, a, aba, bb)$ and $y = (ba, ba, ab, b)$ have a solution?

Solution: Now we have to find out such a sequence that strings formed by x and y are identical. Such a sequence is 1, 2, 1, 3, 3, 4. Hence from x and y list

$$\begin{array}{ccccccccc} 1 & 2 & 1 & 3 & 3 & 4 & & \\ b & a & b & aba & aba & bb & = & ba & ba & ba & ab & ab & b \end{array}$$

Example 3:

Obtain the solution for the following system of posts correspondence problem. A = {100, 0, 1}, B = {1, 100, 00}

Solution: Consider the sequence 1, 3, 2. The string obtained from A = babababb. The string obtained from B = bababbbb. These two strings are not equal. Thus if we try various combination from both the sets to find the unique sequence, we could not get such a sequence. Hence there is no solution for this system.

Example 4:

Obtain the solution for the following system of posts correspondence problem, X = {100, 0, 1}, Y = {1, 100, 00}.

Solution: The solution is 1, 3, 1, 1, 3, 2, 2. The string is

$$\begin{aligned} X_1X_3X_1X_1X_3X_2X_2 &= 100 + 1 + 100 + 100 + 1 + 0 + 0 = 1001100100100 \\ Y_1Y_3Y_1Y_1Y_3Y_2Y_2 &= 1 + 00 + 1 + 1 + 00 + 100 + 100 = 1001100100100 \end{aligned}$$

Chomsky Hierarchy

Chomsky Hierarchy represents the class of languages that are accepted by the different machine. The category of language in Chomsky's Hierarchy is as given below:

1. Type 0 known as Unrestricted Grammar.
2. Type 1 known as Context Sensitive Grammar.
3. Type 2 known as Context Free Grammar.
4. Type 3 Regular Grammar.

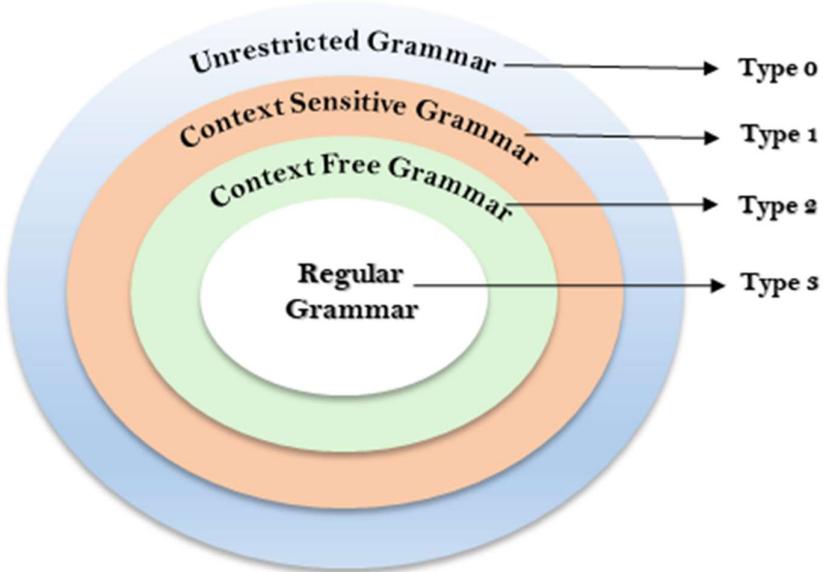


Fig: Chomsky Hierarchy

This is a hierarchy. Therefore every language of type 3 is also of type 2, 1 and 0. Similarly, every language of type 2 is also of type 1 and type 0, etc.

Type 0 Grammar:

Type 0 grammar is known as Unrestricted grammar. There is no restriction on the grammar rules of these types of languages. These languages can be efficiently modeled by Turing machines.

For example:

1. $bAa \rightarrow aa$
2. $S \rightarrow s$

Type 1 Grammar:

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

- o The context sensitive grammar may have more than one symbol on the left hand side of their production rules.
- o The number of symbols on the left-hand side must not exceed the number of symbols on the right-hand side.
- o The rule of the form $A \rightarrow \epsilon$ is not allowed unless A is a start symbol. It does not occur on the right-hand side of any rule.
- o The Type 1 grammar should be Type 0. In type 1, Production is in the form of $V \rightarrow T$

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

For example:

1. $S \rightarrow AT$
2. $T \rightarrow xy$
3. $A \rightarrow a$

Type 2 Grammar:

Type 2 Grammar is known as Context Free Grammar. Context free languages are the languages which can be represented by the context free grammar (CFG). Type 2 should be type 1. The production rule is of the form

1. $A \rightarrow \alpha$

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

For example:

1. $A \rightarrow aBb$
2. $A \rightarrow b$
3. $B \rightarrow a$

Type 3 Grammar:

Type 3 Grammar is known as Regular Grammar. Regular languages are those languages which can be described using regular expressions. These languages can be modeled by NFA or DFA.

Type 3 is most restricted form of grammar. The Type 3 grammar should be Type 2 and Type 1. Type 3 should be in the form of

1. $V \rightarrow T^*V / T^*$

For example:

1. $A \rightarrow xy$