UNIT 2 AUTOMATA AND LANGUAGES

2.0 INTRODUCTION

In the previous unit we have examined sets, relations and functions. In this unit we will introduce a special kind of a set i.e., a language which is a set of strings over an alphabet. First we describe what is a language and what are the operations we perform on languages. Certain set of strings or languages are represented in algebraic fashion, then these algebraic expressions of languages are called **regular expressions**. A language represented by a regular expression is called a regular language. Then we introduce a notion of finite automata, also called finite state machine or deterministic finite automata. that recognizes regular languages. Finally, we introduce another kind of a theoretical machine, called nondeterministic finite automata (NFA). In deterministic finite automata (DFA), there is a unique next state for transition on input in a given state. If we relax this condition of uniqueness of the next state in DFA, we get NFA. But any set accepted by NFA can be also accepted by DFA. However the concept of non-determinism plays an important role in both the theory of computation and design and analysis of algorithm, specially in defining complexity classes. In the next unit we will examine a more powerful machine and explain the terms computability and complexity classes.

2.1 OBJECTIVES

After studying this unit, you should be able to:

Define alphabet, substring;

Define a language and various operations on languages;

Define and use a regular expression;

Define a finite automata for computation of a language;

Design a finite automata for a known language;

Define the term nondeterministic finite automata

Design nondeterministic finite automata for a known language

2.2 REGULAR EXPENSION

In this unit, first we shall discuss the definitions of alphabet, string, and language with some important properties.

2.2.1 Introduction to Defining of Languages

For a language, defining rules can be of two types. The rules can either tell us how to test a string of alphabet letters that we might be presented with, to see if it is a valid word, i.e., a word in the language or the rules can tell us how to construct all the words in the language by some clear procedures.

Alphabet: A finite set of symbols/characters. We generally denote an alphabet by $\sum$. If we start an alphabet having only one letter, say, the letter z, then $\sum = \{z\}$

Letter: Each symbol of an alphabet may also be called a letter of the alphabet or simply a letter.

Language over an alphabet: A set of words over an alphabet. Languages are denoted by letter L with or without a subscript.

String/word over an alphabet: Every member of any language is said to be a string or a world.

Example 1: Let L_1 be the language of all possible strings obtained by L_1 = {z, zz, zzz, zzzz . . . . . .}

This can also be written as
$L_1 = \{z^n\}$ for n = 1, 2, 3, …..

A string of length zero is said to be **null string** and is represented by ∧.

Above given language L_1 does not include the null string. We could have defined it so as to include ∧. Thus, $L = \{z^n \mid n=0, 1, 2, 3\ldots\}$ contains the null string.

In this language, as in any other, we can define the operation of concatenation, in which two strings are written down side by side to form a new longer string. Suppose u = ab and v = baa, then uv is called concatenation of two strings u and v and is uv = abbaa and vu = baaab. The operation of concatenation is analogous to addition:

z^n concatenated with z^m is the word z^{n+m}.

Example 2: If the word zzz is called c and the word zz is called d, then the word formed by concatenating c and d is

cd = zzzzz

When two words in our language L_1 are concatenated they produce another word in the language L_1. However, this may not be true in all languages.

Example 3: If the language is L_2 = {z, zzz, zzzzz, zzzzzzz…..}

$= \{z^{odd}\}$

$= \{z^{2n+1}$ for n = 0, 1, 2, 3….}

then c = zzz and d = zzzzz are both words in L_2, but their concatenation cd = zzzzzzzz is not a word in L_2. The reason is simple that member of L_2 are of odd length while after concatenation it is of even length.

Note: The alphabet for L_2 is the same as the alphabet for L_1.

Example 4: A Language L_3 may denote the language having strings of even lengths include of length 0. In other words, L_3 = { ∧zz, zzzz, …..}

Another interesting language over the alphabet ∑ = {z} may be

Example 5: $L_4 = \{z^p : p \text{ is a prime natural number}\}$.
There are infinitely many possible languages even for a single letter alphabet ∑ = {z},

In the above description of concatenation we find very commonly, that for a single letter alphabet when we concatenate c with d, we get the same word as when we concatenate d with c, that is cd = dc **But this relationship does not hold for all languages**. For example, in the English language when we concatenate "Ram" and "goes" we get "Ram goes". This is, indeed, a word but distinct from "goes Ram".

Now, let us define the reverse of a language L. If c is a word in L, then reverse (c) is the same string of letters spelled backward.
The reverse (L) = {reverse (w), w ∈ L}

Example 6: Reverse (zzz) = zzz

Reverse (173) = 371

Let us define a new language called PALINDROME over the alphabet ∑ = {a,b}.

PALINDROME = {∧ , and all strings w such that reverse (w) = w}

= {∧ , a, b, aa, bb, aaa, aba, bab, bbb, aaaa, abba, …}

Concatenating two words in PALINDROME may or may not give a word in palindrome, e.g., if u = abba and v = abbcba, then uv = abbaabbcbba which is not palindrome.

2.2.2 Kleene Closure Definition

Suppose an alphabet ∑ , and define a language in which any string of letters from ∑ is a word, even the null string. We shall call this language the closure of the alphabet. We denote it by writing * after the name of the alphabet as a superscript, which is written as $\sum^*$. This notation is sometimes also known as **Kleene Star**.

For a given alphabet ∑ , the language $\sum^*$ ($sigma^*$) consists of all possible strings, including the null string.

For example, If ∑ = {z}, then, $\sum^* = L_1$ = {∧, z, zz, zzz …..}

Example 7: If ∑= {0, 1}, then, ∑ * = {∧, 0, 1, 00, 01, 10, 11, 000, 001 …..}
So, we can say that Kleene Star is an operation that makes an infinite language of strings of letters out of an alphabet, if the alphabet, ∑ ≠ ϕ. However, by the definition alphabet ∑ may also be ϕ. In that case, $\sum^*$ is finite. By "infinite language, we mean a language with infinitely many words.

Now, we can generalise the use of the star operator to languages, i.e., to a set of words, not just sets of alphabet letters.

Definition: If s is a set of words, then by s^* we mean the set of all finite strings formed by concatenating words from s, where any word may be used as often.

Example 8: If s = {cc, d}, then

s^* = {∧ or any word composed of factors of cc and d}
= {∧ or all strings of c's and d's in which c's occur in even clumps}.
The string ccdcccd is not in s^* since it has a clump of c's of length 3.
$\{x : x \wedge = \text{or } x = (cc)^{i_1} d^{j_1} (cc)^{i_2} d^{j_2} \ldots\ldots (cc)^{i_m} (d)^{j_m} \}$ where $i_1, j_1, \ldots\ldots i_m, j_m \geq 0$

Positive Closure: If we want to modify the concept of closure to refer to only the concatenation leading to non-null strings from a set s, we use the notation + instead of *. This plus operation is called positive closure.

Theorem 1: For any set s of strings prove that $s^* = (s^*)^* = s^{**}$

Proof: We know that every word in s^{**} is made up of factors from s^*.
Also, every factor from s^* is made up of factors from s.
Therefore, we can say that every word in s^{**} is made up of factors from s.

First, we show $s^{**} \subset s^*$. (i)
Let $x \in s^{**}$…. Then $x = x_1 \ldots\ldots x_n$ for some $x_1 \in s^*$ which implies $s^{**} \subset s^*$

Next, we show $s^* \subset s^{**}$.
$s^* \subset s^{**}$ (ii)

By above inclusions (i) and (ii), we prove that
$s^* = s^{**}$

Now, try some exercises.

Ex.1) If u = ababb and v = baa then find
(i) uv (ii) vu (iii) uv (iv) vu (v) uuv.

Ex.2) Write the Kleene closure of the following
(i) {aa, b}
(ii) {a, ba}

2.2.3 Formal Definition of Regular Expressions

Certain sets of strings or languages can be represented in algebraic fashion, then these algebraic expressions of languages are called **regular expressions**. Regular expressions are in **Bold** face. The symbols that appear in regular use of the letters of the alphabet ∑ are the symbol for the null string ∧ , parenthesis, the star operator, and the plus sign.

The set of regular expressions is defined by the following rules:

1. Every letter of ∑ can be made into a regular expression ∧ itself is a regular expression.
2. Φ is a regular expression

3 If **l** and **m** are regular expressions, then so are

(i) **(l)**

(ii) **lm**
(iii) **l+m**
(iv) $\mathbf{l}^*$
(v) $\mathbf{l}^+ = \mathbf{ll}^*$

4 Nothing else is regular expression.

For example, now we would build expression from the symbols 0,1 using the operations of union, concatenation, and Kleene closure.
(i) **01** means a zero followed by a one (concatenation)
(ii) **0+1** means either a zero or a one (union)
(iii) $\mathbf{0}^*$ means ∧**+0+00+000+**….. (Kleen closure) .

With parentheses, we can build larger expressions. And, we can associate meanings with our expressions. Here's how

Expression	**Set represented**
$\mathbf{(0+1)}^*$	all strings over {0,1}
$\mathbf{0}^*\mathbf{10}^*\mathbf{10}^*$	strings containing exactly two ones
$\mathbf{(0+1)}^*\mathbf{11}$	strings which end with two ones.

The language denoted/represented by the regular expression R is L(R).

Example 9: The language L defined by the regular expression **ab*a** is the set of all strings of a's and b's that begin and end with a's, and that have nothing but b's inside.
L = {∧ aa, aba, abba, abbba, abbbba, }

Example 10: The language associated with the regular expression $\mathbf{a}^*\mathbf{b}^*$ contains all the strings of a's and b's in which all the a's (if any) come before all the b's (if any).

L = {∧, a, b, aa, ab, bb, aaa, aab, abb, bbb, aaa,…)

Note that ba and aba are not in this language. Notice also that there need not be the same number of a's and b's.

Example 11: Let us consider the language L defined by the regular expression $\mathbf{(a+b)}^*$ $\mathbf{a(a+b)}^*$**.** The strings of the language L are obtained by concatenating a string from the language corresponding to $\mathbf{(a+b)}^*$ followed by a followed by a string from the language associated with $\mathbf{(a+b)}^*$. We can also say that the language is a set of all words over the alphabet $\sum$ = {a,b} that have an a in them somewhere.

To make the association/correspondence/relation between the regular expressions and their associated languages more explicit, we need to define the operation of multiplication of set of words.

Definition: If S and T are sets of strings of letters (they may be finite or infinite sets), we define the product set of strings of letters to be. ST = {all combinations of a string from S concatenated with a string from T in that order}.

Example 12: If S = {a, aa, aaa}, T = {bb, bbb} Then, ST = {abb, abbb, aabb, aabbb, aaabb, aaabbb}.

Example 13: If S = {a bb bab}, T = {∧ bbbb}

Then, ST = {a, bb, bab, abbbb, bbbbbb ,babbbbb}

Ex.3) Find a regular expression to describe each of the following languages:

(a) {a,b,c}

(b) {∧,a,abb,abbbb,….}

Ex.4) Find a regular expression over the alphabet {0,1,} to describe the set of all binary numerals without leading zeroes (except 0 itself). So the language is the set

{0,1,10,11,100,101,110,111,…}.

2.2.4 Algebra of Regular Expressions

There are many general equalities for regular expressions. We will list a few simple equalities together with some that are not so simple. All the properties can be verified by using properties of languages and sets. We will assure that R,S and T denote the arbitrary regular expressions.

Properties of Regular Expressions

1. $(R+S)+T = R+(S+T)$
2. $R+R = R$
3. $R+\phi = \phi+R = R.$
4. $R+S = S+R$
5. $R\phi = \phi R = \phi$
6. $R\wedge = \wedge R = R$
7. $(RS)T = R(ST)$
8. $R(S+T) = RS+RT$
9. $(S+T)R = SR+TR$
10. $\phi^* = \wedge^* = \wedge$
11. $R^*R^* = R^* = (R^*)^*$
12. $RR^* = R^*R = R^* = \wedge+RR^*$
13. $(R+S)^* = (R^*S^*)^* = (R^*+S^*)^* = R^*S^* = (R^*S)^*R^* = R^*(SR^*)^*$
14. $(RS)^* = (R^*S^*)^* = (R^*+S^*)^*$

Theorem 2: Prove that R+R = R

Proof : We know the following equalities:

L(**R+R**) = L(**R**)UL(**R**) = L(**R**)

So **R**+**R** = **R**

Theorem 3: Prove the distributive property

R(S+T) = RS+RT

Proof: The following set of equalities will prove the property:

L(R(S+T)) = L(R)L(S+T)
= L(R)(L(S)UL(T))
= (L(R)L(S))U(L(R)L(T))
= L(RS+RT)

Similarly, by using the equalities we can prove the rest. The proofs of the rest of the equalities are left as exercises.

Example 15: Show that $R+RS^*S = a^*bS^*$, where $R = b+aa^*b$ and S is any regular expression.

$R+RS^*S = R \wedge+RS^*S$ (property 6)

$= R(\wedge+S^*S)$ (property 8)

$= R(\wedge+SS^*)$ (property 12)

$= RS^*$ (property 12)

$= (b+aa^*b)S^*$ (definition of R)
$= (\wedge+aa^*)\, bS^*$ (properties 6 and 8)

$= a^*bS^*$. (Property 12)

Try an exercise now.

Ex.5) Establish the following equality of regular expressions:
$b^*(abb^*+aabb^*+aaabb^*)^* = (b+ab+aab+aaab)^*$

As we already know the concept of language and regular expressions, we have an important type of language derived from the regular expression, called **regular language.**

2.3 REGULAR LANGUAGE

Language represented by a regular expression is called a regular language. In other words, we can say that a regular language is a language that can be represented by a regular expression.

Definition: For a given alphabet , the following rules define the regular language associated with a regular expression.

Rule 1: ϕ,{$\wedge$} and {a} are regular languages denoted respectively by regular expressions ϕ and $\wedge$.

Rule 2: For each a in , the set {a} is a regular language denoted by the regular expression **a**.

Rule 3: If **l** is a regular expression associated with the language L and **m** is a regular expression associated with the language M, then:

(i) The language = {xy : x ∈ L and y ∈ M} is a regular expression associated with the regular expression **lm**

(ii) The regular expression **l+m** is associated with the language formed by the union of the sets L and M.

language (**l+m**) = L∪M

(iii) The language associated with the regular expression $(\mathbf{l})^*$ is L^*, the Kleen Closure of the set L as a set of words:

language $(\mathbf{l}^*) = L^*$.

Now, we shall derive an important relation that, all finite languages are regular.

Theorem 4: If L is a finite language, then L can be defined by a regular expression.

In other words, all finite languages are regular.

Proof: A language is finite if it contains only finitely many words.

To make one regular expression that defines the language L, turn all the words in L into bold face type and insert plus signs between them. For example, the regular expression that defines the language L = {baa, abbba, bababa} is **baa + abbba + bababa**

Example16: If L = {aa, ab, ba, bb}, then the corresponding regular expression is **aa + ab +ba + bb.**

Another regular expression that defines this language is **(a+b) (a+b).**

So, a particular regular language can be represented by more than one regular expressions. Also, by definition, each regular language must have at least one regular expression corresponding to it.

Try some exercises.

Ex.6) Find a language to describe each of the following regular expressions:
(a) **a+b** (b) $\mathbf{a+b^*}$ (c) $\mathbf{a^*bc^*+ac}$

Ex.7) Find a regular expression for each of the following languages over the alphabet {a,b}:

(a) strings with even length.
(b) strings containing the sub string aba.

2.4 FINITE AUTOMATA

Finite automata are important in science, mathematics, and engineering. Engineers like them because they are superb models for circuits (and, since the advent of VLSI systems sometimes finite automata represent circuits.) computer scientists adore them because they adapt very likely to algorithm design. For example, the lexical analysis portion of compiling and translation. Mathematicians are introduced by them too due to the fact that there are several nifty mathematical characterizations of the sets they accept.
Can a machine recognise a language? The answer is yes for some machine and some an elementary class of machines called finite automata. Regular languages can be represented by certain kinds of algebraic expressions by Finite automaton and by certain grammars. For example, suppose we need to compute with numbers that are represented in scientific notation. Can we write an algorithm to recognise strings of symbols represented in this way? To do this, we need to discuss some basic computing machines called finite automaton.

An automata will be a finite automata if it accepts all the words of any regular language where language means a set of strings. In other words,

2.4.1 Finite Automata

A system where energy and information are transformed and used for performing some functions without direct involvement of man is called automaton. Examples are automatic machine tools, automatic photo printing tools, etc.

A finite automata is similar to a finite state machine. A finite automata consists of five parts:

(1) a finite set of states;
(2) a finite set of alphabets;
(3) an initial state;
(4) a subset of set of states (whose elements are called “yes” state or; accepting state;) and
(5) a next-state function or a transition state function.

A finite automata over a finite alphabet A can be thought of as a finite directed graph with the property that each node emits one labelled edge for each distinct element of A. The nodes are called states. There is one special state called **the start** (or **initial**) state, and there is a subset of states called **final states** (which could be possibly empty)

For example, the labelled graph in fig.1 given below represents a DFA over the alphabet A = {a,b} with start state 1 and final state 4.

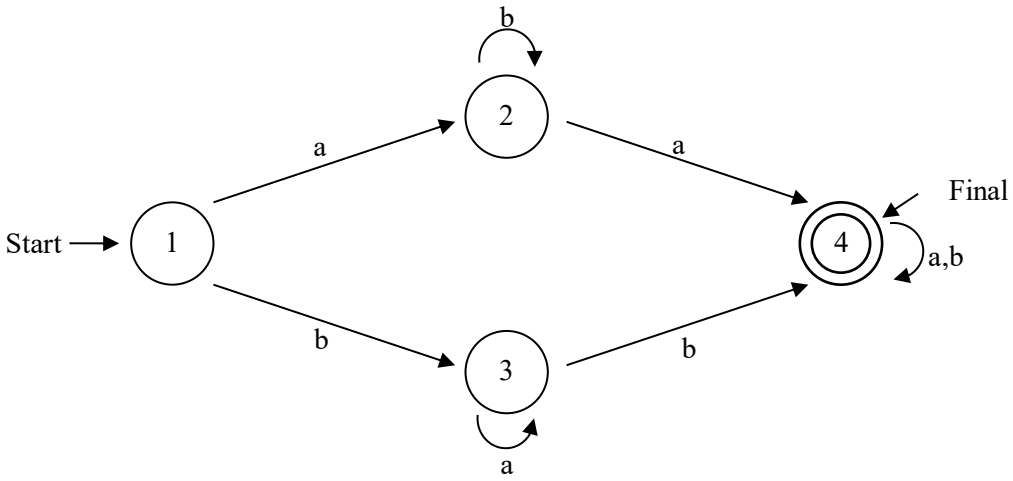


Fig. 1: Finite Automata

We always indicate the start state by writing the word start with an arrow painting to it. Final states are indicated by double circle.

The single arrow out of state 4 labelled with a,b is short hand for two arrows from state 4, going to the same place, one labelled a and one labelled b. It is easy to check that this digraph represents a DFA over {a,b} because there is a start state, and each state emits exactly two arrows, one labelled with a and one labelled with b.

So, we can say that a finite automaton is a collection of three tuples:

1. A finite set of states, one of which is designated as the initial state, called the start state, and some (may be none) of which we designated as final states.

2. An alphabet $\sum$ of possible input letters from which are formed strings that are to be read one letter at a time.

3. A finite set of transitions that tell for each state and for each letter of the input alphabet which state to go to next.

For example the input alphabet has only two letters a and b. Let us also assume that there are only three states, x, y and z. Let the following be the rules of transition:

1. from state x and input a go to state y;
2. from state x and input b go to state z;
3. from state y and input b go to state x;
4. from state y and input b go to state z; and
5. from state z and any input stay at state z.

Let us also designate state x as the starting state and state z as the only final state.

Let us examine what happens to various input strings when presented to this FA. Let us start with the string aaa. We begin, as always, in state x. The first letter of the string is an a, and it tells us to go state y (by rule 1). The next input (instruction) is also an a, and this tells us (by rule 3) to go back to state x. The third input is another a, and (by Rule 1) again we go to the state y. There are no more input letters in the input string, so our trip has ended. We did not finish in the final state (state z), so we have an unsuccessful termination of our run.

The string aaa is not in the language of all strings that leave this FA in state z. The set of all strings that do leave as in a final state is called the language defined by the finite automaton. The input string aaa is not in the language defined by this FA. We may say that the string aaa is not accepted by this FA because it does not lead to a final state. We may also say "aaa is rejected by this FA." The set of all strings accepted is the language associated with the FA. So, we say that L is the language accepted by this FA. FA is also called a language recogniser.

Let us examine a different input string for this same FA. Let the input be abba. As always, we start in state x. Rule 1 tells us that the first input letter, a, takes us to state y. Once we are in state y we read the second input letter, which is a b. Rules 4 now tells us to move to state z. The third input letter is a b, and since we are in state z, Rule 5 tells us to stay there. The fourth input letter is an a, and again Rule 5 says state z. Therefore, after we have followed the instruction of each input letter we end up in state z. State z is designated as a final state. So, the input string abba has taken us successfully to the final state. The string abba is therefore a word in the language associated with this FA. The word abba is accepted by this FA.

It is not difficult for us to predict which strings will be accepted by this FA. If an input string is made up of only the letter a repeated some number of times, then the action of the FA will be jump back and forth between state x and state y. No such word can ever be accepted.

To get into state z, it is necessary for the string to have the letter b in it as soon as a b is encountered in the input string, the FA jumps immediately to state z no matter what state it was before. Once in state z, it is impossible to leave. When the input strings run out, the FA will still be in state z, leading to acceptance of the string.

So, the FA above will accept all the strings that have the letter b in them and strings not of this form are never accepted. Therefore, the language associated with this FA is the one defined by the regular expression $\mathbf{(a+b)^* b(a+b)^*}$.

The list of transition rules can grow very long. It is much simpler to summarise them in a table format. Each row of the table is the name of one of the states in FA, and each column of this table is a letter of the input alphabet. The entries inside the table are the new states that the FA moves into the transition states. The transition table for the FA we have described is:

Table 1

State	Input	
	a	b
Start x	y	z
y	x	z
Final z	z	z

The machine we have already defined by the transition list and the transition table can be depicted by the state graph in Figure 2.

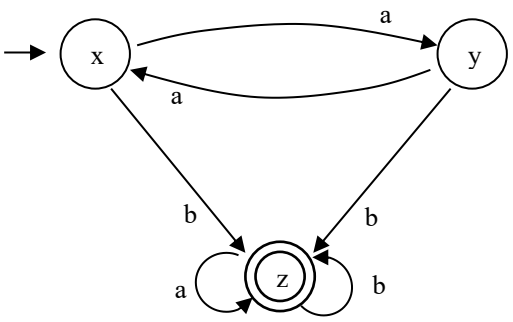


Fig. 2: State Transition graph

Note: A single state can be start as well as final state both. There will be only one start state and none or more than one final states in Finite Automaton.

2.4.2 Another Method to Describe FA

There is a traditional method to describe finite automata which is extremely intuitive. It is a picture called a graph. The states of the finite automaton appear as vertices of the graph while the transitions from state to state under inputs are the graph edges. The state graph for the same machine also appears in Figure3 given below.

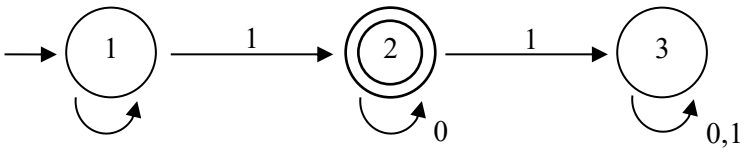


Fig. 3: Finite automata

The finite automata shown in Figure 3 can also be represented in Tabular form as below:

Table 2

	State	Input		Accept?
		0	1	
Start	1	1	2	No
Final	2	2	3	Yes
	3	3	3	No

Before continuing, let's examine the computation of a finite automaton. Our first example begins in state one and reads the input symbols in turn changing states as necessary. Thus, a computation can be characterized by a sequence of states. (Recall that Turing machine configurations needed the state plus the tape content. Since a finite automaton never writes, we always know what is on the tape and need only look at a state as a configuration.) Here is the sequence for the input 0001001.

Input Read:	0		0		0		1		0		0		1	
State:	1	→	1	→	1	→	1	→	2	→	2	→	2	→	3

Example 17 (An elevator controller): Let's imagine an elevator that serves two floors. Inputs are calls to a floor either from inside the elevator or from the floor itself. This makes three distinct inputs possible, namely:

0 - no calls
1 - call to floor one
2.- call to floor two

The elevator itself can be going up, going down, or halted at a floor. If it is on a floor, it could be waiting for a call or about to go to the other floor. This provides us with the six states shown in figure 4 along with the state graph for the elevator controller.

W1 Waiting on first floor
U1 About to go up
UP Going up
DN Going down
W2 Waiting-second floor
D2 About to go down

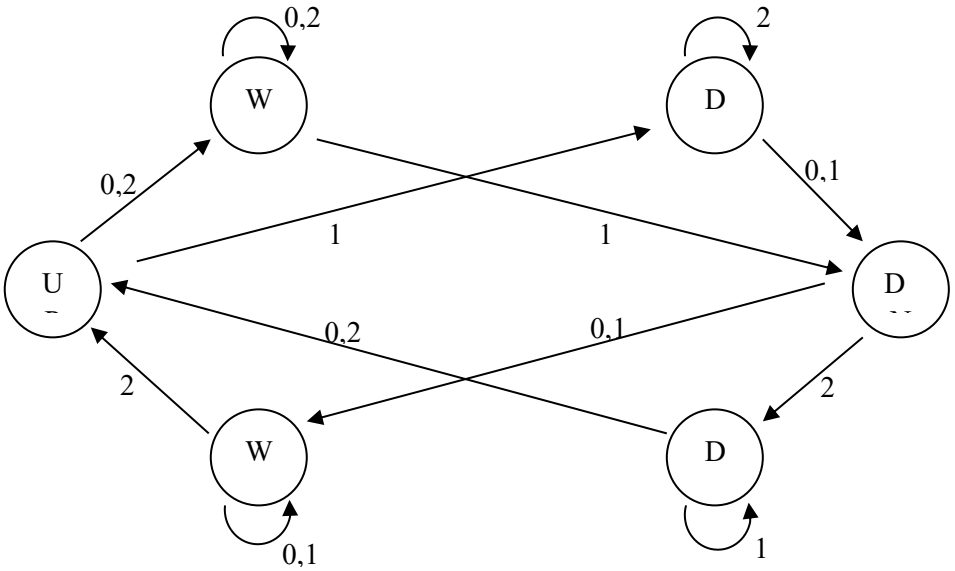


Fig. 4: Elevator Control

A transition state table for the elevator is given in table3:

Table 3 Elevator Control

State	Input		
	None	**call to 1**	**call to 2**
W1 (wait on 1)	W1	W1	UP
U1 (start up)	UP	U1	UP
UP	W2	D2	W2
DN	W1	W1	U1
W2 (wait on 2)	W2	DN	W2
D2 (start down)	DN	DN	D2

Accepting and rejecting states are not included in the elevator design because acceptance is not an issue. If we were to design a more sophisticated elevator, it might have states that indicated:

Finite automata

a) power failure
b) overloading, or
c) breakdown

In this case, acceptance and rejection might make sense.

Let us make a few small notes about the design. If the elevator is about to move (i.e., in state U1 or D2) and it is called to the floor it is presently on it will stay. (This may be good Try it next time you are in an elevator.) And, if it is moving (up or down) and gets called back the other way, it remembers the call by going to the U1 or D2 state upon arrival on the next floor. Of course, the elevator does not do things like open and close doors (these could be states too) since that would have added complexity to the design. Speaking of complexity, imagine having 100 floors.

That is our levity for this section. Now that we know what a finite automaton is, we must (as usual) define it precisely.

Definition : ***A finite automata M is a quintuple M = (Q, $\sum$,δ , q_O, F) where :***

Q is a finite set (of states)
$\sum$ is a finite alphabet (of input symbols)
$\boldsymbol{\delta}$: $Q \times \sum \rightarrow Q$ (next state function)
$q_O \in Q$ (the starting state)
$F \subseteq Q$ (the accepting states)

We also need some additional notation. The next state function is called the transition function and the accepting states are often called final states. The entire machine is usually defined by presenting a transition state table or a transition diagram. In this way, the states, alphabet, transition function, and final states are constructively defined. The starting state is usually the lowest numbered state. Our first example of a finite automaton is:

$M = (\{q_1, q_2, q_3\}, \{0,1\}, , q_1, \{q_2\}$

Let us look again at a computation by our first finite automaton. For the input 010, our machine begins in q_1, reads a 0 and goes to $\delta(q_2,0) = q_2$ after reading the final 0. All that can be put together as:

$\delta(\delta(\delta(q_1,0),1)0) = q_2$

We call this transition on strings δ^* and define it as follows:

Definition : Let M = ($Q,\sum,\delta,q_O,F$). For any input string x, input symbol a, and state q_i, the *transition function on strings δ^* takes the values:*

$\delta^*(q_i,(*e)) = w_i$
$\delta^*(q_i,a) = \delta(q_i,,a)$ a
$\delta^*(q_i,xa) = \delta(\delta^*(q_i,x),a) \forall a \in \sum , x \in \sum^*$

That certainly was terse. But δ^* is really just what one expects it to be. It merely applies the transition function to the symbols in the string.

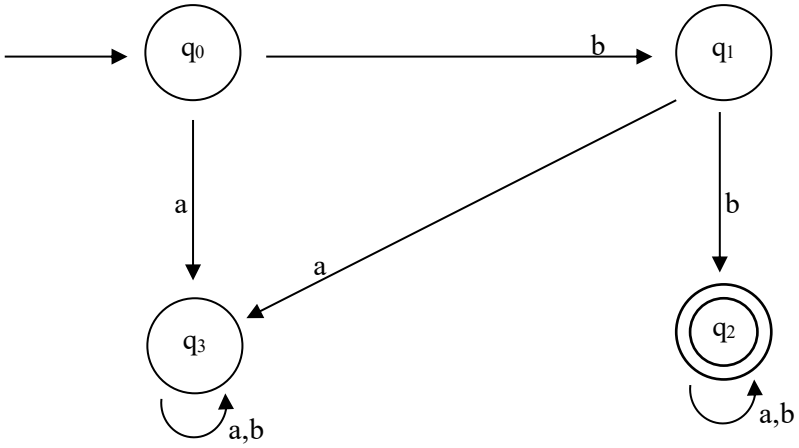


Fig. 5: Finite automata

This machine has a set of states = {q_0, q_1, q_2, q_3) and operates over the input alphabet {a,b}. In the above figure q_0 is the starting state and its set of final or accepting states, F = {q_2} an accepting state can also be shown by two concentric circles as shown in the fig.5

The transition function is fully described twice once in figure 6 as a state graph and once in tasble 4 as a state table.

Table 4

State	Input		Accept?
	A	b	
0	3	1	No
1	3	2	No
2	2	2	Yes
3	3	3	No

If the machine receives the input bbaa, it goes through the sequence of states:

q_0, q_1, q_2, q_2, q_2

While when it gets an input such as abab, it goes through the state transition:

q_0, q_3, q_3, q_3, q_3

Now we shall become a bit more abstract. When a finite automaton receives an input string such as:

$x = x_1 x_2 \ldots x_n$

where the x_i are symbols from its input alphabet, it progresses through the sequence:

$q_{k_1}, q_{k_2}, \ldots q_{k_{n+1}}$

where the states in the sequence are defined as:

$q_{k_1} = q_0$

$q_{k_2} = \delta(q_{k_1}, x_1) = \delta(q_0, x_0)$

$q_{k_3} = \delta(q_{k_2}, x_2) = \delta^*(q_0, x_1 x_2)$

$q_{k_{n+1}} = \delta(q_{k_n}, x_n) = \delta^*(q_0, x_1 x_2 \ldots . x_n)$

Getting back to a more intuitive reality, the following table provides an assignment of values to the symbols used above for an input of bbaba to the finite automaton of figure 3.

i	1	2	3	4	5	6
x_i	b	b	a	b	a	
q_{k_1}	q_0	q_1	q_2	q_2	q_2	q_2

Definition: *The **set** (of strings) **accepted** by the finite automaton* $M = (Q, \sum, \delta, q_0, F)$ *is* $T(M) = \{x / \delta^*(q_0,x) \in F\}$

This set of accepted strings (L(M) to mean for language accepted by M) is merely all of the strings for which M ended up in a final or accepting state after processing the string. For our first example (figure 1) this was all strings of 0's and 1's that contain exactly one 1. Our last example (figure 3) accepted the set of strings over the alphabet {a,b} which began with exactly two b's.

2.4.3 Finite Automata as Output Devices

The automata that we have discussed so far have only a limited output capability to the extent that only outputs are 'accepted' and 'not accepted' to indicating the acceptance or rejection of an input string. We want to introduce two classic models for finite automata that have additional output capability. We will consider machines that transform input strings into output strings. These machines are basically DFAs, except that we associate an output symbol with each state or with each state transition. But there are no final states because we are not interested in acceptance or rejection.

Mealy and Moore Machines

The first model invented by Mealy [1955] is called a Mealy machine. It associates an output letter with each transition. For example, if the output associated with the edge labelled with the letter a is x, we shall write a/x on that edge. A state transition for a Mealy machine can be presented in figure 7 as follows:

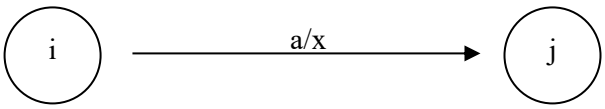


Fig. 7: Mealy machine

Indicating that the machine in state i and on input a gives output x and enters state j.

In a Mealy machine, an output always takes place during a transition of the states. The second model invented by Moore [1956], is called a Moore machine. It associates an output letter with each state. For example, if the output associated with state I is x, we will always write i/x inside the state circle. A typical state transition for a Moore machine can be presented in figure8 as follows:

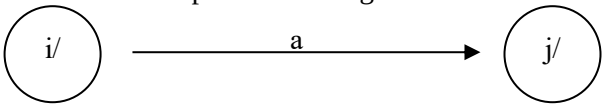


Fig. 8: Moore machine

In a Moore machine, each time a state is entered, simultaneously an output takes place. So, the first output always occurs as soon as the machine is started. Mealy and Moore machines are equivalent. In other words, any problem that is soluble by one type of machine can also be solved by the other type of machine.

Example 18: Suppose we want to compute the number of sub strings of the form bab that occurs in an arbitrary input string over the alphabet {a,b}.

The diagrammatic representation of a Mealy machine for the task is given below in figure 9:

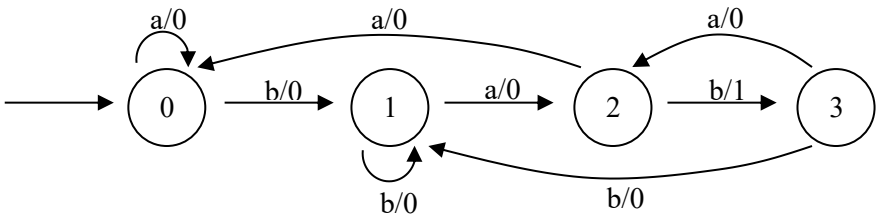


Fig. 9: Mealy machine

For example, the output of this Mealy machine for the sample string abababaababb is 000101000010, where each 1 indicates the availability of a (or an additional) substring up to that point. On the other hand, a 0 indicates that the three previous inputs including the current input do not form a substring of the form bab.

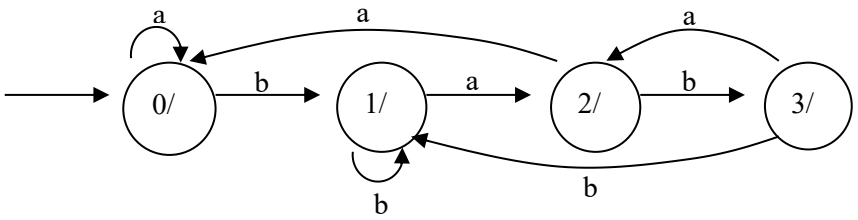


Fig. 10: Moore machine

For example, the output of this Moore machine for the simple string. Abababaababb is 0000101000010. We can count the number of 1's in the output string to obtain the number of occurrence of the sub string bab.

Example 19: A Simple Traffic Signal : Suppose we have a simple traffic intersection, where a north-south highway intersects an east-west highway. We will assume that the east-west highway always has a green light unless some north-south traffic is detected by sensors. When north-south traffic is detected, after a certain time delay the signals change and stay that way for a fixed period of time. We are required to design an appropriate circuit to capture the desired result stated above. We construct a Moore machine as a model of the required circuit as follows:

The input symbols for the required Moore machine are 0 (no traffic detected) and 1 (traffic detected). Let G, Y and R mean the colours Green, Yellow and Red, respectively. The output strings are GR, YR, RG, AND RY, where the first letter of a string is the colour of the east-west light and the second letter of a string is the colour of the north-sought light. The Moore machine model for this simple traffic intersection problem is given below diagrammatically:

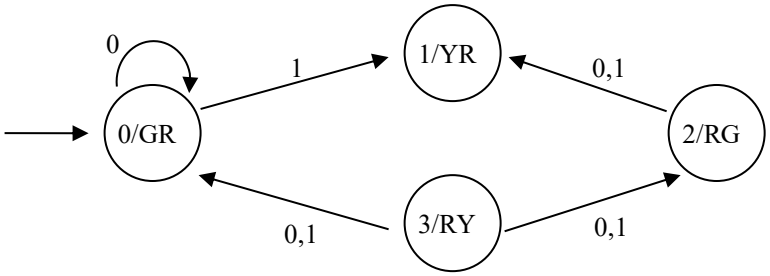


Fig. 11: Traffic signal transition diagram

Mealy machines appear to be more useful than Moore machines. But problems like traffic signal control have hic Moore machine solutions because each state is associated with a new output configuration.

Let us try some exercises:

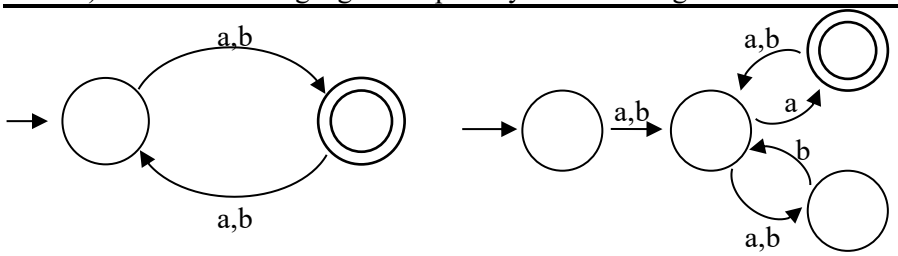
Ex.8) Build a new FA that accepts only the word ∧. Also write the corresponding regular expression.

Ex.9) Build an FA that accepts only those words that have even lengths. Also write the regular expression.

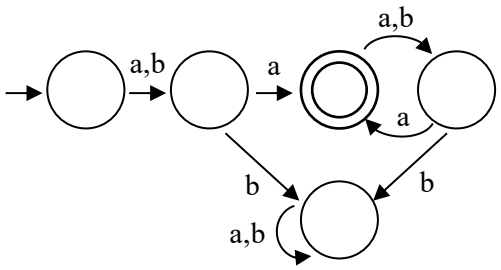
Ex.10) Build an FA that accepts only the word baa, ab and abb and no other words. Also write the corresponding regular expression.

Ex.11) Build an FA that will accept the language of all words each having twice as many a's as the number of b's. Also write the corresponding regular expression.

Ex.12) Describe the languages accepted by the following FA's:



(a) **(b)**



(c)

Figure 12:

2.5 NON DETERMINISTIC FINITE AUTOMATA

You have already studied finite automata (though 'automata' is a plural form of the

noun 'automaton', the word 'automata' is also used in singular sense). Now consider an automata that accepts all and only strings ending in 01, represented diagrammatically, as follows:

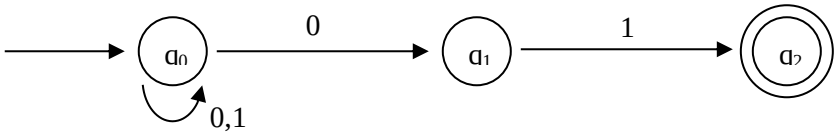


Fig. 13: Transition Diagram

In the case of the finite automata shown in figure 1, the following points may benoted:

(i) On input 0 in state q_0, the next state may be either of the two states viz., q_0 or q_1.
(ii) There is no next state on input 0 in the state q_1.
(iii) There is no next state on input 0 and 1 in the state q_2.

In this transition system, what happens when this automata processes the input .00101?

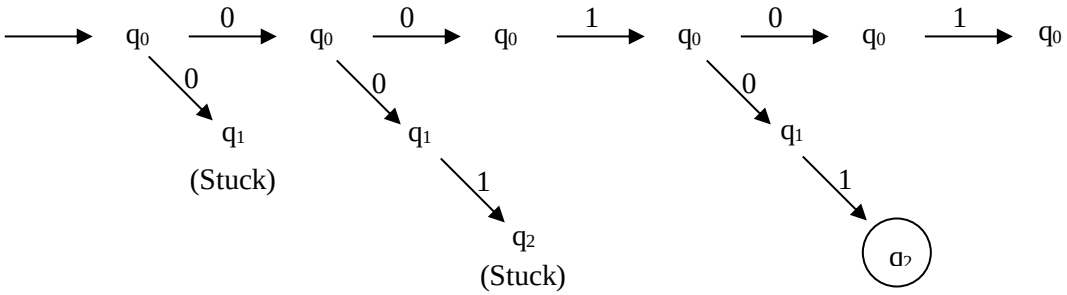


Fig. 14: Processing of string 00101

Here from the initial state q_0, for the processing of alphabet 0, there are two states at once or viewed another way, it can be 'guessed" which state to go to next. Such a finite automata allows to have a choice of 0 or more next states for each state input pair and is called a non-deterministic finite automata. An NFA can be in several states at once.

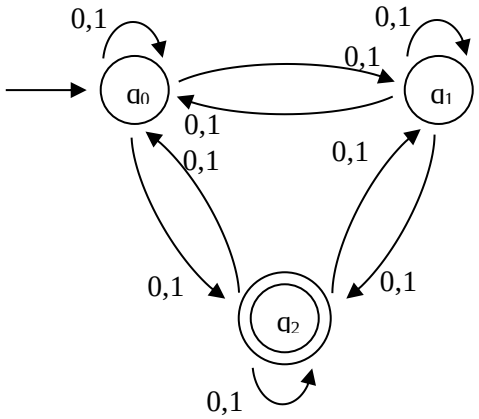


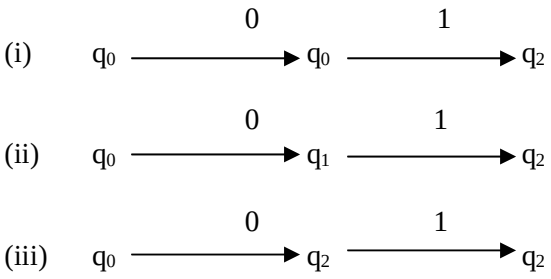
Fig. 15: Transition diagram

Before going to the formal definition of NFA, let us discuss one more case of non-determinism of finite automata. Suppose Q = {q_0, q_1, q_2}, $\sum$ = {0, 1}, q_0 is an initial state and q_2 is final state. Again, suppose the processing of any input symbol does not result in the transition to a unique state, but results a chain of states. Let us consider a

machine given in figure 3.

For the sake of convenience, let us check the processing of any input symbol. Fromthe state q_0, after processing 0, resulting states are q_0, q_1, q_2 and for input symbol 1, there are three possible states q_0, q_1 and q_2 not a unique state. It clarifies that a non-deterministic automata can have more than one possible state or none state after processing any input symbol from $\sum$.

Let us check how the string 01 is processed by the above automata. Here we havethree paths to reach to the final state:



A generalisation which is obtained here by allowing of several states as a result of the processing of an input symbol is called non-determinism. If from any state, we can reach to several states or none state, then the finite automata becomes non-deterministic in nature.

Formally, a non-deterministic finite automata is a quintuple

$A = (Q, \sum, \delta, q_0, F)$
Where

* Q is a finite set of states
* $\sum$ is a finite alphabet for inputs
* δ is a transition function from $Q \times \sum$ to the power set of Q i.e. to 2^Q
* $q_0 \in Q$ is the start/initial state
* $F \subseteq Q$ is a set of final/accepting states.

The NFA, for the example just considered, can be formally represented as:
$(\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$

Where the transition function, is given by the table 1:

Table1

States	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
q_2	Ø	Ø

Now, let us prove that the NFA

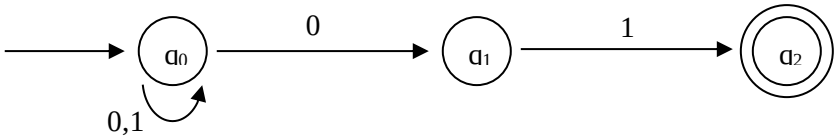


Fig. 16: NFA accepting x01

accepts the language $\{x01 : x \in \sum^*\}$ of all the strings that terminate with the sub-string 01. A mutual induction on the three statements below proves that the NFA accepts the given language.

1. $w \in \sum^* \Rightarrow q_0 \in \delta(q_0, w)$
2. $q_1 \in \delta(q_0, w) \Leftrightarrow w = x0$
3. $q_2 \in \delta(q_0, w)\ w = x01$

If $|w| = 0$ then $w = \wedge$. Then statement (1) follows from def., and statement & (2) and (3) show that all the string x01 will be accepted by the above non-deterministic automata.

Example 20: Consider the NFA with the formal description as $(Q, \sum, \delta, q_0, F)$ where $Q = \{q_0, q_1, q_2\}$, $\sum = \{a, b\}$, q_0 is the initial state and q_1 is only the final state, and δ is given by the following table:

Table 2

State	Input from	
	a	b
$\rightarrow q_0$	q_1, q_2	q_0
(q_1)	q_1	-
q_2	q_1	q_2

In NFA, though the function maps to a sub-set of the set of states, yet we generally drop braces, i.e., instead of $\{q_0, q_1\}$, we just write q_0, q_1.

The computation for an NFA is also similar to that of DFA. Let $N = (Q, \sum, \delta, q_0, F)$ be an NFA and w is a string over the alphabet $\sum$. The string w is accepted by NFA if corresponding to the input sequence, there exists a sequence of transitions from the initial state to any of the possible final states.

Now, let us check computations (in NFA, there are many possible computations) ofthe string aba.

$\delta(q_0, aba) = \delta(\delta(q_0, a), ba)$

$= \delta(q_1, ba)$ or $\delta(q_2, ba)$

$= \delta(\delta(q_1, b), a)$ or $\delta(\delta(q_2, b), a)$

= stuck or $\delta(q_2, a)$

= q_1 (an accepting state)

The above sequence of states shows the final state q_1 which is an accepting state. Hence, the string aba is accepted by the system and the input sequence of states for the input is $\rightarrow q_0 \xrightarrow{a} q_2 \xrightarrow{b} q_2 \xrightarrow{a} (q_1)$

Try some exercises:

Ex.13) Consider an NFA given in figure 5. Check whether the strings 001, 011101, 01110, 010 are accepted by the machine, or not?

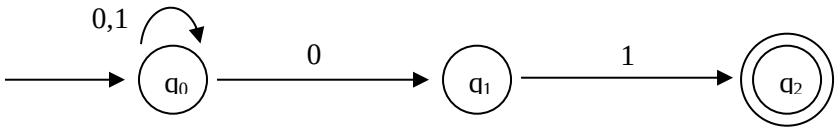


Fig. 17

Ex.14) Give an NFA which accepts all the strings starting with ab over {a,b}.

2.6 SUMMARY

In this unit we introduced the concepts of languages, regular expressions and regular langyages. Finite Automata are machines that recognize regular languages. From regular expressions, we can derive regular languages. In deterministic finite automata (DFA), there is a unique next state for transition on input in a given state. If we relax this condition of uniqueness of the next state in DFA, we get NFA.

2.7 SOLUTION/ANSWERS

Ex.1) (i) ababbbaa
(ii) baaababb
(iii) ab abb ab abb
(iv) baa baa
(v) ababbababb baa

Ex.2) (i) Suppose aa = x

Then $\{ x, b\}^* = \{\wedge , x, b, xx, bb, xb, bx, xxx, bxx, xbx, xxb, bbx, bxb, xbb, bbb\}$
substituting x = aa
$\{aa,b\}^* = \{\wedge , aa, b, aaaa, bb, aab, baa, aaaaaa, baaaa, aabaa, \ldots\}$

(ii) $\{a,ba\}^* = \{\wedge , a, ba, aa, baba, aba, baa, \ldots \}$

Ex.3) (a) **a+b+c**

(b) $\mathbf{ab^*+ba^*}$

(c) $\wedge+\mathbf{a(bb)^*}$

Ex.4) $\mathbf{0+1(0+1)^*}$

Ex.5) Starting with the left side and using properties of regular expressions, we get

$\mathbf{b^*(abb^* + aabb^*+aaabb^*)^*}$
$= \mathbf{b^*((ab+aab+aaab)b^*)^*}$ (property 9)
$= \mathbf{(b + ab + aab + aaab)^*}$ (property 7).

Ex.6) (a) {a,b}
(b) {a, ∧,b,bb,….b^n,….}
(c) {a,b,ab,bc,abb,bcc,…ab^n,bc^n,….}

Ex.7) (a) **(aa+ab+ba+bb)***
(b) **(a+b)*aba(a+b)***

Ex.8)

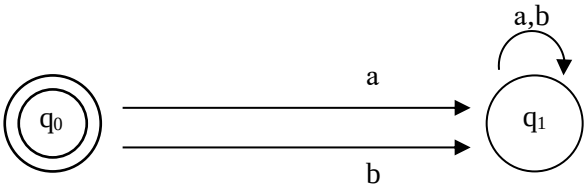


Fig. 18: Regular Expression of a null string

Ex.9)

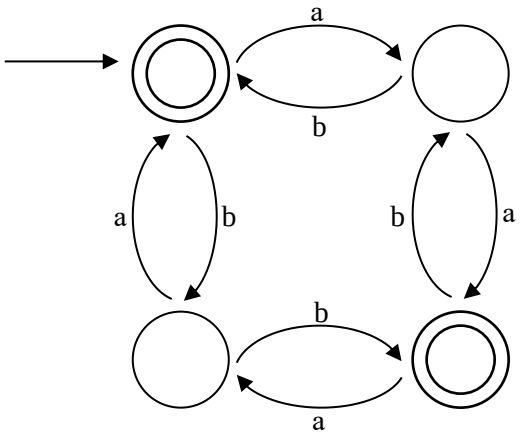


Fig. 19: Regular Expression is (aa+ba+ab+bb)

Ex.10) R.E. is **(baa + ab + abb)**

Ex.11) (i) All the words of odd lengths.
(ii) All the words ended with a.
(iii) All the words with a at even places.

Ex.12) is given by

State	Input	
	0	1
q_0	q_0, q_1	q_0
q_1	-	-
(q^2)	-	-

$\delta(q_0, 001) = \delta(q_0, 01) = \delta(q_1, 1) = (q_2)$ (Accepting state)

$\delta(q_0, 011101) = \delta(q_0, 11101)$

$= \delta(q_0, 1101)$

$= \delta (q_0, 101)$

$= \delta (q_0, 01)$

$= \delta (q_0, 1)$

$= q_2$ (accepting state)

$\delta (q_0, 01110) = \delta (q_0, 1110)$

$= \delta (q_0, 110)$

$= \delta (q_0, 10)$

$= \delta (q_0, 0)$

$= q_0$ or q_1 (Not an accepting state)

$\delta (q_0, 010) = \delta (q_0, 10)$

$= \delta (q_0, 0)$

$= q_0$ or q_1 (Not an accepting state)

So, strings 001 and 011101 are accepted by the given automata.

Ex.13)

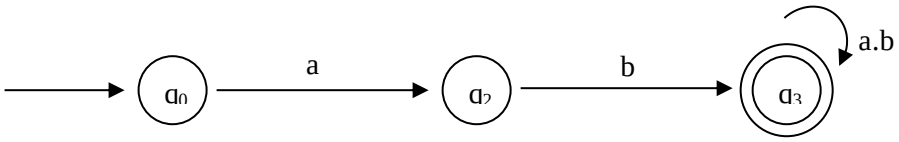


Fig. 20