AUTOMATA THEOARY AND COMPILER DESIGN

UNIT - I

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

Introduction to Automata

Automata theory is a branch of the theory of computation.
It deals with the study of abstract machines and their computations.
An abstract machine is called the automata. It includes the design and analysis of automata, which are mathematical models that can perform computations on strings of symbols according to a set of rules.
It includes Regular languages and finite automata, Context free Grammar and Context-free language, turning machines, etc.
The term "Automata" is derived from the Greek word which means "self-acting
An automaton with a finite number of states is called a Finite Automaton(FA) or FiniteState Machine (FSM).

Structural Representation of Finite Automata

Finite automata are abstract machines used to recognize patterns in input sequences, forming the basis for understanding regular languages in computer science. They consist of states, transitions, and input symbols, processing each symbol step-by-step. If the machine ends in an accepting state after processing the input, it is accepted; otherwise, it is rejected. Finite automata come in deterministic (DFA) and non-deterministic (NFA), both of which can recognize the same set of regular languages. They are widely used in text processing, compilers, and network protocols.

AUTOMATA THEOARY AND COMPILER DESIGN

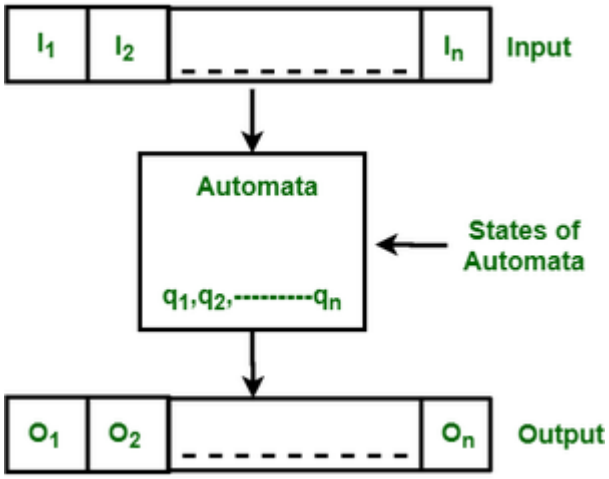


Figure: Features of Finite Automata

Features of Finite Automata
Input: Set of symbols or characters provided to the machine.
Output: Accept or reject based on the input pattern.
States of Automata: The conditions or configurations of the machine.
State Relation: The transitions between states.
Output Relation: Based on the final state, the output decision is made.
Formal Definition of Finite Automata
A finite automaton can be defined as a tuple:
{ Q, Σ, q, F, δ }, where:
Q: Finite set of states
Σ: Set of input symbols
q: Initial state
F: Set of final states
δ: Transition function

Automata and Complexity

Finite Automata (FA)

Deterministic Finite Automata (DFA): Has exactly one transition for each input symbol in every state.
Nondeterministic Finite Automata (NFA): Allows multiple transitions, including epsilon (ε) transitions.
Recognizes **regular languages**.

Pushdown Automata (PDA):

A finite automaton with an additional **stack** for memory.
Recognizes **context-free languages**.
Used in parsing and syntax analysis.

AUTOMATA THEOARY AND COMPILER DESIGN

Turing Machines:

The most powerful automaton with an **infinite tape** for memory.
Can simulate any computation.
Recognizes **recursively enumerable languages**.
Variants include Multi-Tape Turing Machines and Non-Deterministic Turing Machines.

Mealy and Moore Machines:

Mealy Machine: Outputs depend on both the current state and the input.
Moore Machine: Outputs depend only on the current state.

Complexity:

Time Complexity Classes:

P: Problems solvable in polynomial time. (e.g., using a DFA or basic algorithms).

NP: Problems whose solutions can be verified in polynomial time. (e.g., solvable by non deterministic automata).

Central Concepts of Automata Theory

Alphabets (Σ):

An **alphabet** is a finite set of symbols used to construct strings.

Example: Binary alphabet $\Sigma=\{0,1\}$, English alphabet $\Sigma=\{a,b,c,\ldots,z\}$.

Alphabets are the building blocks for creating languages.

String

A string is a **finite** sequence of symbols from some alphabet. A string is generally denoted as **w** and the length of a string is denoted as **|w|**.
Empty string is the string with zero occurrence of symbols, represented as ε.

Number of Strings (of length 2)
that can be generated over the alphabet {a, b}:

- -
a a
a b

b a
b b

Length of String |w| = 2
Number of Strings = 4

Closure Representation

L+: It is a **Positive Closure** that represents a set of all strings except Null or ε-strings.

L*: It is "**Kleene Closure**", that represents the occurrence of certain alphabets for given language alphabets from zero to the infinite number of times. In which ε-string is also included.

Example:

(a) Regular expression for language accepting all combination of g's over Σ={g}:

R = g*

R={ε,g,gg,ggg,gggg,ggggg,...}

(b) Regular Expression for language accepting all combination of g's over Σ={g} :

R = g+

R={g,gg,ggg,gggg,ggggg,gggggg,...}

Language

A language is a set of strings, chosen from some Σ* or we can say- **'**A language is a subset of Σ* **'**. A language that can be formed over ' Σ ' can be **Finite** or **Infinite**.

Example of Finite Language:

L1 = { set of string of 2 }
L1 = { xy, yx, xx, yy }

Example of Infinite Language:

L1 = { set of all strings starts with 'b' }
L1 = { babb, baa, ba, bbb, baab, ... }

Finite Automaton can be classified into two types –

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

AUTOMATA THEOARY AND COMPILER DESIGN

NON DETERMINISTIC FINITE AUTOMATA:

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

An NDFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Formal Definition of an NDFA An NDFA can be represented by a 5-tuple (Q, ∑, δ, q0, F)
Where

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

δ: Q × ∑ → 2 Q

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

- q0 is the initial state from where any input is processed (q0 ∈ Q).
- F is a set of final state/states of Q (F ⊆ Q).

Graphical Representation of an NDFA: (same as DFA)

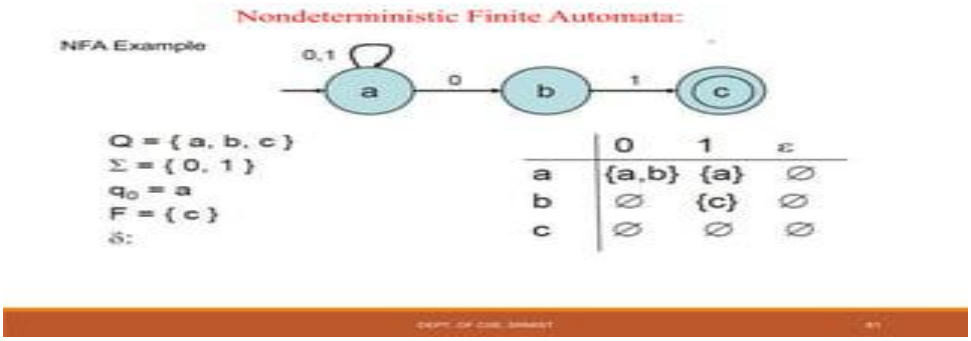
- An NFA is a five-tuple: M = (Q, Σ, δ, q0, F)

Q	A finite set of states
Σ	A finite input alphabet
q0	The initial/starting state, q0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, which is a total function from Q x Σ to 2^Q

δ: (Q x Σ) -> 2^Q -2^Q is the power set of Q,
the set of all subsets of Q δ(q,s) -The set of all states p such that there is a transition labeled s from q to p δ(q,s) is a function from Q x S to 2^Q (but not to Q)

- Let M = (Q, Σ, δ, q0, F) be an NFA and let w be in Σ*. Then w is accepted by M iff δ({q0}, w) contains at least one state in F

• Let M = (Q, Σ, δ,q0,F) be an NFA. Then the language accepted by M is the set: L(M) = {w | w is in Σ* and δ({q0},w) contains at least one state in F}



An Application - Text Search

Automata theory is the study of abstract machines and automata. It also includes the computational problems that can be solved using them . In the theory of computation, the simpler abstract machine is finite automata.

The other important abstract machines are

1. Pushdown Automata
2. Turing Machine.

The finite automata proposed to model brain function of the human. The simplest example for finite automata is the switch with two states "on" and "off" [1].The Finite Automata is the five tuples combination focusing on states and transition through input characters. In figure 1 the ending state is ON, starting state is OFF and collection of states are OFF and ON. It is having only a single input PUSH for making the transition from the state OFF to ON, then ON to OFF. The switch is the simplest practical application of finite automata.

Examples of NFA

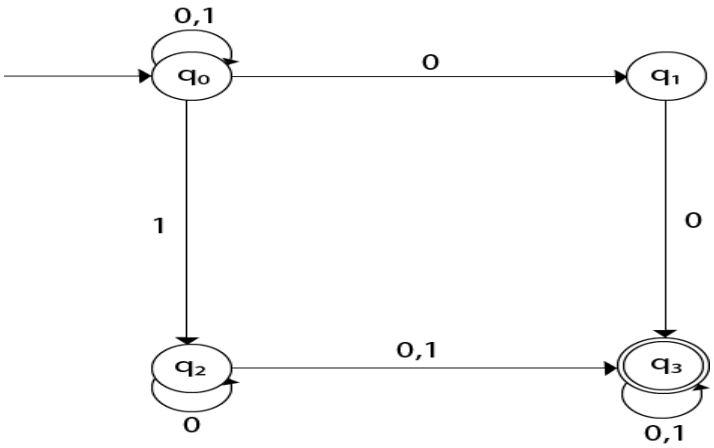
Example 1:
Design a NFA for the transition table as given below:

Present State	0	1
→Q0	Q0,Q1	Q0,Q2

Q1	Q3	E
Q2	Q2,Q3	Q3
→Q3	Q3	Q3

Solution:

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



δ(q0, 0) = {q0, q1}

δ(q0, 1) = {q0, q2}

Then, δ(q1, 0) = {q3}

Then, δ(q2, 0) = {q2, q3}

δ(q2, 1) = {q3}

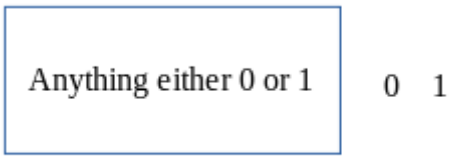
Then, δ(q3, 0) = {q3}

δ(q3, 1) = {q3}

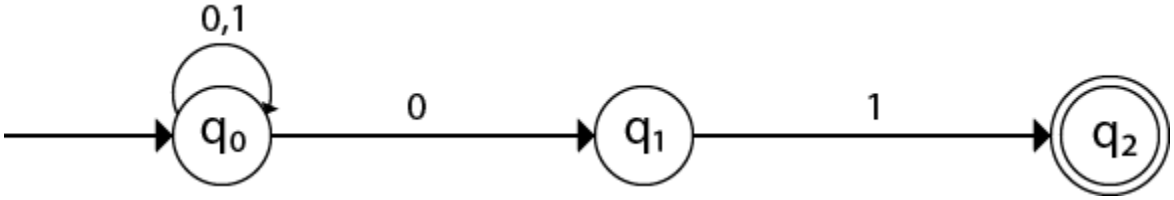
Example 2:

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

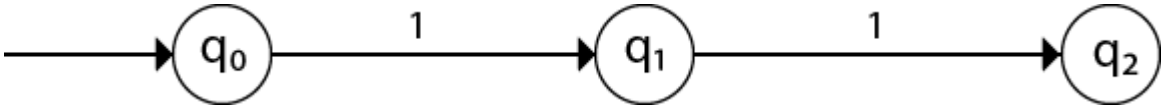
Solution:



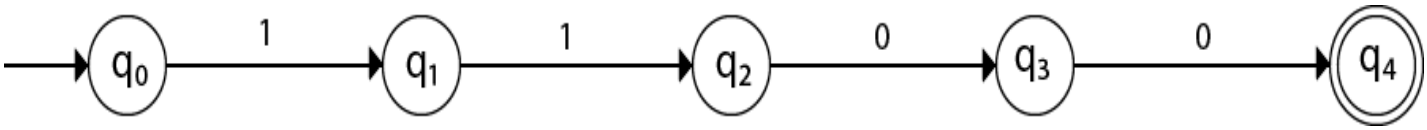
Hence, NFA would be:



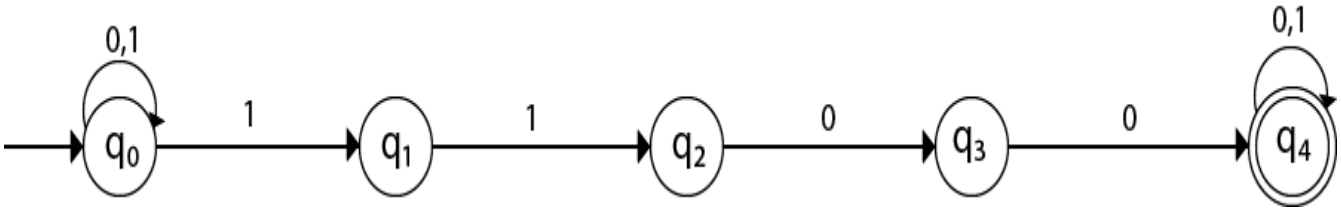
Example 3:
Design an NFA with ∑ = {0, 1} in which double '1' is followed by double '0'.
Solution:
The FA with double 1 is as follows:
Advertisement



It should be immediately followed by double 0.
Then,



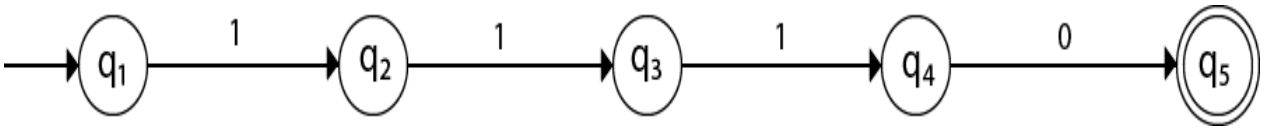
Now before double 1, there can be any string of 0 and 1. Similarly, after double 0, there can be any string of 0 and 1.
Hence the NFA becomes:



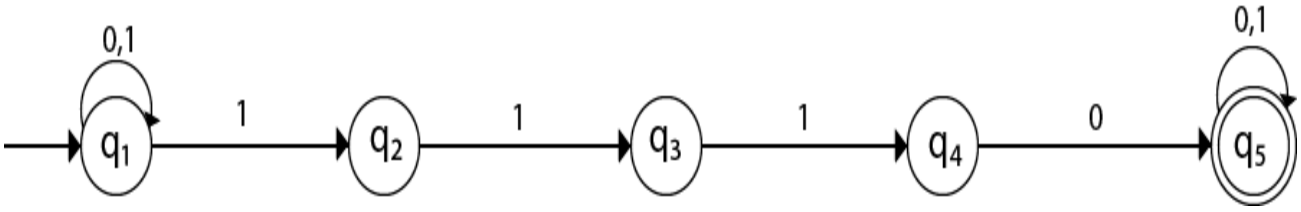
Now considering the string 01100011
q0 → q1 → q2 → q3 → q4 → q4 → q4 → q4

Example 4:

Design an NFA in which all the string contain a substring 1110.
Solution:
The language consists of all the string containing substring 1010. 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
→q1	q1	q1, q2
q2		q3
q3		q4
q4	q5	
*q5	q5	q5

Consider a string 111010,

$\delta(q1, 111010) = \delta(q1, 1100)$
$= \delta(q1, 100)$
$= \delta(q2, 00)$

Got stuck! As there is no path from q2 for input symbol 0. We can process string 111010 in another way.

$\delta(q1, 111010) = \delta(q2, 1100)$
$= \delta(q3, 100)$
$= \delta(q4, 00)$
$= \delta(q5, 0)$
$= \delta(q5, \varepsilon)$

NFA Example 01

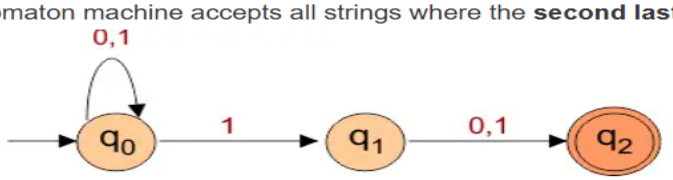
Design an NFA with ∑ = {0, 1} for all binary strings where the **second last bit is 1.**

Solution

The language generated by this example will include all strings in which the **second-last bit is 1.**

> L= {10, 010, 000010, 11, 101011……..}

The following NFA automaton machine accepts all strings where the **second last bit is 1.**



Where,

- {q0, q1, q2} refers to the set of states
- {0,1} refers to the set of input alphabets
- δ refers to the transition function

- q0 refers to the initial state
- {q2} refers to the set of final states

Transition function δ is defined as

- δ (q0, 0) = q0
- δ (q0, 1) = q0,q1
- δ (q1, 0) = q2
- δ (q1, 1) = q2
- δ (q2, 0) = ϕ
- δ (q2, 1) = ϕ

Transition Table for the above Non-Deterministic Finite Automata is-

States	0	1
q0	q0	q0,q1
q1	q2	q2
q2	–	–

Finite Automata with Epsilon Transitions

• The ε-transitions in NFA are given in order to move from one state to another without having any symbol from input set Σ.

Consider the NFA with ε as:

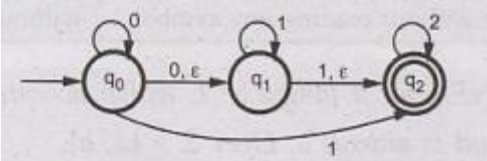


Fig. 1.8.1 NFA with ε - transitions

• In this NFA with ε, q_0 is a start state with input 0 one can be either in state qo or in state q_1. If we get at the start a symbol 1 then with ε - move we can change state from q_0 to q_1 and then with input we can be in state q_1.

• On the other hand, from start state q_0, with input 1 we can reach to state q_2.

• Thus it is not definite that on input 1 whether we will be in state q_1 or q_2. Hence it is called nondeterministic finite automata (NFA) and since there are some ε moves by which we can simply change the states from one state to other. Hence it is called NFA with ε.

Definition of NFA with ε

The language L accepted by NFA with &, denoted by M = (Q, Σ, δ, q_0, F) can be defined as:

Let, M = (Q, Σ, δ, q_0, F) be a NFA with ε.

Where

Q is a finite set of states.

Σ is input set.

δ is a transition or a mapping function for transitions from Q×{ΣUε} to 2^Q.

q_0 is a start state.

F is a set of final states such that F ϵ Q.

Example 1.8.1 *Construct NFA with ε which accepts a language consisting the strings of any number of a's followed by any number of b's. Followed by any number of c's.*

Solution: Here any number of a's or b's or c's means zero or more in number. That means there can be zero or more a's followed by zero or more b's followed by zero or more c's. Hence NFA with ε can be -

Normally ε's are not shown in the input string. The transition table can be -

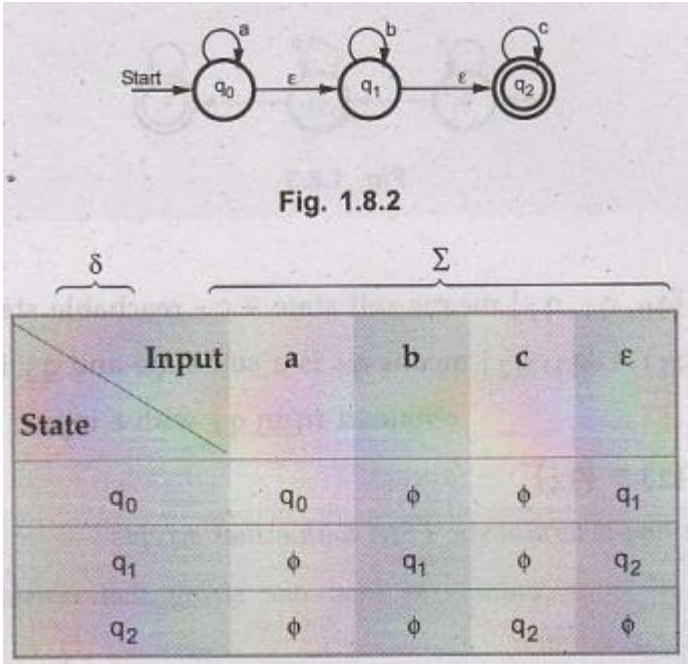


Fig. 1.8.2

State \ Input	a	b	c	ε
q_0	q_0	ϕ	ϕ	q_1
q_1	ϕ	q_1	ϕ	q_2
q_2	ϕ	ϕ	q_2	ϕ

We can parse the string aabbcc as follows -

$\delta (q_0, aabbcc) \vdash \delta (q_0, abbcc)$

$\vdash \delta (q_0, bbcc)$

$\vdash \delta (q_0, \varepsilon bbcc)$

$\vdash \delta (q_1, bbcc)$

$\vdash \delta (q_1, bcc)$

$\vdash \delta (q_1, cc)$

$\vdash \delta (q_1, \varepsilon cc)$

$\vdash \delta (q_2, cc)$

$\vdash \delta (q_2, c)$

$\vdash \delta (q_2, \varepsilon)$

Thus we reach to accept state, after scanning the complete input string.

Definition of ε - closure

The ε - closure (p) is a set of all states which are reachable from state p on ε - transitions such that:

i) ε - closure (p) = p where p ϵ Q.

ii) If there exists ε closure (p) = {q} and δ(q,ε) = r then

ε - closure (p) = {q, r}

Example 1.8.2 *Find ε - closure for the following NFA with ε.*

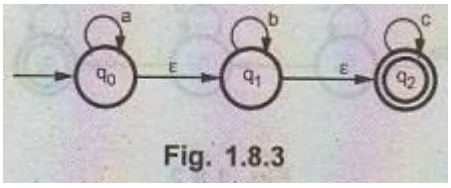


Fig. 1.8.3

Solution:

ε - closure (q_0) = {q_0, q_1, q_2} means self state + ε - reachable states.

ε - closure (q_1) = {q_1, q_2} means q_1 is a self state and q_2 is a state obtained from q_1 with ε input.

ε - closure (q_2) = {q_2}

Example 1.8.3 *Build a non-deterministic FSM with & that accepts*

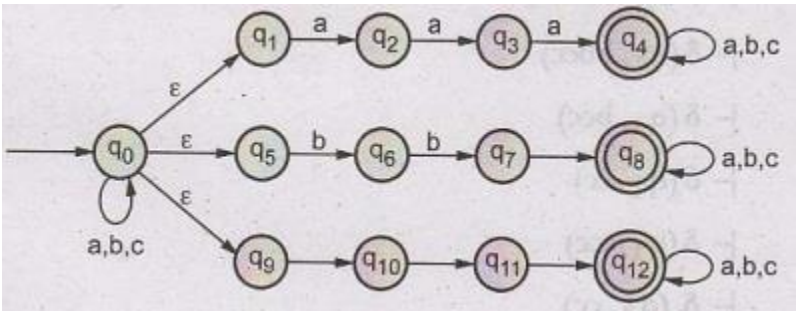
L = {w = {a, b, c): w contains at least one string that consists of three identical symbolds in a row). For example*

• *The following strings are in L are aabbb, baabbbccc*

• *The following strings are not in L are ε, aba, abababab, abcbcab.*

Solution: The NDFSM with ε is

AUTOMATA THEOARY AND COMPILER DESIGN



Acceptors,Classifiers,andTransducers Acceptor (Recognizer)

Acceptors

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

Classifier

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

Transducer

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

DETERMINISTIC FINITE AUTOMATA:

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

Formal Definition of a DFA

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

Q is a finite set of states.
∑ is a finite set of symbols called the alphabet.
Δ is the transition function where δ:Q ×∑→ Q
q0 is the initial state from where any input is processed (q0∈Q).

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

Graphical Representation of a DFA

A DFA is represented by digraphs called state diagram.

The vertices represent the states.
The arcs labeled with an input alphabet show the transitions.
The initial state is denoted by an empty single incoming arc.
The final state is indicated by double circles.

Example

Let a deterministic finite automaton be→

Q={a, b, c},
∑={0, 1},
q0={a},
F={c},and

Transition function δ as shown by the following table –

Present State	Next state For input0	Next state for input 1
A	A	B
B	C	A
C	B	C

DFA vs NDFA

DFA	NDFA
The transition from a state is to a single particular next state for each input symbol. Hence it is called deterministic.	The transition from a state can be to multiple next states for each input symbol. Hence it is called non-deterministic.

Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Back tracking is allowed in DFA	In NDFA, back tracking is not always possible.
Requires more space.	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

How A DFA Process Strings

A DFA processes a string by starting in the start state and reading the input symbols one at a time. For each input symbol, the DFA transitions to the next state according to the transition function, and it accepts the string if it ends in an accepting state.
An algorithm that describes how a DFA processes a string:

1. Start in the start state.
2. Read the next input symbol.
3. Transition to the next state according to the transition function.
4. If the current state is an accepting state, then accept the string.
5. If the current state is not an accepting state, and there are no more input symbols to read, then reject the string.

Repeat steps 2-5 until the end of the string is reached.

For example,
Consider the DFA that identifies whether the given decimal is even or odd.
Here we consider 3 states, one start state qstart, one even state qeven and one odd state qodd.
If the machine stops at qeven the given number is even.
If it stops at qodd the given number is odd.

1. Deterministic Finite Automata (DFA)

A DFA is represented as {Q, Σ, q, F, δ}. In DFA, for each input symbol, the machine transitions to one and only one state. DFA does not allow any null transitions, meaning every state must have a transition defined for every input symbol.
DFA consists of 5 tuples {Q, Σ, q, F, δ}.
Q : set of all states.
Σ : set of input symbols. (Symbols which machine takes as input)
q : Initial state. (Starting state of a machine)
F : set of final state.
δ : Transition Function, defined as δ : Q X Σ --> Q.

Example:
Construct a DFA that accepts all strings ending with ‘a’.
Given:
Σ = {a, b},
Q = {q0, q1},
F = {q1}

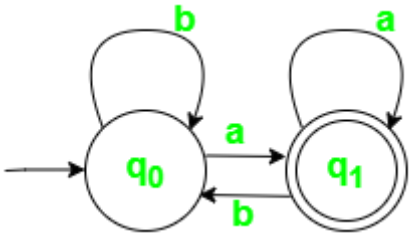


Fig 1. State Transition Diagram for DFA with Σ = {a, b}

State\Symbol	a	b
q0	q1	q0
q1	q1	q0

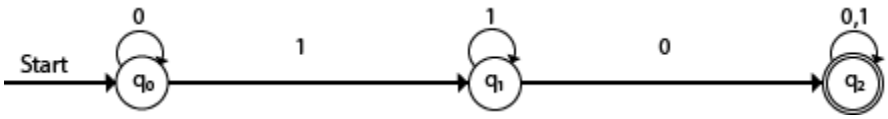
In this example, if the string ends in ‘a’, the machine reaches state q1, which is an accepting state.

Example 1:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q2}

Solution:

Transition Diagram:



Transition Table:

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

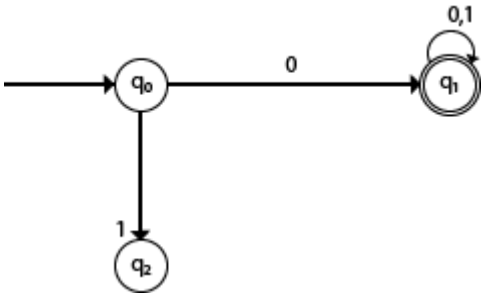
q1	q2	q1
*q2	q2	q2

(0|1)*10(0|1)*

Example 2:
DFA with ∑ = {0, 1} accepts all starting with 0.

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

Solution:

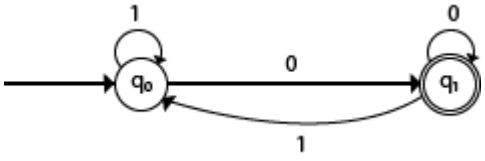


Explanation:

- In the above diagram, we can see that on given 0 as input to DFA in state q0 the DFA changes state to q1 and always go to final state q1 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 ∑ = {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 q0, the DFA changes state to q1. 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 q1 on 1 input, so the string ending with 1, will not be accepted or will be rejected.

Example 3:
Design FA with ∑ = {0, 1} accepts even number of 0's and even number of 1's.

Solution:

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

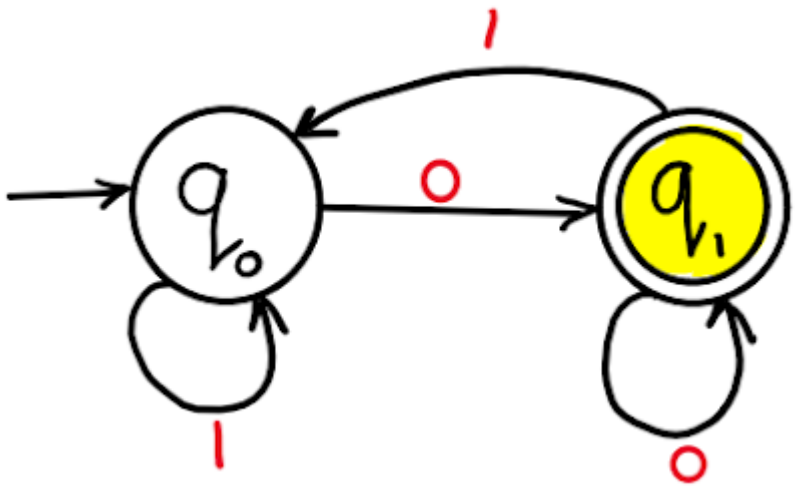
L= {e, 00,11,0101,1100,0110,1010,……}

1. Even 0's and even 1's
2. Even 0's and odd 1's
3. Odd 0's and even 1's
4. Odd 0's and odd 1's

	0	1
A	C	B
B	D	A
C	A	D
D	B	C

Example : Draw a DFA for the language accepting strings ending with '0' over input alphabets ∑={0, 1} ?

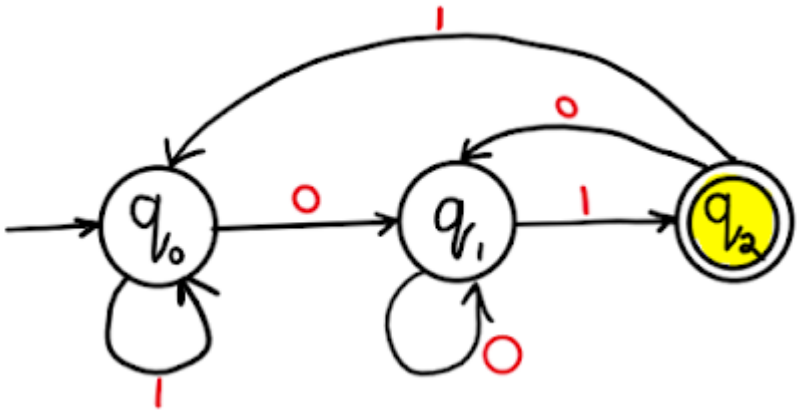
Solution:



AUTOMATA THEOARY AND COMPILER DESIGN

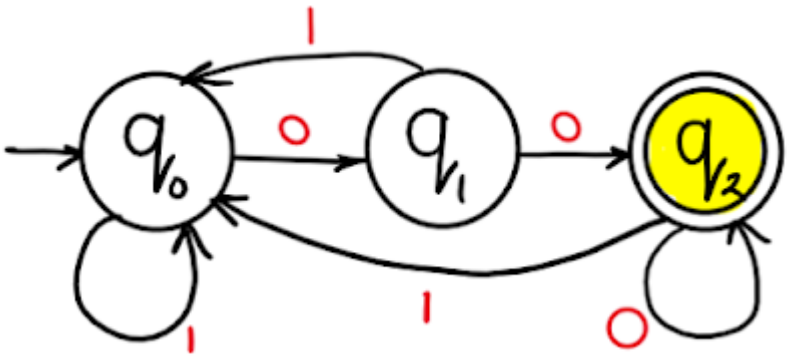
Example 2: Draw a DFA for the language accepting strings ending with ‘01’ over input alphabets ∑={0, 1} ?

Solution:



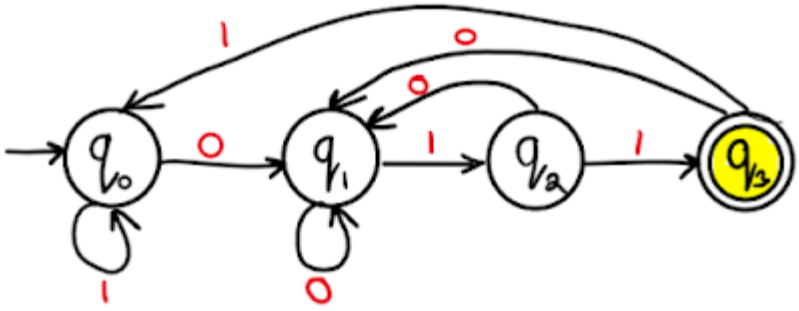
Example 3: Draw a DFA for the language accepting strings ending with ‘00’ over input alphabets ∑={0, 1} ?

Solution:



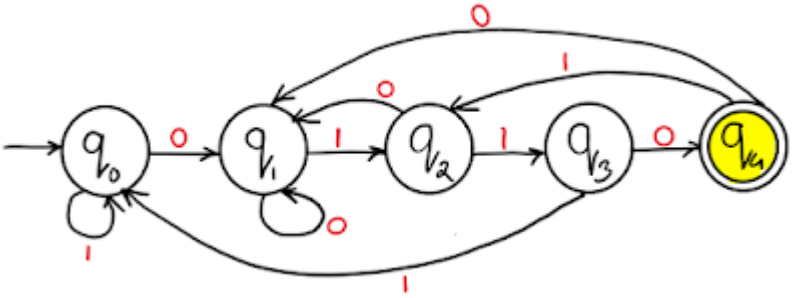
Example 4: Draw a DFA for the language accepting strings ending with ‘011’ over input alphabets ∑ = {0, 1} ?

Solution:



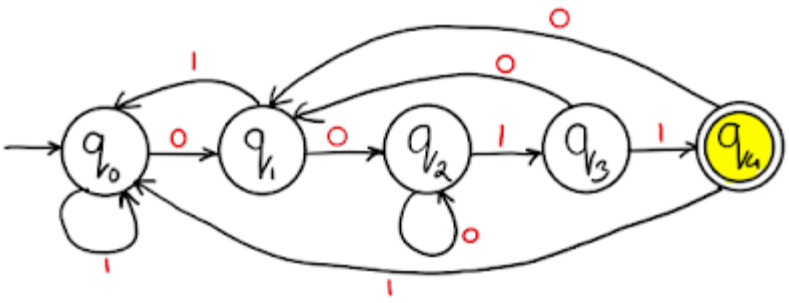
Example 5: Draw a DFA for the language accepting strings ending with ‘0110’ over input alphabets ∑ = {0, 1} ?

Solution:



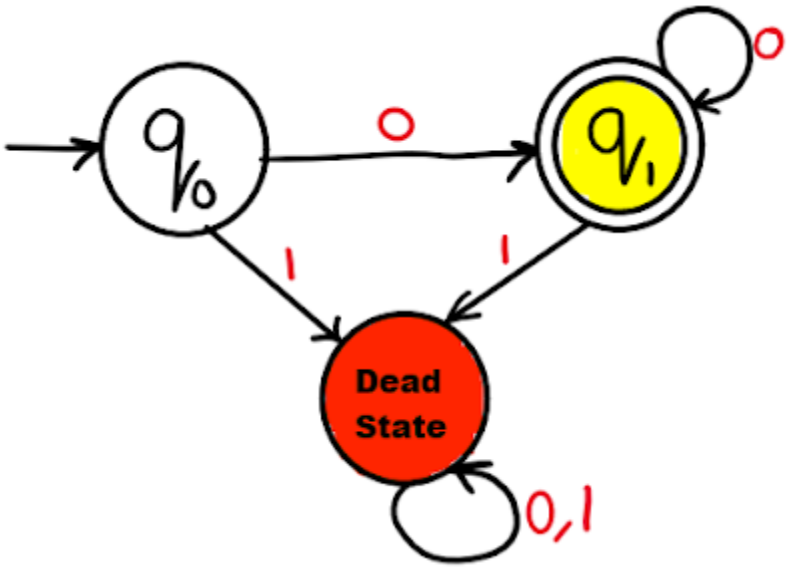
Example 6: Draw a DFA for the language accepting strings ending with ‘0011’ over input alphabets ∑ = {0, 1} ?

Solution:



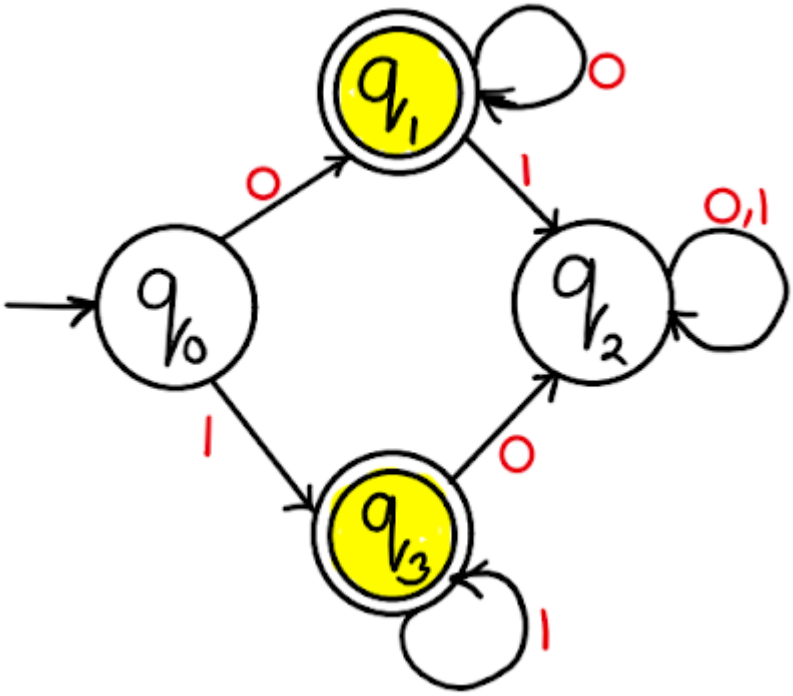
Example 7: Draw a DFA for the language accepting strings with ‘0’ only over input alphabets ∑={0, 1} ?

Solution:



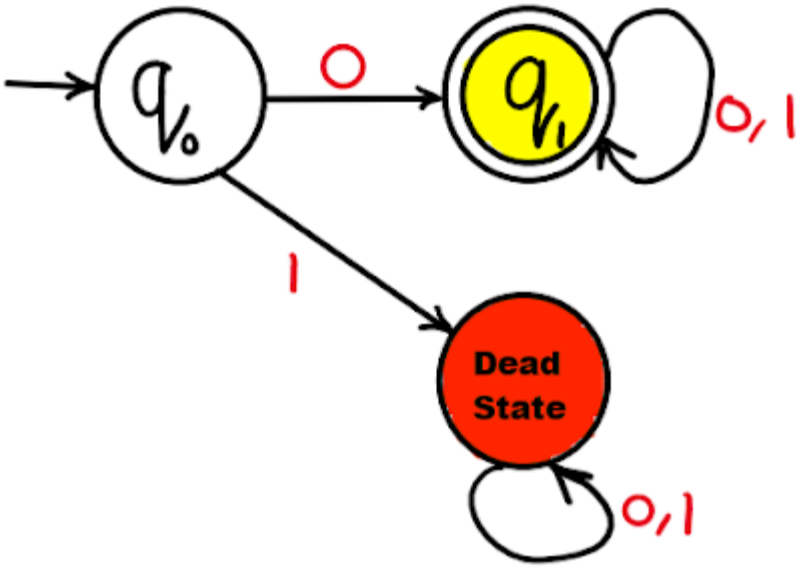
Example 8: Draw a DFA for the language accepting strings with ‘0’ and ‘1’ only over input alphabets ∑={0, 1} ?

Solution:



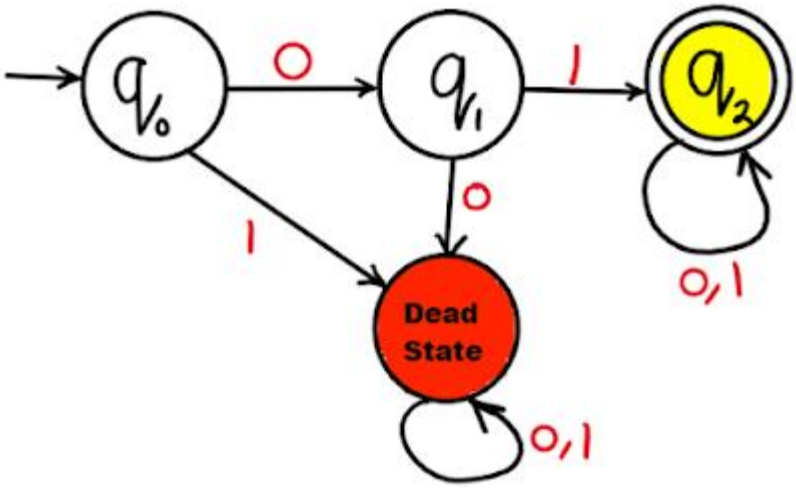
Example 9: Draw a DFA for the language accepting strings starting with ‘0’ over input alphabets ∑={0, 1} ?

Solution:



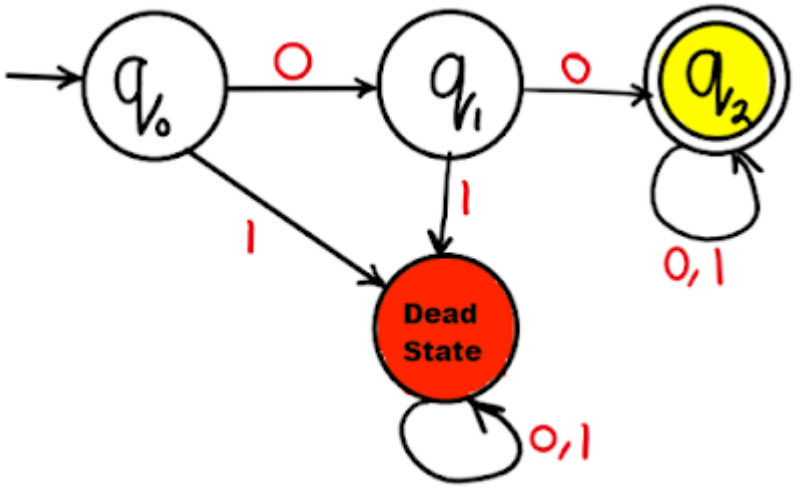
Example 10: Draw a DFA for the language accepting strings starting with '01' over input alphabets ∑={0, 1} ?

Solution:



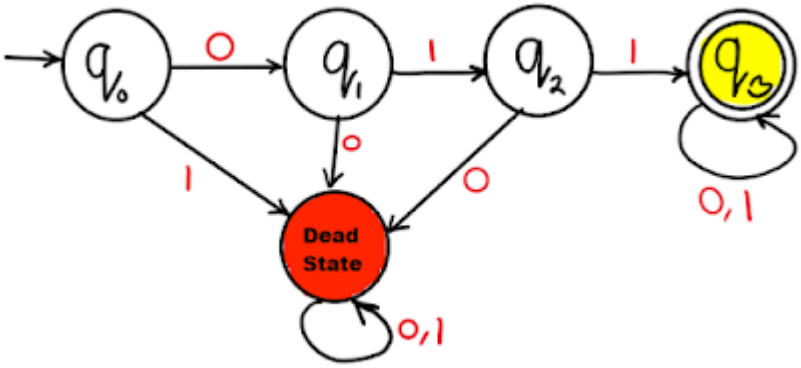
Example 11: Draw a DFA for the language accepting strings starting with '00' over input alphabets ∑={0, 1} ?

Solution:



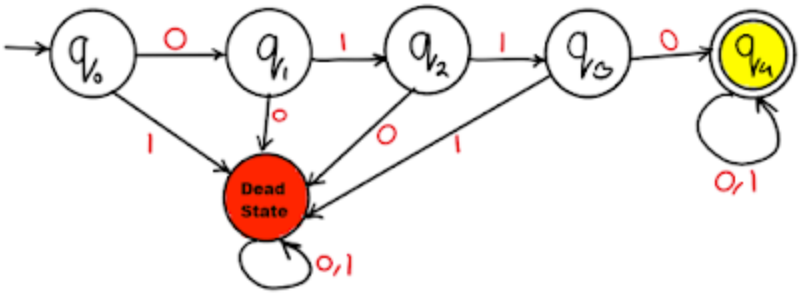
Example 12: Draw a DFA for the language accepting strings starting with '011' over input alphabets ∑ = {0, 1} ?

Solution:



Example 13: Draw a DFA for the language accepting strings starting with '0110' over input alphabets ∑ = {0, 1} ?

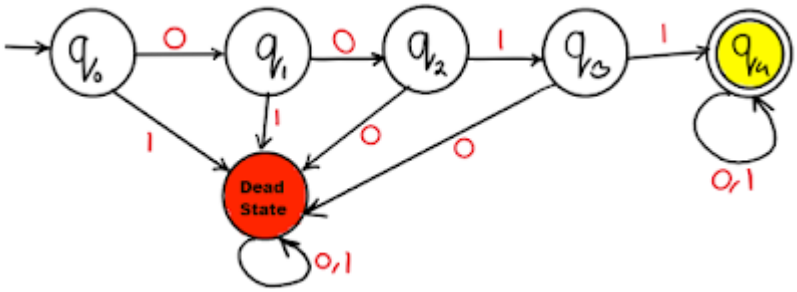
Solution:



AUTOMATA THEOARY AND COMPILER DESIGN

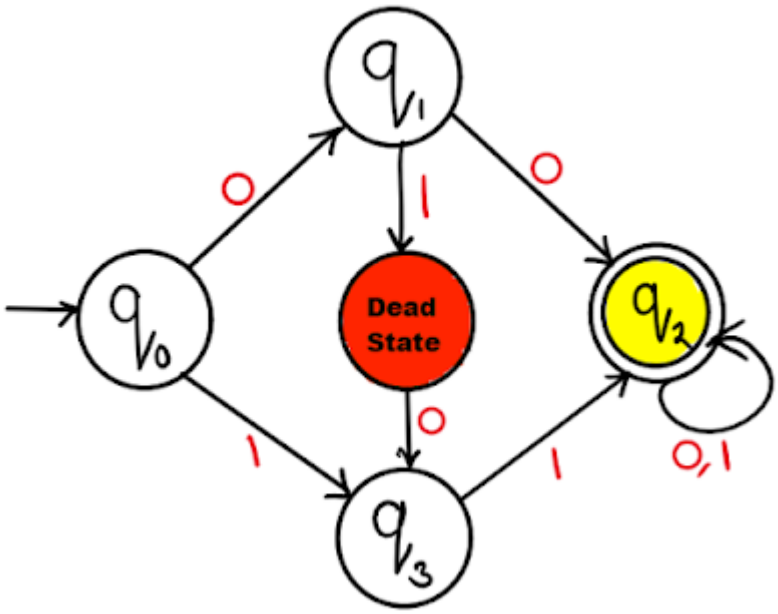
Example 14: Draw a DFA for the language accepting strings starting with ‘0011’ over input alphabets ∑ = {0, 1} ?

Solution:



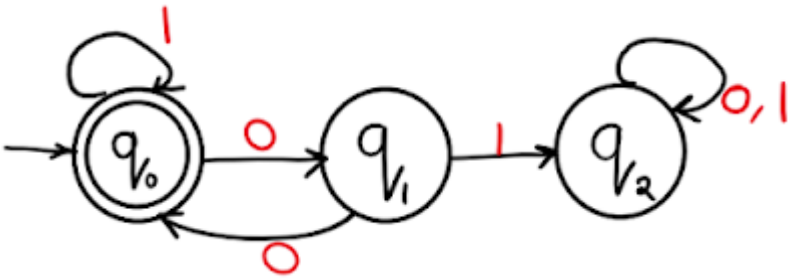
Example 15: Draw a DFA for the language accepting strings starting with ‘00’ or ’11’ over input alphabets ∑ = {0, 1} ?

Solution:



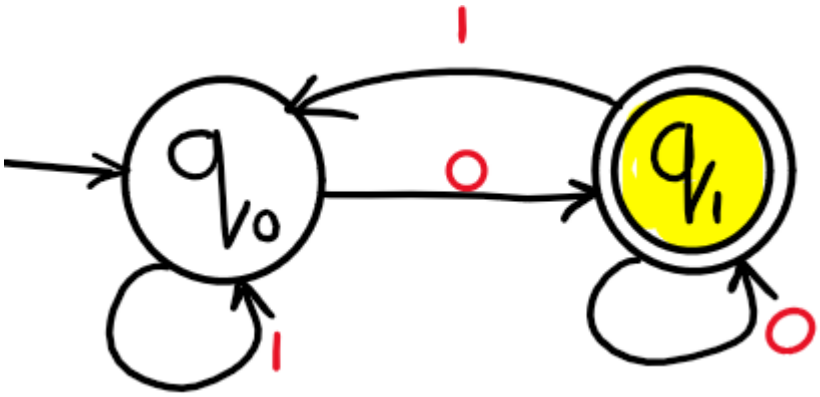
Example 16: Draw a DFA for the language accepting strings without substring ‘00’ over input alphabets ∑ = {0, 1} ?

Solution:



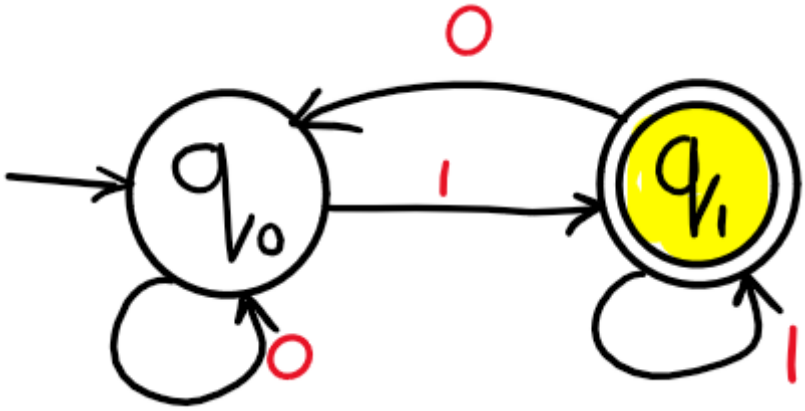
Example 17: Draw a DFA for the language accepting even binary numbers strings over input alphabets ∑ = {0, 1} ?

Soluntion:



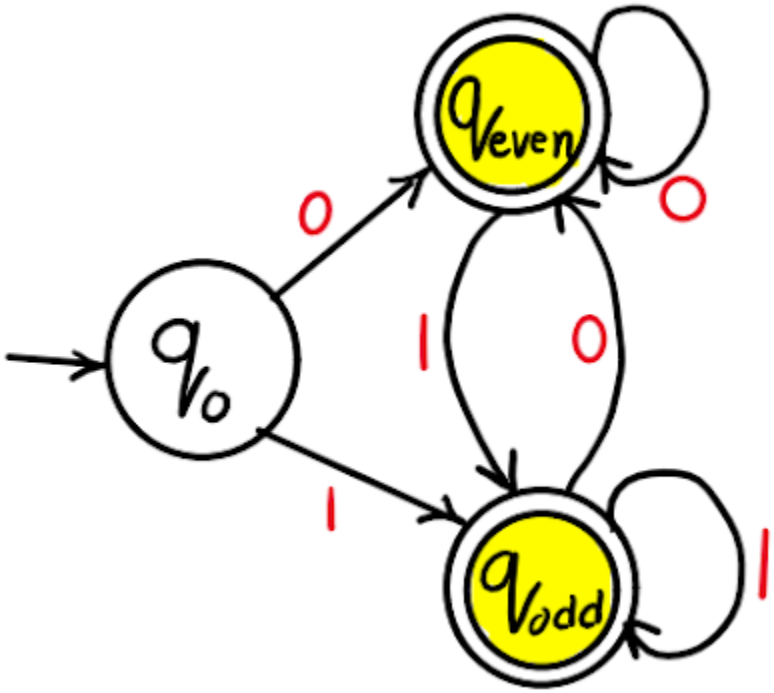
Example 18: Draw a DFA for the language accepting odd binary numbers strings over input alphabets ∑ = {0, 1} ?

Solution:



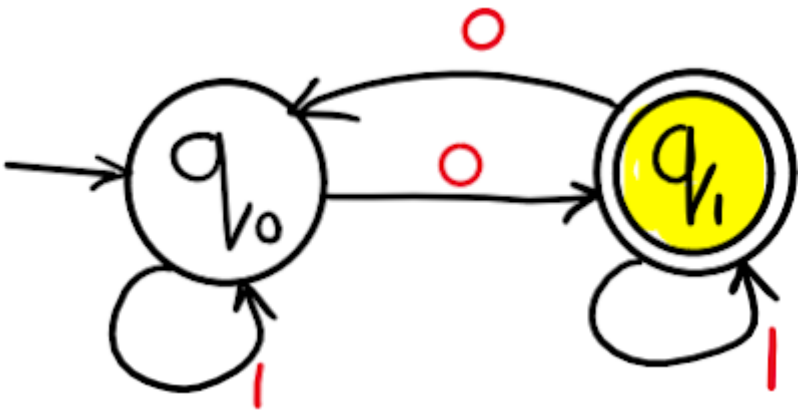
Example 19: Draw a DFA for the language accepting odd or even binary numbers strings over input alphabets ∑ = {0, 1} ?

Solution:



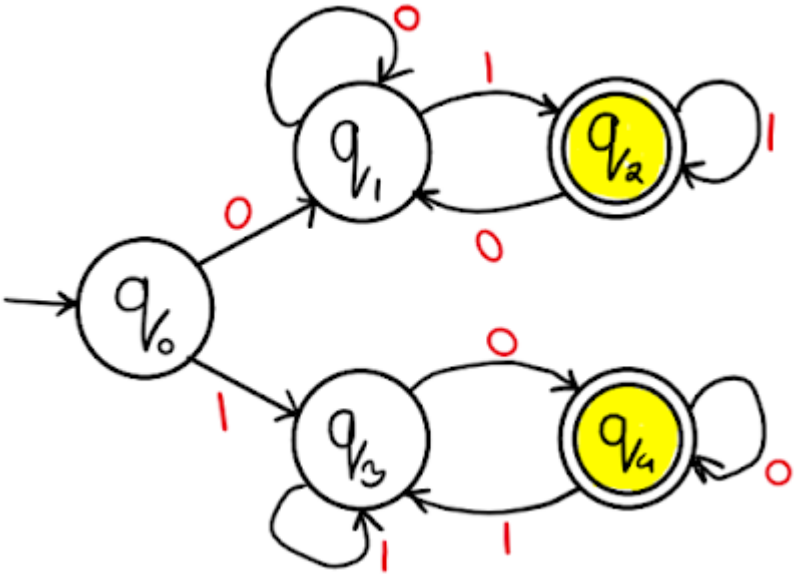
Example 20: Draw a DFA for the language accepting strings containg even number of total zeros over input alphabets ∑ = {0, 1} ?

Solution:



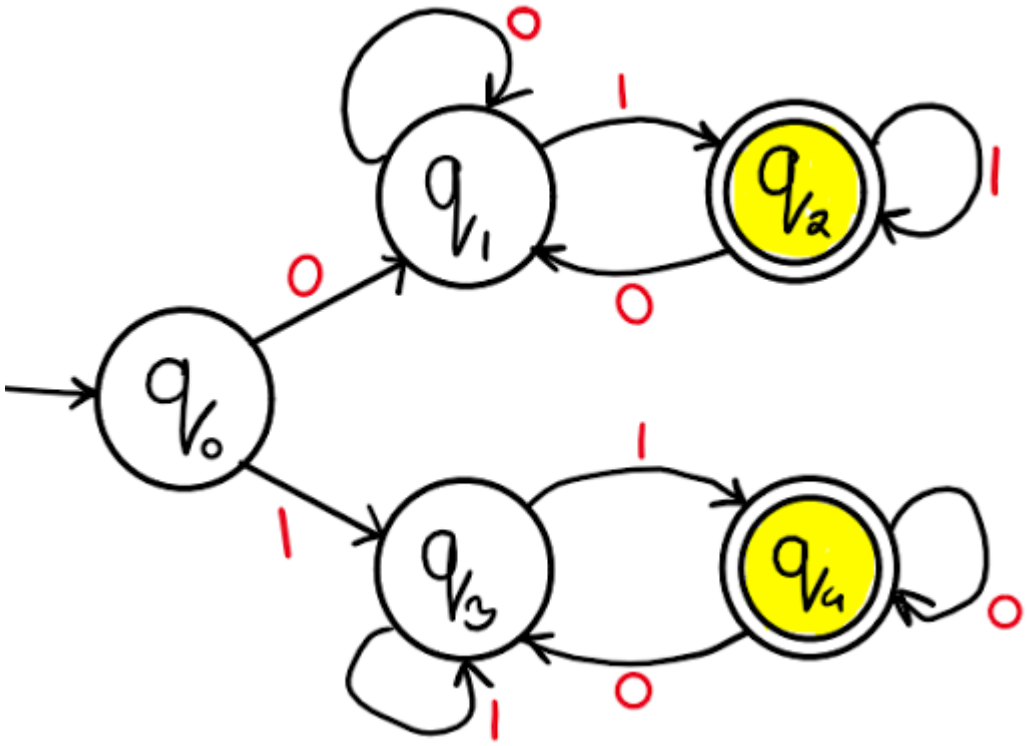
Example 21: Draw a DFA for the language accepting strings starting and ending with different characters over input alphabets ∑ = {0, 1} ?

Soluiton:



Example 22: Draw a DFA for the language accepting strings starting and ending with same character over input alphabets ∑ = {0, 1} ?

Solution:

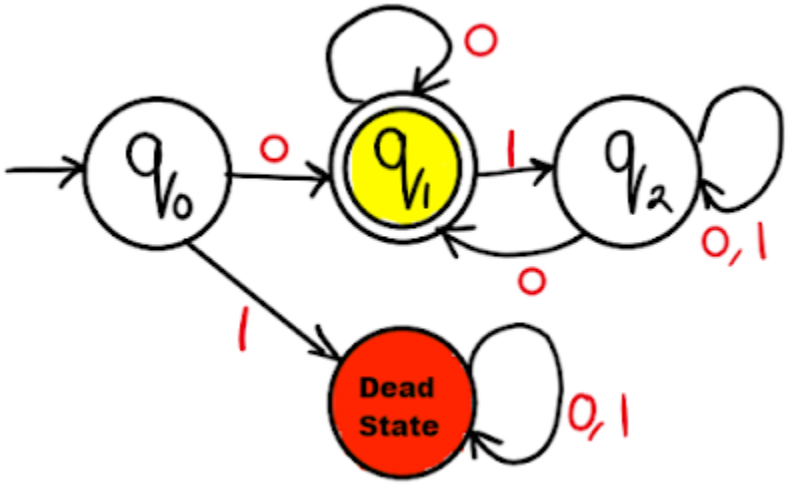


SSS

AUTOMATA THEOARY AND COMPILER DESIGN

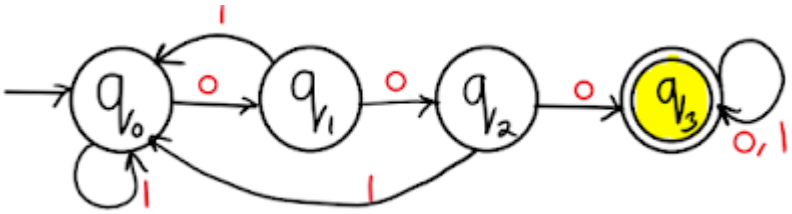
Example 23: Draw a DFA for the language accepting strings starting and ending with '0' always over input alphabets ∑ = {0, 1} ?

Solution:



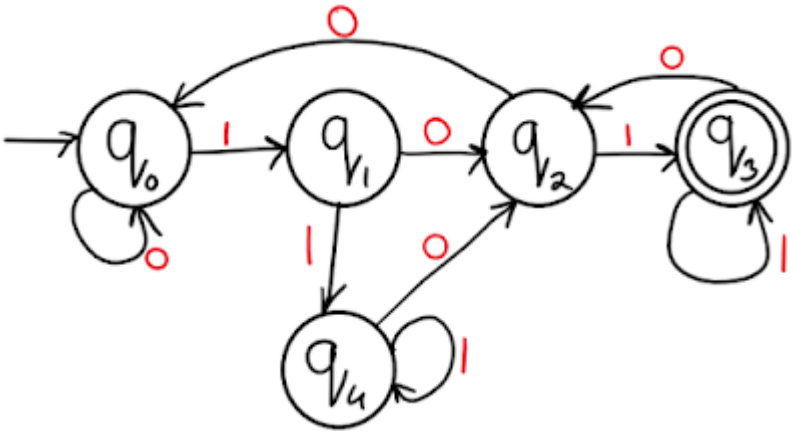
Example 24: Draw a DFA for the language accepting strings containing three consecutives '0' always over input alphabets ∑ = {0, 1} ?

Solution:



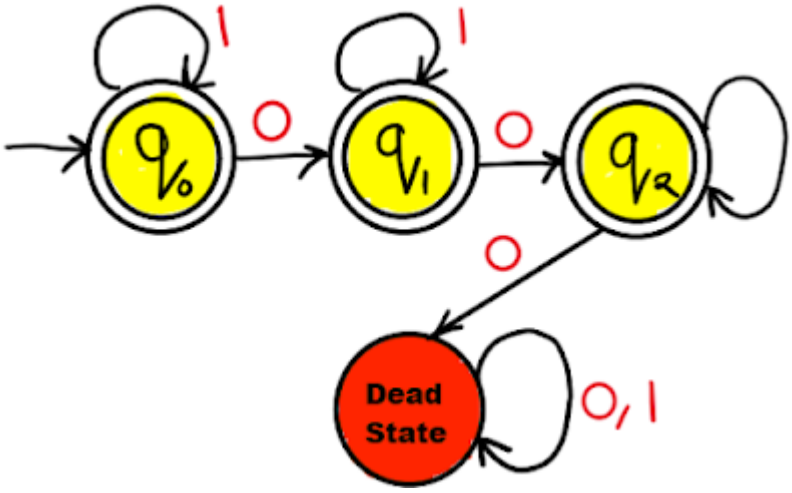
Example 25: Draw a DFA for the language accepting strings such that each '0' is immediately preceded and followed by '1' over input alphabets ∑ = {0, 1} ?

Solution:



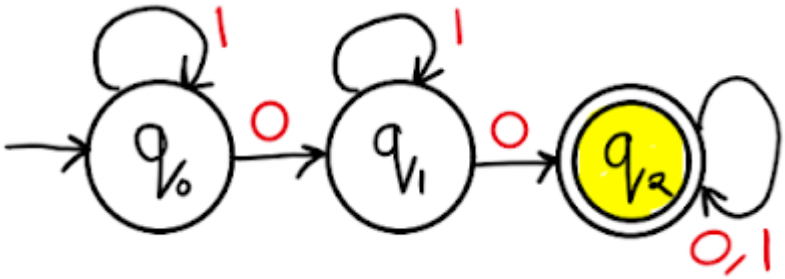
Example 26: Draw a DFA for the language accepting strings containing at most two ‘0’ over input alphabets ∑ = {0, 1} ?

Solution:



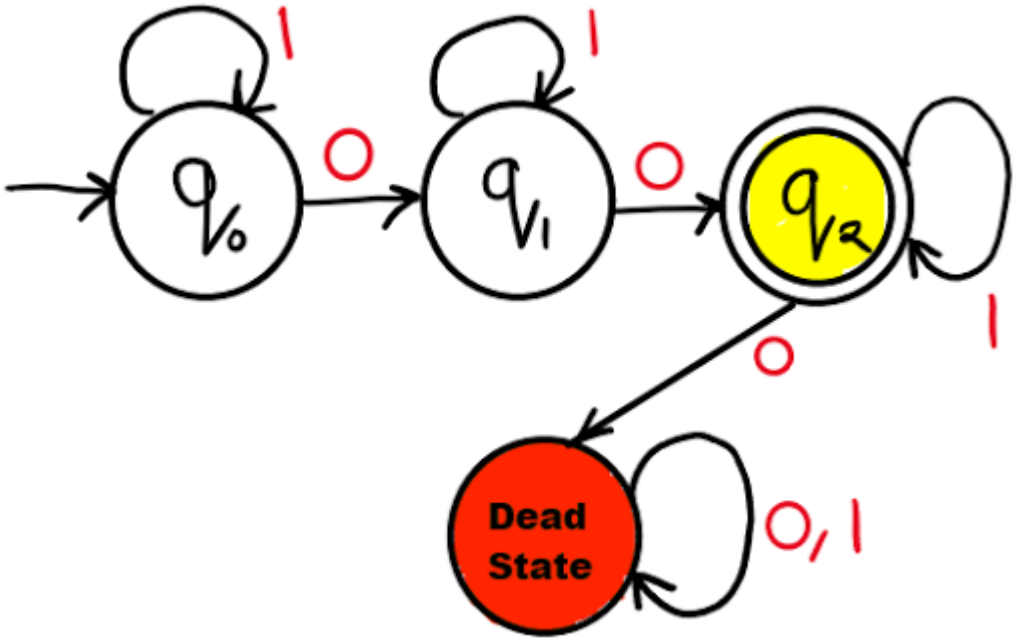
Example 27: Draw a DFA for the language accepting strings containing at least two ‘0’ over input alphabets ∑ = {0, 1} ?

Solution:



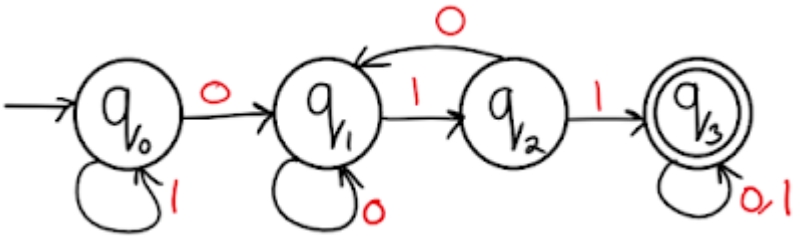
Example 28: Draw a DFA for the language accepting strings containing exactly two '0' over input alphabets ∑ = {0, 1} ?

Solution:



Example 29: Draw a DFA for the language accepting strings with '011' as substring over input alphabets ∑ = {0, 1} ?

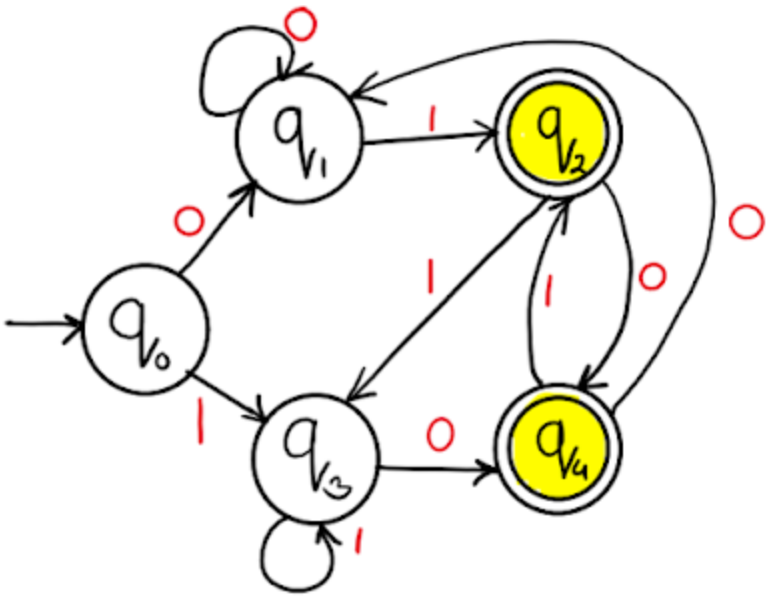
Solution:



AUTOMATA THEOARY AND COMPILER DESIGN

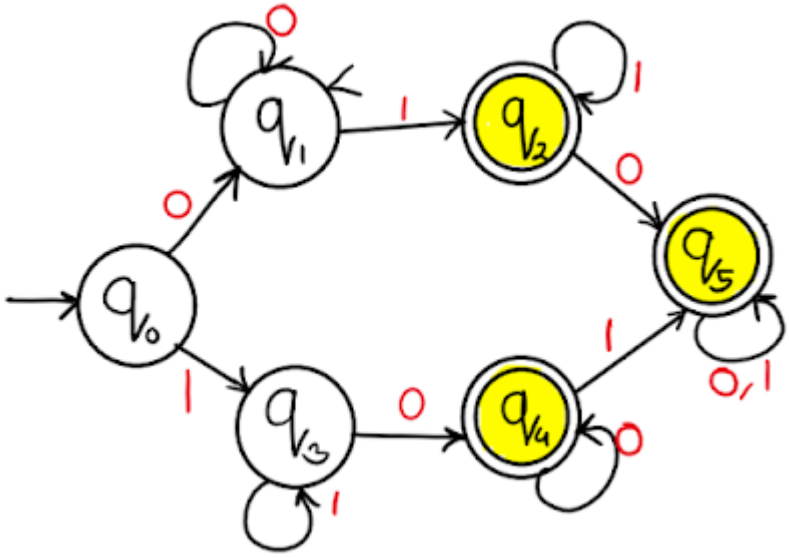
Example 30: Draw a DFA for the language accepting strings ending in either ’01’, or ’10’ over input alphabets ∑ = {0, 1} ?

Solution:

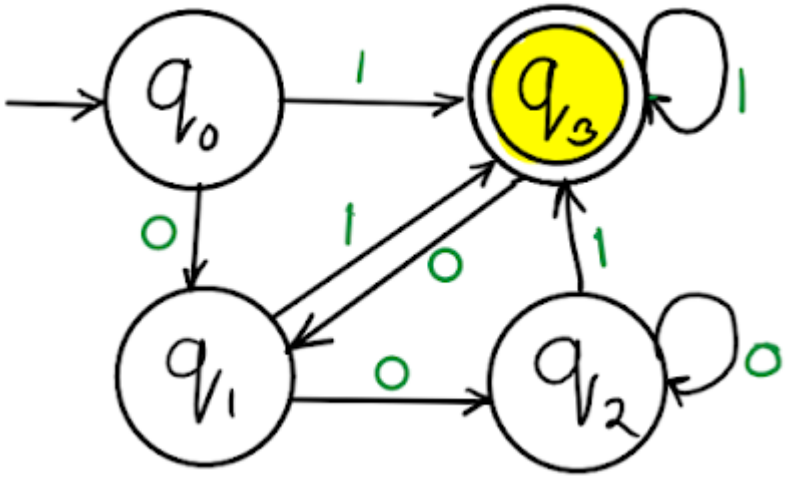


Example 31: Draw a DFA for the language accepting strings containing ’01’, or ’10’ as substring over input alphabets ∑ = {0, 1} ?

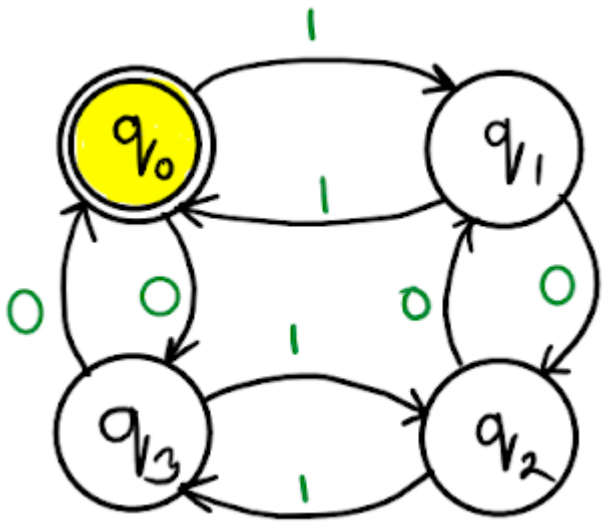
Solution:



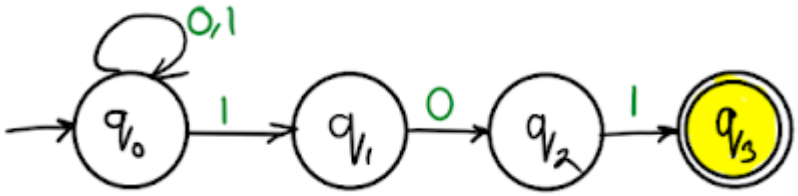
Example 32: Draw DFA that accepts any string which ends with 1 or it ends with an even number of 0's following the last 1.
Alphabets are {0,1}.
Solution:



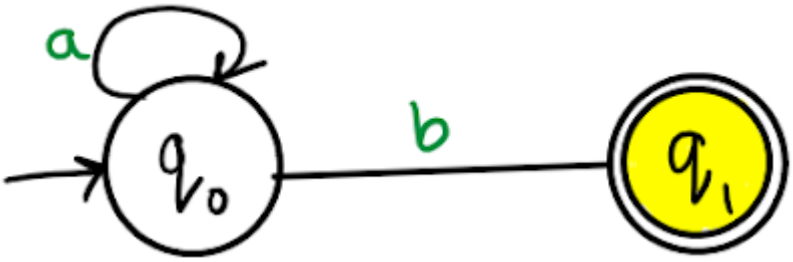
Example 33: Construct DFA accepting set of all strings containing even no. of a's and even no. of b's over input alphabet {a,b}.
Solution:



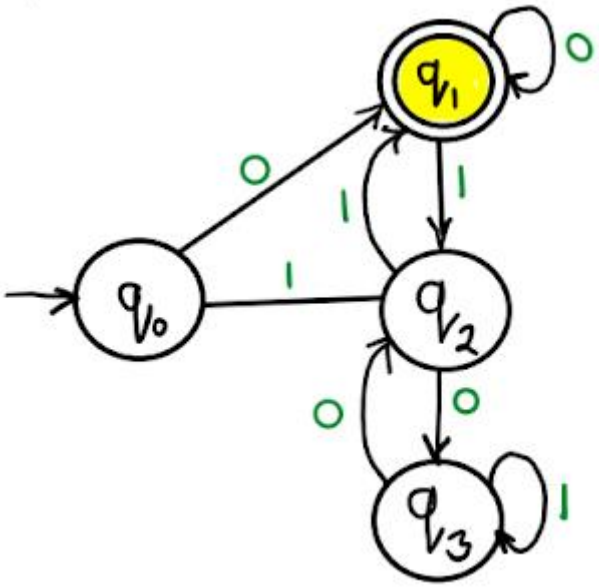
Example 34: Give DFA accepting the language over alphabet {0,1} such that all strings of 0 and 1 ending in 101.
Solution:



Example 35: Construct DFA for anb | n>=0.
Solution:

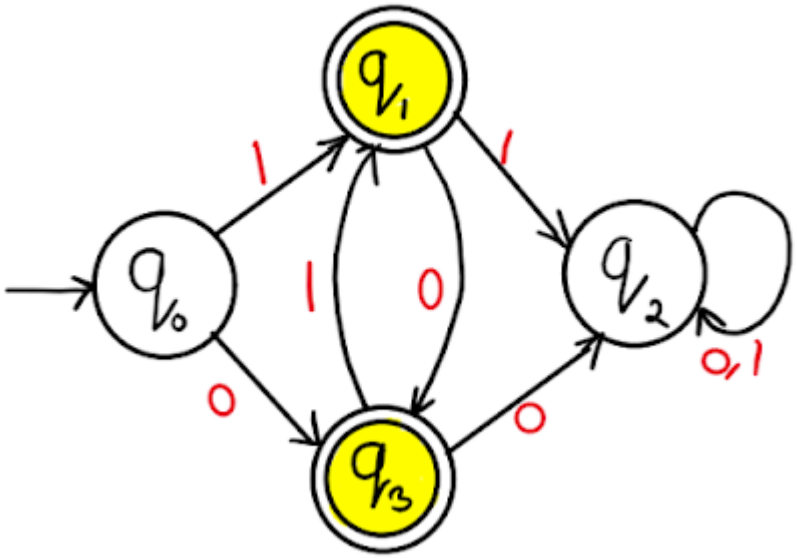


Example 36: construct DFA for binary integer divisible by 3 ?
Solution:



Example 37: Draw a DFA for the language accepting strings containing neither ’00’, nor ’11’ as substring over input alphabets ∑ = {0, 1} ?

Solution:



Conversion of Epsilon-NFA to NFA

Non-deterministic Finite Automata (NFA) is a finite automata having zero, one, or more than one moves from a given state on a given input symbol. Epsilon NFA is the NFA that contains epsilon move(s)/Null move(s). To remove the epsilon move/Null move from epsilon-NFA and to convert it into NFA, we follow the steps mentioned below.

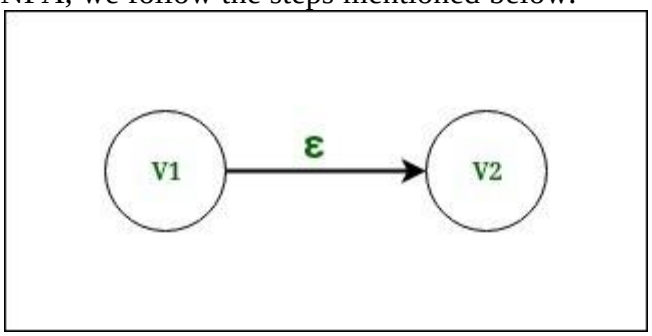
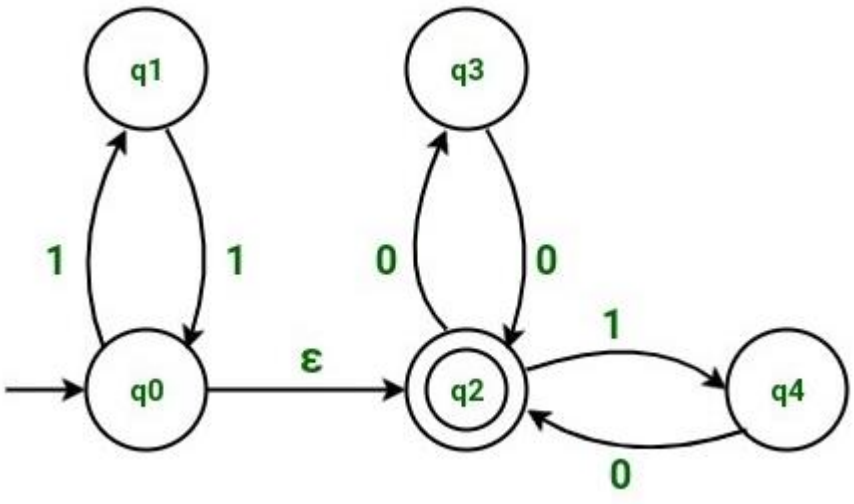


Figure – Vertex v1 and Vertex v2 having an epsilon move

Step-1: Consider the two vertexes having the epsilon move. Here in *Fig.1* we have vertex v1 and vertex v2 having epsilon move from v1 to v2.
Step-2: Now find all the moves to any other vertex that start from vertex v2 *(other than the epsilon move that is considering).* After finding the moves, duplicate all the moves that start from vertex v2, with the same input to start from vertex v1 and remove the epsilon move from vertex v1 to vertex v2.

Step-3: See that if the vertex v1 is a start state or not. If vertex v1 is a start state, then we will also make vertex v2 as a start state. If vertex v1 is not a start state, then there will not be any change.
Step-4: See that if the vertex v2 is a final state or not. If vertex v2 is a final state, then we will also make vertex v1 as a final state. If vertex v2 is not a final state, then there will not be any change. Repeat the steps*(from step 1 to step 4)* until all the epsilon moves are removed from the NFA. Now, to explain this conversion, let us take an example.
Example: Convert epsilon-NFA to NFA. Consider the example having states q0, q1, q2, q3, and q4.



In the above example, we have 5 states named as q0, q1, q2, q3 and q4. Initially, we have q0 as start state and q2 as final state. We have q1, q3 and q4 as intermediate states.
Transition table for the above NFA is:

States/Input	Input 0	Input 1	Input epsilon
q0	–	q1	q2
q1	–	q0	–
q2	q3	q4	–
q3	q2	–	–
q4	q2	–	–

According to the transition table above,

- state q0 on getting input 1 goes to state q1.
- State q0 on getting input as a null move *(i.e. an epsilon move)* goes to state q2.
- State q1 on getting input 1 goes to state q0.
- Similarly, state q2 on getting input 0 goes to state q3, state q2 on getting input 1 goes to state q4.
- Similarly, state q3 on getting input 0 goes to state q2.
- Similarly, state q4 on getting input 0 goes to state q2.

We can see that we have an epsilon move from state q0 to state q2, which is to be removed. To remove epsilon move from state q0 to state q1, we will follow the steps mentioned below.

Step-1: Considering the epsilon move from state q0 to state q2. Consider the state q0 as vertex v1 and state q2 as vertex v2.

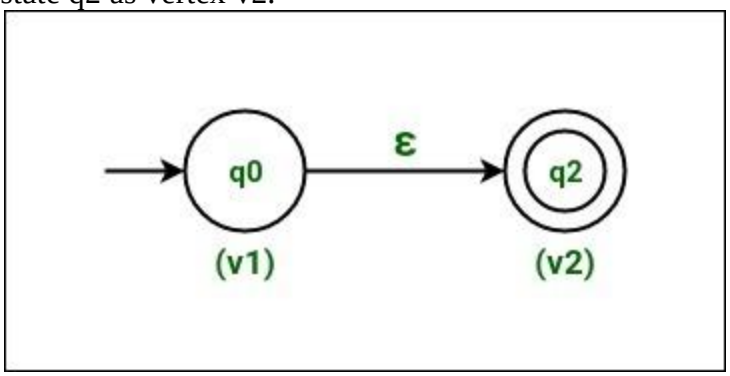


Figure – State q0 as vertex v1 and state q2 as vertex v2

Step-2: Now find all the moves that starts from vertex v2 *(i.e. state q2)*. After finding the moves, duplicate all the moves that start from vertex v2 *(i.e state q2)* with the same input to start from vertex v1 *(i.e. state q0)* and remove the epsilon move from vertex v1 *(i.e. state q0)* to vertex v2 *(i.e. state q2)*. Since state q2 on getting input 0 goes to state q3. Hence on duplicating the move, we will have state q0 on getting input 0 also to go to state q3.

Similarly state q2 on getting input 1 goes to state q4. Hence on duplicating the move, we will have state q0 on getting input 1 also to go to state q4.

So, NFA after duplicating the moves is:

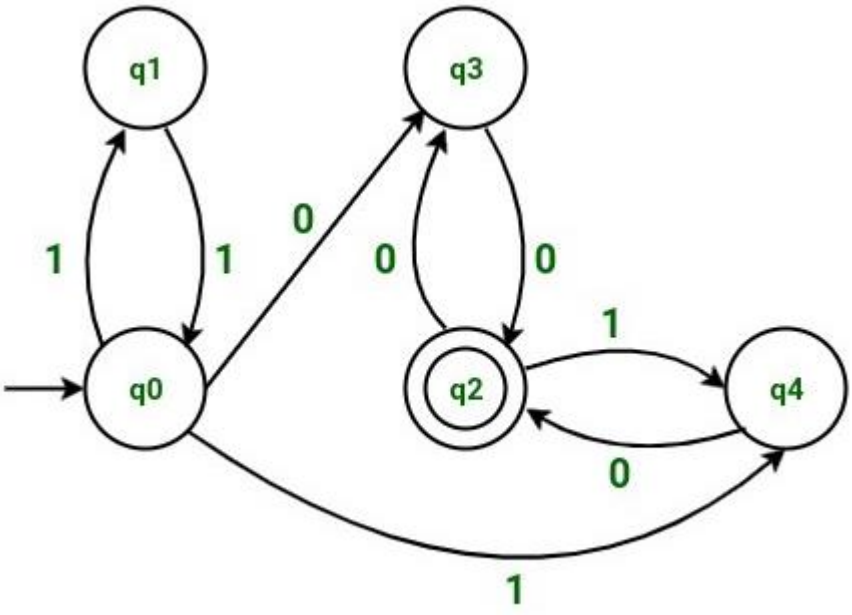
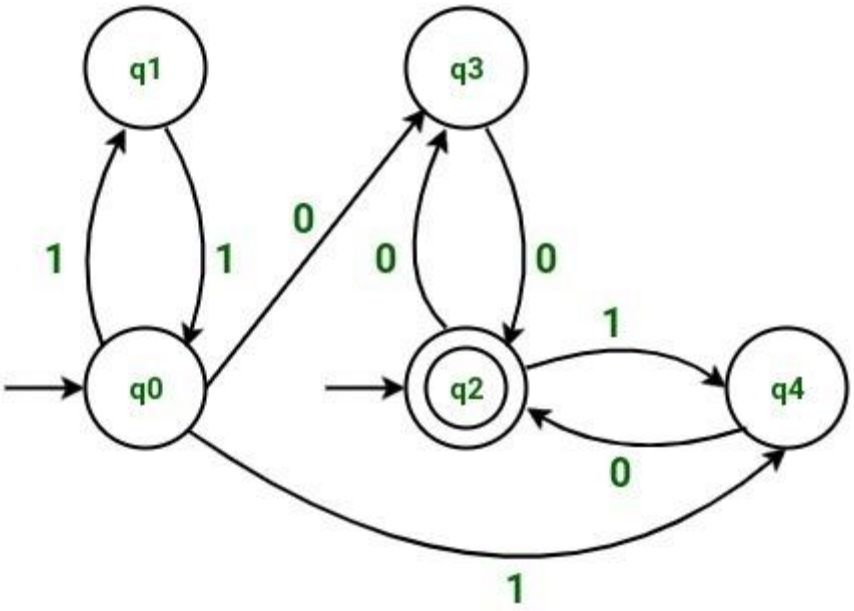


Figure – NFA on duplicating moves

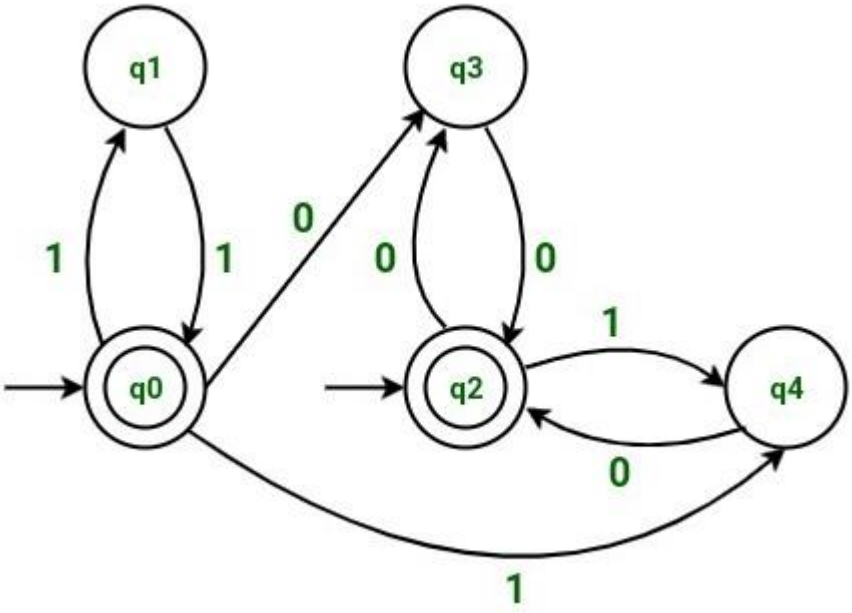
Step-3: Since vertex v1 *(i.e. state q0)* is a start state. Hence we will also make vertex v2 *(i.e. state q2)* as a start state. Note that state q2 will also remain as a final state as we had initially. NFA after making state q2



also as a start state is: **Figure –** NFA

after making state q2 as a start state

Step-4: Since vertex v2 *(i.e. state q2)* is a final state. Hence we will also make vertex v1 *(i.e. state q0)* as a final state. Note that state q0 will also remain as a start state as we had initially. After making state q0 also



as a final state, the resulting NFA is:

Figure – Resulting NFA (state q0 as a final state)The transition table for the above resulting NFA is:

States/Input	Input 0	Input 1
q0	q3	q1,q4
q1	–	q0

States/Input	Input 0	Input 1
q2	q3	q4
q3	q2	–
q4	q2	–

Conversion from NFA to DFA

Steps for converting NFA to DFA:

Step 1: Convert the given NFA to its equivalent transition table
To convert the NFA to its equivalent transition table, we need to list all the states, input symbols, and the transition rules. The transition rules are represented in the form of a matrix, where the rows represent the current state, the columns represent the input symbol, and the cells represent the next state.
Step 2: Create the DFA's start state
The DFA's start state is the set of all possible starting states in the NFA. This set is called the "epsilon closure" of the NFA's start state. The epsilon closure is the set of all states that can be reached from the start state by following epsilon (?) transitions.
Step 3: Create the DFA's transition table
The DFA's transition table is similar to the NFA's transition table, but instead of individual states, the rows and columns represent sets of states. For each input symbol, the corresponding cell in the transition table contains the epsilon closure of the set of states obtained by following the transition rules in the NFA's transition table.
Step 4: Create the DFA's final states
The DFA's final states are the sets of states that contain at least one final state from the NFA.
Step 5: Simplify the DFA
The DFA obtained in the previous steps may contain unnecessary states and transitions. To simplify the DFA, we can use the following techniques:

- Remove unreachable states: States that cannot be reached from the start state can be removed from the DFA.
- Remove dead states: States that cannot lead to a final state can be removed from the DFA.
- Merge equivalent states: States that have the same transition rules for all input symbols can be merged into a single state.

Step 6: Repeat steps 3-5 until no further simplification is possible
After simplifying the DFA, we repeat steps 3-5 until no further simplification is possible. The final DFA obtained is the minimized DFA equivalent to the given NFA.
Example: Consider the following NFA shown in Figure 1.

AUTOMATA THEOARY AND COMPILER DESIGN

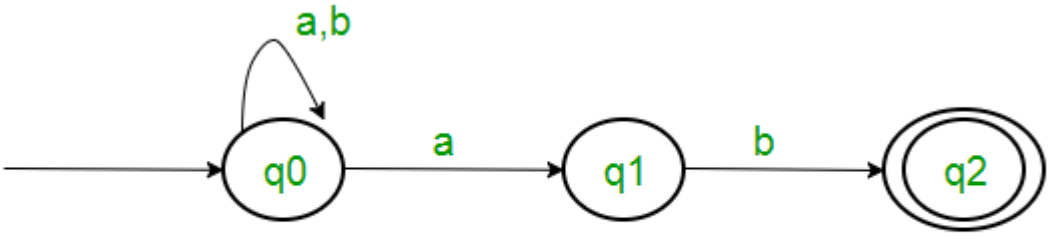


Figure 1

Following are the various parameters for NFA. Q = { q0, q1, q2 } ? = (a, b) F = { q2 } ? (Transition Function of NFA)

State	a	b
q0	q0,q1	q0
q1		q2
q2		

Step 1: Q' = ? Step 2: Q' = {q0} Step 3: For each state in Q', find the states for each input symbol. Currently, state in Q' is q0, find moves from q0 on input symbol a and b using transition function of NFA and update the transition table of DFA. ?' (Transition Function of DFA)

State	a	b
q0	{q0,q1}	q0

Now { q0, q1 } will be considered as a single state. As its entry is not in Q', add it to Q'. So Q' = { q0, { q0, q1 } } Now, moves from state { q0, q1 } on different input symbols are not present in transition table of DFA, we will calculate it like: ?' ({ q0, q1 }, a) = ? (q0, a) ? ? (q1, a) = { q0, q1 } ?' ({ q0, q1 }, b) = ? (q0, b) ? ? (q1, b) = { q0, q2 } Now we will update the transition table of DFA. ?' (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}

Now { q0, q2 } will be considered as a single state. As its entry is not in Q', add it to Q'. So Q' = { q0, { q0, q1 }, { q0, q2 } } Now, moves from state {q0, q2} on different input symbols are not present in transition table of DFA, we will calculate it like: ?' ({ q0, q2 }, a) = ? (q0, a) ? ? (q2, a) = { q0, q1 } ?' ({ q0, q2 }, b) = ? (q0, b) ? ? (q2, b) = { q0 } Now we will update the transition table of DFA. ?' (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

As there is no new state generated, we are done with the conversion. Final state of DFA will be state which has q2 as its component i.e., { q0, q2 } Following are the various parameters for DFA. Q' = { q0, { q0, q1 }, { q0, q2 } } ? = (a, b) F = { { q0, q2 } } and transition function ?' as shown above. The final DFA for above NFA has been shown in Figure 2.

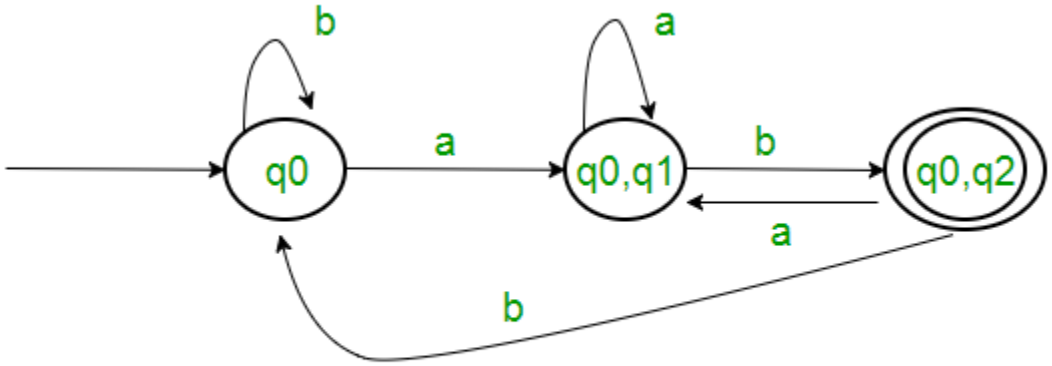


Figure 2

Note : Sometimes, it is not easy to convert regular expression to DFA. First you can convert regular expression to NFA and then NFA to DFA.

Question : The number of states in the minimal deterministic finite automaton corresponding to the regular expression (0 + 1)* (10) is _____.

Solution : First, we will make an NFA for the above expression. To make an NFA for (0 + 1)*, NFA will be in same state q0 on input symbol 0 or 1. Then for concatenation, we will add two moves (q0 to q1 for 1 and q1 to q2 for 0) as shown in Figure 3.

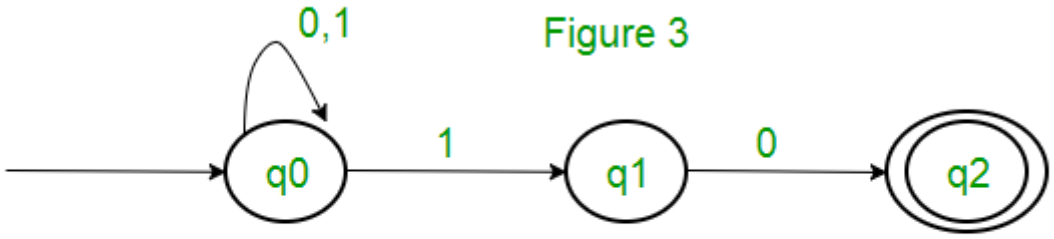


Figure 3

Using above algorithm, we can convert NFA to DFA as shown in Figure 4

State	0	1
q0	q0	q0,q1
q0,q1	q0,q2	q0,q1
q0,q2	q0	q0,q1

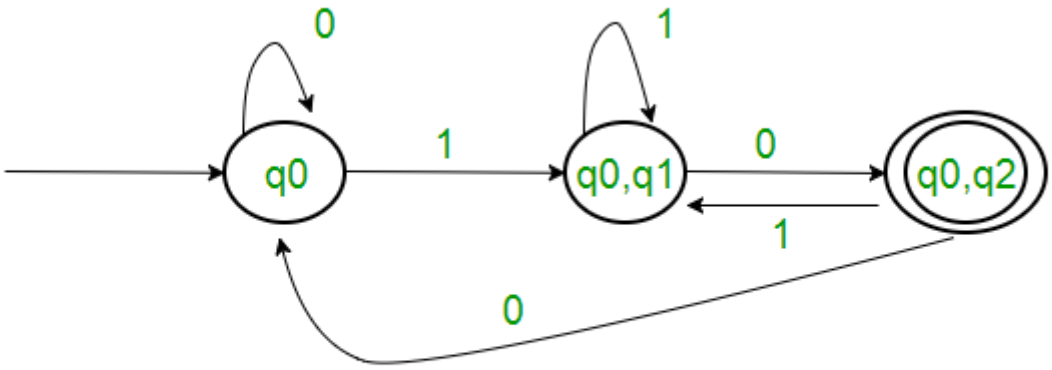


Figure 4

Applications of Automata

Automata is a machine that can accept the Strings of a *Language L* over an *input alphabet Σ* . So far we are familiar with the Types of Automata. Now, let us discuss the expressive power of Automata and further understand its Applications.

Automata have various applications in computer science, such as **finite automata** for pattern recognition and lexical analysis, **pushdown automata** in parsing and syntax analysis,

and **Turing machines** for simulating algorithms and complex computation. These models are essential for formal language processing, compilers, and artificial intelligence.

Expressive Power of various Automata: The Expressive Power of any machine can be determined from the class or set of Languages accepted by that particular type of Machine. Here is the increasing sequence of expressive power of machines :

FA<DPDA<PDA<LBA<TM $FA<DPDA<PDA<LBA<TM$

As we can observe that FA is less powerful than any other machine. It is important to note that DFA and NFA are of same power because every NFA can be converted into DFA and every DFA can be converted into NFA . The Turing Machine i.e. TM is more powerful than any other machine

(i) Finite Automata (FA) equivalence:

```
Finite Automata
≡ PDA with finite Stack
≡ TM with finite tape
≡ TM with unidirectional tape
≡ TM with read only tape
```

(ii) Pushdown Automata (PDA) equivalence:

```
PDA ≡ Finite Automata with Stack
```

(iii) Turing Machine (TM) equivalence:

```
Turing Machine
≡ PDA with additional Stack
≡ FA with 2 Stacks
```

The **Applications** of these Automata are given as follows:

1. Finite Automata (FA) –

- For the designing of lexical analysis of a compiler.
- For recognizing the pattern using regular expressions.
- For the designing of the combination and sequential circuits using Mealy and Moore Machines.
- Used in text editors.
- For the implementation of spell checkers.
- Can be used as a model for learning and decision making.
- Can parse text to extract information and structure data.

2. Push Down Automata (PDA) –

- For designing the parsing phase of a compiler (Syntax Analysis).
- For implementation of stack applications.
- For evaluating the arithmetic expressions.
- For solving the Tower of Hanoi Problem.
- Can be used in software engineering, to verify and validate the correctness of software models.
- Can be used in network protocols, to parse and validate incoming messages and to enforce specific message formats.
- Can be used in cryptography, to implement secure algorithms for encryption and decryption.
- Used in string matching and pattern recognition, to search for specific patterns in input strings.
- Used in XML parsing.
- Used in natural language processing applications like parsing sentences, recognizing parts of speech, and generating syntax trees.
- Used in automatic theorem proving and formal verification.
- Used in formal verification of hardware and software systems.

3. Linear Bounded Automata (LBA) –

- For implementation of genetic programming.
- For constructing syntactic parse trees for semantic analysis of the compiler.
- For recognition of context-sensitive languages.
- Used in game theory to model and analyze interactions between agents.

4. Turing Machine (TM) –

- For solving any recursively enumerable problem.
- For understanding complexity theory.
- For implementation of neural networks.
- For implementation of Robotics Applications.
- For implementation of artificial intelligence.
- Used as a theoretical model to analyze the time and space complexity of algorithms.
- Used in computational biology to model and analyze biological systems.
- Used in artificial intelligence as a model for learning and decision making.
- Used to study the relationship between classical computing and quantum computing.
- Used in digital circuit design to model and verify the behavior of digital circuits.

AUTOMATA THEOARY AND COMPILER DESIGN

- Used in human-computer interaction to model and analyze the interaction between humans and computers.