----------MODULE-1----------

Introduction to Automata Theory: Central Concepts of Automata theory, Deterministic Finite Automata (DFA), Non- Deterministic Finite Automata(NFA) ,Epsilon- NFA, NFA to DFA Conversion, Minimization of DFA

Introduction to Compiler Design: Language Processors, Phases of Compilers

INTRODUCTION TO FINITE AUTOMATA

Introduction: Automata theory is the study of abstract computing devices, or machines. Automata are essential for the study of the limits of computations. Finite Automata are a useful model for many important parts of hardware and software.

Alphabets: A symbol is an abstract entity. Letters and digits are examples of frequently used symbol. An alphabet is a finite, non-empty set of symbol and is denoted by ∑.

Example :∑= {0,1} , ∑={a,b,c}

Strings: *is a finite set sequence of symbols chosen from alphabet.*

Example: 011101 is a string from alphabet ∑= *{0,1}.*

Operations on string:

Concatenation: of two strings is formed by writing first string followed by second string with no space.*Ex. V='a', W='cat' then V.W='acat'*

Reverse: of the string W is obtained by writing the symbols of the string in reverse order and is denoted as W^R. *ex. W='the' then* W^R*='eht'.*

Length: of a string W, denoted |W| is the number of symbols composing the string. *Ex.W='the' then |W|=3*

Empty String: Denoted by Ɛ is the string with zero symbol.

Power of Alphabets:if ∑ is alphabet, the set of all strings of certain length can be expressed from that alphabet by using exponential notation. Ex. ∑={a,b} then

$\sum^0 = \{ \varepsilon\}$

$\sum^1=\{a,b\}$

$\sum^2$={aa,ab,bb,ba}

$\sum^3$ ={aaa,aab,aba,abb,bbb,baa,bab,bba}

$\sum^* = \sum^0 U \sum^1 U \sum^2 U \ldots \sum^n$

Language: set of strings, all are chosen from $\sum^*$, where $\sum$ is a particular alphabet is called a language.

$L \subseteq \sum^*$.

Empty Language: represented as Ø where Ø={ }, does not contain any element.

Introduction to Finite Automata and formal language

An automaton is a construct that possesses all the indispensable features of a digital computer. It accepts input, produces output, may have some temporary storage and can make decisions in transforming the input into the output.

A formal language is an abstraction of the general characteristics of programming languages. A formal language consists of a set of symbols and some rules of by which these symbols can be combined into entities called sentences.

Finite automata are computing devices that accept/recognize regular languages and are used to model operations of many systems. Their operations can be simulated by a very simple computer program.

Automaton:

A finite automaton (FA, also called a *finite-state automaton* or a *finite-state machine*) is a mathematical tool used to describe processes involving inputs and outputs. An FA can be in one of several states and can switch between states depending on symbols that it inputs. Once it settles in a state, it reads the next input symbol, performs some computational task associated with the new input, outputs a symbol, and switches to a new state depending on the input. Notice that the new state may be identical to the current state.

DFA: Deterministic Finite Automata

Definition: DFA is a finite automaton in which for each input symbol there is exactly one transition out of each state

A DFA is 5-tuple or quintuple **M = (Q, ∑, δ, q0, F)** where

Q is non-empty, finite set of states.

∑ is non-empty, finite set of input alphabets.

δ is transition function, which is a mapping from Q X ∑ -> Q.

$q_0 \in Q$ is the start state.

$F \subseteq Q$ is set of accepting or final states.

Note: For each input symbol *a*, from a given state there is exactly one transition (there can be no

transitions from a state also) and it is sure (or can determine) to which state the machine enters. So, the machine is called *Deterministic machine*. Since it has finite number of states the machine is called ***Deterministic finite machine or Deterministic Finite Automaton*** or Finite State Machine (FSM).

The language accepted by DFA is

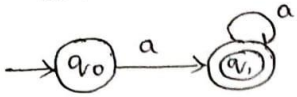
$L(M) = \{ w \mid w \in \sum^* \text{ and } \delta^*(q_0, w) \in F \}$

Designing a FA: The automaton can be described by

1. *Transition diagram:* is a graph
   a. For each state in Q there is a node
   b. For each state q in Q and each input symbol a in $\sum$,let $\delta(q,a) = p$, then the transition diagram has an arc from node ***p*** labeled ***a.***
   c. There is an arrow into the start state ***q0*** labeled start
   d. Nodes corresponding to accepting states are marked by double circle
2. *Transition table* : is a conventional tabular representation of a function like δ that take two arguments and returns a value
   a. Rows of the table corresponds to the states and the columns correspond to the input

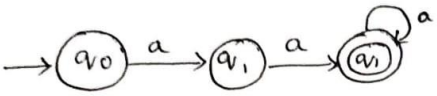
Examples

1. Draw a DFA to accept string of a's having atleast one 'a'.



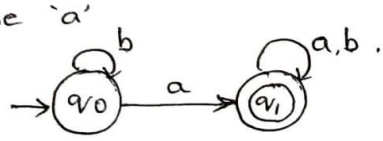
δ	a
→ q_0	q_1
* q_1	q_1

2. Draw a DFA to accept string of a's having atleast 2 a's.

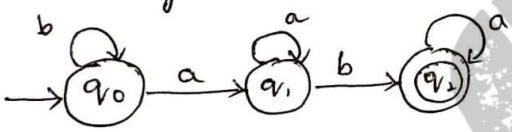


δ	a
→ q_0	q_1
q_1	q_2
* q_2	q_2

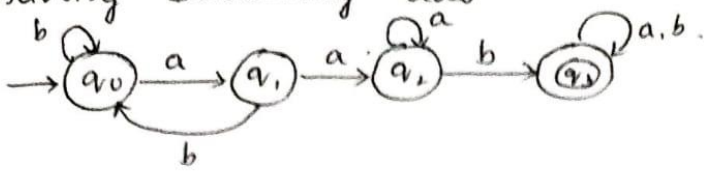
3. Draw DFA to accept strings of a's & b's having atleast one 'a'



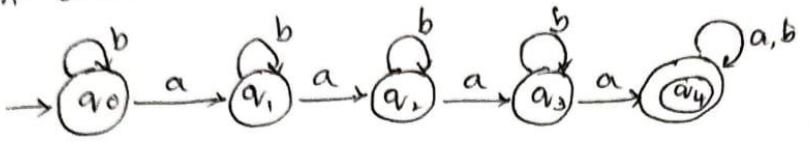
4. Draw a DFA to accept string of a's & b's having substring 'ab'



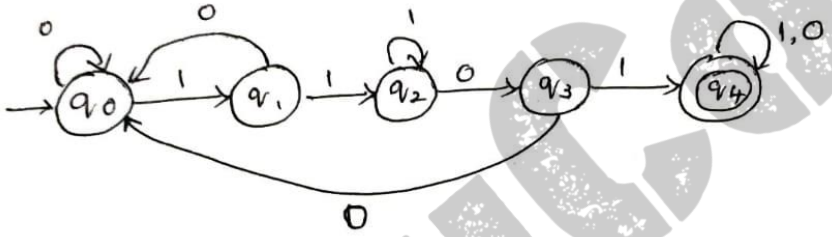
5. Draw a DFA to accept strings of a's & b's having substring 'aab'



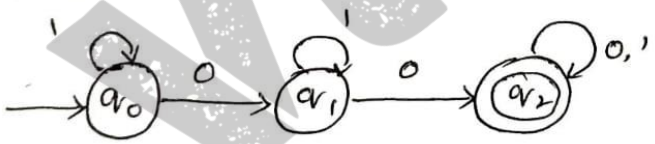
6. Obtain DFA to accept strings of a's & b's having 4 a's.



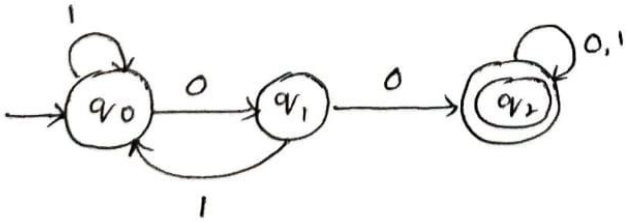
7. Obtain DFA to accept the set of all strings containing 1101 as a substring.



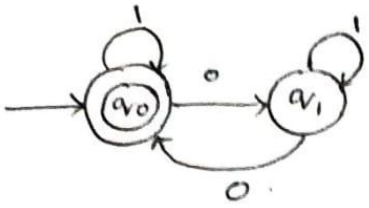
8(a) Obtain DFA to accept set of all strings containing atleast 2 0's. [not consecutive].



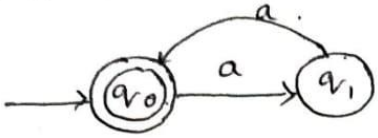
(b) 2 consecutive 0's.



9. Language of all strings in which the number of 0's is even.

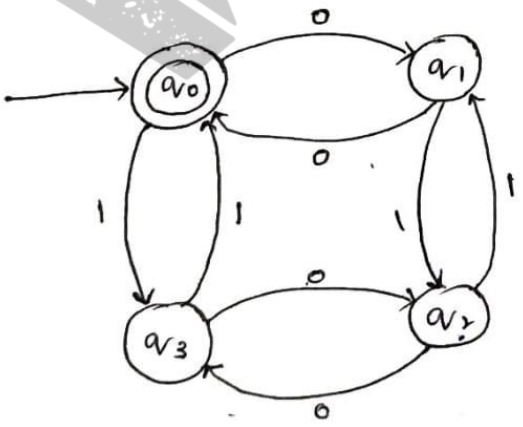


10. Obtain a dfa that recognises all even number of a's where $\Sigma = \{a\}$.

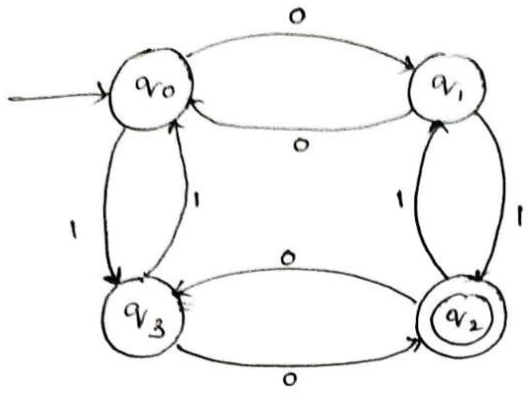


10. Obtain a DFA that recognises all odd number of a's where $\Sigma = \{a\}$.

11. Language of all strings in which both the number of 0's & 1's are even.

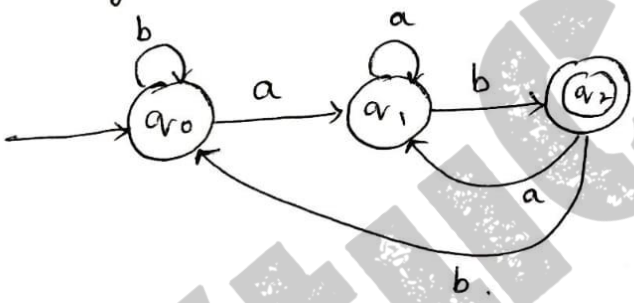


12. Language of all strings in which both 0's & 1's are odd.

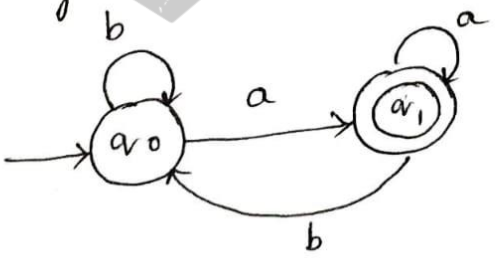


Ending

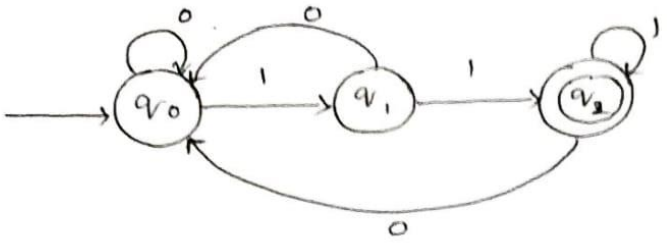
2. Draw a dfa to accept the strings of a & b ending with ab $\Sigma = \{a, b\}$.



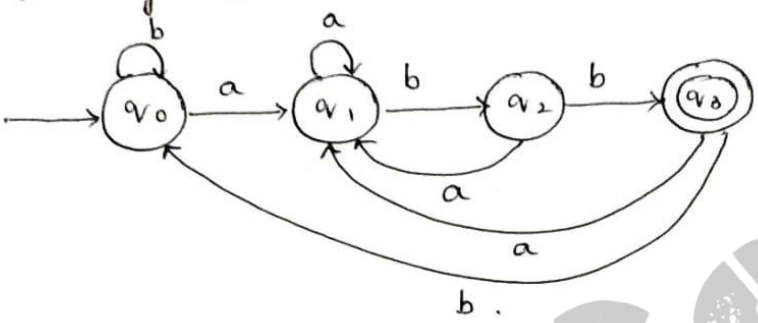
1. Draw a dfa to accept the strings of a & b ending with 'a'.



3. Obtain dfa accepting the set of all strings that end with 11.

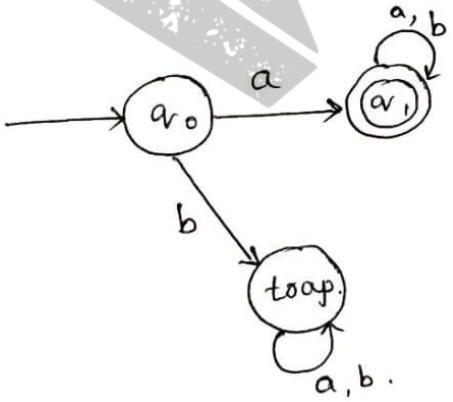


4. Draw a dfa to accept the strings of a & b ending with abb. $\Sigma = \{a, b\}$.

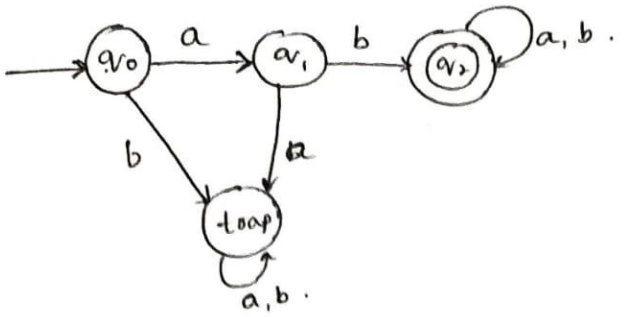


Starting :-

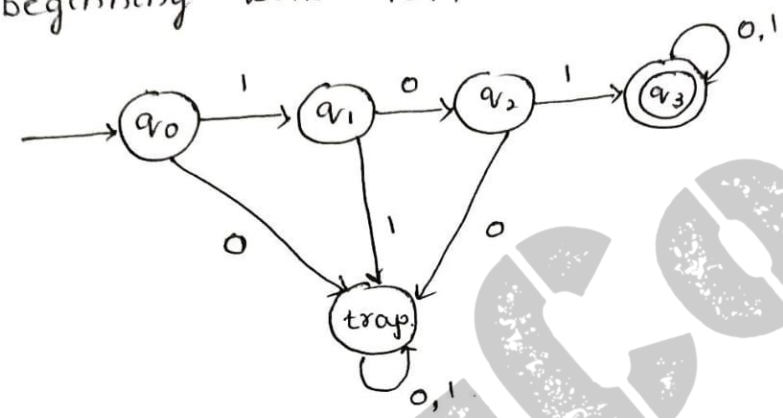
1. Draw a dfa to accept the set of all strings that starts with 'a'. $\Sigma = \{a, b\}$.



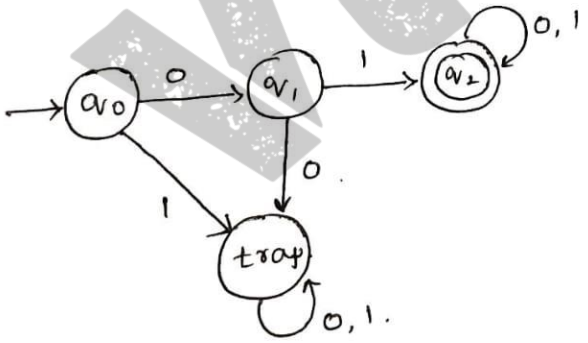
2. Draw a dfa to start with 'ab'



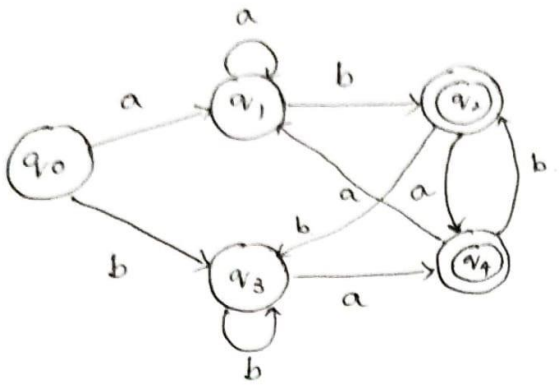
3. obtain a DFA accepting the set of all strings beginning with 101.



4. obtain DFA accepting the set of all strings that begin with 01

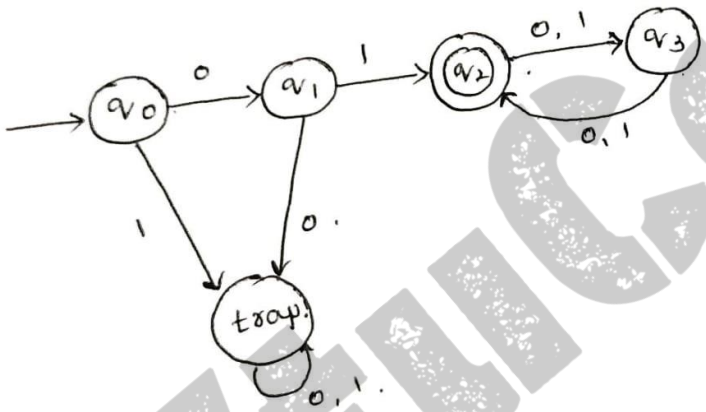


Construct a DFA to accept strings of a's & b's ending with ab or ba.



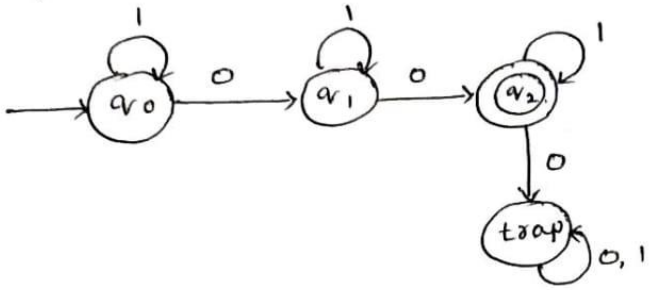
Design a DFA to accept the language.

$L = \{w \mid w$ is of even length & begins with 01$\}$.



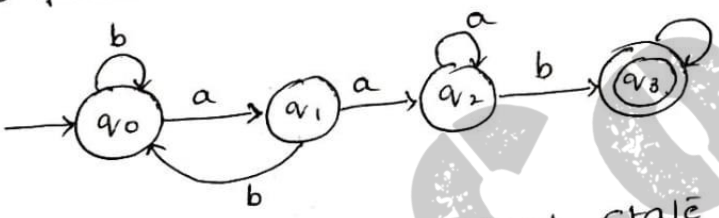
Misc:-

1. Language of all strings containing exactly two 0's.

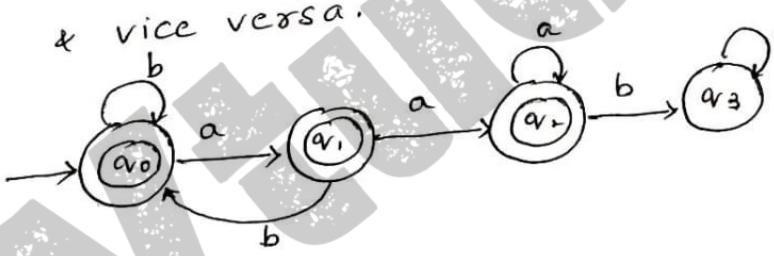


2. Obtain a DFA to accept strings of a's & b's except those containing substring "aab"

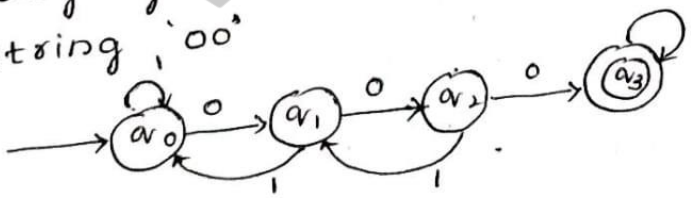
(i) find DFA to accept substring 'aab'.



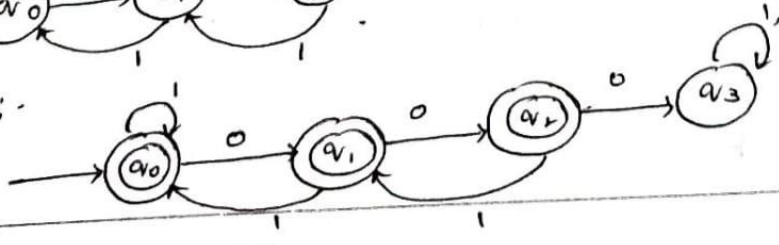
(ii) Now change all final state to non-final state & vice versa.



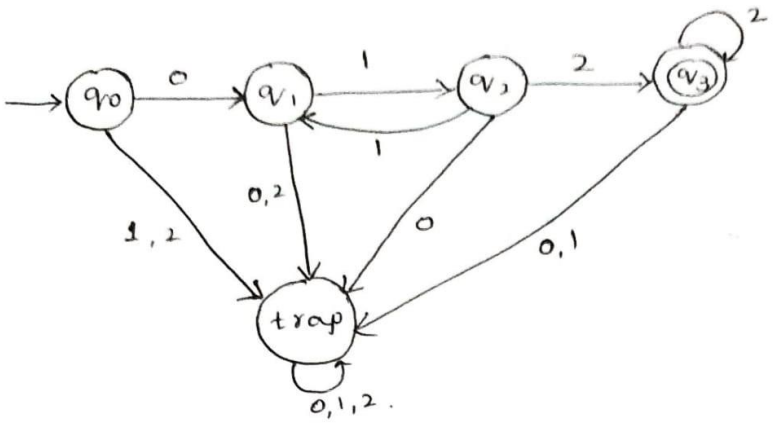
3. Language containing no more than 1 occurence of string '00'



vice versa:-

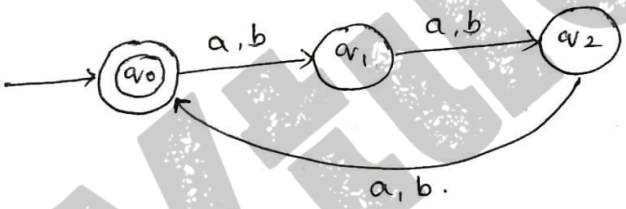


1. Obtain a DFA to accept strings of 0's, 1's & 2's beginning with '0' followed by odd number of 1's and ending with 2.



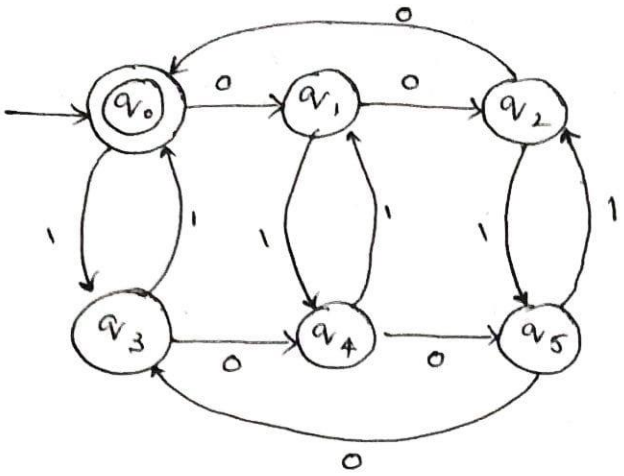
8. Obtain DFA to accept the language $L = \{w : |w| \bmod 3 = \}$ on $\Sigma = \{a, b\}$.

$L = \{\in, aaa, aab, aba, abb, baa, bab, \ldots\}$.



9. obtain DFA to accept strings of a's & b's such that

obtain DFA accepting the set of all strings such that number of 1's is even & the number of 0's is multiple of 3.

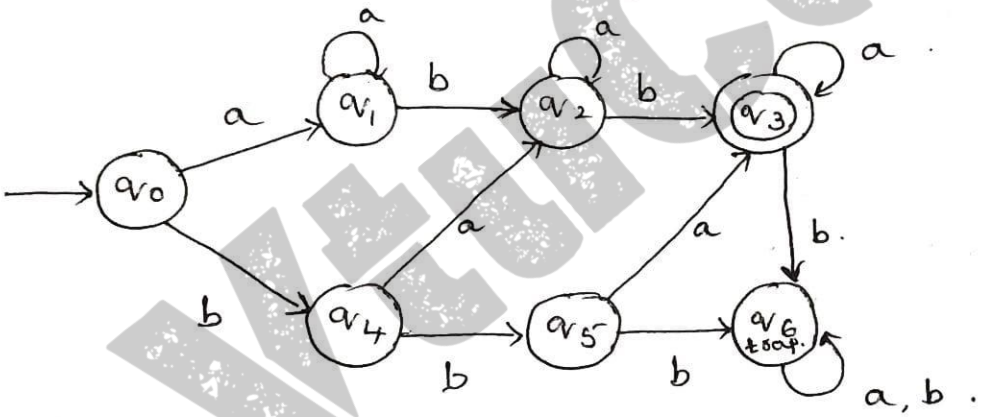


$n_0(w) \bmod 3 = 0$

$n_1(w) \bmod 2 = 0$.

Draw a DFA to accept the language

$L = \{w \mid n_a(w) \geq 1, n_b(w) = 2\}$.



Nondeterministic Finite Automaton: NFA

NFA is defined as follows: at some point in processing a string on a machine, the machine has a choice of moves; when this happens, it selects a move in an unspecified way. In other words, there can be zero, one or more than one transition out of a state with the same label. So the machine must choose which path to take. This leads us to the observation that a string may have more than one path through the machine that is each entry in the table for NFA is a set. This represents a relaxation of the rules for defining FA's.

Definition: An NFA is a 5-tuple or quintuple M = (Q, ∑, δ, q_0, F) where

Q is non empty, finite set of states.

∑ is non empty, finite set of input alphabets.

δ is transition function which is a mapping from Q X ∑ to subsets of 2^Q. This function shows the change of state from one state to a set of states based on the input symbol.

$q_0 \in Q$ is the start state.

$F \subseteq Q$ is set of accepting or final states

Language Accepted by NFA:

Definition: Let M = (Q, ∑, δ, q_0, A) be a DFA where Q is set of finite states, ∑ is set of input alphabets (from which a string can be formed), δ is transition function from Q x {∑U ε} to 2^Q, q_0 is the start state and A is the final or accepting state. The string (also called language) *w* accepted by an NFA can be defined in formal notation as:

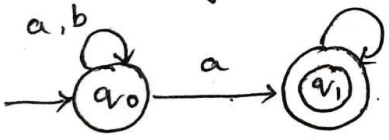
L(M) = { w | w ∈ ∑*and δ*(q_0, w) = Q with at least one Component of Q in A}

Examples

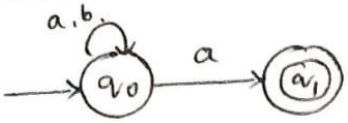
I obtain an NFA to accept set of all strings over the alphabet $\Sigma = \{a, b\}$.

(a) starting with 'a'.

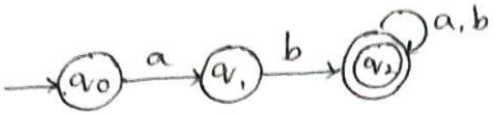
(b) containing 'a'



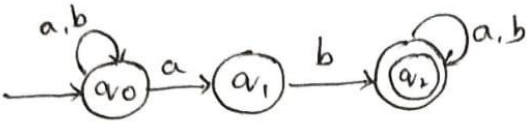
(c) Ends with 'a'



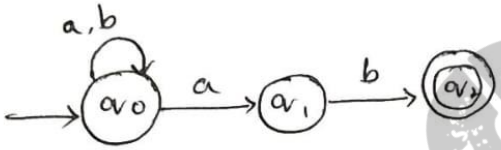
(d) starts with a, b.



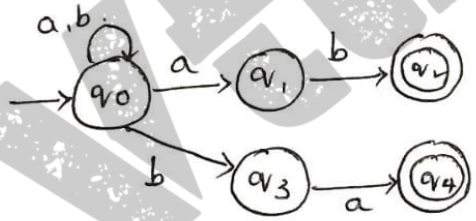
(e) contain 'ab'



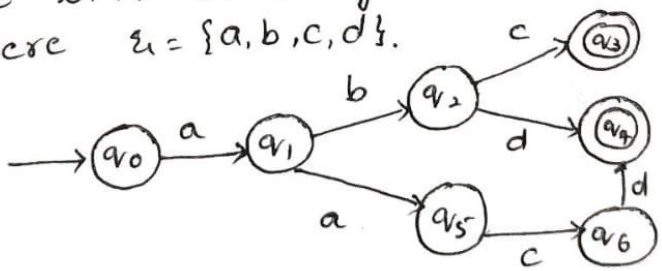
(f) Ends with ab.



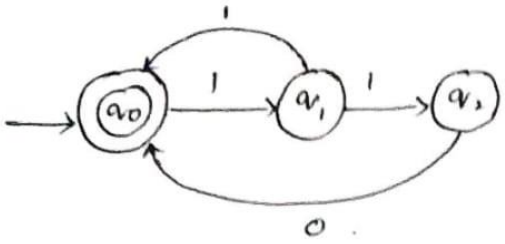
2. Obtain an NFA to accept strings ending with 'ab' or 'ba' over $\Sigma = \{a, b\}$.



3. Write NFA to recognise the string abc, abd, aacd where $\Sigma = \{a, b, c, d\}$.

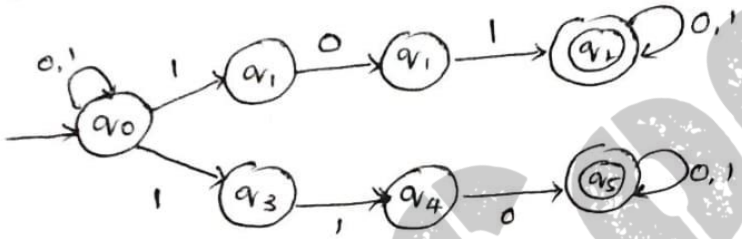


4. Write NFA to accept $\{11, 110\}^*$.

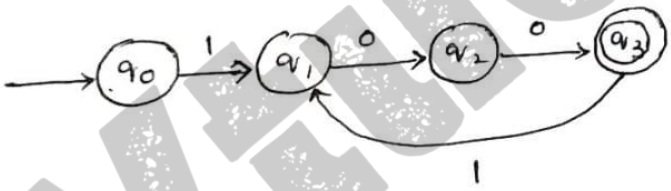


5. Write NFA to accept the following languages over $\{0, 1\}^*$.

(a) Set of all strings such that containing either 101 or 110 as substring.



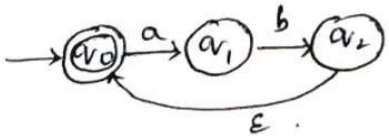
(b) Every ~~one~~ '1' followed by '00'.



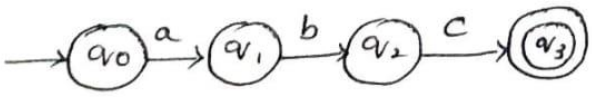
6. Design an NFA with 4 states that accepts the language $\{ab, abc\}^*$.

Sol: To accept ab.

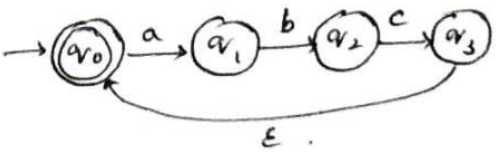
ab^*

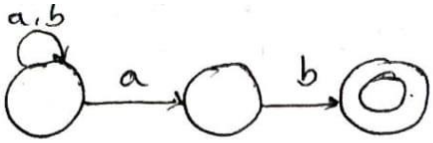


To accept abc.



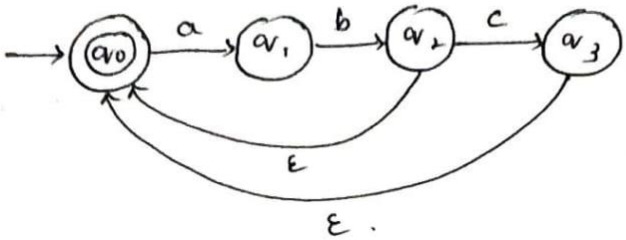
abc^*





abaab:

combining these 2.



7. Design NFA with no more than 5 states for the set $\{abab^n \mid n \geq 0\} \cup \{aba^n \mid n \geq 0\}$.

Sol: consider the set $\{ab\,ab^n \mid n \geq 0\}$

for $n=0$ the string accepted is aba.

Draw a path for string $aba\,b^n$ where $n \geq 0$.

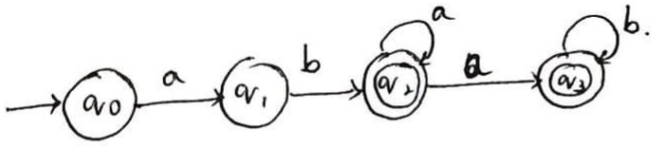
→ (1).

III[ly] for the set $\{aba^n \mid n \geq 0\}$.

for $n=0$ the string accepted is ab.

→ (2).

for set $\{abab^n\} \cup \{aba^n\}$. combine (1) & (2).



Extended Transition* δ: Describes what happens when we start in any state and follow sequence of inputs.

Definition:

Let **M = (Q, ∑, δ, q0, F)** where

Q is non-empty, finite set of states.

∑ is non-empty, finite set of input alphabets.

$q_0 \in Q$ is the start state.

$F \subseteq Q$ is set of accepting or final states.

δ* is extended transition function, which is a mapping from Q X ∑ -> Q. as follows:

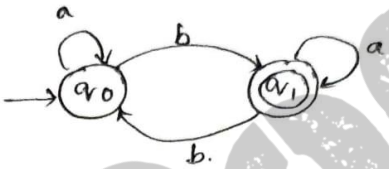
i. For any $q \in Q$, **$\delta^*(q, \in)=q$**

ii. For any $q \in Q$, $y \in \sum^*$, $a \in \sum$

$\delta^*(q, ya)= \delta(\delta^*(q,y),a)$

Example:

consider a DFA



Find whether the string **aabbab** is accepted.

Sol: y = aabbab.

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

$\delta^*(q_0, a) = \delta(\delta^*(q_0, \in), a) \Rightarrow \delta(q_0, a) = q_0$.

$\delta^*(q_0, aa) = \delta(\delta^*(q_0, a), a) \Rightarrow \delta(q_0, a) = q_0$.

$\delta^*(q_0, aab) = \delta(\delta^*(q_0, aa), b) \Rightarrow \delta(q_0, b) = q_1$

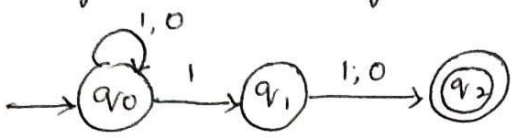
$\delta^*(q_0, aabb) = \delta(\delta^*(q_0, aab), b) \Rightarrow \delta(q_1, b) = q_0$.

$\delta^*(q_0, aabba) = \delta(\delta^*(q_0, aabb), a) \Rightarrow \delta(q_0, a) = q_0$.

$\delta^*(q_0, aabbab) = \delta(\delta^*(q_0, aabba), b) \Rightarrow \delta(q_0, b) = q_1$.

q_1 is final state hence the string aabbab is accepted.

Describe the processing of input 00101 by the following NFA using extended transition.



Sol^:

$\hat{\delta}(q_0, \epsilon) = \{q_0\}$.

$\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0\}$.

$\hat{\delta}(q_0, 00) = \hat{\delta}(\hat{\delta}(q_0, 0), 0) = \delta(q_0, 0) = \{q_0\}$

$\hat{\delta}(q_0, 001) = \delta(\hat{\delta}(q_0, 00), 1) = \delta(q_0, 1) = \{q_0, q_1\}$.

$\hat{\delta}(q_0, 0010) = \delta(\hat{\delta}(q_0, 001), 0) = \delta(\{q_0, q_1\}, 0) =$

$= \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_2$

$\hat{\delta}(q_0, 00101) = \delta(\hat{\delta}(q_0, 0010), 1) = \delta(\{q_0, q_2\}, 1)$

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

$= \{q_0, q_2\}$.

since $q_2 \in F$. string is accepted.

FINITE AUTOMATA WITH Ɛ TRANSITIONS

Ɛ-NFA Epsilon NFA

Is an NFA with Epsilon transitions. The NFA which includes transitions on the empty input Ɛ is called Ɛ-NFA

Definition: AnƐ-NFA is a 5-tuple or quintuple M = (Q, ∑, δ, q_0, F) where

Q is non empty, finite set of states.

∑ is non empty, finite set of input alphabets.

δ is transition function which is a mapping from Q X{∑UƐ} to subsets of 2^Q.

This function shows the change of state from one state to a set of states based on the input symbol.

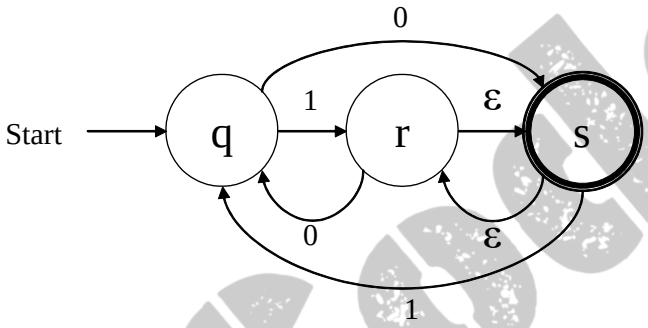
$q_0 \in Q$ is the start state.
$F \subseteq Q$ is set of accepting or final states

Ɛ-Closure of a state
Definition:

Ɛ-closure is a set of all vertices p such that there is a path from q to p labelled Ɛ.

Let M = (Q, ∑, δ, q_0, F) be an NFA with M = (Q, ∑, δ, q_0, F) transitions and let S be any subset of Q. The Ɛ-closure of S denoted as Ɛ(S) is defined by

1. Every element of S is an element of Ɛ(S).
2. For any q Є Ɛ(S) every element of δ(q, Ɛ) is in Ɛ(S)
3. No other element are in Ɛ(S)

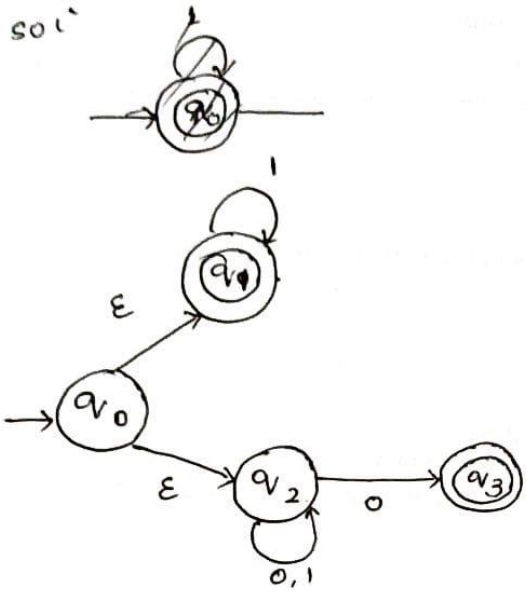


- *ε-closure(q) = { q }*
- *ε-closure(r) = { r, s}*

Examples

2. Draw a ε-NFA that accepts all binary strings where the last symbol is '0' or that contain only 1's.

Soln:



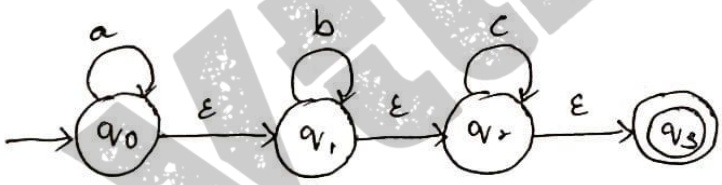
ε-closure $(q_0) = \{q_0, q_1, q_2\}$.

ε-closure $(q_1) = \{q_1\}$.

ε-closure $(q_2) = \{q_2\}$.

ε-closure $(q_3) = \{q_3\}$.

3. Design ε-NFA that accepts set of all strings consisting of zero or more a's followed by zero or more b's followed by zero ore more c's.



ε-closure $(q_0) = \{q_0, q_1, q_2, q_3\}$.

ε-closure $(q_1) = \{q_1, q_2, q_3\}$.

ε-closure $(q_2) = \{q_2, q_3\}$

ε-closure $(q_3) = \{q_3\}$.

Conversion from NFA-DFA (Subset Construction)

With an NFA, at any point in a scanning of the input string we may be faced with a choice of any number of paths to be followed to reach a final state. With a DFA there is never a choice of paths. So, when we construct a DFA which accepts the same language as a particular NFA, the conversion process effectively involves merging all possible states which can be reached on a particular input character, from a particular state, into a single, composite, state which represents all those paths.

Let $M_N = (Q_N, \sum, \delta_N, q_0, F_N)$ be an NFA and accepts the language $L(M_N)$. There should be an equivalent DFA $M_D = (Q_D, \sum_D, \delta_D, q_0, F_D)$ such that $L(M_D) = L(M_N)$. The procedure to convert an NFA to its equivalent DFA is shown below

Step1: The start state of NFA M_N is the start state of DFA M_D. So, add q_0(which is the start state of NFA) to Q_D and find the transitions from this state.

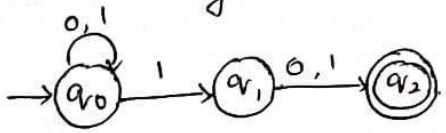
Step2: For each state $[q_i, q_j,\ldots.qk]$ in Q_D, the transitions for each input symbol in $\sum$ can be obtained as shown below:

- $\delta_D([q_i, q_j,\ldots.qk], a) = \delta_N(q_i, a) \cup \delta_N(q_j, a) \cup \ldots\ldots\delta_N(q_k, a)$
    - a =$[q_l,q_m,\ldots.q_n]$say.
- Add the state $[q_l, q_m,\ldots.q_n]$ to Q_D, if it is not already in Q_D.
- Add the transition from [qi, qj,….qk] to $[q_l, q_m,\ldots.q_n]$ on the input symbol α iff the state $[q_l, q_m,\ldots.q_n]$ is added to Q_D in the previous step.

Step3: The state $[q_a, q_b,\ldots.q_c] \in Q_D$ is the final state, if at least one of the state in $q_a, q_b, \ldots.. q_c \in A_N$ i.e., at least one of the component in $[q_a, q_b,\ldots.q_c]$ should be the final state of NFA.

Examples:-

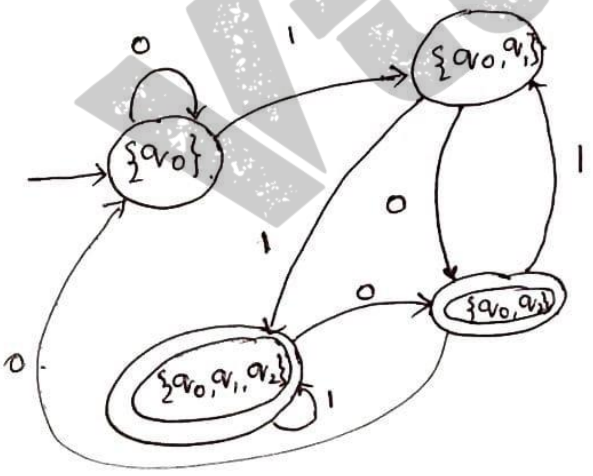
1. convert the given NFA to DFA.



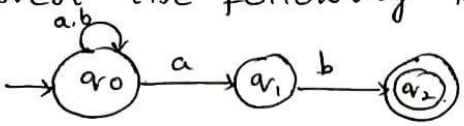
Soln:-

δ_{NFA}	0	1
→ q_0	$\{q_0\}$	$\{q_0, q_1\}$.
q_1	$\{q_2\}$	$\{q_2\}$.
q_2	ϕ	ϕ.

δ_{DFA}	0	1
→ $\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$.
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$.
* $\{q_0, q_2\}$	$\{q_0\}$	$\{q_0, q_1\}$.
* $\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$.



2. Convert the following NFA to DFA.

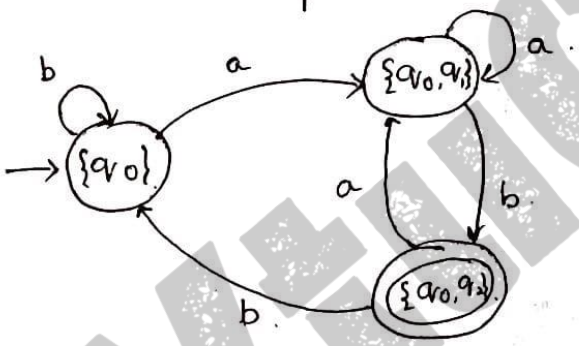


Sol^: given

δ_{NFA}	a	b
→ q_0	$\{q_0, q_1\}$	$\{q_0\}$.
q_1	ϕ	$\{q_2\}$.
q_2	ϕ	ϕ.

conversion

δ_{DFA}	a	b
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$.
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$.
*. $\{q_0, q_2\}$	$\{q_0, q_1\}$.	$\{q_0\}$.

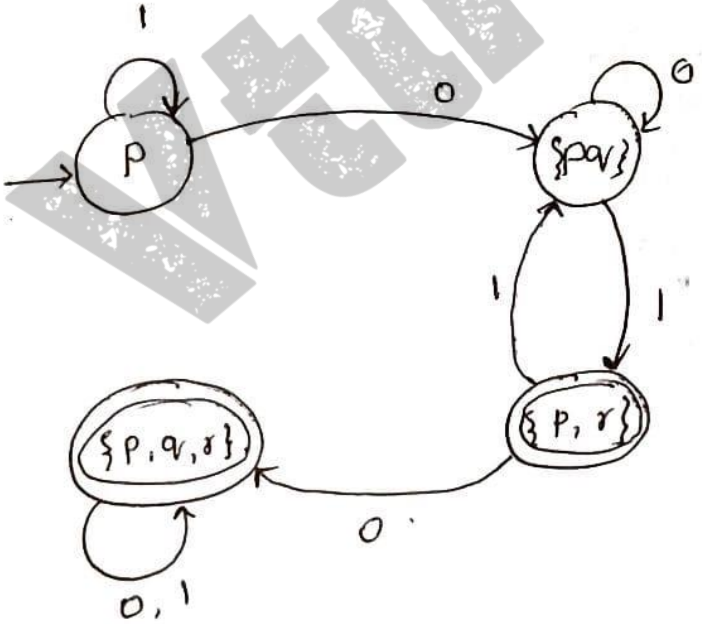


3. Convert the following NFA to DFA.

δ_{NFA}	0	1
→ p	{p, q}	{p}
q	q	{r}
* r	{p, r}	{q}

Sol^: -

δ_{DFA}	0	1
→ {p}	{p, q}	{p}
{p, q}	{p, q}	{p, r}
* {p, r}	{p, q, r}	{p, q}
* {p, q, r}	{p, q, r}	{p, q, r}



4. Convert the given NFA to DFA.

δ_{NFA}	0	1
→ p	{p, q}	{p}
q	{r, s}	{p}
* r	{p, s}	{r}
* s	{q, r}	ϕ

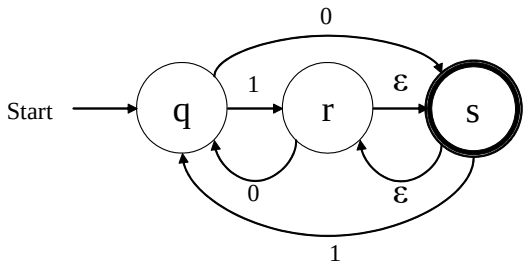
Sol^ :-

δ_{DFA}	0	1
→ {p}	{p, q}	{q}
{q}	{r, s}	{p}
{p, q}	{p, r, s}	{q, r}
{r, s}	{p, s, q, r}	{r}
{p, q, r, s}	{p, r, s, q}	{q, p, r}
{p, r, s}	{p q s}	{p, s}

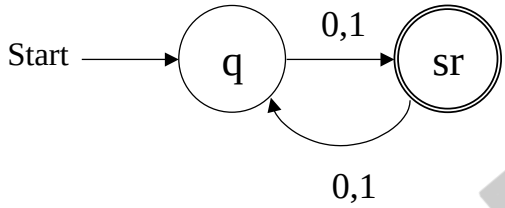
Conversion from Ɛ-NFA-DFA (Subset Construction)

To eliminate ε-transitions, use the following to convert to a DFA

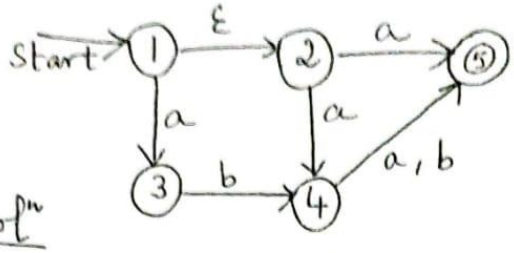
1. Compute ε-closure for the current state, resulting in a set of states S.
2. $\delta_D(S,a)$ is computed for all a in Σ by
   a. Let S = $\{p_1, p_2, \ldots p_k\}$
   b. Compute $\bigcup_{i=1}^{k} \delta(p_i, a)$ and call this set $\{r_1, r_2, r_3 \ldots r_m\}$ This set is achieved by following input a, not by following any ε-transitions
   c. Add the ε-transitions in by computing $\delta(S,a) = \bigcup_{i=1}^{m} \varepsilon - closure(r_i)$
3. Make a state an accepting state if it includes any final states in the ε-NFA.



Converts to



Convert the following ε - NFA to DFA



Soln

* The start state is the ε - closure of the start state of the ε-NFA
* The final states of the DFA will correspond to the subsets of the ε-NFA that contain a final state.

E - CLOSURE (1) = {1, 2}

E - CLOSURE of (δ(1, a) ∪ δ(2, a))

= E-CLOSURE of ({3} ∪ {4, 5})

= E - CLOSURE of {3, 4, 5} = {3, 4, 5}

E - CLOSURE of (δ(1, b) ∪ δ(2, b))

= E - CLOSURE of ({ϕ} ∪ {ϕ})

= {ϕ}

E - CLOSURE of (δ(3, a) ∪ δ(4, a) ∪ δ(5, a))

= E - CLOSURE of ({ϕ} ∪ {5} ∪ {ϕ})

= {5}

E-CLOSURE of $(\delta(3,b) \cup \delta(4,b) \cup \delta(5,b))$

$= E\text{-CLOSURE of } (\{4\} \cup \{5\} \cup \{\phi\})$

$= E\text{-CLOSURE of } (\{4,5\})$

$= \{4,5\}$

E-CLOSURE of $(\delta(5,a))$

$= \{\phi\}$

E-CLOSURE of $(\delta(5,b)) = \{\phi\}$

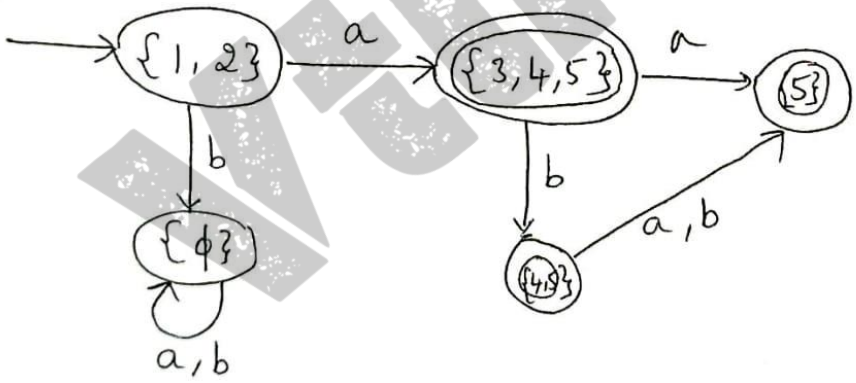
E-CLOSURE of $(\delta(4,a) \cup \delta(5,a))$

$= E\text{-CLOSURE of } (\{5\} \cup \{\phi\})$

$= \{5\}$

E-CLOSURE of $(\delta(4,b) \cup \delta(5,b))$

$= E\text{-CLOSURE of } (\{5\} \cup \{\phi\}) = \{5\}$



One can show the connection between state {5} & {φ} or rule out.

If $M_D = (Q_D, \sum_D, \delta_D, q_0, F_D)$ is the DFA constructed from NFA $M_N = (Q_N, \sum, \delta_N, q_0, F_N)$ by the subset construction, then $L(M_D) = L(M_N)$.

Proof: Let |w| =0, that is w= ε. By the basis definitions of **δ*** for DFA's and NFA's both δ*({ q_0 }, ε) and δ*(q_0, ε) are {q_0}

Let w be of length n+1, and assume the statement for length *n*. break w as w=xa, where a is the final symbol of w. by the inductive hypothesis δ*({ q_0 }, x)= δ*(q_0, x). let both these sets of N's states be {$p_1, p_2, \ldots p_k$}

The inductive part of the definition of δ*for NFA's tells that:

$$\delta^*(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \text{ ----------1}$$

The subset construction, on the other hand, tells that :

$$\delta_D(\{p_1, p_2, \ldots p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \text{----------2}$$

Using eqn 2 and the fact that δ*({ q_0 }, x)={$p_1, p_2, \ldots p_k$}in the inductive part of the definition of δ* for DFA's

$$\delta^*(\{q_0\}, x) = \delta_D(\delta_D^*(\{q_0\}, x), a) = \delta_D(\{p_1, p_2, \ldots p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \text{---------3}$$

Thus equation 2 and 3 demonstrate that $\delta^*(\{q_0\}, w) = \delta_N^*(\{q_0\}, w)$

Hence proved that $L(M_D) = L(M_N)$.

Difference between DFA, NFA, ε-NFA

DFA	NFA	ε-NFA
DFA is a 5-tuple **M = (Q, ∑, δ, q0, F) where** • Q is non-empty, finite set of states. • ∑ is non-empty, finite set of input alphabets. • δ : **Q X ∑ -> Q**. • $q_0 \in Q$ is the start state. • F⊆ Q is set of accepting or final states.	M = (Q, ∑, δ, q_0, F) where • Q is non empty, finite set of states. • ∑ is non empty, finite set of input alphabets. • δ : **Q X ∑ to 2^Q**. • $q_0 \in$ Q is the start state. • F ⊆ Q is set of accepting or final states	M = (Q, ∑, δ, q_0, F) where • Q is non empty, finite set of states. • ∑ is non empty, finite set of input alphabets • δ : from Q X{∑Uε} to 2^Q. • $q_0 \in$ Q is the start state. • F ⊆ Q is set of accepting or final states.

There can be zero or one transition from a state on an input symbol;	There can be zero, one or more number of transitions from a state on an input symbol	There can be zero, one or more number of transitions from a state with or without an input symbol
Difficult to design	The NFA are easier to design	Easy to construct using regular expression
More number of transitions	Less number of transitions	More number of transitions compared to NFA
Less powerful since at any point of time it will be in only one state	More powerful; than DFA since it can be in more than one state	More powerful than NFA since at any point of time it will be in more than one state with or without giving any input.

INTRODUCTION TO COMPILER DESIGN

OVERVIEW OF LANGUAGE PROCESSING SYSTEM

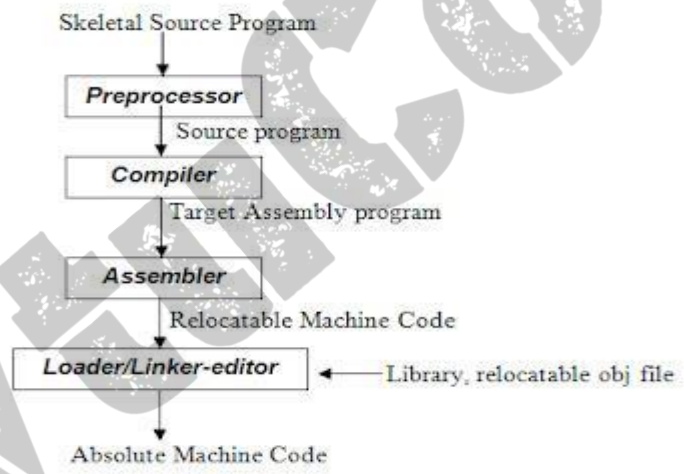
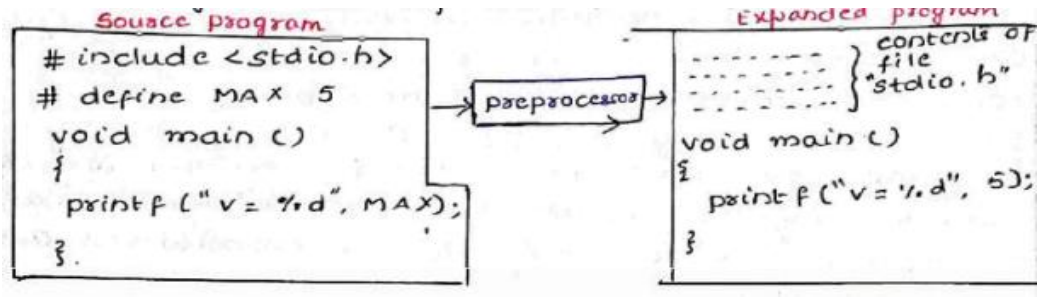


Fig 1.1 Language –processing System

Preprocessor

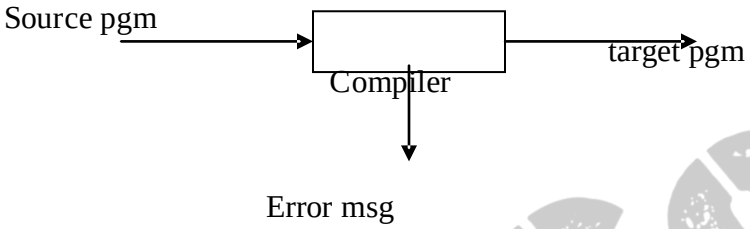
A preprocessor produce input to compilers. They may perform the following functions.

1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

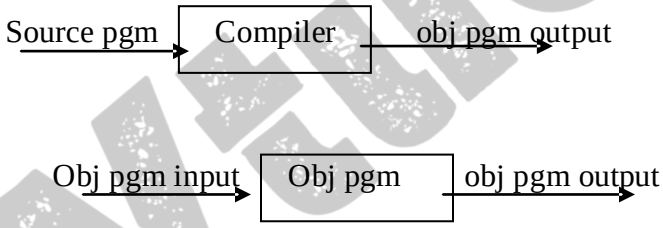


COMPILER

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



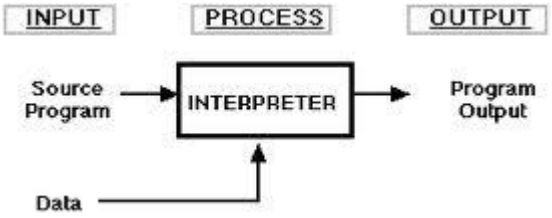
Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.



ASSEMBLER

programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

INTERPRETER: An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Direct Execution

Advantages:

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes a various may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for Interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages:

- The execution of the program is *slower*.
- *Memory* consumption is more

DIFFERENCE BETWEEN COMPILER AND INTERPRETER

Compiler	Interpreter
Compiler scans the whole program in one go.	Translates program one statement at a time.
As it scans the code in one go, the errors (if any) are shown at the end together.	Considering it scans code one line at a time, errors are shown line by line.
Main advantage of compilers is it's execution time.	Due to interpreters being slow in executing the object code, it is preferred less.
It converts the source code into object code.	It does not convert source code into object code instead it scans it line by line
It does not require source code for later execution.	It requires source code for later execution.
C, C++, C# etc.	Python, Ruby, Perl, SNOBOL, MATLAB, etc.

Loader and Link-editor:

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would

waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To over come this problem of wasted translation time and memory. System programmers developed another component called loader

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could "relocate" directly behind the user's program. The task of adjusting programs or they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of translator are:

1 Translating the hll program input into an equivalent ml program.

2 Providing diagnostic messages wherever the programmer violates specification of the hll.

TYPE OF TRANSLATORS:-

- INTERPRETOR
- COMPILER
- PREPROSSESSOR

STRUCTURE OF THE COMPILER DESIGN

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **'phases'**.

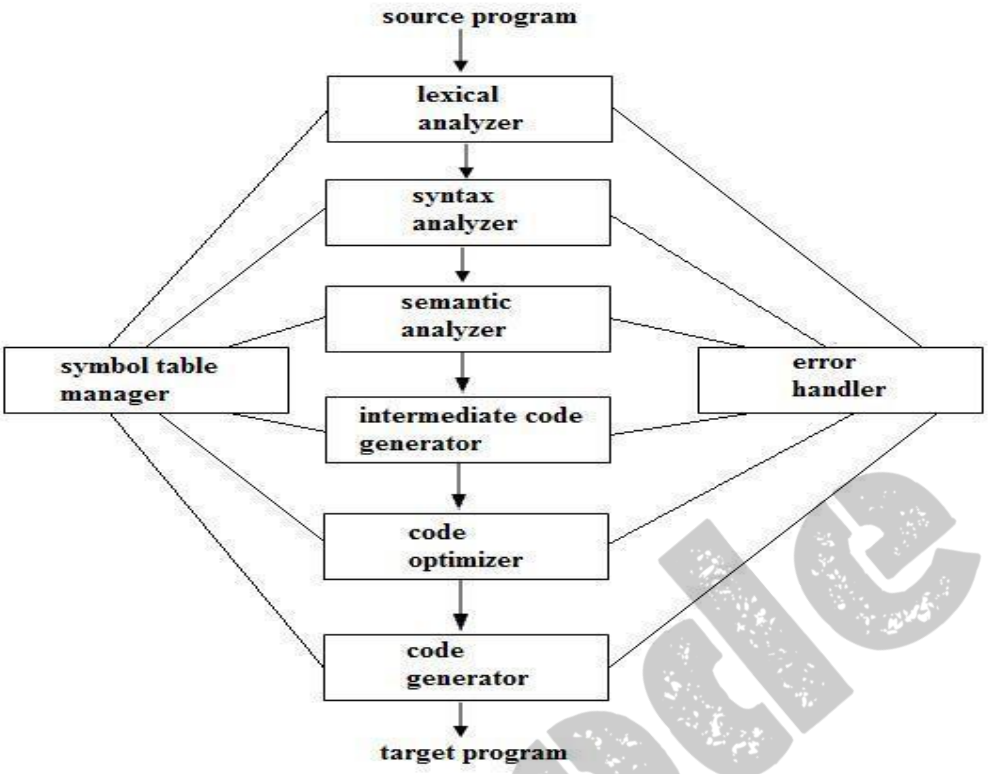


Fig 1.5 Phases of a compiler

Lexical Analysis:- Lexical Analyzer or *Scanners* reads the source program one character at a time, On reading character stream of source program, it groups them into meaningful sequences called "*Lexemes*". For each lexeme analyzer produces an output called **tokens.**

<token_name, attribute value>

Abstract symbol used in parser

points to entry in the symbol table for this token

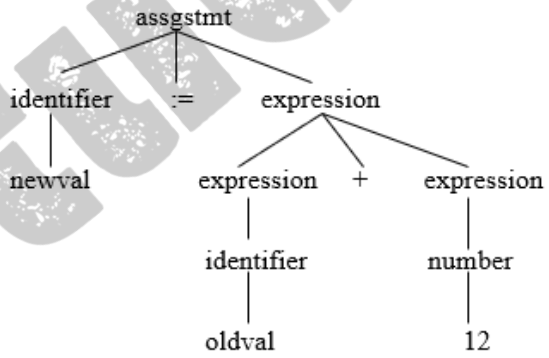
A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimiters and so on)

Ex newval := oldval + 12 => tokens:

newval	*identifier*
:=	*assignment operator*
oldval	*identifier*
+	*add operator*
12	*a number*

Syntax Analysis:-The second stage of translation is called Syntax analysis or *parsing*. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program.
A syntax analyzer is also called as a parser. A parse tree describes a syntactic structure.



Semantic Analysis: Uses syntax tree and information in symbol table to check source program for semantic consistency with language definition. It gathers type information and saves it in either syntax tree or symbol table for use in Intermediate code generation.
Type checking- compiler checks whether each operator has the matching operands.
Coercions-language specification may permit some type of conversion.

Intermediate Code Generations:- An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.