UNIT – I

Formal Language and Regular Expressions: Languages, Definition Languages regular expressions, Finite Automata – DFA, NFA. Conversion of regular expression to NFA, NFA to DFA. Applications of Finite Automata to lexical analysis, lex tools.

Languages

Theory of automata is a theoretical branch of computer science and mathematical. It is the study of abstract machines and the computation problems that can be solved using these machines. The abstract machine is called the automata. The main motivation behind developing the automata theory was to develop methods to describe and analyze the dynamic behavior of discrete systems.

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

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

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

Symbols:

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

Example:

1, a, b, #

Alphabets:

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

Examples:

∑ = {a, b}
∑ = {A, B, C, D}
∑ = {0, 1, 2}
∑ = {0, 1, ..., 5]
∑ = {#, β, Δ}

String:

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

Example 1:

If ∑ = {a, b}, various string that can be generated from ∑ are {ab, aa, aaa, bb, bbb, ba, aba.....}.
A string with zero occurrences of symbols is known as an empty string. It is represented by ε.
The number of symbols in a string w is called the length of a string. It is denoted by |w|.

Example 2:

w = 010
Number of Sting |w| = 3

Language:

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

Example: 1

L1 = {Set of string of length 2}
= {aa, bb, ba, bb} **Finite Language**

Example: 2

L2 = {Set of all strings starts with 'a'}
= {a, aa, aaa, abb, abbb, ababb} **Infinite Language**

Definition Languages regular expressions

The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.

The languages accepted by some regular expression are referred to as Regular languages.

A regular expression can also be described as a sequence of pattern that defines a string.

Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

For instance:

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

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

Operations on Regular Language

The various operations on regular language are:

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

L U M = {s | s is in L or s is in M}

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

L ⋂ M = {st | s is in L and t is in M}

Kleen closure: If L is a regular language then its Kleen closure L1* will also be a regular language.

L* = Zero or more occurrence of language L.

Example 1:

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

Solution:

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

R = a*

That is Kleen closure of a.

Example 2:

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

Solution:

The regular expression has to be built for the language

L = {a, aa, aaa, ...}

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

$R = a^+$

Example 3:

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

Solution:

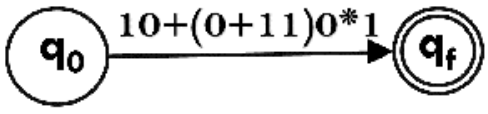
The regular expression will be:

r.e. = (a + b)*
This will give the set as L = {ε, a, aa, b, bb, ab, ba, aba, bab, ...}, any combination of a and b. The (a + b)* shows any combination with a and b even a null string.
Example 1: Write the regular expression for the language accepting all the string which are starting with 1 and ending with 0, over ∑ = {0, 1}.
Solution:
In a regular expression, the first symbol should be 1, and the last symbol should be 0. The r.e. is as follows:
R = 1 (0+1)* 0
Example 2:
Write the regular expression for the language starting and ending with a and having any having any combination of b's in between.
Solution:
The regular expression will be:
R = a b* b
Example 3:
Write the regular expression for the language starting with a but not having consecutive b's.
Solution: The regular expression has to be built for the language:
L = {a, aba, aab, aba, aaa, abab, ...}
The regular expression for the above language is:
R = {a + ab}*
Example 4:
Write the regular expression for the language accepting all the string in which any number of a's is followed by any number of b's is followed by any number of c's.
Solution: As we know, any number of a's means a* any number of b's means b*, any number of c's means c*. Since as given in problem statement, b's appear after a's and c's appear after b's. So the regular expression could be:
R = a* b* c*
Example 5:
Write the regular expression for the language over ∑ = {0} having even length of the string.
Solution:
The regular expression has to be built for the language:
L = {ε, 00, 0000, 000000, ...}
The regular expression for the above language is:
R = (00)*
Example 6:
Write the regular expression for the language having a string which should have atleast one 0 and alteast one 1.
Solution:
The regular expression will be:
R = [(0 + 1)* 0 (0 + 1)* 1 (0 + 1)*] + [(0 + 1)* 1 (0 + 1)* 0 (0 + 1)*]
Example 7:
Describe the language denoted by following regular expression
r.e. = (b* (aaa)* b*)*
Solution:

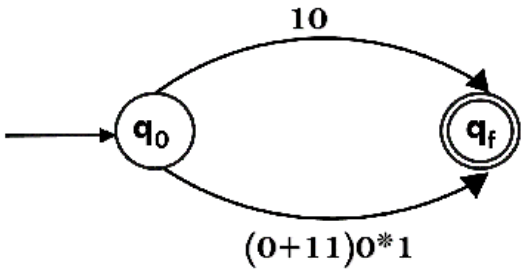
The language can be predicted from the regular expression by finding the meaning of it. We will first split the regular expression as:
r.e. = (any combination of b's) (aaa)* (any combination of b's)
L = {The language consists of the string in which a's appear triples, there is no restriction on the number of b's}
Example 8:
Write the regular expression for the language L over $\sum = \{0, 1\}$ such that all the string do not contain the substring 01.
Solution:
The Language is as follows:
L = {ε, 0, 1, 00, 11, 10, 100, ...}
The regular expression for the above language is as follows:
R = (1* 0*)
Example 9:
Write the regular expression for the language containing the string over {0, 1} in which there are at least two occurrences of 1's between any two occurrences of 1's between any two occurrences of 0's.
Solution: At least two 1's between two occurrences of 0's can be denoted by (0111*0)*.
Similarly, if there is no occurrence of 0's, then any number of 1's are also allowed. Hence the r.e. for required language is:
R = (1 + (0111*0))*
Example 10:
Write the regular expression for the language containing the string in which every 0 is immediately followed by 11.
Solution:
The regular expectation will be:
R = (011 + 1)*

Conversion of RE to FA

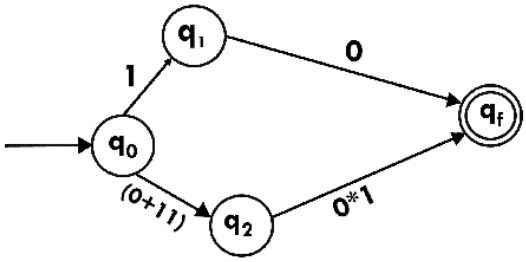
To convert the RE to FA, we are going to use a method called the subset method. This method is used to obtain FA from the given regular expression. This method is given below:
Step 1: Design a transition diagram for given regular expression, using NFA with ε moves.
Step 2: Convert this NFA with ε to NFA without ε.
Step 3: Convert the obtained NFA to equivalent DFA.
Example 1:
Design a FA from given regular expression 10 + (0 + 11)0* 1.
Solution: First we will construct the transition diagram for a given regular expression.
Step 1:



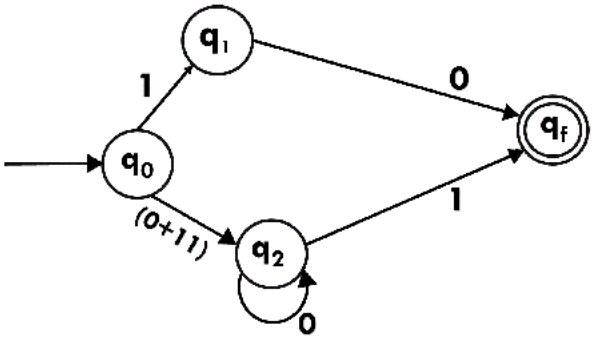
Step 2:



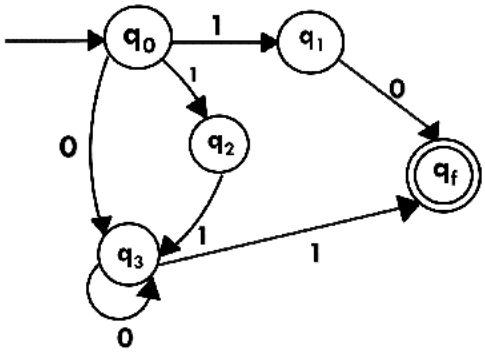
Step 3:



Step 4:



Step 5:



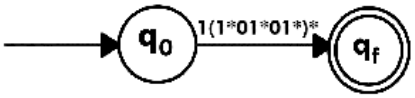
Now we have NFA without ε. Now we will convert it into required DFA for that, we will first write a transition table for this NFA.

State	0	1
→q0	q3	{q1, q2}
q1	qf	ϕ
q2	ϕ	q3
q3	q3	qf
*qf	ϕ	ϕ

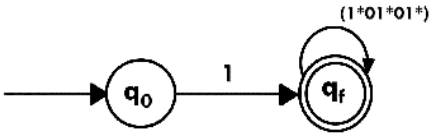
The equivalent DFA will be:

State	0	1
→[q0]	[q3]	[q1, q2]
[q1]	[qf]	Φ
[q2]	ϕ	[q3]
[q3]	[q3]	[qf]
[q1, q2]	[qf]	[qf]
*[qf]	ϕ	Φ

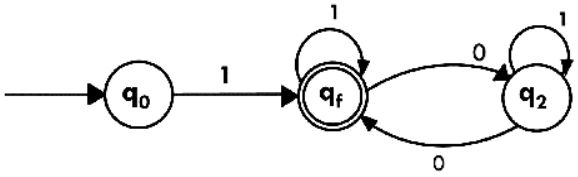
Example 2:
Design a NFA from given regular expression 1 (1* 01* 01*)*.
Solution: The NFA for the given regular expression is as follows:
Step 1:



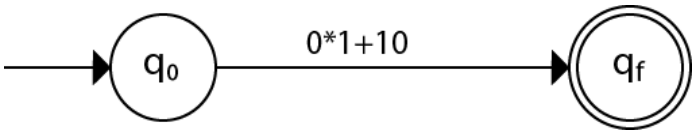
Step 2:



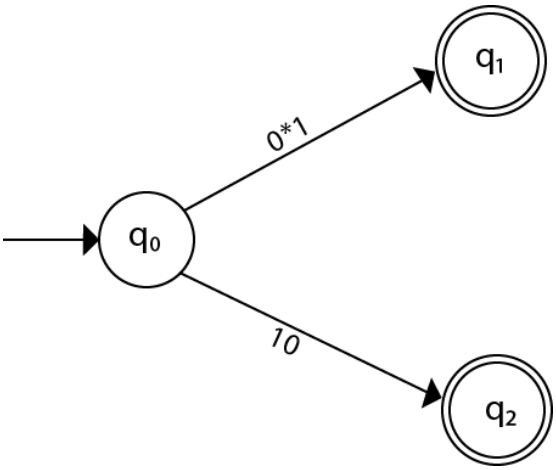
Step 3:



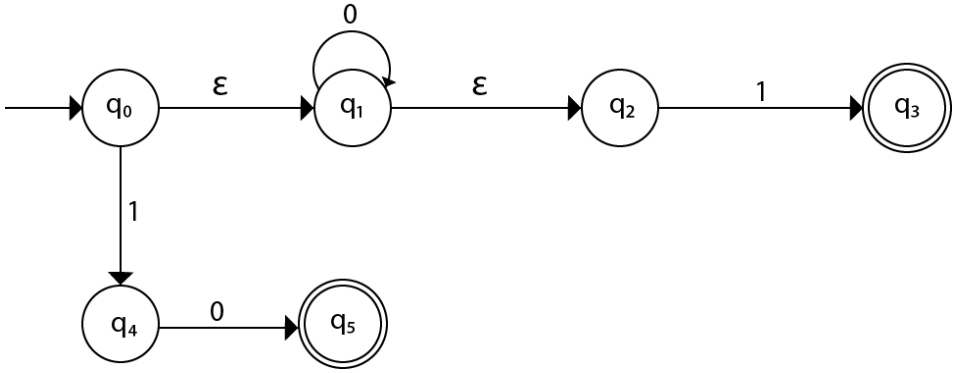
Example 3:
Construct the FA for regular expression 0*1 + 10.
Solution:
We will first construct FA for R = 0*1 + 10 as follows:
Step 1:



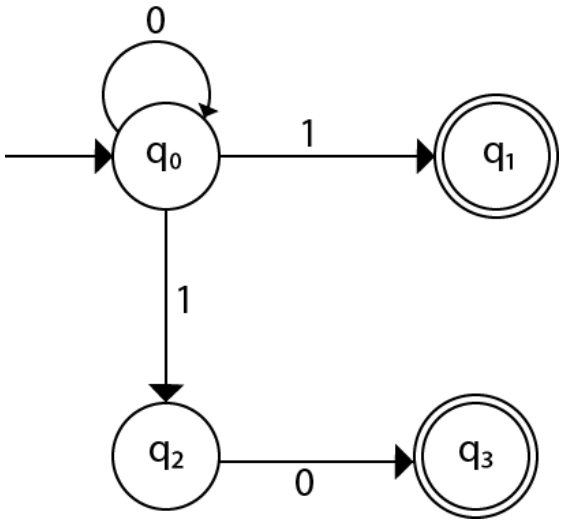
Step 2:



Step 3:



Step 4:



Finite Automata – DFA, NFA

Finite automata are used to recognize patterns.

It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.

At the time of transition, the automata can either move to the next state or stay in the same state.

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

Formal Definition of FA

A finite automaton is a collection of 5-tuple (Q, ∑, δ, q0, F), where:

Q: finite set of states

∑: finite set of the input symbol

q0: initial state

F: **final** state

δ: Transition function

Finite Automata Model:

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

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

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

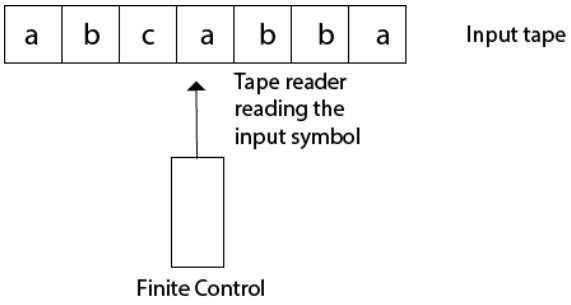


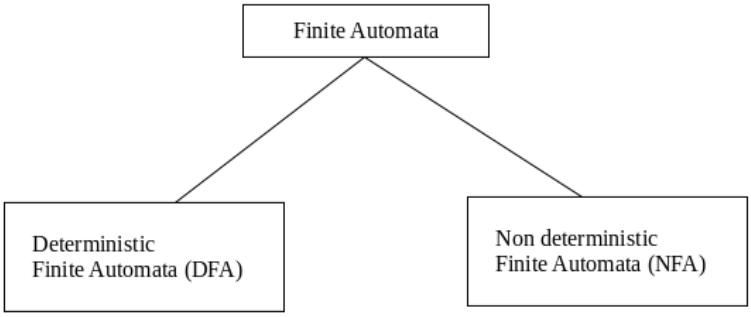
Fig :- Finite automata model

Types of Automata:

There are two types of finite automata:

DFA(deterministic finite automata)

NFA(non-deterministic finite automata)



1. DFA

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

2. NFA

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

Some important points about DFA and NFA:

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

Transition Diagram

A transition diagram or state transition diagram is a directed graph which can be constructed as follows:
There is a node for each state in Q, which is represented by the circle.
There is a directed edge from node q to node p labeled a if $\delta(q, a) = p$.
In the start state, there is an arrow with no source.
Accepting states or final states are indicating by a double circle.
Some Notations that are used in the transition diagram:

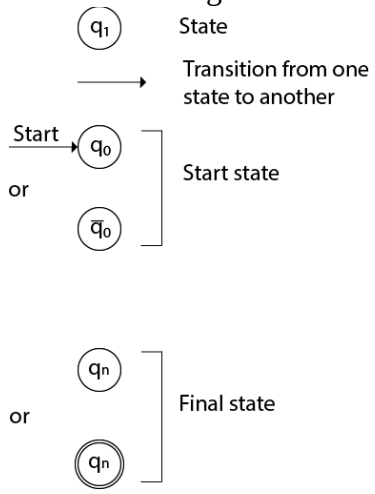


Fig:- Notations

There is a description of how a DFA operates:

In DFA, the input to the automata can be any string. Now, put a pointer to the start state q and read the input string w from left to right and move the pointer according to the transition function, δ. We can read one symbol at a time. If the next symbol of string w is a and the pointer is on state p, move the pointer to δ(p, a). When the end of the input string w is encountered, then the pointer is on some state F.

The string w is said to be accepted by the DFA if $r \in F$ that means the input string w is processed successfully and the automata reached its final state. The string is said to be rejected by DFA if $r \notin F$.

Example 1:

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

Solution:

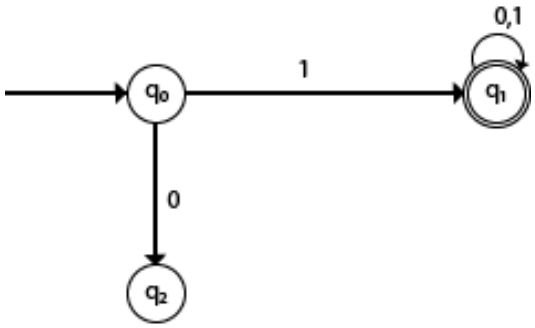


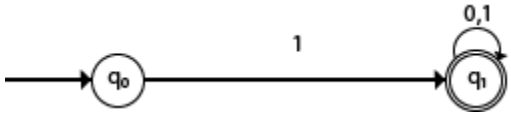
Fig: Transition diagram

The finite automata can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q0 on receiving 0, the machine changes its state to q2, which is the dead state. From q1 on receiving input 0, 1 the machine changes its state to q1, which is the final state. The possible input strings that can be generated are 10, 11, 110, 101, 111...., that means all string starts with 1.

Example 2:

NFA with ∑ = {0, 1} accepts all strings starting with 1.

Solution:



The NFA can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q1 on receiving input 0, 1 the machine changes its state to q1. The possible input string that can be generated is 10, 11, 110, 101, 111....., that means all string starts with 1.

Conversion of regular expression to NFA, NFA to DFA

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

Let, M = (Q, ∑, δ, q0, F) is an NFA which accepts the language L(M). There should be equivalent DFA denoted by M' = (Q', ∑', q0', δ', F') such that L(M) = L(M').

Steps for converting NFA to DFA:

Step 1: Initially Q' = ϕ

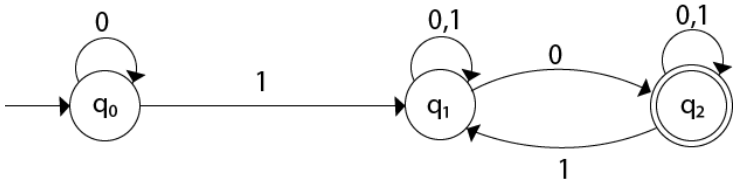
Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.

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

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

Example 1:

Convert the given NFA to DFA.



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

State	0	1
→q0	q0	q1
q1	{q1, q2}	q1
*q2	q2	{q1, q2}

Now we will obtain δ' transition for state q0.

δ'([q0], 0) = [q0]

δ'([q0], 1) = [q1]

The δ' transition for state q1 is obtained as:

δ'([q1], 0) = [q1, q2] (**new** state generated)

δ'([q1], 1) = [q1]

The δ' transition for state q2 is obtained as:

δ'([q2], 0) = [q2]

δ'([q2], 1) = [q1, q2]

Now we will obtain δ' transition on [q1, q2].

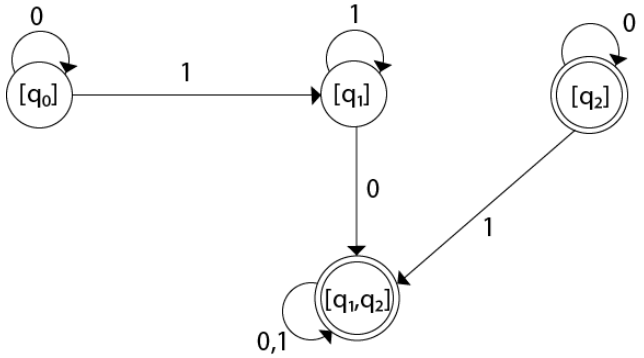
δ'([q1, q2], 0) = δ(q1, 0) ∪ δ(q2, 0)
= {q1, q2} ∪ {q2}
= [q1, q2]

δ'([q1, q2], 1) = δ(q1, 1) ∪ δ(q2, 1)
= {q1} ∪ {q1, q2}
= {q1, q2}
= [q1, q2]

The state [q1, q2] is the final state as well because it contains a final state q2. The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]
*[q1, q2]	[q1, q2]	[q1, q2]

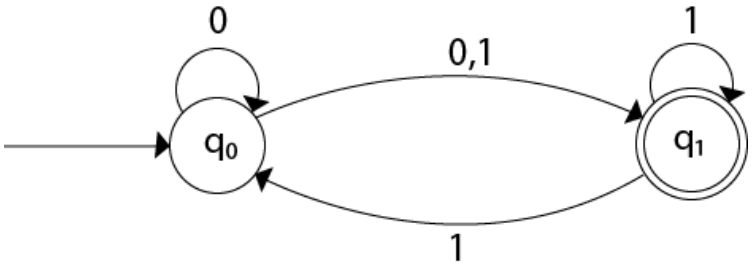
The Transition diagram will be:



The state q2 can be eliminated because q2 is an unreachable state.

Example 2:

Convert the given NFA to DFA.



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

State	0	1
→q0	{q0, q1}	{q1}
*q1	Φ	{q0, q1}

Now we will obtain δ' transition for state q0.

δ'([q0], 0) = {q0, q1}

= [q0, q1] (**new** state generated)

δ'([q0], 1) = {q1} = [q1]

The δ' transition for state q1 is obtained as:

δ'([q1], 0) = ϕ

δ'([q1], 1) = [q0, q1]

Now we will obtain δ' transition on [q0, q1].

δ'([q0, q1], 0) = δ(q0, 0) ∪ δ(q1, 0)

= {q0, q1} ∪ ϕ

= {q0, q1}

= [q0, q1]

Similarly,

δ'([q0, q1], 1) = δ(q0, 1) ∪ δ(q1, 1)

= {q1} ∪ {q0, q1}

= {q0, q1}

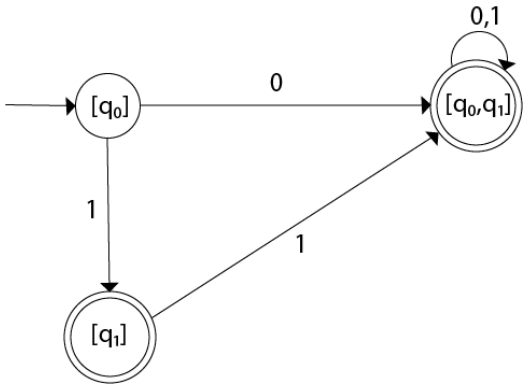
= [q0, q1]

As in the given NFA, q1 is a final state, then in DFA wherever, q1 exists that state becomes a final state. Hence in the DFA, final states are [q1] and [q0, q1]. Therefore set of final states F = {[q1], [q0, q1]}.

The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0, q1]	[q1]
*[q1]	Φ	[q0, q1]
*[q0, q1]	[q0, q1]	[q0, q1]

The Transition diagram will be:



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

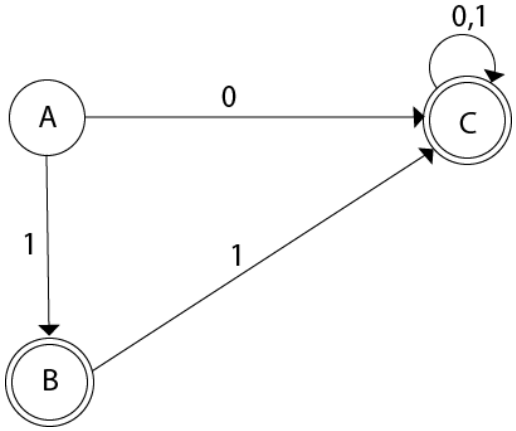
Suppose

A = [q0]

B = [q1]

C = [q0, q1]

With these new names the DFA will be as follows:



Applications of Finite Automata to lexical analysis

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

Expressive Power of various Automata:

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

FA<DPDA<PDA<LBA<TM

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

The Turing Machine i.e. TM is more powerful than any other machine.

(i) Finite Automata (FA) equivalence: Finite Automata

≡ PDA with finite Stack

≡ TM with finite tape

≡ TM with unidirectional tape

≡ TM with read only tape

(ii) Pushdown Automata (PDA) equivalence:

PDA ≡ Finite Automata with Stack

(iii) Turing Machine (TM) equivalence:

Turing Machine

≡ PDA with additional Stack

≡ FA with 2 Stacks

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

1. Finite Automata (FA) **–**

For the designing of lexical analysis of a compiler.

For recognizing the pattern using regular expressions.

For the designing of the combination and sequential circuits using Mealy and Moore Machines.

Used in text editors.

For the implementation of spell checkers.

2. Push Down Automata (PDA) **–**

For designing the parsing phase of a compiler (Syntax Analysis).

For implementation of stack applications.

For evaluating the arithmetic expressions.

For solving the Tower of Hanoi Problem.

3. Linear Bounded Automata (LBA) –

For implementation of genetic programming.

For constructing syntactic parse trees for semantic analysis of the compiler.

4. Turing Machine (TM) **–**

For solving any recursively enumerable problem.

For understanding complexity theory.

For implementation of neural networks.

For implementation of Robotics Applications.

For implementation of artificial intelligence.

Lex tools.

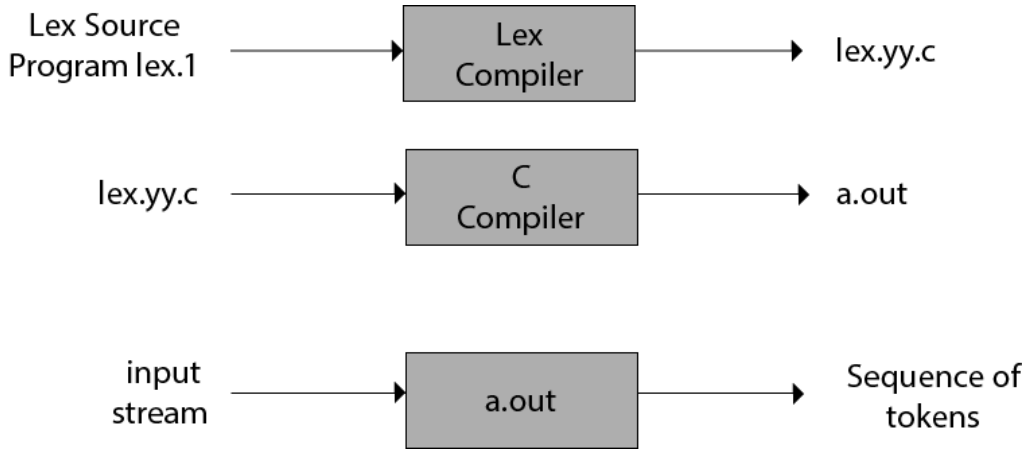
Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.

Finally C compiler runs the lex.yy.c program and produces an object program a.out.

a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.

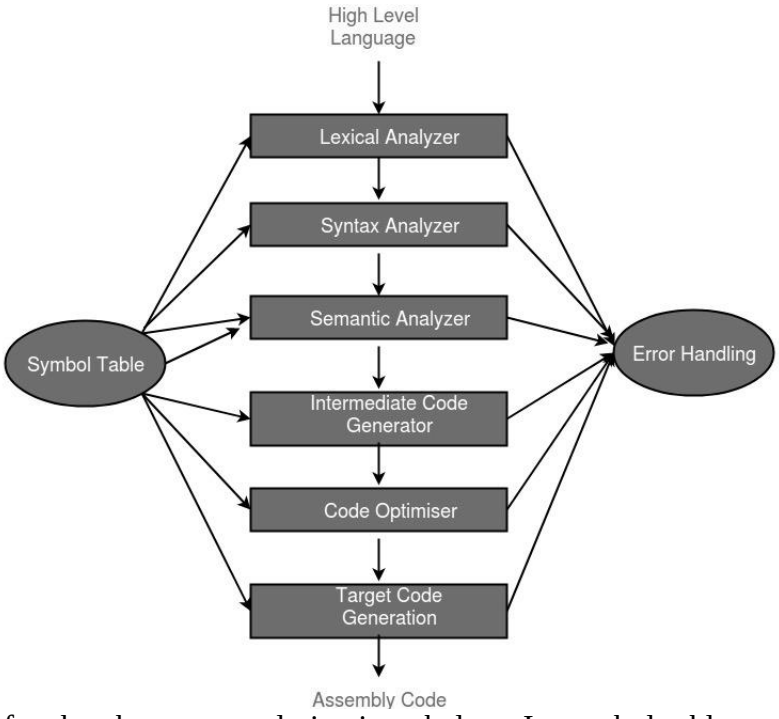
Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

Compiler Phases

Compiler phases like lexical analyzer, Syntax analyzer, Semantic Analyzer, Intermediate code generator, code Optimizer, and Target code generation. let's consider an example.

x = a+b*50



The symbol table for the above example is given below. In symbol table are clearly mentions the variable name and variable types.

S. No.	Variable name	Variable type
1	X	float
2	A	Float
3	B	Float

Now, here you will see how you can execute the compiler phase at each level and how it works.

Lexical Analyzer **:**

In this phase, you will see how you can tokenize the expression.

x -> Identifier- (id, 1)
= -> Operator - Assignment
a -> Identifier- (id, 2)
+ -> Operator - Binary Addition
b -> Identifier- (id, 3)
* -> Operator - Multiplication
50 -> Constant - Integer

Now, the final tokenized expression is given below.

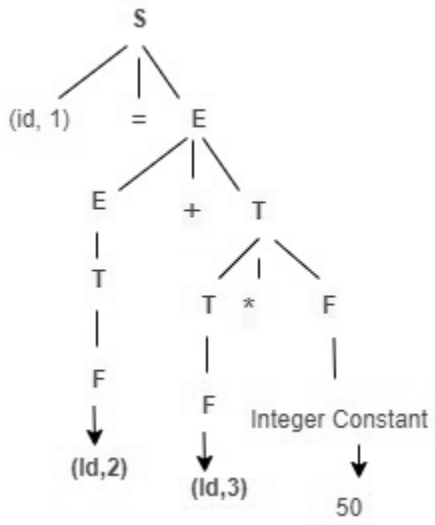
(id, 1) = (id, 2) + (Id, 3)*50

Syntax Analyzer **:**

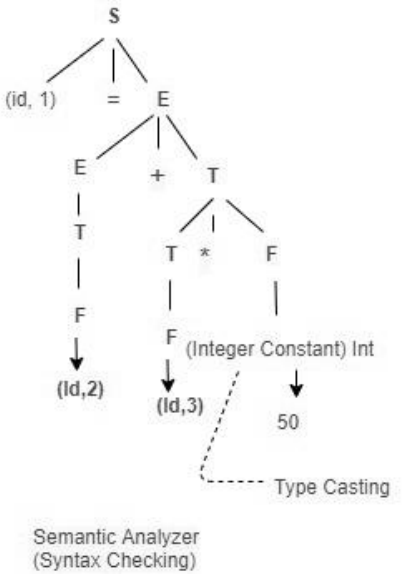
In this phase, you will see how you can check the syntax after tokenized the expression.

S -> Id = E
E -> E+T | T
T -> T*F | F
F -> Id | Integer constant

SDT for the above expression is given below.



Semantic Analyzer :
In this phase, you will see how you can check the type and semantic action for the syntax tree. Given below is the diagram of the semantic analyzer.



Intermediate Code Generator**:**
In this phase as an input, you will give a modified parse tree and as output after converting into Intermediate code will generate 3 -Address Code. Given below is an expression of the above-modified parse tree.

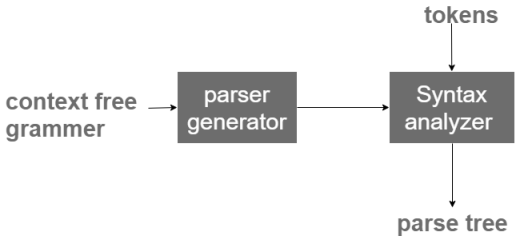
Three Address Code –

t1 = b * 50.0
t2 = a+t1
x = t2
Code Optimizer**:**
In this phase, you will see as an input will give 3 address code and as an output, you will see optimize code. Let's see how it will be converted.
t1 = b* 50.0
x = a+ t1
Target Code Generator:
It is the last phase and in this, you will see how you can convert the final expression into assembly code. so, that it will be easy to understand for the processor.
Mul
Add
Store

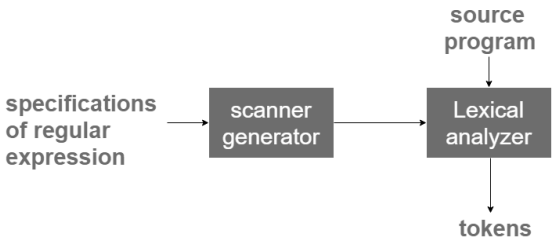
Compiler Construction Tools:

The compiler writer can use some specialized tools that help in implementing various phases of a compiler. These tools assist in the creation of an entire compiler or its parts. Some commonly used compiler construction tools include:

Parser Generator –
It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.
Example: PIC, EQM



Scanner Generator –
It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automaton to recognize the regular expression.
Example: Lex



Syntax directed translation engines –
It generates intermediate code with three-address format from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produce the intermediate code. In this, each node of the parse tree is associated with one or more translations.

Automatic code generators –
It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. A template matching process is used. Its equivalent machine language statement using templates replaces an intermediate language statement.

Data-flow analysis engines –
It is used in code optimization. Data flow analysis is a key part of the code optimization that gathers the information, which is the values that flow from one part of a program to another.

Compiler construction toolkits –
It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.