REGULAR EXPRESSIONS AND LANGUAGES:

Introduction

Instead of focusing on the power of a computing device, let's look at the task that we need to perform. Let's consider problems in which our goal is to match finite or repeating patterns.

For example regular expressions are used as pattern description language in

- Lexical analysis.-- compiler
- Filtering email for spam.
- Sorting email into appropriate mailboxes based on sender and/or content words and phrases.
- Searching a complex directory structure by specifying patterns that are known to occur in the file we want.

A regular expression is ***a pattern description*** language, which is used to describe particular patterns of interest. A regular expression provides a concise and flexible means for "**matching**" strings of text, such as particular characters, words, or patterns of characters.

Example: **[ ]** : A character class which matches any character within the brackets

[^ \t\n] matches any character except space, tab and newline character.

Regular expression

What are regular languages**?**

A language accepted by finite- automata is called as regular language.

A regular- languages can be described using regular expressions, in the form of algebraic notations consisting of the symbols such as alphabets in Σ, the operators such as + . and *, where + is used for union operations, **.** is used for concatenation and * is used for closure operations. Thus the regular expressions are the structural representation of finite-automata using algebraic notations, which can serve as the input language for many systems that process strings. Example such as Lex program, Unix Grep etc…

Definition of Regular expression

Define Regular expression.

A regular expression is defined as follows:

Φ is a regular expression denoting an empty language.

ε is regular expression denoting the language containing empty string.

a is regular expression denoting the language containing only {a}.

If R is a regular expression denoting the language L_R and S is a regular expression denoting the language L_S then R + S is a regular expression corresponding to the language $L_R \cup L_S$.

R.S is a regular expression corresponding to the language $L_R . L_S$.

R^* is a regular expression corresponding to the language L_R. Thus the expressions obtained by applying any of the rules are regular expressions.

Examples of Regular expressions

Regular expression	**Meaning**
a^*	String consisting of any number of a's. (zero or more a's)
a^+	String consisting of at least one a. (one or more a's)
(a + b)	String consisting of either a or b
$(a+b)^*$	String consisting of any nuber of a's and b's including ε
$(a+b)^*$ ab	Strings of a's and b's ending with ab.
ab$(a+b)^*$	Strings of a's and b's starting with ab.
$(a + b)^*$ ab $(a+b)^*$	Strings of a's and b's with substring ab.

Write the regular expressions for the following languages**:**

a. Strings of a's and b's having length 2:

Regular expression = **(a + b) (a+ b)**.

b. Strings of a's and b's of length ≤ 10:

Regular expression = $(\varepsilon + a + b)^{10}$.

c. Strings of a's and b's of even length.

Regular expression = $[(a + b) (a + b)]^*$

d. Strings of a's and b's of odd length

Regular expression = $(a + b) [(a + b) (a + b)]^*$

e. Strings of a's of even length

Regular expression = $(aa)^*$

f. Strings of a's of odd length

Regular expression = $a(aa)^*$

Strings of a's and b's with alternate a's and b's**.**

Alternate a's and b's can be obtained by concatenating the string (ab) zero or more times. ie $(ab)^*$ and adding an optional **b** to the front ie: (ε +b) and adding an optional **a** at the end, ie: (ε +a)

The *regular expression* = **(ε +b) (ab)*(ε +a)**

Obtain regular expression to accept the language containing at least one a and one b over Σ = { a, b, c}. **OR**

Obtain regular expression to accept the language containing at least one 0 and one 1 over Σ = { 0, 1, 2}.

String should contain at least one a and one b, so the regular expression corresponding to this is given by = ab + ba

There is no restriction on c's. Insert any number of a's, b's and c;s ie: (a+b+c)* in between the above regular expression.

So the final *regular expression* = **(a+b+c)* a (a+b+c)* b(a+b+c)* + (a+b+c)* b(a+b+c)*a(a+b+c)***

Obtain regular expression to accept the language containing at least 3 consecutive zeros.

Regular expression for string containing 3 consecutive 0's = 000

The above regular expression can be preceded or followed by any number of 0's and 1's, ie: (0+1)*

Regular expression = **(0+1)*000(0+1)***

Obtain regular expression to accept the language containing strings of a's and b's ending with b and has no substring aa.

Regular expression for strings of a's and b's ending with b and has no substring aa is nothing but the string containing any combinations of either b or ab without ε.

Regular expression = **(b + ab) (b +ab)***

Obtain regular expression to accept the language containing strings of a's and b's such that L = { a^{2n} b^{2m} | n, m ≥ 0 **}**

a^{2n} means even number of a's, regular expression = (aa)*

b^{2m} means even number of b's, regular expression = (bb)*.

The *regular expression* for the given language = **(aa)* (bb)***

Obtain regular expression to accept the language containing strings of a's and b's such that L = { a^{2n+1} b^{2m} | n, m ≥ 0 }.

a^{2n+1} means odd number of a's, regular expression = a(aa)*

b^{2m} means even number of b's, regular expression = (bb)*

The *regular expression* for the given language = **a(aa)* (bb)***

Obtain regular expression to accept the language containing strings of a's and b's such that L = { $a^{2n+1}\ b^{2m+1}$ | n, m ≥ 0 }.

a^{2n+1} means odd number of a's, regular expression = $a(aa)^*$

b^{2m+1} means odd number of b's, regular expression = $b(bb)^*$

The *regular expression* for the given language = **$a(aa)^*b(bb)^*$**

Obtain regular expression to accept the language containing strings of 0's and 1's with exactly one 1 and an even number of 0's.

Regular expression for exactly one 1 = 1

Even number of 0's = $(00)^*$

So here 1 can be preceded or followed by even number of 0's or 1 can be preceded and followed by odd number of 0's.

The regular expression for the given language = **$(00)^*\ 1\ (00)^* + 0(00)^*\ 1\ 0(00)^*$**

Obtain regular expression to accept the language containing strings of 0's and 1's having no two consecutive 0's. **OR**

Obtain regular expression to accept the language containing strings of 0's and 1's with no pair of consecutive 0's.

Whenever a 0 occurs it should be followed by 1. But there is no restriction on number of 1's. So it is a string consisting of any combinations of 1's and 01's, ie regular expression = $(1+01)^*$

Suppose string ends with 0, the above regular expression can be modified by inserting (0 + ε) at the end.

Regular expression for the given language = **$(1+01)^*\ (0 + \varepsilon)$**

Obtain regular expression to accept the language containing strings of 0's and 1's having no two consecutive 1's. **OR**

Obtain regular expression to accept the language containing strings of 0's and 1's with no pair of consecutive 1's.

Whenever a 1 occurs it should be followed by 0. But there is no restriction on number of 0's. So it is a string consisting of any combinations of 0's and 10's, ie regular expression = $(0+10)^*$

Suppose string ends with 1, the above regular expression can be modified by inserting (1 + ε) at the end.

Regular expression for the given language = **$(0+10)^*\ (1 + \varepsilon)$**

Obtain regular expression to accept the following languages over Σ = { a, b}.

i. Strings of a's and b's with substring aab.

Regular expression = **$(a+b)^* aab(a+b)^*$**

ii. Strings of a's and b's such that 4th symbol from right end is **b** and the 5th symbol from right end is **a**.

Here the 4th symbol from right end is b and the 5th symbol from right end is a the corresponding regu;lar expression = ab(a+b)(a+b)(a+b).

But the above regular expression can be preceded with any number of a's and b's.

Therefore the regular expression for the given language = **$(a+b)^*ab(a+b)(a+b)(a+b)$.**

iii. Strings of a's and b's such that 10th symbol from right end is **b**.

The regular expression for the given language = **$(a+b)^*b(a+b)^9$.**

iv. Strings of a's and b's whose lengths are multiple of 3.

OR

L = { |w| mod 3 = 0, where w is in Σ = { a, b}

Length of string w is multiple of 3, the regular expression = **$[(a+b)\ (a+b)\ (a+b)]^*$**

v. Strings of a's and b's whose lengths are multiple of 5.

OR

L = { |w| mod 5 = 0, where w is in Σ = { a, b}

Length of string w is multiple of 5,

The regular expression = **$[(a+b)\ (a+b)\ (a+b)\ (a+b)(a+b)]^*$**

vi. Strings of a's and b's not more than 3 a's:

Not more than 3 a's, regular expression= (ε+a) (ε+a) (ε+a).

But there is no restriction on b's, so we can include b^* in between the above regular expression.

The regular expression for the given language = **$b^*(ε+a)\ b^*(ε+a)\ b^*\ (ε+a)\ b^*$**

vii. Obtain the regular expression to accept the words with two or more letters but beginning and ending with the same letter. Σ = { a, b}

Regular expression beginning and ending with same letter is = a a + b b. In between include any number of a's and b's.

Therefore the regular expression = **$a\ (a+b)^*\ a + b\ (a+b)^*\ b$**

viii. Strings of a's and b's of length is either even or multiple of 3.

Multiple of regular expression = $[(a+b)\ (a+b)\ (a+b)]^*$

Length is of even, regular expression = $[(a+b)\ (a+b)]^*$

So the regular expression for the given language = **[(a+b) (a+b) (a+b)]*+ [(a+b) (a+b)]***

ix. Obtain the regular expression to accept the language L = { a^nb^m | m+n is even **}**

Here n represents number of a's and m represents number of b's.

m+n is even results in two possible cases;

case i. when even number of a's followed by even number of b's.

regular expression : $(aa)^*(bb)^*$

case ii. Odd number of a's followed by odd number of b's.

regular expression = $a(aa)^* b(bb)^*$.

So the regular expression for the given language = **$(aa)^*(bb)^* + a(aa)^* b(bb)^*$**

x. Obtain the regular expression to accept the language L = { a^nb^m | n ≥ 4 and m ≤ 3 }**.**

Here n ≥ 4 means at least 4 a's, the regular expression for this = $aaaa(a)^*$

m ≤ 3 means at most 3 b's, regular expression for this = (ε+b) (ε+b) (ε+b).

So the regular expression for the given language = **$aaaa(a)^*$ (ε+b) (ε+b) (ε+b).**

xi. Obtain the regular expression to accept the language L = { $a^nb^m c^p$ | n ≥ 4 and m ≤ 3 p ≤ 2}**.**

Here n ≥ 4 means at least 4 a's, the regular expression for this = $aaaa(a)^*$

m ≤ 3 means at most 3 b's, regular expression for this = (ε+b) (ε+b) (ε+b).

p ≤ 2 means at most 2 c's, regular expression for this = (ε+c) (ε+c)

So the regular expression for the given language = **$aaaa(a)^*$(ε+b) (ε+b) (ε+b) (ε+c) (ε+c).**

xii. All strings of a's and b's that do not end with **ab.**

Strings of length 2 and that do not end with ab are ba, aa and bb.

So the regular expression = **$(a+b)^*$(aa + ba +bb)**

xiii. All strings of a's, b's and c's with exactly one **a**.

The regular expression = **$(b+c)^*$ a $(b+c)^*$**

xiv. All strings of a's and b's with at least one occurrence of each symbol in Σ = {a, b}.

At least one occurrence of a's and b's means **ab** + **ba**, in between we have n number of a's and b's.

So the regular expression =**$(a+b)^*$ a $(a+b)^*$ b$(a+b)^*$ +$(a+b)^*$ b$(a+b)^*$ a$(a+b)^*$**

Obtain the regular expression for the language L = { $a^n b^m$ | m ≥ 1, n ≥ 1, nm ≥ 3 }.

Solution:

Case i. Since **nm ≥ 3,** if **n** = 1 then **m** should be **≥ 3.** The equivalent regular expression is given by: RE = **a bbb(b)***

Case ii. Since **nm ≥ 3,** if m = 1 then **n** should be **≥ 3.** The equivalent regular expression is given by: RE = **aaa(a)* b**

Case iii. Since **nm ≥ 3,** if m ≥ 2 and n ≥ 2 then the equivalent regular expression is given by:

RE = **aa(a)* bb(b)***

So the final regular expression is obtained by adding all the above regular expression.

Regular expression = **abbb(b)* + aaa(a)*b + aa(a)*bb(b)***

Application of Regular expression:

1. Regular expressions are used in UNIX.
2. Regular expressions are extensively used in the design of Lexical analyzer phase.
3. Regular expressions are used to search patterns in text.

FINITE AUTOMATA AND REGULAR EXPRESSIONS

1. ****Converting Regular Expressions to Automata:

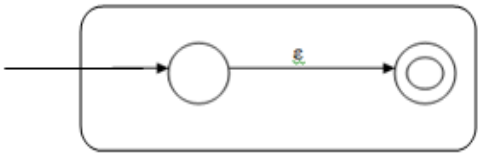
Prove that every language defined by a regular expression is also defined by a finite automata**.**

Proof:

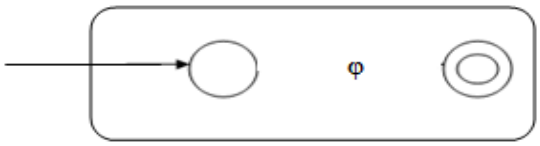
Suppose L = L(R) for a regular expression R, we show that L = L(E) for some ε-NFA E with:

a. Exactly one accepting state.
b. No arcs into the initial state.
c. No arcs out of the accepting state.

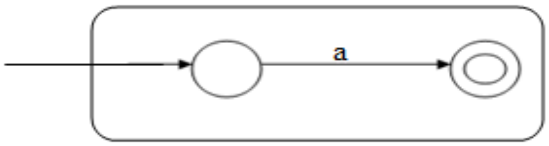
The proof must be discussed with the following transition diagrams for the basis of the construction of an automaton.



R = ε

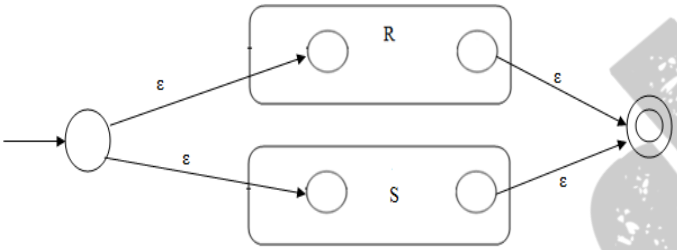


R = φ



R = a

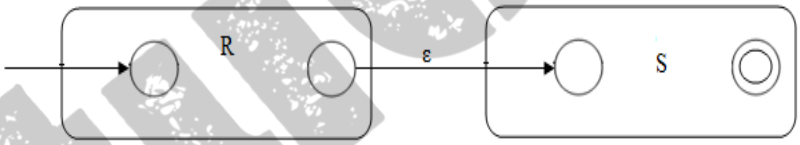
By definition of regular expression, if R is a RE and S is a RE then R+S is also a RE corresponding to the language L (R+S), its automaton is given by:



L (R+ S)

Starting at new start state, we can go to the start state of either the automaton for R or S. We then reach the accepting state of one of these automata R or S. We can follow one of the ε- arcs to the accepting state of the new automaton.

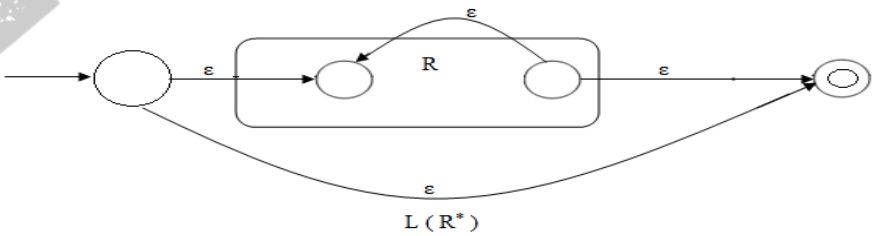
Automaton for R.S is given by**:**



L (R.S)

The start state of the first(R) automata becomes the start state of the whole and the final state of the second(S) automata becomes the final state of the whole.

Automaton for R* is given by**:**

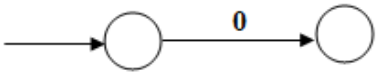


L (R^*)

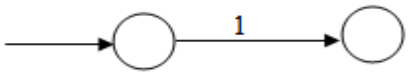
From start state to final state one arc labeled ε (for ε in R^*) or the to the start state of automaton R through that automaton one or more time and then to the final state.

Convert the regular expression $(0 + 1)^* 1 (0 + 1)$ to an ε- NFA.

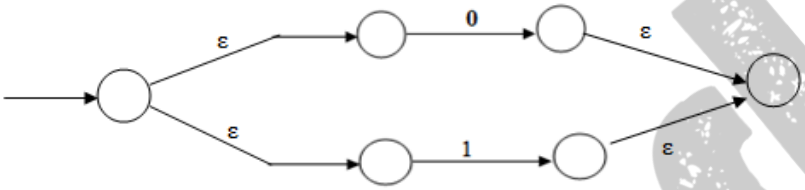
The automaton for L = 0 is given by:



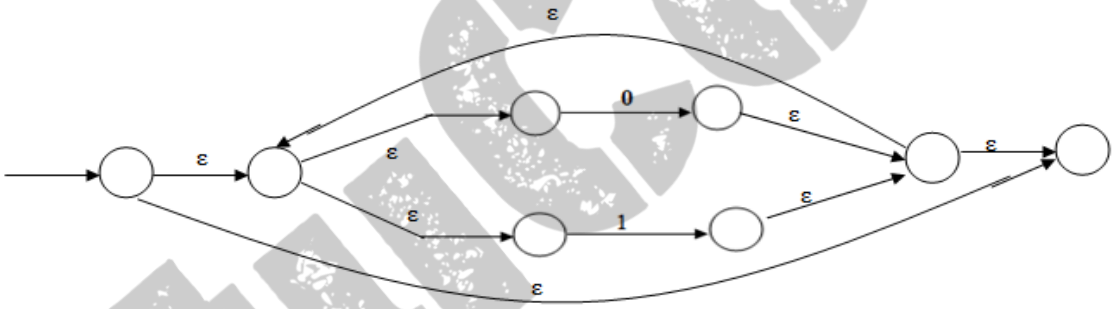
The automaton for L = 1 is given by:



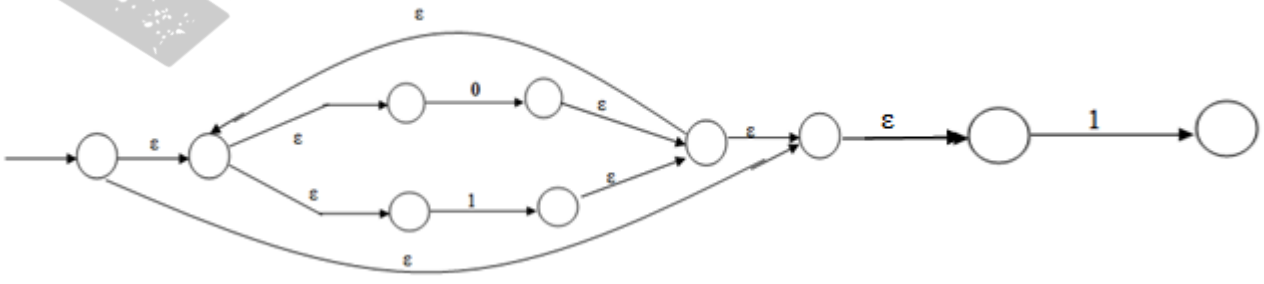
The automaton for L = 0+1 is given by:



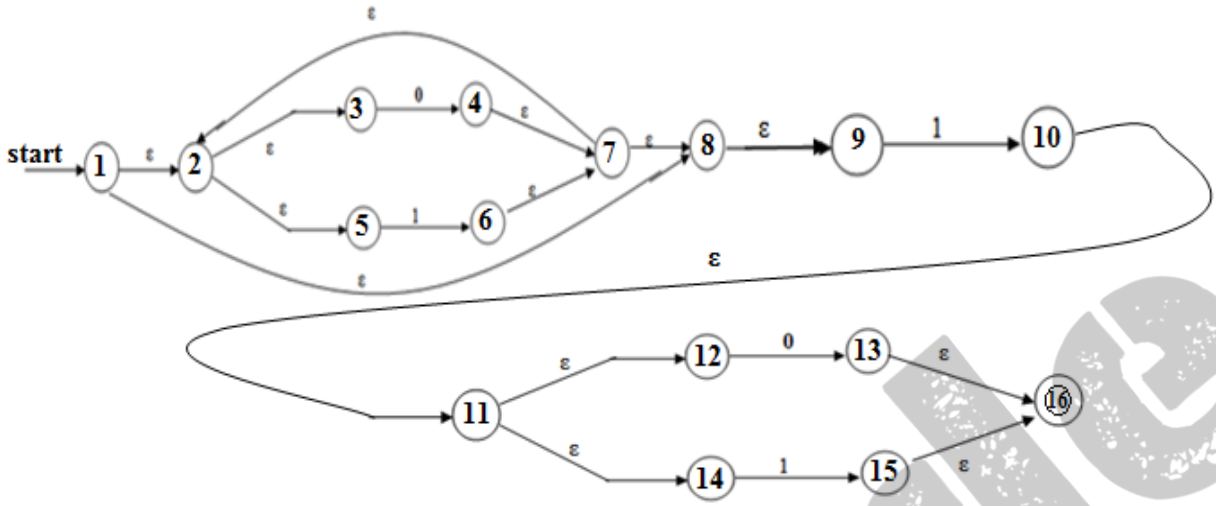
The automaton for L = $(0+1)^*$ is given by:



The automaton for L = $(0+1)^* 1$ is given by:



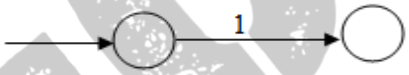
Finally the ε-**NFA for the regular expression: (0+1)*1(0+1) is given by:**



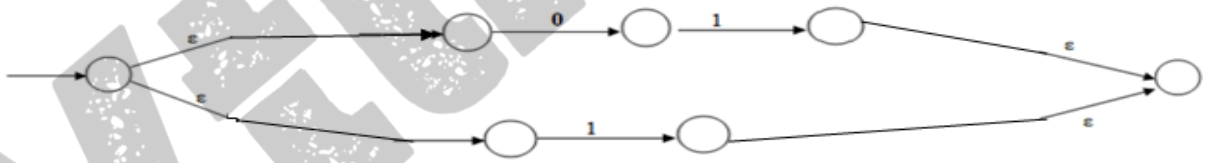
Convert the regular expression (01+ 1)* to an ε- NFA**.**

The automaton for L = 01 is given by:

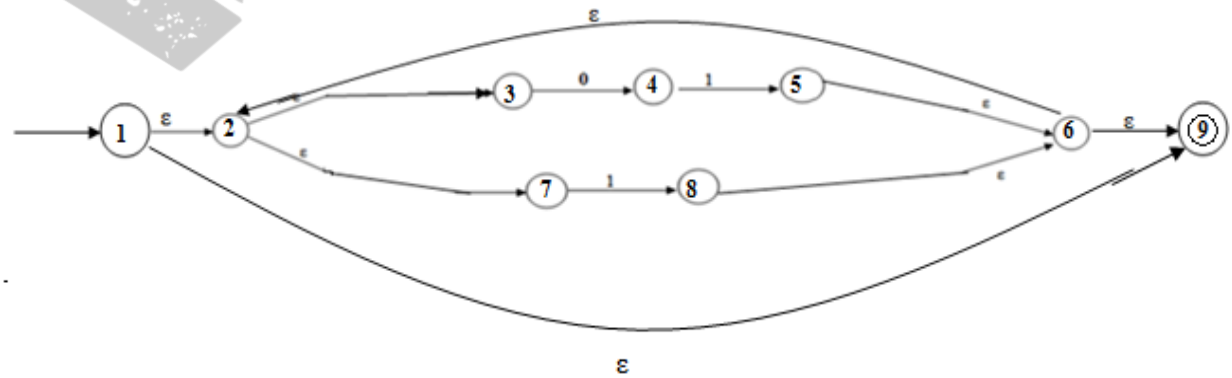
The automaton for L = 1 is given by:



The automaton for L = 01+1 is given by:

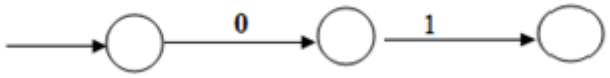


Finally **ε- NFA** for L = (01+1)* is given by:

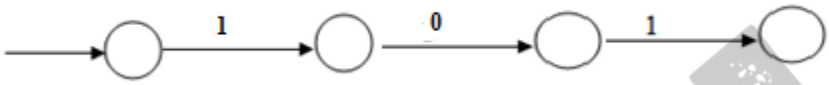


Convert the regular expression (01+ 101) to an ε- NFA.

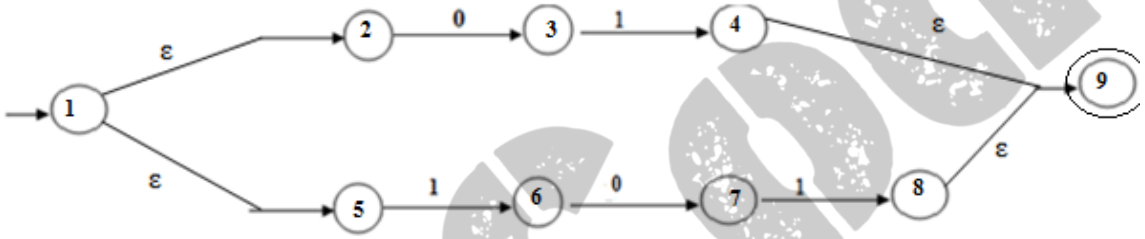
The automaton for L = 01 is given by:



The automaton for L = 101 is given



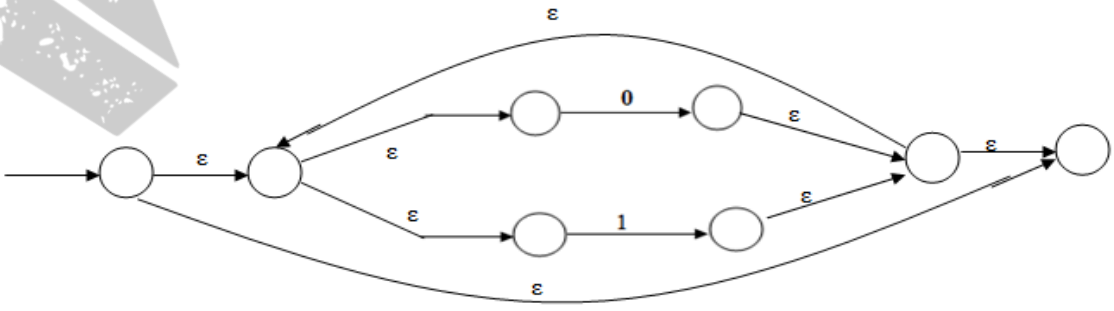
The automaton for L = (01+101) is given by:



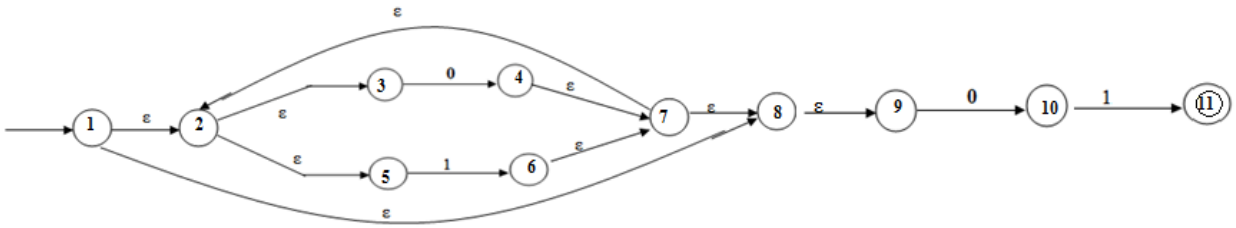
Convert the regular expression $(0+1)^*01$ to an ε- NFA.

The automaton for L = 01 is given by:

The automaton for L = $(0+1)^*$ is given by:

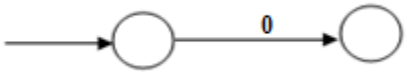


Epsilon-NFA for the regular expression $(0+1)^*01$ is given by:

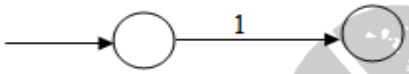


Convert the regular expression $0^* + 1^* + 2^*$ to an ε- NFA.

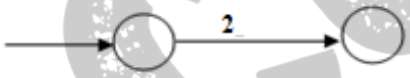
The automaton for L = 0 is given by:



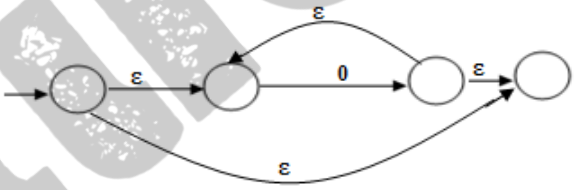
The automaton for L = 1 is given by:



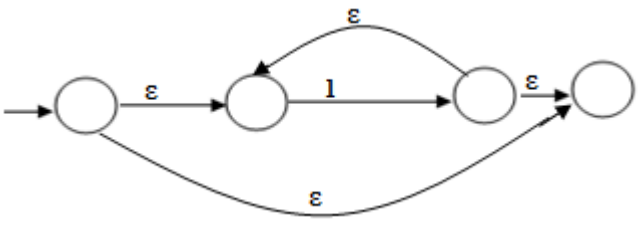
The automaton for L = 2 is given by:



The automaton for L = 0^* is given by:

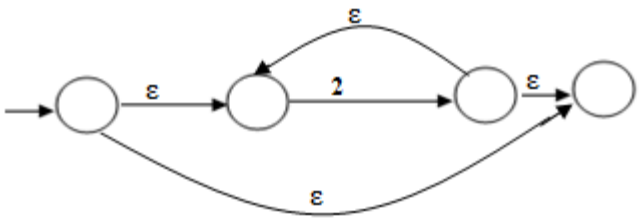


The automaton for L = 1^* is given by:

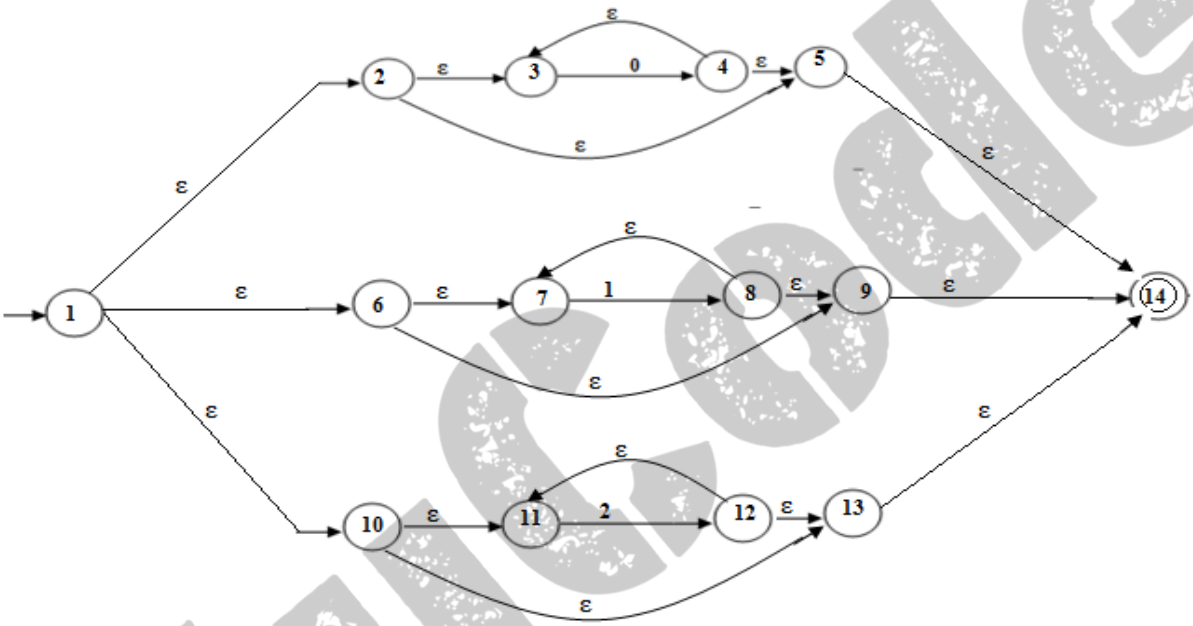


The automaton for L = 2^* is given by:

vtucode



Epsilon-NFA for the regular expression $0^* + 1^* + 2^*$ is given by:



CONVERTING DFA's TO REGULAR EXPRESSION USING STATE ELIMINATION TECHNIQUE

How to build a regular expression for a FSM. Instead of limiting the labels on the transitions of an FSM to a single character or *ε*, we will allow entire regular expressions as labels.

- For a given input FSM/FA M, we will construct a machine M' such that M and M' are equivalent and M' has only *two states*, start state and a single accepting state.
- M' will also have just one transition, which will go from its start state to its accepting state. The label on that transition will be a regular expression that describes all the strings that could have driven the original machine *M from its start state to* some accepting state

Algorithm to create a regular expression from FSM: (State elimination)

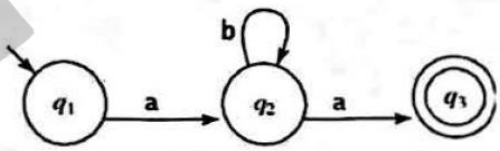
1. Remove any states from given FSM M that are unreachable from the start state
2. If FSM M has no accepting states then halt and return the simple regular expression Ø.

3. If the start state of FSM M is part of a loop (i.e: it has any transitions coming into it), then create a new start state **s** and connects to M 's start state via an ε-transition. This new start state s will have no transitions into it.
4. If a FSM M has ***more than one*** accepting state or if there is ***just one*** but there are any transitions out of it, create a new accepting state and connect each of **M's** accepting states to it via an ε-transition. Remove the old accepting states from the set of accepting states. Note that the new accepting state will have no transitions out from it.
5. At this point, if **M** has only one state, then that state is both the ***start state*** and the ***accepting state*** and **M** has no transitions. So L (M} = {**ε**}. Halt and return the simple regular expression as ***ε.***
6. Until only the start state and the accepting state remain do:
   - **6.1.** Select some state ***s*** *of M which is of a*ny state except the start state or the accepting state.
   - **6.2** Remove that state ***s*** *from M.*
   - **6.3** Modify the transitions among the remaining states so that *M accepts the* same strings The labels on the rewritten transitions may be any regular expression.
7. Return the regular expression that labels the one remaining transition from the start state to the accepting state

Consider the following FSM ***M***: Show a regular expression for L(M).

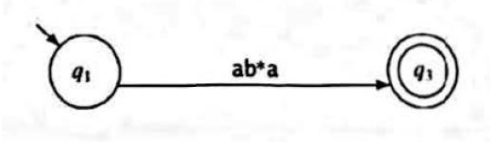
OR

Obtain the regular expression for the following finite automata using state elimination method.



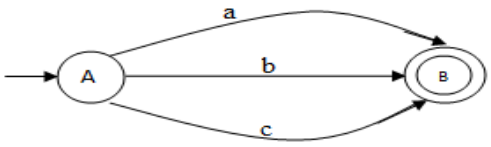
We can build an equivalent machine ***M'*** *by eliminating state* q_2 *and replacing it by a* transition from q_1 to q_3 labeled with the regular expression **ab*a**.

So *M' is:*

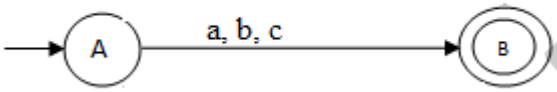


Regular Expression = **ab*a**

Obtain the regular expression for the following finite automata using state elimination method.



There is no incoming edge into the initial state as well as no outgoing edge from final state. So there is only two states, initial and final.

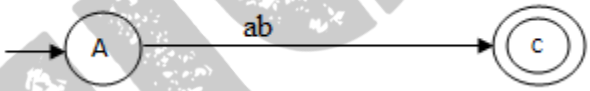


Regular expression = **(a+b+c)** or **(a U b U c)**

Obtain the regular expression for the following finite automata using state elimination method.

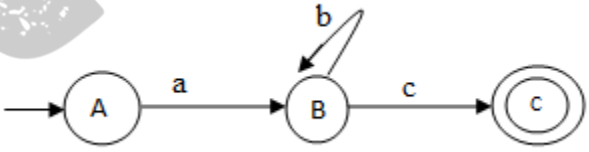
There is no incoming edge into the initial state as well as no outgoing edge from final state.

After eliminating the state B:



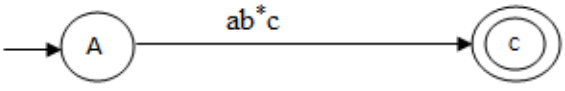
Regular expression = **ab**

Obtain the regular expression for the following finite automata using state elimination method.



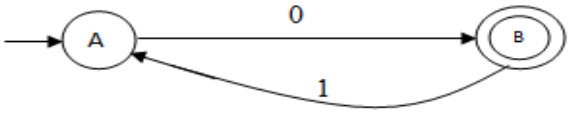
There is no incoming edge into the initial state as well as no outgoing edge from final state.

After eliminating the state B:

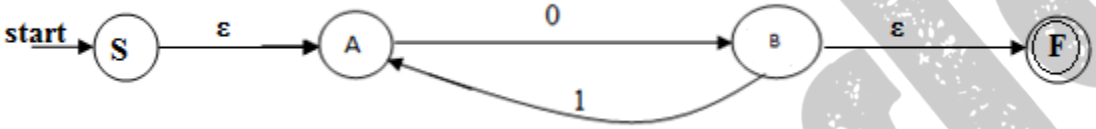


Regular expression = **ab*c**

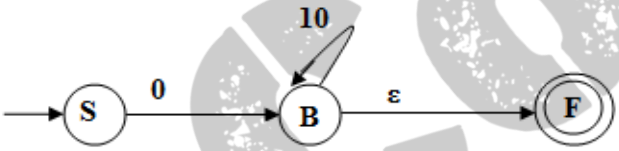
Obtain the regular expression for the following finite automata using state elimination method.



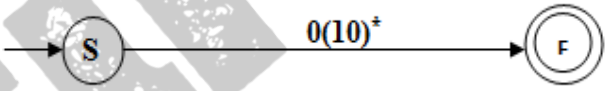
Since initial state has incoming edge, and final sate has outgoing edge, we have to create a new iniatial and final state by connecting new initial state to old initial state through ε and old final state to new final state through ε. Make old final state has non-final state.



After removing state A:

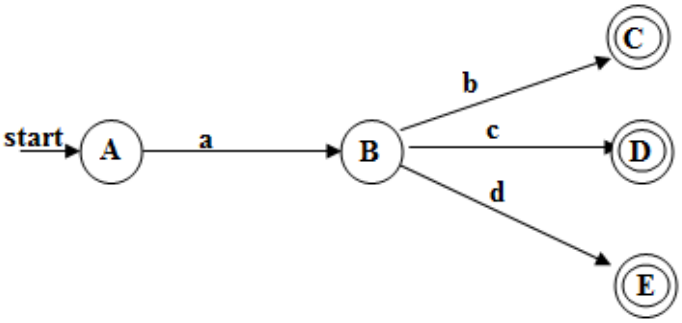


After removing state B:

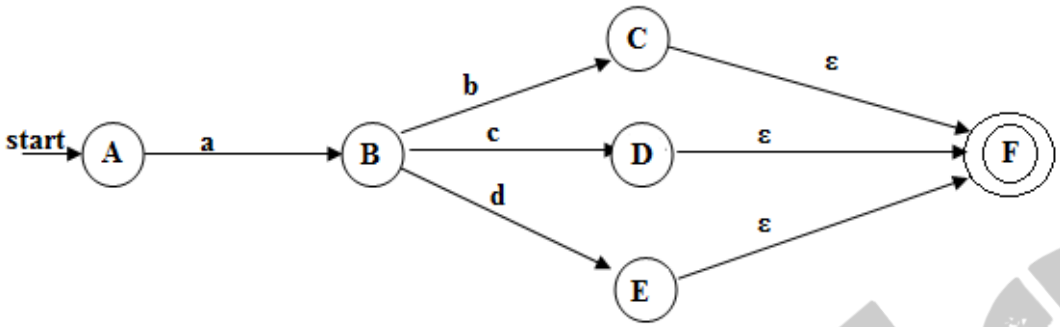


Regular expression: **0(10)***

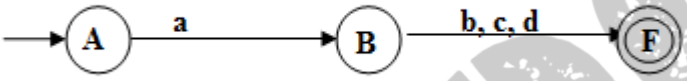
Obtain the regular expression for the following finite automata using state elimination method.



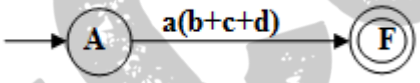
Since there are multiple final states, we have to create a new final state.



After removing states C, D and E:

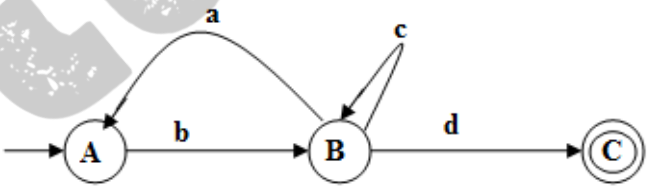


After removing state B:

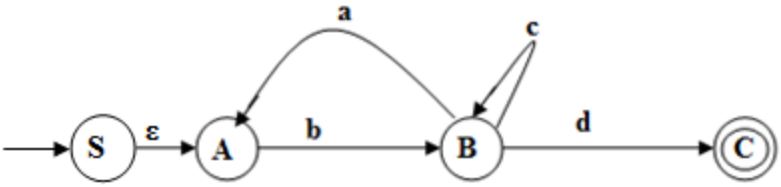


Regular Expression: **a(b+c+d)**

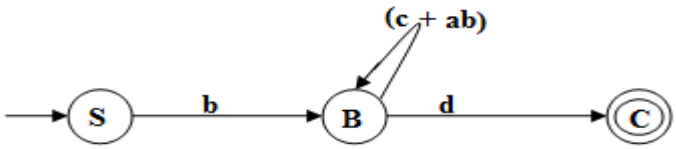
Obtain the regular expression for the following finite automata using state elimination method.



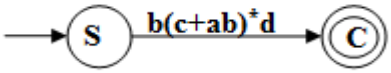
After inserting new start state:



After removing state A:

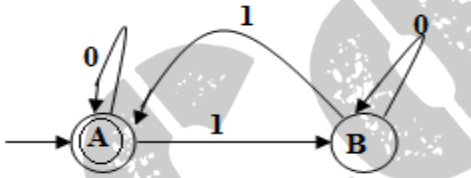


After removing state B:

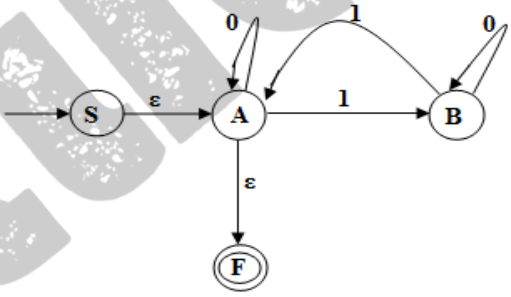


Regular expression: **b(c +ab)*d**

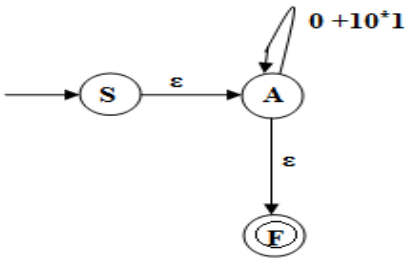
Obtain the regular expression for the following finite automata using state elimination method.



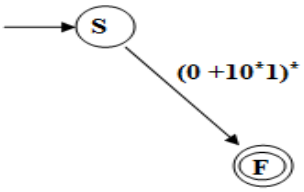
By creating new start and final states:



After removing state B:

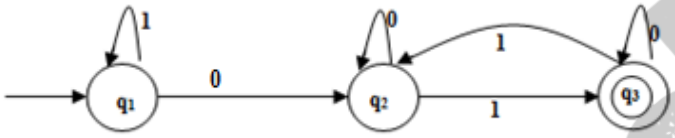


After removing state A:

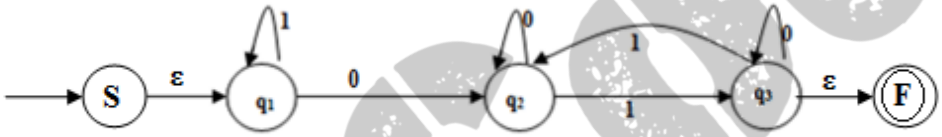


Regular expression: **(0+10*1)***

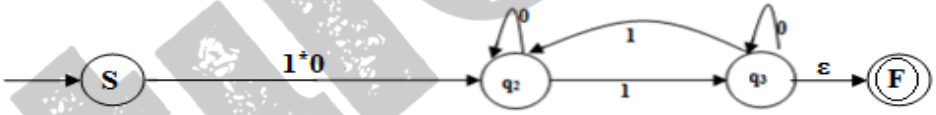
Obtain the regular expression for the following finite automata using state elimination method.



By creating new start and final states:

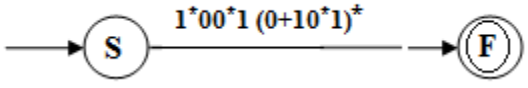


After removing state q_1:



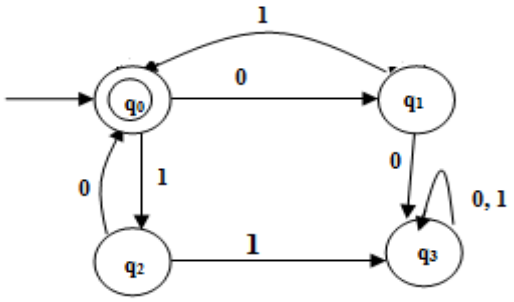
After removing state q_2:

After removing state q_3:

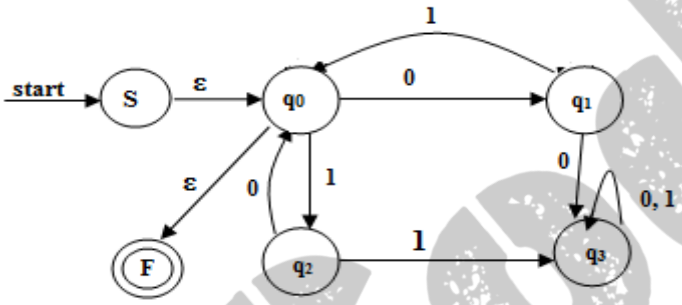


Regular expression: **1*00*1(0+10*1)***

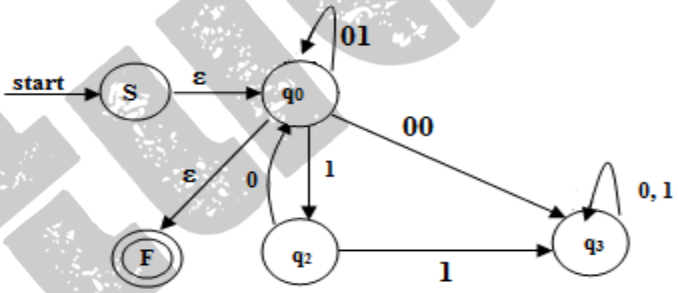
Obtain the regular expression for the following finite automata using state elimination method.



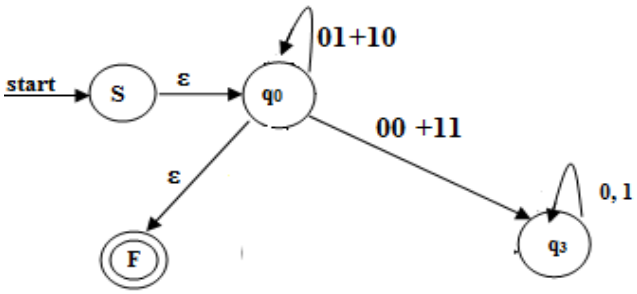
By creating new start state and final state:



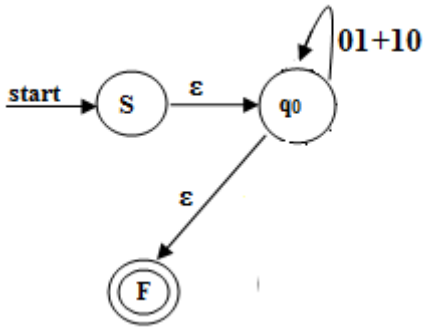
After removing q_1 state:



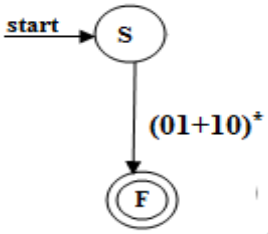
After removing q_2 state:



After removing q_3 state:



After removing q_0 state:

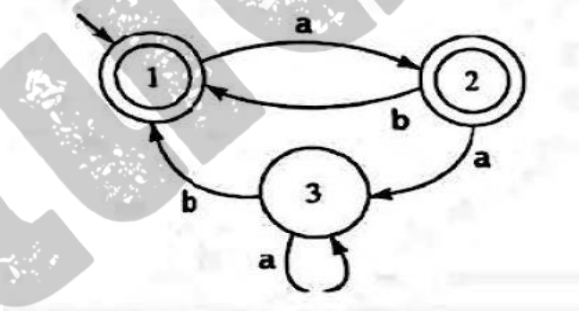


Regular expression: **(01+10)***

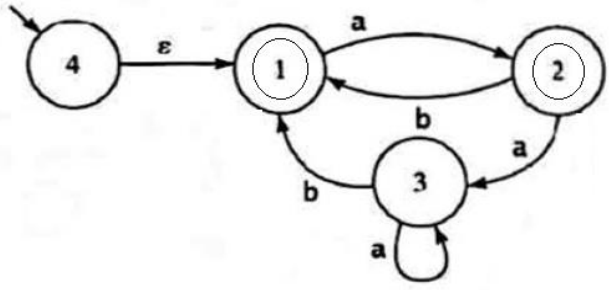
Consider the following FSM M: Show a regular expression for L(M).

OR

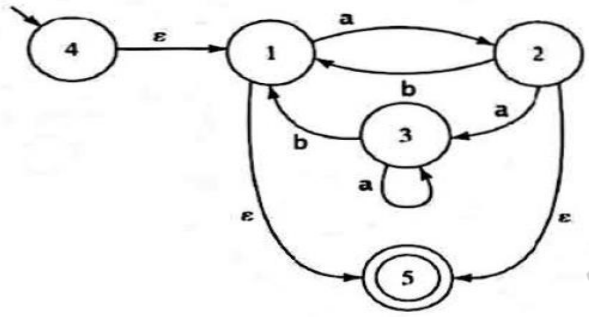
Obtain the regular expression for the following finite automata using state elimination method.



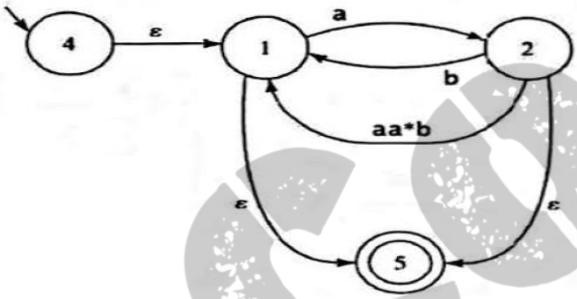
Since start state 1 has incoming transitions, we create a new start state and link that state to state 1 through **ε.**



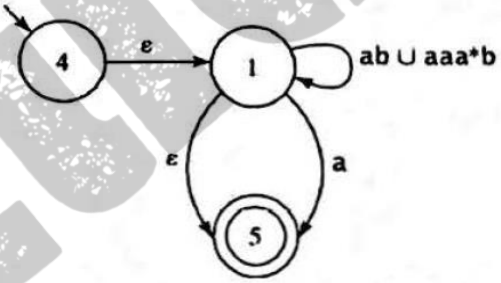
Since accepting state 1 and 2 has outgoing transitions, we create a new accepting state and link that state to state 1 and state 2 through **ε.** Remove the old accepting states from the set of accepting states. **(ie:** consider 1 and 2 has non final states)



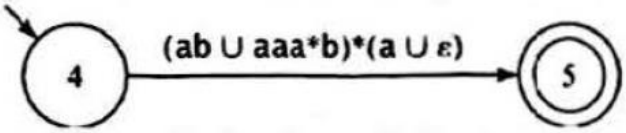
Consider the state 3 and remove that state:



Consider the state 2 and remove that state:



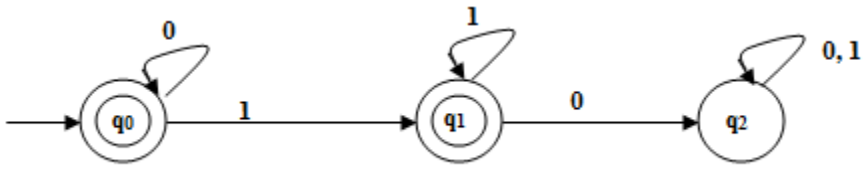
Consider the state 1 and remove that state:



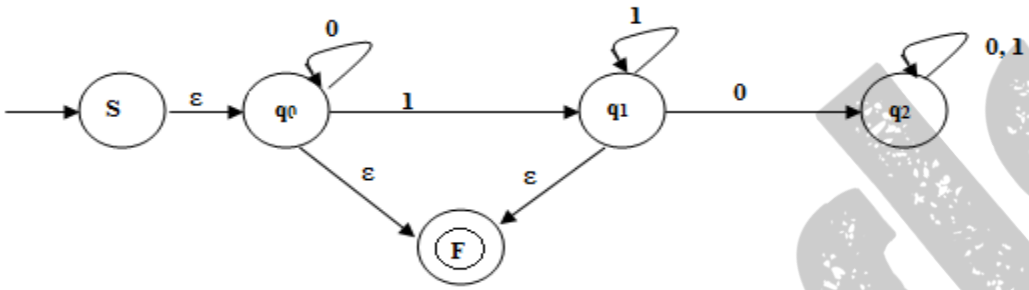
Finally we have only start and final states with one transition from start state 1 to final state 2, The labels on transition path indicates the regular edpression.

Regular Expression = **(ab U aaa* b)* (a U ε)**

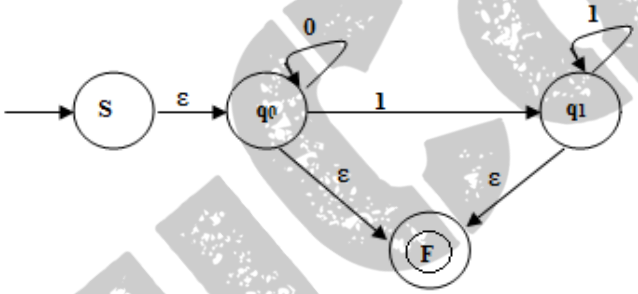
Consider the following FSM M: Show a regular expression for L(M).



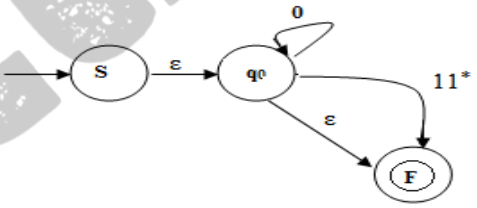
After creating new start and final states:



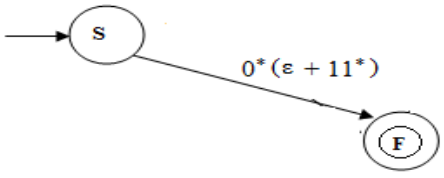
After removing q_2 state:



After removing q_1 state:



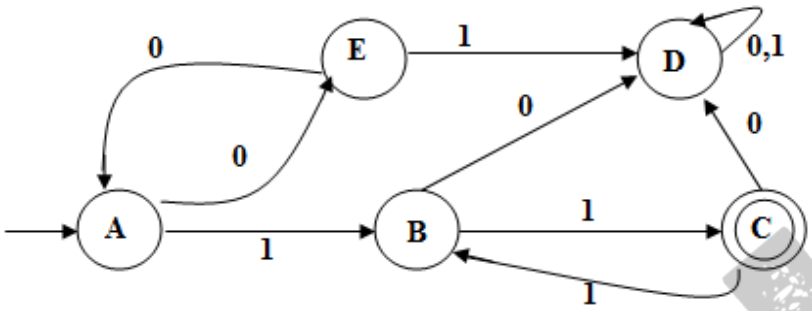
After removing q_0 state:



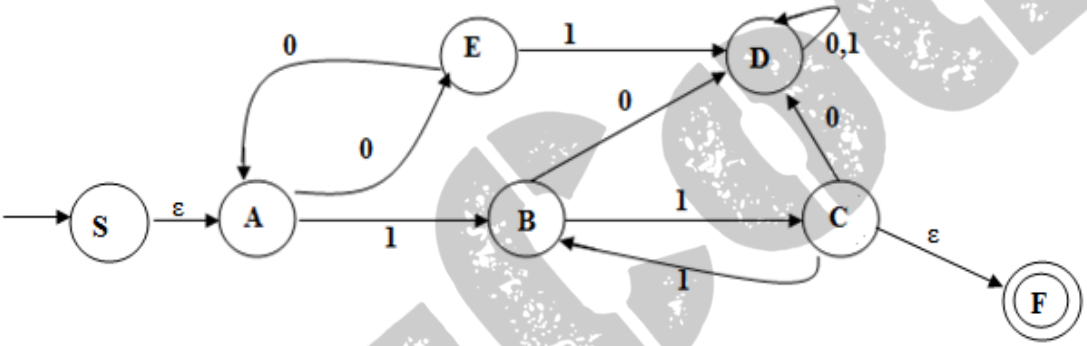
Regular expression: $0^* (\varepsilon + 1^+) = 0^* 1^*$

Consider the following FSM M: Show a regular expression for L(M). **OR**

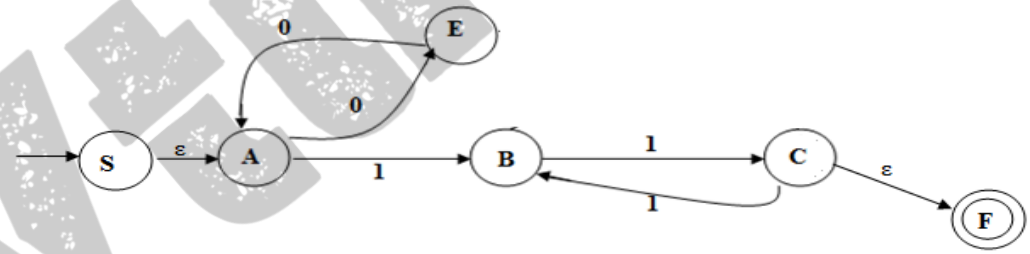
Construct regular expression for the following FSM using state elimination method.



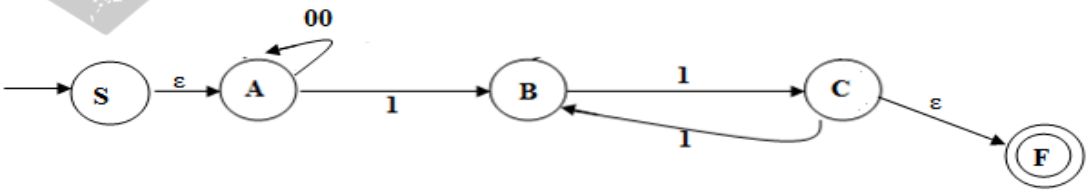
By creating new state and final states.



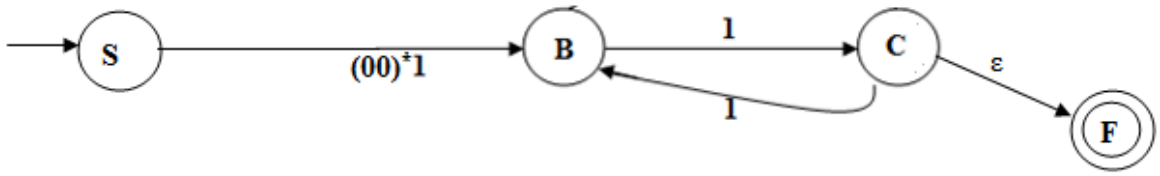
After removing D state:



After removing E state:

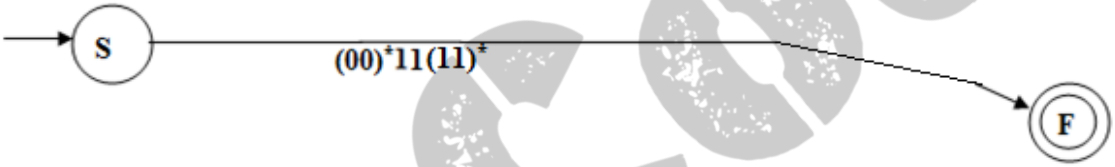


After removing A state:



After removing B state:

After removing C state:

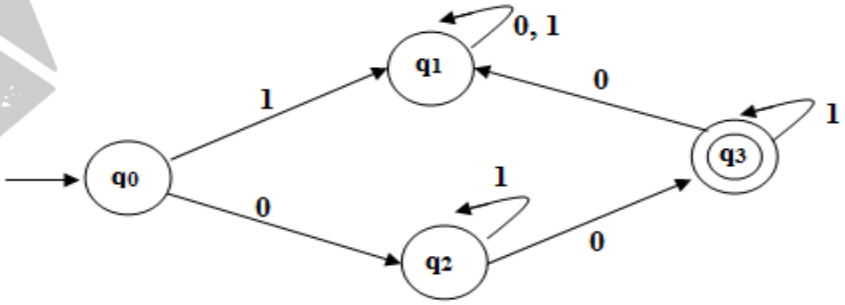


Regular expression = $(00)^*11(11)^*$

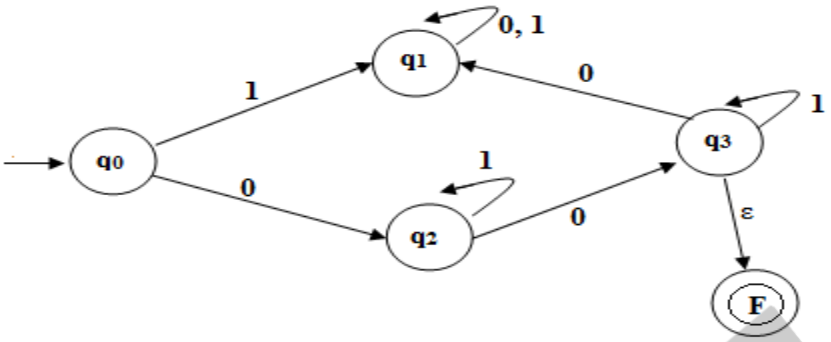
Consider the following FSM M: Show a regular expression for L(M).

OR

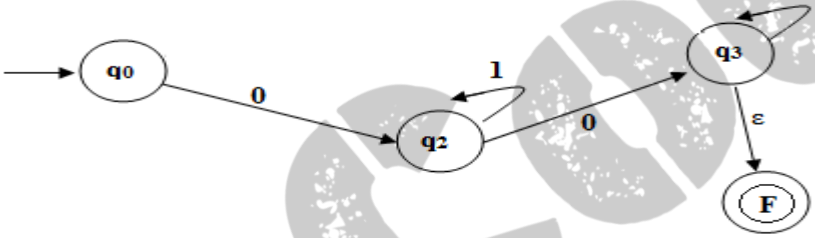
Construct regular expression for the following FSM using state elimination method.



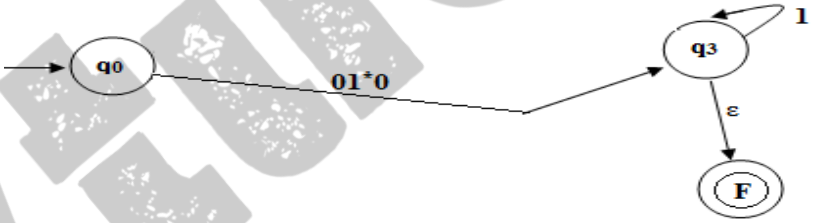
By creating final state.



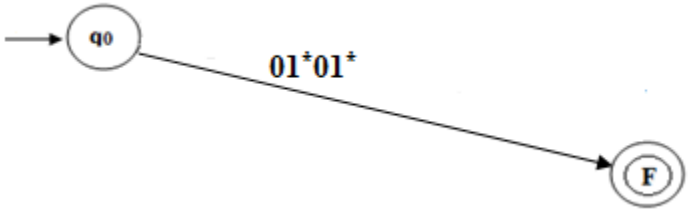
After removing $q_1$state:



After removing $q_2$state:



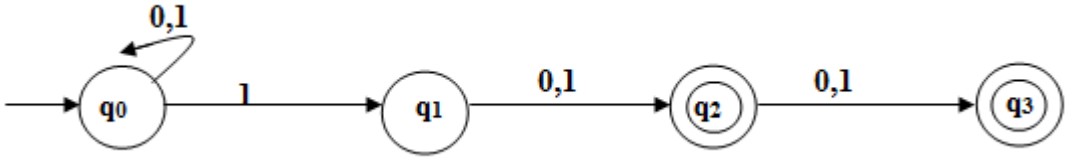
After removing $q_3$state:



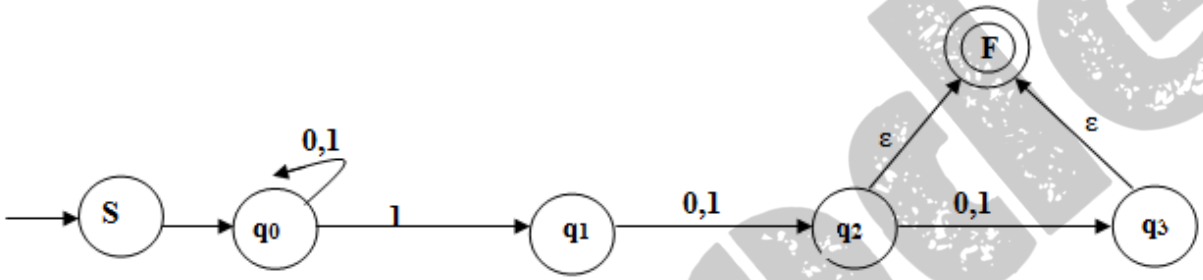
Regular expression= 01^*01^*

Consider the following FSM M: Show a regular expression for L(M). **OR**

Construct regular expression for the following FSM using state elimination method.

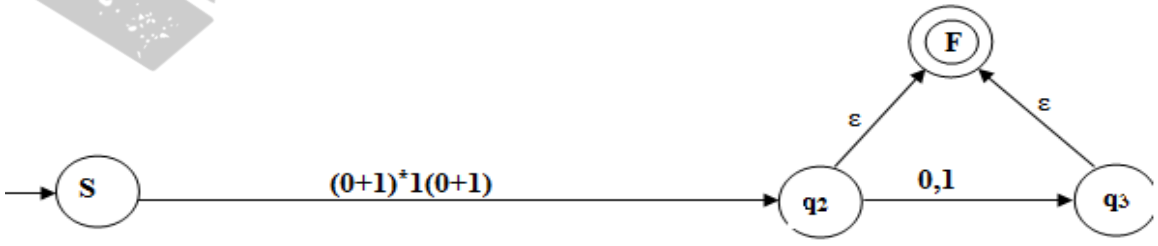


By creating new start and final states:

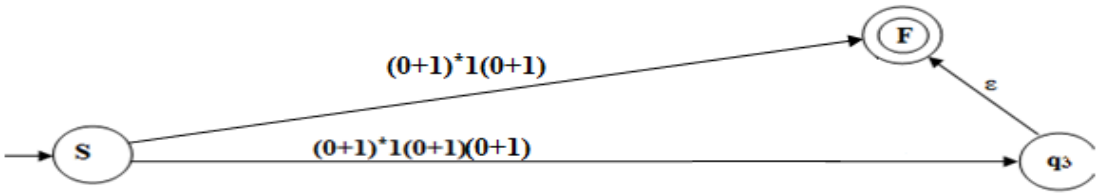


After removing q_0 state:

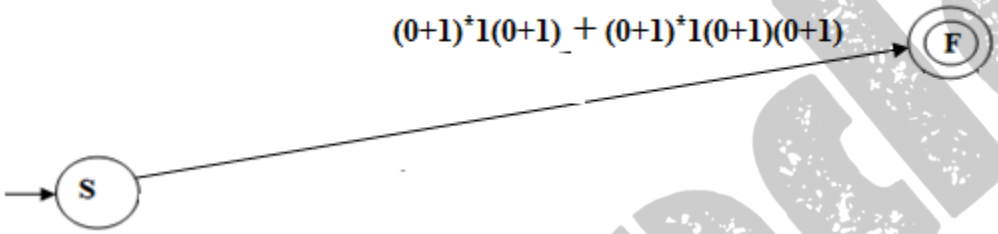
After removing q_1 state:



After removing q_2 state:



After removing q_3 state:



Regular expression: **$(0+1)^*1(0+1) +(0+1)^*1(0+1)(0+1)$**

CONVERTING DFA's TO REGULAR EXPRESSION USING KLEEN'S THEOREM

The construction regular expression using this method describes sets of strings that label certain paths in the DFA's transition diagram. However the paths are allowed to pass through only a limited subset of the states. We start with the simplest expressions that describe paths that are not allowed to pass through any states (ie: they are single state or single arc), and inductively build the expressions that let the paths go through progressively larger sets of states. Finally the paths are allowed to go through any state. At the end these expressions represent all possible paths.

Let us consider a DFA with **'n'** number of states and use **R_{ij}^k** as the name of regular expression whose language is the set of strings **w** is the label of a path from state I to state j in a given DFA and that path has no intermediate node whose number is greater than **k**. Note that beginning and end points of the path are not intermediate, so there is no constraint that **i** and/or **j** be less than or equal to **k**.

To construct the regular expression R_{ij}^k we use the inductive definition, starting at k = 0 and finally k is reaching **k = n** which is the number of states in DFA.

When k=0

Regular expressions for the paths that can go through no intermediate state at all, there are 2 kinds of paths that meet such condition:

1. An arc from node (state) i to node j

2. A path of length 0 that consists of only some node i.

If i ≠ j then only case (1) is possible. We must examine DFA and find those input symbols a such that there is a transition from state i to state j on symbol a.

a) If there is no such symbol a, then $R_{ij}^{0} = Ø$

b) If there is exactly one such symbol a, then $R_{ij}^{0} = a$

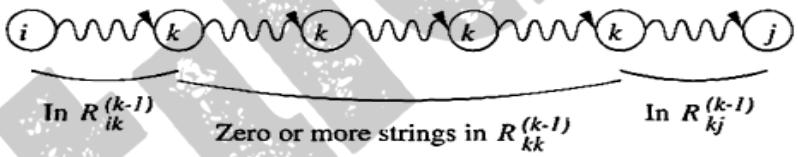
c) If there are symbols $a_1, a_2, a_3, \ldots\ldots\ldots\ldots a_k$ that label arcs from state I to state j, then

$R_{ij}^{0} = a_1 + a_2 + a_3, \ldots\ldots\ldots\ldots + a_k$

If i = j, then legal paths are the path of length 0 and all loops from i to itself. The path length 0 is represented by the regular expression ε. Thus we add ε to the various expressions devised in a) through c) above. That is in case a) expression becomes Ø + ε = ε, in case b) expression becomes ε+a, in case c) expression becomes $\varepsilon + a_1 + a_2 + a_3, \ldots\ldots\ldots\ldots + a_k$

Suppose there is a path from state i to state j that goes through no state higher than k. There are two possible cases to consider.

1. The path does not go through state k at all. In this case, the label of path is in the language of R_{ij}^{k-1}
2. The path goes through state k at least once. Then we can break the path into several pieces, as suggested in below figure:



The first goes from state i to state k without passing through k, the last piece goes from k to j without passing through k, and all the pieces in the middle go from k to itself, without passing through k. When we combine the expressions for the paths of the two types above, we have the expression for the labels of all paths from state i to state j that go through no state higher than k.

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} (R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

$\mathbf{R_{ij}^{0}}$ = Regular expressions for the paths that can go through no intermediate states at all.

$\mathbf{R_{ij}^{1}}$ = Regular expressions for the paths that can go through an intermediate state 1 only.

$\mathbf{R_{ij}^{2}}$ = Regular expressions for the paths that can go through an intermediate state 1 and state 2 only.

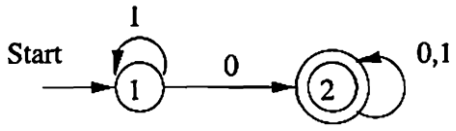
$\mathbf{R_{ij}^{3}}$ = Regular expressions for the paths that can go through an intermediate state 1, state 2 and

State3 only and so on….

NOTE:

Identity rule	Example
ε R = R ε = R	**1 ε = ε 1 = 1**
ØR = RØ = Ø	1Ø = Ø1 = Ø
$\varepsilon^* = \varepsilon$	
$(Ø)^* = \varepsilon$	
Ø + R = R+ Ø = R	Ø +1 =1
R +R = R	1 U 1 = 1
$RR^* = R^*R = R^+$	$00^* = 0^+$
$(R^*)^* = R^*$	$(1^*)^* = 1^*$
$R^* R^* = R^*$	
$\varepsilon + RR^* = R^*$	**$\varepsilon + 1^+ = 1^*$**
(P+Q)R = PR +QR	
$(P+Q)^* = (P^*Q^*) = (P^*+Q^*)^*$	
$R^*(\varepsilon + R) = (\varepsilon + R)\, R^* = R^*$	
$(\varepsilon + R)^* = R^*$	
$\varepsilon + R^* = R^*$	
$(PQ)^* P = P(QP)^*$	
$R^*R + R = R^*R = R^+$	

Write the regular expression for the language accepted by the following DFA:



Answer:

When k =0; (passing through no intermediate state), the various regular expressions are:

$R_{11}^{(0)}$	$\epsilon + 1$
$R_{12}^{(0)}$	0
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$(\epsilon + 0 + 1)$

When k =1; (passing through sate 1 as intermediate state), the various regular expressions are:

$R_{11}^{(1)}$	1^*
$R_{12}^{(1)}$	1^*0
$R_{21}^{(1)}$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon+0+1$

Therefore the regular expression corresponding to the language accepted by the DFA is given by:

R_{12}^{2} (state 1(i) is the start state and state 2(j) is the final state). By using the formula:

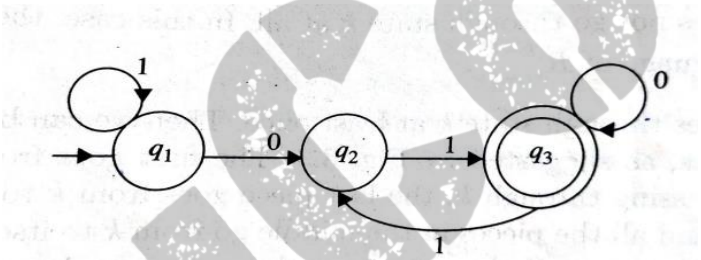
$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

$R_{12}^{2} = R_{12}^{1} + R_{12}^{1} (R_{22}^{1})^* R_{22}^{1}$

$R_{12}^{(2)} \mid 1^*0 + 1^*0(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$

Regular expression = **$1^*0\ (0 + 1)^*$**

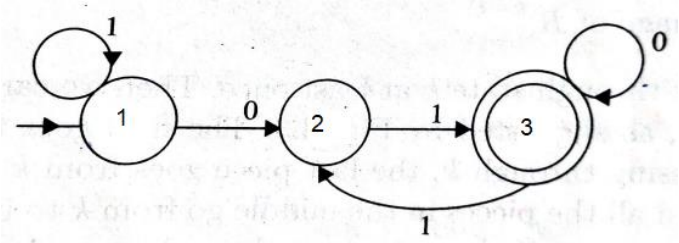
Write the regular expression for the language accepted by the following DFA:



Answer:

Number of states in DFA = 3; ie: **k = 3**

By renaming the states of DFA:



$R_{11}^{(0)}$	$1+\epsilon$
$R_{12}^{(0)}$	0
$R_{13}^{(0)}$	ϕ
$R_{21}^{(0)}$	ϕ
$R_{22}^{(0)}$	ϵ
$R_{23}^{(0)}$	1
$R_{31}^{(0)}$	ϕ
$R_{32}^{(0)}$	1
$R_{33}^{(0)}$	$0+\epsilon$

(a)

$R_{11}^{(1)}$	1*
$R_{12}^{(1)}$	1*0
$R_{13}^{(1)}$	ϕ
$R_{21}^{(1)}$	ϕ
$R_{22}^{(1)}$	ϵ
$R_{23}^{(1)}$	1
$R_{31}^{(1)}$	ϕ
$R_{32}^{(1)}$	1
$R_{33}^{(1)}$	$0+\epsilon$

(b)

$R_{11}^{(2)}$	1*
$R_{12}^{(2)}$	1*0
$R_{13}^{(2)}$	1*01
$R_{21}^{(2)}$	ϕ
$R_{22}^{(2)}$	ϵ
$R_{23}^{(2)}$	1
$R_{31}^{(2)}$	ϕ
$R_{32}^{(2)}$	1
$R_{33}^{(2)}$	$0+\epsilon+11$

(c)

Regular expressions for paths that can go through a) no state, b) state 1 only and c) states 1 and 2 only.

Therefore the regular expression corresponding to the language accepted by the DFA is given by: R_{13}^{3} (state 1(i) is the start state and state 3 (j) is the final state). By using the formula:

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} (R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

Where i =1, j= 3 and k = 3; we get

$R_{13}^{3} = R_{13}^{2} + R_{13}^{2} (R_{33}^{2})^* R_{33}^{2}$

$= 1^*01 + 1^*01 (0 + \varepsilon + 11)^* (0 + \varepsilon + 11)$

$= 1^*01 + 1^*01 (0 + \varepsilon + 11)^+]$

Regular expression = **$1^*01 (0 + 11)^*$**

Give all the regular expressions $R_{ij}^{(0)}$, $R_{ij}^{(1)}$, $R_{ij}^{(2)}$ and also write the regular expression corresponding to the language accepted by the automaton given below:

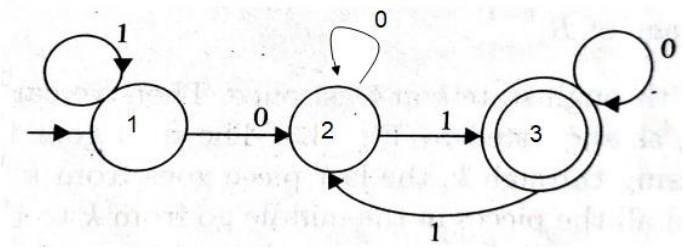
∂	**0**	**1**
$\rightarrow q_1$	q_2	q_1
q_2	q_2	q_3
$*q_3$	q_3	q_2

Answer:

Number of states in DFA = 3; ie: k =3

By renaming the states of DFA as $q_1 = 1$, $q_2 = 2$, $q_3 = 3$

Transition diagram of DFA:



$R_{ij}^{(0)}$	
$R_{11}^{(0)}$	ε + 1
$R_{12}^{(0)}$	0
$R_{13}^{(0)}$	Ø
$R_{21}^{(0)}$	Ø
$R_{22}^{(0)}$	0 + ε
$R_{23}^{(0)}$	1
$R_{31}^{(0)}$	Ø
$R_{32}^{(0)}$	1
$R_{33}^{(0)}$	0 + ε

$R_{ij}^{(1)}$	
$R_{11}^{(1)}$	$1^* + ε = 1^*$
$R_{12}^{(1)}$	1^*0
$R_{13}^{(1)}$	$1^*Ø = Ø$
$R_{21}^{(1)}$	Ø
$R_{22}^{(1)}$	0 + ε
$R_{23}^{(1)}$	1
$R_{31}^{(1)}$	Ø
$R_{32}^{(1)}$	1
$R_{33}^{(1)}$	0 + ε

$R_{ij}^{(2)}$	
$R_{11}^{(2)}$	1^*
$R_{12}^{(2)}$	$1^* 0 0^*$
$R_{13}^{(2)}$	$1^* 0 0^* 1$
$R_{21}^{(2)}$	Ø
$R_{22}^{(2)}$	0^*
$R_{23}^{(2)}$	0^*1
$R_{31}^{(2)}$	Ø
$R_{32}^{(2)}$	$1 0^*$
$R_{33}^{(2)}$	$0 + ε + 10^*1$

Therefore the regular expression corresponding to the language accepted by the DFA is given by:

R_{13}^{3} (state 1(i) is the start state and state 3 (j) is the final state). By using the formula:

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

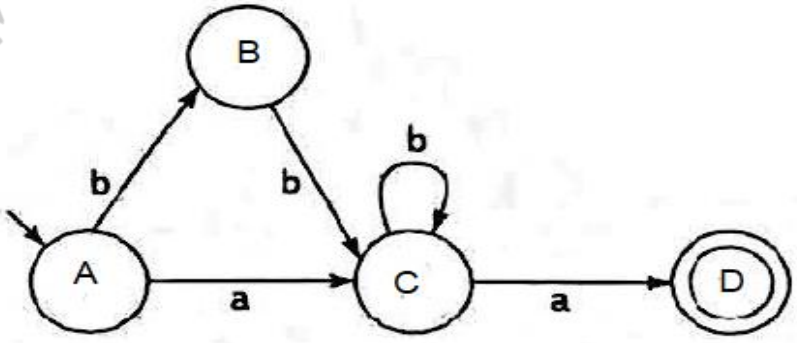
Where i =1, j= 3 and k = 3; we get

$R_{13}^{3} = R_{13}^{2} + R_{13}^{2} (R_{33}^{2})^* R_{33}^{2}$

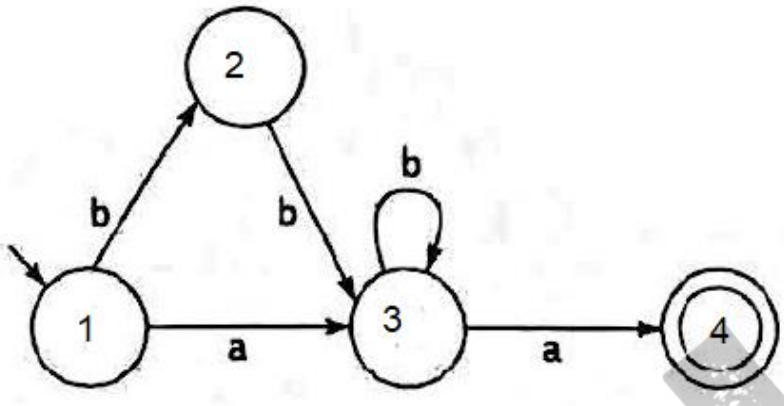
$= 1^* 0 0^* 1 + 1^* 0 0^* 1 ((0 + 10^*1)^*)^* (0 + 10^*1)^*$

Regular expression = $\mathbf{1^* 0^+ 1 (0 + 10^*1)^*}$

Construct the regular expression for the following FSM using Kleen's Theorem.



By renaming the states of DFA, we get:



Regular expressions for paths that can go through 3 intermediate states: states 1, states 2 and states 3 only.

$R_{ij}^{(3)}$	
$R_{11}^{(3)}$	$Ø + ε = ε$
$R_{12}^{(3)}$	b
$R_{13}^{(3)}$	$(a + bb)b^*$
$R_{14}^{(3)}$	$ab^*a + bbb^*a$
$R_{21}^{(3)}$	Ø
$R_{22}^{(3)}$	$Ø + ε = ε$
$R_{23}^{(3)}$	bb^*
$R_{24}^{(3)}$	bb^*a
$R_{31}^{(3)}$	Ø
$R_{32}^{(3)}$	Ø
$R_{33}^{(3)}$	b^*
$R_{34}^{(3)}$	b^*a
$R_{41}^{(3)}$	Ø
$R_{42}^{(3)}$	Ø
$R_{43}^{(3)}$	Ø
$R_{44}^{(3)}$	$Ø + ε = ε$

The regular expression corresponding to the language accepted by the DFA is given by: R_{14}^4 (state 1(i) is the start state and state 4 (j) is the final state). By using the formula:

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

Where i =1, j= 4 and k = 4; we get

$R_{14}^4 = R_{14}^3 + R_{14}^3 (R_{44}^3)^* R_{44}^3$

$= (ab^*a + bbb^*a) + (ab^*a + bbb^*a) (\varepsilon)^* \varepsilon$

We know that $\varepsilon^* = \varepsilon$ and $\varepsilon R = R$

$= (ab^*a + bbb^*a) + (ab^*a + bbb^*a)$

Therefore Regular expression reduces to

$= ab^*a + bbb^*a$

REGULAR LANGUAGES AND PROPERTIES OF REGULAR LANGUAGES.

Regular Languages:

The regular languages are the languages accepted by Finite automata (DFA's, NFA's and ε – NFA's) and defined by regular expressions.

Example: L = { Strings of a's and b's ending abb}

L ={ even of a's and even number of b's} etc….

There are many languages which are not regular. We can prove that certain languages are not regular using one powerful tool called pumping lemma.

Example: L= $\{a^n b^n \mid n>= 0\}$

L = { equal number of 0's and 1's} etc……

PROVING LANGUAGES NOT TO BE REGULAR

Pumping Lemma (PL) for Regular Languages:

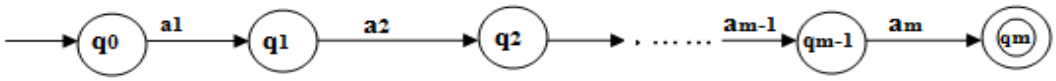
***Theorem (Statement) :**

Let L be a regular language. Then there exists a constant 'n' (which depends on L) such that for every string '**w**' in L such that $|w| \geq n$, we can break w into three strings, w=xyz, such that:

1. $|y| > 0$ ie: $y \neq \varepsilon$
2. $|xy| \leq n$
3. For all $k \geq 0$, the string xy^kz is also in L.

Proof: Suppose L = L(A) for some DFA 'A' and regular language L. Suppose 'A' ha s'n' number of states. Consider any string w = $a_1a_2a_3$……a_m of length 'm' where m >= n and

each a_i is an input symbol. Since we have 'm' input symbols, naturally we should have 'm+1' states, in sequence $q_0, q_1, q_2 \ldots\ldots\ldots\ldots q_m$ where q_0 is → start state and q_m is → final state.



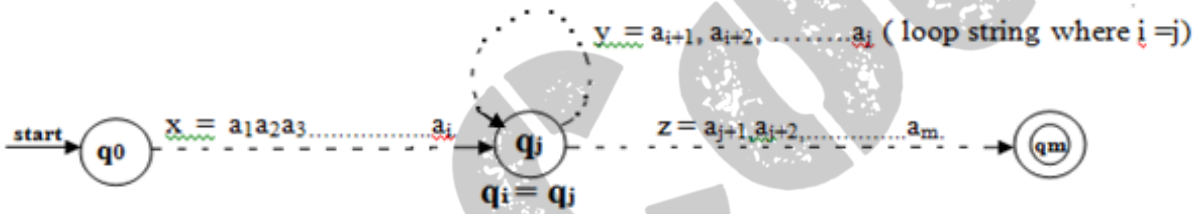
Since $|w| \geq n$, by the pigeonhole principle it is not possible to have distinct transitions, since there are only **'n'** different states. So one of the state can have a loop. Thus we can find two different integers **i** and **j** with $0 \leq \mathbf{i} < \mathbf{j} \leq n$, such that $q_i = q_j$. Now we can break the string **w** = xyz as follows:

$x = a_1a_2a_3\ldots\ldots\ldots\ldots a_i$.

$y = a_{i+1}, a_{i+2}, \ldots\ldots\ldots a_j$ (loop string where i =j)

$z = a_{j+1}a_{j+2},\ldots\ldots\ldots\ldots a_m$.

The relationships among the strings and states are given in figure below:



'x' may be empty in the case that **i=** 0. Also '**z**' may be empty if j = n = m. However, y cannot be empty, since '**i**' is strictly less than '**j**'.

Thus for any $k \geq o$, xy^kz is also accepted by DFA 'A'; that is for a language L to be a regular, xy^kz is in L for all $k \geq o$.

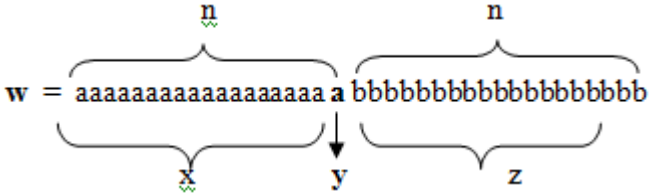
Applications of Pumping lemma:

1. It is useful to prove certain languages are non-regular.
2. It is possible to check whether a language accepted by FA is finite or infinite.

Show that $L= \{a^n b^n \mid n>= 0\}$ is not regular.

Let L is regular language and 'n' be the number of states in FA.

since $|w| = n + n = 2n \geq n$, we can split 'w' into xyz such that $|xy| \leq n$ and $|y| \geq 1$ as



Where |x| = n-1 and |y| = 1 so that |xy| = n-1 +1 = $n \leq n$, which is true.

According to pumping lemma $xy^kz \in L$ for all $k \geq o$.

If '**k**' = 0, the string '**y**' does not appear, so the number of 'a's will be less than number of 'b's ie: $w = a^{n-1} b^n$.

But according to pumping lemma 'n' number of 'a's should be followed by 'n' of 'b's, which is a contradiction to the assumption that the language is regular.

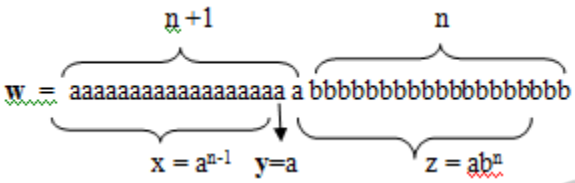
So the language $L= \{a^n b^n \mid n>= 0\}$ is not regular language.

Show that $L= \{a^i b^j \mid i > j\}$ is not regular**.**

Let L is regular language and 'n' be the number of states in FA.

Consider the string $w = a^{n+1} b^n$

since $|w| = n+1 + n = 2n+1 \geq n$, we can split 'w' into '**xyz**' such that $|xy| \leq n$ and $|y| \geq 1$ as



Where |**x**| = n-1 and |y| = 1 so that |xy| = n-1 +1 = $n \leq n$, which is true.

According to pumping lemma $xy^kz \in L$ for all $k \geq o$.

If '**k**' = 0, the string '**y**' does not appear, so the string 'w' has 'n' number of 'a's followed by 'n' number of 'b's. ie: $w = a^n b^n$.

But according to pumping lemma 'n+1'number of 'a's should be followed by 'n' of 'b's, which is a contradiction to the assumption that the language is regular.

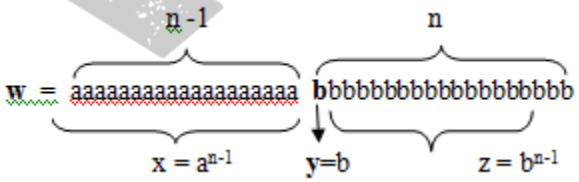
So the language $L= \{a^i bj \mid i>j\}$ is not regular language.

Show that $L= \{w \mid n_a(w) < n_{b(w)} \}$ is not regular**.**

Let L is regular language and 'n' be the number of states in FA.

Consider the string $w = a^{n-1} b^n$

since |**w**| = n-1 + n = 2n-1 ≥ n, we can split '**w**' into '**xyz**' such that $|xy| \leq n$ and $|y| \geq 1$ as



Where |**x**| = n-1 and |y| = 1 so that |xy| = n-1 +1 = $n \leq n$, which is true.

According to pumping lemma $xy^kz \in L$ for all $k \geq o$.

If '**k**' = 0, the string '**y**' does not appear, so the string 'w' has 'n-1' number of 'a's followed by 'n-1' number of 'b's. ie: $w = a^{n-1} b^{n-1}$.

But according to pumping lemma 'n-1'number of 'a's should be followed by 'n' of 'b's, which is a contradiction to the assumption that the language is regular.

So the language $L= \{w \mid n_a(w) < n_{b(w)} \}$ is not regular.

Show that $L= \{w \mid n_a(w) = n_{b(w)} \}$ is not regular**.**

We can prove that L is not regular by taking string $w= a^n b^n \quad \mid n>=0$.

For solution refer problem1**.**

Show that $L= \{a^i b^j \mid i \neq j\}$ is not regular**.**

ie: **i ≠ j** means i > j or i < j; so we can take string 'w' = $a^{n+1}b^n$ or $w= a^{n-1}b^n$.

Solution is similar to the previous problems.

Show that $L= \{a^n b^m c^{n+m} \mid n,m >= 0\}$ is not regular**.**

Let L is regular language and 'n' be the number of states in FA.

Since L is regular it is closed under homomorphism. So we can take h(a) = a, h(b) = a and h(c) = c.

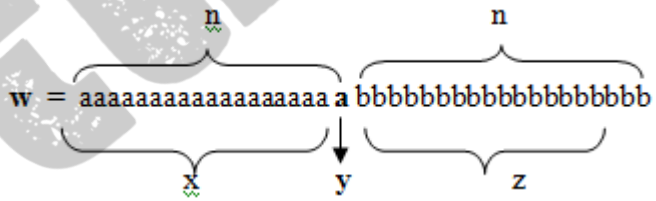
Now the language L is reduced to $L = \{a^n a^m c^{n+m} \mid n+m >= 0\}$

ie: $L= \{a^{n+m} c^{n+m} \mid n+m >= 0\}$ which is in the form

$L = \{ a^i b^j \mid i >=0\}$,

Consider $w = a^n b^n$

since $|w| = n +n = 2n \geq n$, we can split 'w' into xyz such that $|xy| \leq n$ and $|y| \geq 1$ as



Where **|x|** = n-1 and |y| = 1 so that |xy| = n-1 +1 = $n \leq n$, which is true.

According to pumping lemma $xy^kz \in L$ for all $k \geq o$.

If '**k**' = 0, the string '**y**' does not appear, so the number of 'a's will be less than number of 'b's ie: $w = a^{n-1} b^n$.

Which is a contradiction to the assumption that the language is regular. So the given language

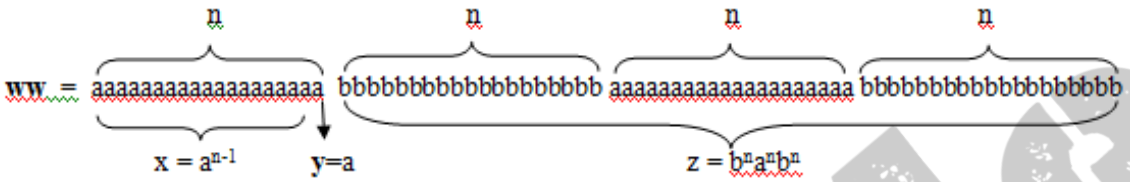
So the language $L= \{a^n b^n \mid n>= 0\}$ is not regular language $L= \{a^n b^m c^{n+m} \mid n,m >= 0\}$ is not regular**.**

Show that L= {ww | w € $(a+b)^*$ } is not regular.

Let L is regular language and 'n' be the number of states in FA.

Consider the string w = $a^n b^n$, therefore ww = $a^n b^n a^n b^n$

since |**w**| = n+n+n + n = 4n ≥ n, we can split '**w**' into '**xyz**' such that |xy| ≤ n and |y |≥ 1 as



Where |**x**| = n-1 and |y| = 1 so that |xy| = n-1 +1 = n ≤ n, which is true.

According to pumping lemma xy^kz € L for all k ≥ o.

If '**k**' = 0, the string '**y**' does not appear, so the number of 'a's on the left of first b will be less than number of 'a's after the first b

ie: ww = $a^{n-1} b^n a^n b^n$.

Which is a contradiction to the assumption that the language is regular.

So the language L= {ww | w € $(a+b)^*$} is not regular is not regular language.

Show that L= {ww^R | w € $(a+b)^*$ } is not regular.

Let L is regular language and 'n' be the number of states in FA.

Consider the string w = $a^n b^n$, therefore ww^R = $a^n b^n b^n a^n$

since |**w**| = n+n+n + n = 4n ≥ n, we can split '**w**' into '**xyz**' such that |xy| ≤ n and |y |≥ 1 as

x = a^{n-1}

y = a

z = $b^n b^n a^n$

Where |**x**| = n-1 and |y| = 1 so that |xy| = n-1 +1 = n ≤ n, which is true.

According to pumping lemma xy^kz € L for all k ≥ o.

If '**k**' = 0, the string '**y**' does not appear, so the number of 'a's on the left of first b will be less than number of 'a's after the first b

ie: ww^R = $a^{n-1} b^n b^n a^n$.

Which is a contradiction to the assumption that the language is regular.

So the language L= {ww^R | w € $(a+b)^*$} is not regular is not regular language.

Show that L= {$a^{n!}$ | n ≥ 0 } is not regular.

Let L is regular language and 'n' be the number of states in FA.

Consider the string $w = a^{n!}$

since $|w| = n \geq n$, we can split '**w**' into '**xyz**' such that $|xy| \leq n$ and $|y| \geq 1$ as

$x = a^i$

$y = a^j$

$z = a^{n! - i - j}$

Where $|x| = i$ and $|y| = j$ so that $|xy| = i + j \leq n$ and $|y| = j \geq 1$

According to pumping lemma $xy^kz \in L$ for all $k \geq o$.

If '**k**' = 0, the string '**y**' does not appear, ie: $a^i (a^j)^k a^{n!-i-j} = a^{n!-j} \notin L$

It is clear that $n! > n! - j$ ie: when j=1, $n! > n! - 1$

But according to pumping lemma, language to be regular, $n! = n! - 1$, which is not true and it is a contradiction. So L can not be regular.

Show that $L= \{0^n \mid n \text{ is prime } \}$ is not regular.

Let L is regular language and 'n' be the number of states in FA.

Consider the string $w = 0^n$

since $|w| = n \geq n$, we can split '**w**' into '**xyz**' such that $|xy| \leq n$ and $|y| \geq 1$ as

$x = 0^i$

$y = 0^j$

$z = 0^{n - i - j}$

Where $|x| = i$ and $|y| = j$ so that $|xy| = i + j \leq n$ and $|y| = j \geq 1$

According to pumping lemma $xy^kz \in L$ for all $k \geq o$.

If '**k**' = 0, the string '**y**' does not appear, ie: $0^i (0^j)^k 0^{n-i-j} = 0^{n-j} \notin L$

It is clear that when j=1, $n \neq (n-1)$ as a prime

But according to pumping lemma, language to be regular, when n is prime, n-1 is also prime, which is not true and it is a contradiction. So L can not be regular.

Show that $L= \{0^n \mid n \text{ is a perfect square } \}$ is not regular.

Let L is regular language and 'n' be the number of states in FA.

Consider the string $w = 0^{n^2}$

since $|w| = n^2 \geq n$, we can split '**w**' into '**xyz**' such that $|xy| \leq n$ and $|y| \geq 1$ as

$x = 0^i$

$y = 0^j$

$z = 0^{n^2 - i - j}$

Where $|x| = i$ and $|y| = j$ so that $|xy| = i + j \leq n$ and $|y| = j \geq 1$

According to pumping lemma $xy^kz \in L$ for all $k \geq o$.

If '**k**' = 0, the string '**y**' does not appear, ie: $0^i (0^j)^k 0^{n^2 - i - j} = 0^{n^2 - j} \notin L$

It is clear that when j=1, $(n^2 - 1)$ is not a perfect square.

But according to pumping lemma, language to be regular, when n^2 is a perfect square, n^2-1 is also a perfect square, which is not true and it is a contradiction. So L can not be regular.

Show that $L= \{0^n \mid n \text{ is a perfect cube } \}$ is not regular**.**

Let L is regular language and 'n' be the number of states in FA.

Consider the string $w = 0^{n^3}$

since $|\mathbf{w}| = n^3 \geq n$, we can split '**w**' into '**xyz**' such that $|xy| \leq n$ and $|y| \geq 1$ as

$x = 0^i$

$y = 0^j$

$z = 0^{n^3 - i - j}$

Where $|\mathbf{x}| = i$ and $|y| = j$ so that $|xy| = i + j \leq n$ and $|y| = j \geq 1$

According to pumping lemma $xy^kz \in L$ for all $k \geq o$.

If '**k**' = 0, the string '**y**' does not appear, ie: $0^i (0^j)^k 0^{n^3 - i - j} = 0^{n^3 - j} \notin L$

It is clear that when j=1, $(n^3 - 1)$ is not a perfect cube.

But according to pumping lemma, language to be regular, when n^3 is a perfect cube, n^3-1 is also a perfect cube, which is not true and it is a contradiction. So L can not be regular.

LEXICAL ANALYSIS PHASE OF COMPILER DESIGN

Lexical Analysis

Lexical analysis reads characters from left to right and groups into tokens. A simple way to build lexical analyzer is to construct a diagram to illustrate the structure of tokens of the source program. We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a *lexical-analyzer generator* and compiling those patterns into code that functions as a lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patterns, not the entire program.

Three general approaches for implementing lexical analyzer are:

1. Use lexical analyzer generator (LEX) from a regular expression based specification that provides routines for reading and buffering the input.
2. Write lexical analyzer in conventional language (C) using I/O facilities to read input.
3. Write lexical analyzer in assembly language and explicitly manage the reading of

input.

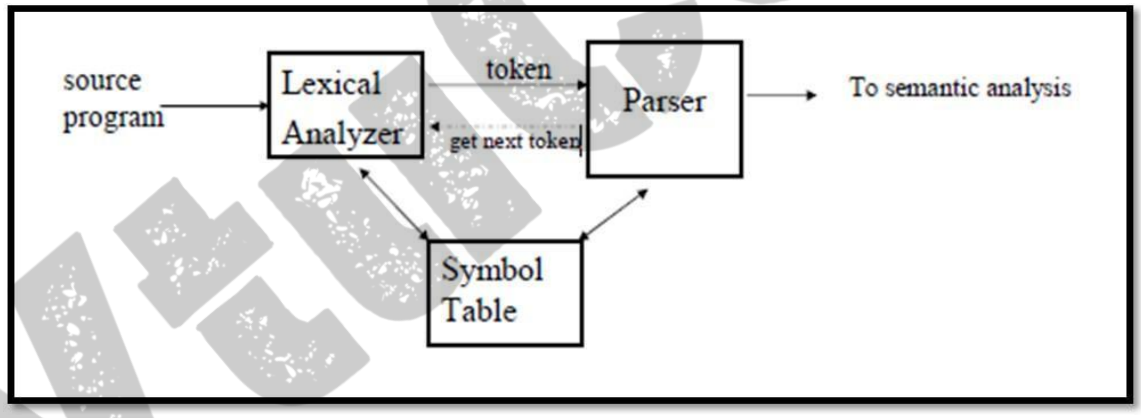
Note: The speed of lexical analysis is a concern in compiler design, since only this phase reads the source program character-by character.

Discuss the various issues of lexical analysis.

1. Lexical analyzer reads the source program character by character to produce tokens.
2. Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.
3. Normally L.A. don't return a comment as a token. It skips a comment, and return the next token (which is not a comment) to the parser.
4. Correlating error messages: It can associate a line number with each error message. In some compilers it makes a copy of the source program with the error messages inserted at the appropriate positions.
5. If the source program uses a **macro-preprocessor**, the expansion of macros may be performed by the lexical analyzer.

Role of Lexical Analyzer

Explain the role of lexical analyzer with a block diagram.



- Read the input characters of the source program, group them into lexemes and produces output as a sequence of tokens.
- It interacts with the symbol table.
- Initially parser calls the lexical analyzer, by means of getNextToken command.
- In response to this command LA read characters from its input until it can identify the next lexeme and produce a token for that lexeme, which can be returned to parser.
- It eliminates comments and white space.

- It displays error messages with line number.

Lexical Analysis versus Parsing:

Why analysis phase of compiler is separated into lexical analysis and parser**.**

The main reasons are;

1. **Simplicity of design**: Separation allows us to simplify the task. For example a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the LA.
2. **Compiler efficiency is improved**: A specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. **Compiler portability is enhanced**: Input device peculiarities can be restricted to the lexical analyzer.

Tokens, Patterns and Lexemes:

Define the following terms with examples:

i. Token.

ii. Pattern.

iii. Lexeme.

Token: It describes the class or category of input string. A *token* is a pair consisting of a token name and an optional attribute value.

For example, identifier, keywords, constants are called tokens.

Pattern: Set of rule that describes the tokens. It is a description of the form that the lexemes of a token may take.

Example: letter [A-Za-z].

Lexeme: Sequence of characters in the source program that are matched with the pattern of the token.

Example: int, a, num, ans etc.

Token representation:

In many programming languages, the following classes cover most or all of the tokens:

i. One token for each keyword; The pattern for a keyword is the same as the keyword itself.

ii. Tokens for the operators, either individually or in classes such as the token comparison.

iii. One token representing all identifiers.

iv. One or more tokens representing constants, such as numbers and literal.

v. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Attributes for tokens

A token has only a single attribute that is a pointer to the symbol-table entry in which the information about the token is kept.

Example: The token names and associated attribute values for the statement E = M * C ** 2 are written below as a sequence of pairs.

<id, pointer to symbol-table entry for E>

<assign_op>

<id, pointer to symbol-table entry for M>

<mult_op>

<id, pointer to symbol-table entry for C>

<exp_op>

<number, integer value 2>

Lexical errors:

1. It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string fi is encountered for the first time in a C program in the context:

   fi (a == f (x))

   A lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier. Since fi is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler- probably the parser in this case handle an error due to transposition of the letters.

2. Suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Possible error-recovery actions are:

1. Delete one character from the remaining input.

2. Insert a missing character into the remaining input.

3. Replace a character by another character.

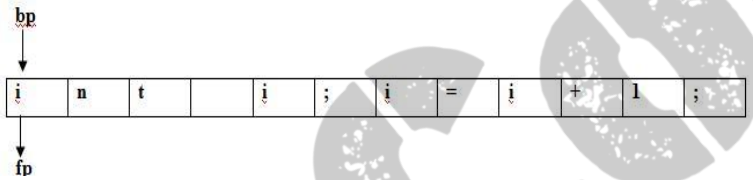
4. Transpose two adjacent characters.

INPUT BUFFERING

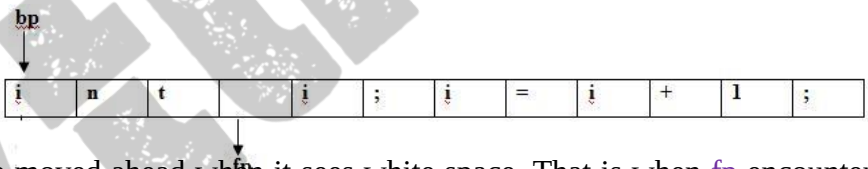
Lexical analyzer scans the input string from left to right, one character at a time. It uses two pointers as:

- beginLexeme pointer
- forward pointer

To keep track of the position of the input scanned. Initially both the pointers point to the first character of the input string.



The forward pointer moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as forward pointer encounters a blank space, the lexeme is identified.



The fp will be moved ahead when it sees white space. That is when fp encounters white space it ignores and moves ahead. Then both fp and bp is set at next token.

What is meant by input buffering?

To recognize tokens reading data/source program from hard disk is done. Accessing hard disk each time is costly and time consuming so special buffer technique has been developed to reduce the amount of overhead required. This process of reading source program into a buffer is called input buffering.

A block of data is first read into a buffer and scanned by lexical analyzer. There are two methods used

1. One buffer
2. Two buffer

One buffer scheme:

Here only one buffer is used to store the input string. But the problem with this scheme is that, if a lexeme is very long, then it crosses the buffer boundary. To scan the remaining part of lexeme, the buffer has to be refilled, that makes overwriting of first part of lexeme. Sometimes it may result in loss of data due to the user misinterpretation.

Two Buffer scheme:

Why two buffer schemes is used in lexical analysis? Explain.

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized two buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

- Here a buffer (array) divided into two N-character halves, where N = number of characters on one disk block Ex: 4096 bytes – If fewer than N characters remain in the input file , then special character, represented by **eof**, marks the end of source file and it is different from input character.
- One read command is used to read N characters. Two pointers are maintained: beginning of the lexeme pointer and forward pointer.
- Initially, both pointers point to the first character of the next lexeme.
- Using this method we can overcome the problem faced by one buffer scheme, even though the input is lengthier the user knows from where he has to begin in the next buffer, as he can see the contents of previous buffer. Thus there is no scope for loss of any data.

Sentinels:

In two buffering scheme we must check the forward pointer, each time it is incremented. Thus we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine these two tests, if we use a sentinel character at the end of buffer.

How is input buffering of lexical analyzer is implemented with sentinels?

OR

Define sentinels. Write an algorithm for look ahead code with sentinels.

Sentinel is a special character inserted at the end of buffer, that cannot be a part of source program; **eof** is used as sentinel.

Look ahead code:

```
Switch(*forward++)
 {
 case eof:
        if (forward is at end of first buffer)
        { reload second buffer;
          forward = beginning of second buffer;
        }
    else if (forward is at end of second buffer)
        { reload first buffer;
          forward = beginning of first buffer;
        }
    else /* eof within a buffer marks the end of input terminate lexical analysis*/
    break;
     case for the other characters
}
```

Operations on Languages:

Give the formal definitions of operations on languages with notations.

In lexical analysis the most important operations on languages are:

i. Union
ii. Concatenation
iii. Star closure
iv. Positive closure.

These operations are formally defined as follows

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)

- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.
- *, concatenation and | has highest to lowest precedence with left associative.
- If two regular expression 'r' and 's' denote the same language, we say 'r' and 's' are equivalent and write r=s

Write the definition of regular expression

Regular expression over alphabet ∑ can be defined as follows:

ε- is a regular expression for the set containing empty string.

a is a regular expression, if a belongs to ∑ that is {**a**} set containing the string a.

Suppose **r** and **s** are regular expression denoting the languages L(r) and L(s) then,

(r|s) is a regular expression denoting L(r) U L(s).

rs is a regular expression denoting L(r)L(s)

(r)* is a regular expression denoting (L(r))*

The following table shows some of the regular expressions along with their possible regular sets:

Regular expression	Set
a|b	{a, b}
(a|b)(a|b) or aa|ab|ba|bb	{aa, ab, ba, bb}
a*	{ε, a, aa, aaa,…}
(a|b)* or (a*b*)*	{ε, a,aa,b,bb,…}
a|a*b	{a, b, ab,aab,aaab,…}

Algebraic properties

r|s = s|r

r|(s|t)= (r|s)|t

(rs)t = r(st)

r(s|t) = rs|rt

(s|t)r = sr|tr

ε r = r

r ε = r

r* = (r| ε)* r** = r*

[abc] denotes the regular expression a|b|c

[a-z] denotes the regular expression a|b|……|z

Regular definition:

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

Define regular definition and write the regular definition for C identifier.

(5)

A regular definition is a sequence of the definitions of the form:

d1 → r1 where $\mathbf{d_n}$ is a distinct name and $\mathbf{r_n}$ is a regular expression over symbols in $\sum \cup \{ d_1, d_2, d_3, \ldots\ldots\ldots\ldots d_{i-1} \}$

d2 → r2

. .

basic symbols previously defined names

. .

d_n → r_n

Regular definition for 'C' identifier:

letter → A | B |…….|Z | a| b |…| z | _

digit → 0 | 1 | ….. | 9

id → letter (letter | digit)*

OR

letter → [A-Za-z_]

digit → [0-9]

id → letter (letter | digit)*

Write the regular definition for an unsigned number

digit → 0 | 1| …..| 9

digits → (digit)$^+$

optionalFraction → **.** digits | ε

optionalExponent → **(** E **(** + | - | ε **)** digits **) | ε**

number → digits optionalFraction optionalExponent

OR

digit → [0-9]

digits → digit$^+$

number → digits (. digits) ? (E [+-]? digits)?

Recognition of Tokens

Our current goal is to perform the lexical analysis needed for the following grammar.

stmt → if expr then stmt
| if expr then stmt else stmt
| ε
expr → term relop term // relop is relational operator =, >, etc
| term
term → id
| number

Recall that the terminals are the tokens, the nonterminals produce terminals.

A regular definition for the terminals is

digit → [0-9]
digits → digits$^+$
number → digits (. digits)? (E[+-]? digits)?
letter → [A-Za-z]
id → letter (letter | digit)*
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>

Lexeme	Token	Attribute
Whitespace	ws	—
if	if	—
then	then	—
else	else	—
An identifier	id	Pointer to table entry
A number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

We also want the lexer to remove whitespace so we define a new token

ws → (blank | tab | newline) +

where blank, tab, and newline are symbols used to represent the corresponding ascii characters.

Recall that the lexer will be called by the parser when the latter needs a new token. If the lexer then recognizes the token ws, it does *not* return it to the parser but instead goes on to recognize the next token, which is then returned. Note that you can't have two consecutive ws tokens in the input because, for a given token, the lexer will match the **longest** lexeme starting at the current position that yields this token. The table on the right summarizes the situation.

For the parser, all the relational ops are to be treated the same so they are all the same token, relop. Naturally, other parts of the compiler, for example the code generator, will need to distinguish between the various relational ops so that appropriate code is generated. Hence, they have distinct attribute values.

vtucode

Specification of Token

To specify tokens Regular Expressions are used.

Recognition of Token: To recognize tokens there are 2 steps

1. Design of Transition Diagram
2. Implementation of Transition Diagram

Transition Diagrams

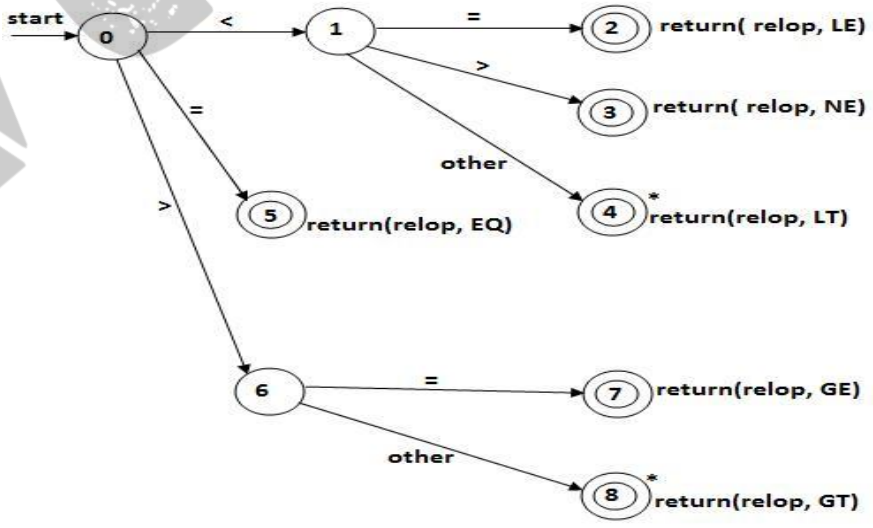
A transition diagram is similar to a flowchart for (a part of) the **lexer.** We draw one for each possible token. It shows the decisions that must be made based on the input seen. The two main components are circles representing *states* (think of them as decision points of the lexer) and arrows representing *edges* (think of them as the decisions made).

It is fairly clear how to write code corresponding to this diagram. You look at the first character, if it is <, you look at the next character. If that character is =, you return (relop, LE) to the parser. If instead that character is >, you return (relop, **NE**). If it is another character, return (relop, LT) and adjust the input buffer so that you will read this character again since you have not used it for the current lexeme. If the first character was =, you return (relop, **EQ**).

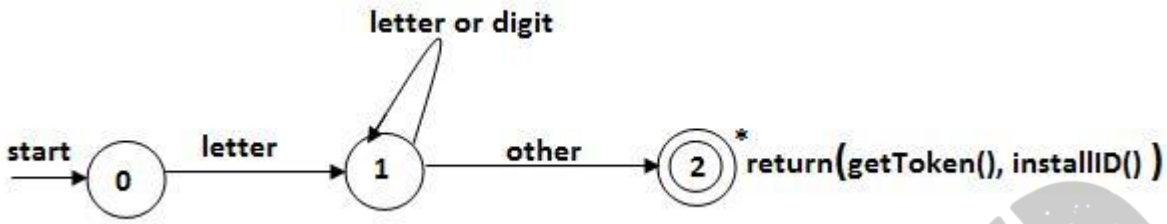
Write the transition diagram to recognize the token given below:

i. relop (relational operator)
ii. Identifier and keyword
iii. Unsigned number
iv. Integer constant
v. Whitespace

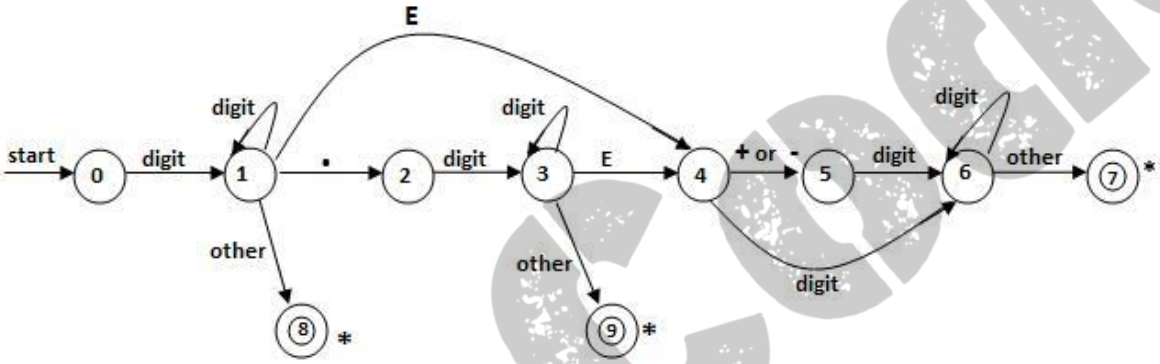
i. Transition diagram for relop:



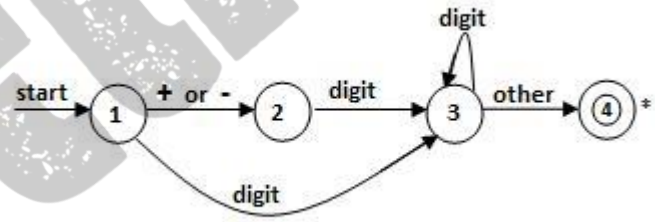
ii. Transition diagram for identifier and keyword:



iii. Unsigned number:

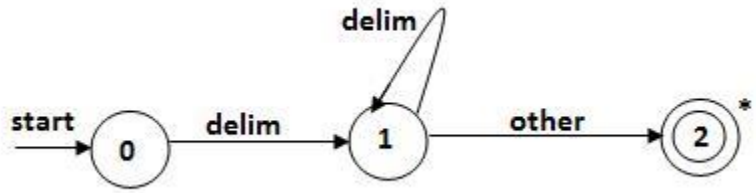


iv. Integer constant:



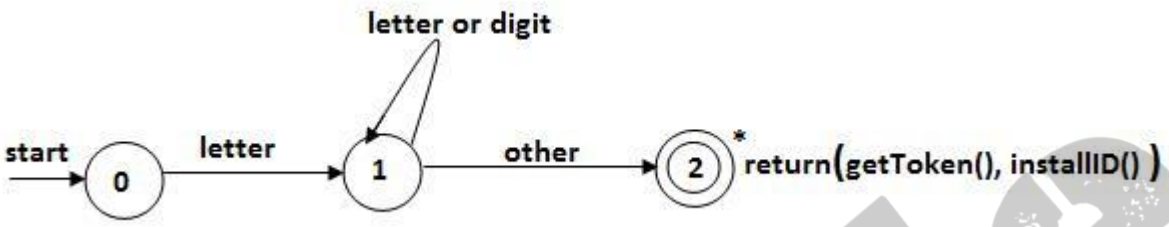
v. Whitespace:

Whitespace characters are represented by delimiter, where **delim** includes the characters like blank, tab, new line and other characters that are not considered by the language design to be part of any token.



Recognition of reserved words and identifiers.

Draw the transition diagram for identifiers and keywords. How do you handle reserve words that look like identifiers?



There are two ways we can handle reserved words that look like identifiers:

1. **Install the reserved words in the symbol table initially:** When we find an identifier, a call to installID( ) function places that identifier into the symbol table if it is not already there and returns a pointer to the symbol table entry. The function getToken( ) examines the symbol table for the lexeme found, and returns token name as either **id** or one of the keyword token that was initially installed in the table.
2. Create separate transition diagrams for each keyword

Architecture of a transition diagram based lexical analyzer

The idea is that we write a piece of code for each decision diagram. This piece of code contains a case for each state, which typically reads a character and then goes to the next case depending on the character read. **nextchar()** is used to read a next char from the input buffer. The numbers in the circles are the names of the cases. Accepting states often need to take some action and return to the parser. Many of these accepting states (the ones with stars) need to restore one character of input. This is called **retract()** in the code.

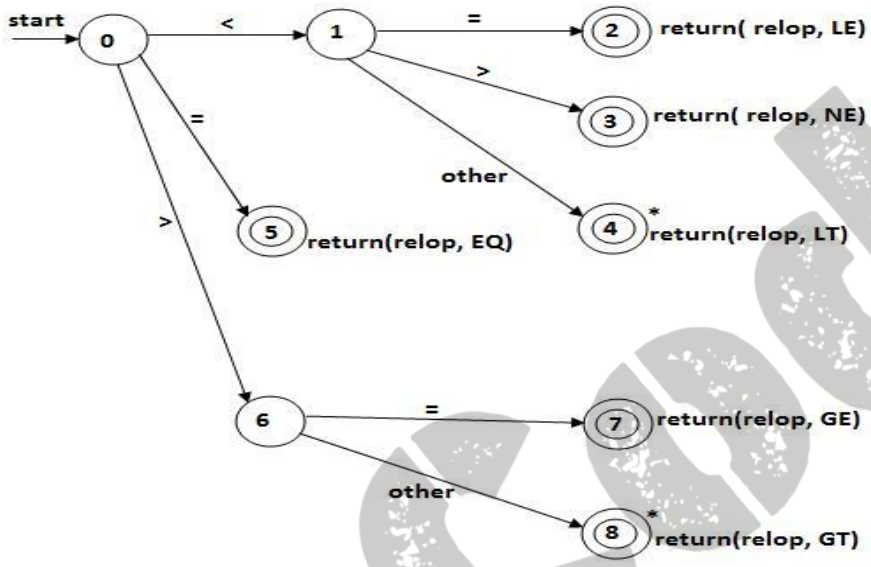
What should the code for a particular diagram do if at one state the character read is not one of those for which a next state has been defined? That is, what if the character read is not the label of any of the outgoing arcs? This means that we have failed to find the token corresponding to this diagram.

The code calls **fail(),** is **not** an error case. It simply means that the current input does not match this particular token. So we need to go to the code section for another diagram after restoring the input pointer so that we start the next diagram at the point where this failing diagram **started**. If we have tried all the diagram, then we have a real failure and need to print an error message and

perhaps try to repair the input.

Construct the transition diagram for relational operators (=, <, <=, >, >=, and < >). Write a lexical analyzer to recognize the above mentioned relational operators.(Write code for START state, one intermediate state and one final state).

Transition diagram:



Coding part:

```
TOKEN getRelop( )
{
        TOKEN  retToken  =  new(RELOP);
          while(1)
        {                       /* repeat character processing until a return or failure occurs */

Switch (state)
          {
        case 0: c = nextChar( );
                if ( c == '< ‘ ) state = 1; else
                if ( c == '=' ) state = 5; else if
                ( c == '>' ) state = 6;
                else fail( );                   /* lexeme is not a relational operator…other */
                break;
```

```
        case 1: c = nextChar( );
                if ( c == '=' ) state = 2;
                else if ( c == '>' ) state = 3;
                else if ( c == other character ) state = 4;
                else fail( );                   /* lexeme is not a relational operator…other */
                break;
                ……..
                ……..
        case 8: retract( );
                retToken.attribute = GT;
                return(retToken);
    }
  }
}
```

vtucode